# SET10117 Multi-Agent Tutorial Timetable Generation

Calum Hamilton

40205163

### Introduction:

In this coursework, I was assigned the task of prototyping a decentralised multi-agent system which would assign timetable slots to students based on their preferences. Timetable slots are initially assigned to Student Agents (acting on behalf of a student) who attempt to swap their current slots until they receive preferential slots.

### Design:

My Design report will address the following five requirements:

1. A meaningful and relevant ontology, allowing agents to communicate
2. A communication protocol, allowing agents to communicate without revealing their preferences
3. A utility function, allowing the student to know when they’re satisfied with their slots
4. A strategy to determine which exchange requests to make, which to accept and which to reject
5. A metric to evaluate system effectiveness in terms of satisfying student preferences

#### 1. Ontology Design:

I include my Ontological Relationships in Appendix A. and examine the Ontological elements here.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Element** | **Properties** | **Data Type** | **Restrictions** | **Type** |
| **Pleased With** | Student Identifier | AID | Mandatory | Predicate |
| **Timeslot** | Day  Time | Integer  Integer | Mandatory  Mandatory | Concept |
| **Tutorial Group** | **Timeslot**  Tutorial ID  Class Size  Tutorial Number | Ontology Element  String  Integer  Integer | Mandatory  Mandatory  Mandatory | Concept |
| **Swap Initial** | **Tutorial Group**  Agent From | Ontology Element  AID | Mandatory  Mandatory | Agent Action |
| **Swap Final** | **Swap Initial**  **Tutorial To**  Agent To | Ontology Element  Ontology Element  AID | Mandatory  Mandatory  Mandatory | Agent Action |
| **Unhappy Slot** | **Swap Initial** | Ontology Element |  | Agent Action |
| **Slots Requested** | ArrayList<**Swap Final**> | List<Ontology Elements> | Mandatory | Predicate |
| **Happy With** | ArrayList<**Swap Final**> | List<Ontology Elements> | Mandatory | Agent Action |
| **Message Board** | ArrayList<**Swap Initial**> | List<Ontology Elements> |  | Concept |
| **Slots Available** | **Message Board** | Ontology Element | Mandatory | Predicate |

#### 2. Communication Protocol:

There were several ways I envisaged the communication protocol for my multi-agent system:

* A free-for-all messaging system
* Students sending slots to a central agent that broadcasted the slots
* Students sending slots to a central agent that iteratively swapped slots

My biggest concern when planning the communication protocol, was reducing the number of messages. This included live messages and total messages.

My concern with a free-for-all messaging system is it would pass too many messages (live and otherwise). This meant some messages may not be received and that the solution scaled badly. It may be reasonable for 5 agents to send and request slots at the same time, but there are nn messages sent per step, where n is the number of agents. Plus, anyone who’s been at a large Christmas gathering can tell you that everyone speaking over each other leads to inefficient communication.

An Agent receiving and broadcasting slots to Student Agents would help limit the number of messages of the prior protocol. Allowing for offering one slot per step, the complexity of this is still 2n per step (one message to the broadcaster, the message broadcasted, the broadcast reply and a reply from the broadcaster). You could attempt to limit messages by having disinterested agents not respond, but it still has most of the issues the free-for-all messaging system has.

An underlying reason for these issues is that Student Agents can’t expose their timeslot preferences (i.e. can only respond yes/no). Since preferences can’t be exposed, the receiver is unable to know gauge how much a slot would benefit each Student Agent and can’t choose the result that maximises social welfare. This led me to understand, the best approach is present a selection of slots and have the Student Agent choose which slots best suit them, as opposed to passing single slots and returning yes or no. This further points towards a centralised approach – having the ability to store and pass slots simultaneously.

I realised that the Communication Protocol would require a controller agent to limit the number of live messages and store a bank or ‘message board’ of unwanted slots. I started using a ‘Timetabling Agent’, to hold a list of slots for agents to swap. This was how I started to design my Ontology.

I detail the Communication Protocol design in the Sequence Diagram in Appendix B.

#### 3. Utility Function:

There were several ways that I saw the Agent utility being calculated:

* Individual calculation
* Mean utility
* Specialised calculation

You could individually calculate the utilities of each Student Agent module and use the calculated value to choose whether the Agent is ‘happy’ or needs to swap a slot by iterating through the values. This is suitable for selecting slots from a message board or comparing existing slots with a new slot. When it comes to evaluating an Agent’s ‘happiness’, it becomes more difficult with >1 slot.

Using a mean utility function, you could still use a calculation to compare slots individually for posting or evaluating swap requests. Additionally, it allows a mean happiness to be calculated based on the Agent’s slots. Mean utility appealed to me, as it makes sense that the ‘happiness’ of the Agent is based on the average slot happiness. From a human-based perspective: “I really hate that 5pm Friday tutorial, but my other slots are nice, so overall my timetable isn’t too bad”.

A specialised calculation, or an adaption of the mean utility, could be useful for the task. It’s difficult to visualise this ahead of time, but it would likely include additional dissatisfaction if an Agent couldn’t attend a slot.

##### Choice of Calculation:

I saw the choice as either use the preference number or using an inverse of the preference. For me, it made more sense using an inverted range to provide Agent preferences, since I’d like Student Agents to aim for 0. My Student Agent preferences range from 0-4:

1. Loves the slot (high preference)
2. Likes the slot
3. Indifferent about the slot (medium preference)
4. Doesn’t like the slot
5. Can’t attend the slot (low preference)

I found it made sense using a range of 0-4 instead of 0-2, to capture the range of human emotion (instead of liking/indifference/hating) more fully, without being too complex.

#### 4. Strategy for Considering Requests and Exchanges to Make

In the system I designed, Student Agents had three opportunities to swap slots:

1. Informing the Timetable Agent of slots they were interested in
2. Agreeing to/Refusing slots that other agents had requested
3. Posting new slots on the message board they didn’t want

I examine these more deeply below:

1. The Student Agent needed to evaluate its slots against slots on the message board. It would then select the best slot(s) and request those slots. I limited the number of unique slots a Student Agent could request to one per module. I found this a good balance between requesting enough slots each cycle and keeping the system running without having to iteratively verify a list of requests.
2. I had Student Agents refuse swaps if the new slot fitness was worse than their current slot fitness. They would only accept a new slot if it were better or equal to their current slot fitness. This meant that Agents would post slots they didn’t like but wouldn’t swap them if the one offered was worse. The reason for accepting equal slots is that the social welfare could increase, even if the Student didn’t directly benefit.
3. Agents check the message board and then post their worst slot if it’s not present and if they don’t love the slot – alternately post no slots.

#### 5. A Metric to Evaluate Overall Effectiveness of the System

I thought the best way to evaluate the system effectiveness was a Utilitarian approach in trying to understand the social welfare of Agents in the system.

Individually, I evaluate the ‘happiness’ of each Student. Since this is what I want the system to maximise, it seemed appropriate to average the ‘happiness’ of Students and get a Mean. The metric I’m using to evaluate overall system effectiveness is the mean value of the Utility Function – the social welfare of the system.

### Implementation

My Implementation Report will address the following three requirements:

1. Show the utility calculation, where it decides which swap requests to accept and which to reject, and state where the student’s preferences are represented
2. State where in the code the timetabling agent ensures that each student attends exactly one tutorial for each module.
3. For each conversation in your communication protocol, you should reference the relevant screenshot from the JADE sniffer, and state which agent behaviours implement it.

#### 1. Student Utility Calculation and Preference Exchange:

I originally planned to implement a mean calculation. The mean utility function worked very well on my first 3 test cases (less users and modules). After testing the mean utility on my fourth test case, I realised I was still accepting Student Agents if they had a slot but couldn’t attend. I tried to fix this by raising and dropping the fixed threshold for ‘happy students’, but this didn’t solve the issue. I decided that the mean calculation was unfit for purpose. I adapted a specialised calculation from the original mean value calculation I was using. The formula I settled on was:

Where *p* = Slot Preference (ranging 0 – 5) – 0 means the Student Agent loves the slot; 5 means the student can’t attend that slot. -5 is a low enough value where the student can never accept the slots without swapping all slots they can’t attend. Slots the student loves and can’t attend are heavily weighted. This calculation is shown in Appendix C.

Output values (for a single example):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Slot Pref:** | 0 - loves the slot | 1 - likes the slot | 2 - indifferent | 3 - doesn't like | 4 - can't attend |
| **Value:** | 1.0 | 0.5 | 0.33° | 0.25 | -5 |

I found if I started the utility function’s target happiness at a reachable near-optimal level (e.g. 0.7-0.9), Student Agents would swap several slots and then exit, without considering any other slots available. This left less Student Agents in the environment, with less opportunity for Student Agents to swap bad slots. I tested various happiness thresholds and found that 1.5 with a reduction of 0.02 upon each time they were notified to request slots was a good level. It didn’t run for too long and allowed Student Agents that had selected optimal slots to exit the environment early.

##### Preference Representation & Swap Requests:

I examine the three areas above, in design section 4. For clarity, I will re-explain the areas of calculation below:

1. Informing the Timetable Agent of slots they were interested in
2. Agreeing to/Refusing slots that other agents had requested
3. Posting new slots on the message board they didn’t want

Appendix D shows how the Student Agent selects ‘interested slots’ – the individual best slots to request for each tutorial class. This is in the **AwaitTimetableBehaviour** behaviour in the Student Agent class.

Appendix E shows how the Student Agent selects which slots to agree to swap and which to refuse. This is in the **AwaitSlotVerifyBehaviour** behaviour in the Student Agent class.

Appendix F shows how the Student Agent selects which new slots to post to the message board. This is in the **AwaitHappyBehaviour** behaviour in the Student Agent class.

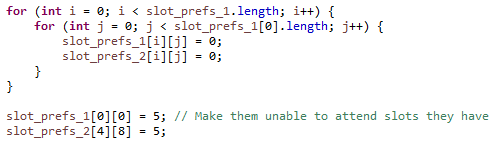
I examine the example conversations in section 3 of the implementation, but the screenshots of the test case 1 conversation are shown in Appendix G.

The Student Agent slot preferences are generated in a 2D array:



Corresponding to slot\_preferences[days][hours].

Shown below is how the slot preferences have been generated for the Test Case 1:



#### 2. Ensuring Unique Module Attendance:

When the program begins, Student Agents are each assigned their week’s slot preferences, and the tutorial slots which they will be attending. The Timetable Agent is also passed the tutorial slots (but not the slot preferences).

After requesting the list of Student Agents from the DF Agent, the Timetable Agent then sorts a list of Student Agents, before assigning the Agent AIDs to a 2D Array of ArrayLists. This allows the Timetable Agent to track the requests that Agents make and verify they requests are valid.

The StudentSlots structure is in the format: - StudentSlots[day][time].

It uses references to the TutorialMap – a HashMap with the Tutorial ID as keys and a corresponding number, referring to the slot ordering.

Appendix H shows the Timetable Agent checking:

* The SETID of the slot requested is the same as the SET ID of the slot being given
* The Student Agent owns the slot it’s trying to swap

Appendix I shows the Timetable Agent updating the slots for Requester and Poster Agents.

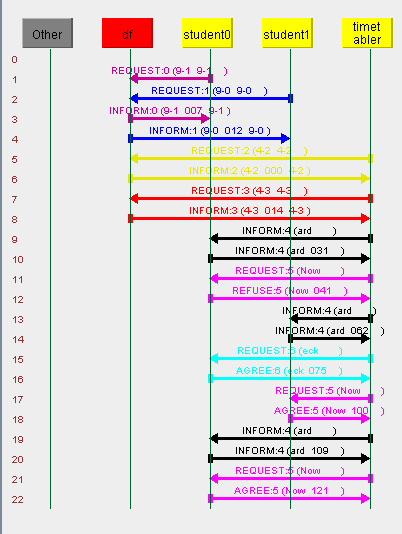
Appendix J shows the Timetable Agent checking to ensure that the Poster Agent owns the posted slot.

This approach will work provided the Timetable Agent is correctly assigned slots. In the slot assignment, when Student Agents are assigned modules, they won’t be assigned more than one module with a single SET ID.

When the Timetable Agent facilitates swaps from one Student Agent to another, it checks that the slots requested and offered belong to the same module, that the Poster Agent owns the slot being posted and the Requester Agent owns the slot it’s requesting.

#### 3. Exploring Agent and Timetable Agent Conversations

In this section, I explore the messages exchanged between two Student Agents and the Timetable Agent in a simple environment. Below is a screenshot of messages exchanged, taken through the Sniffer agent:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Line(s)** | **Agent from** | **Agent to** | **Purpose** | **Code Function** |
| 0-6 | All | DF | System registration | Register |
| 7-8 | Timetable | DF | Requesting Student list | FindStudentsBehaviour |
| 9 | Timetable | Student0 | Informing of timetable slots (0 slots total) | SwapStudentBehaviour |
| 10 | Student0 | Timetable | Informing of interested slots | AwaitTimetableBehaviour |
| 11 | Timetable | Student0 | Asking if unhappy or has slots | SwapStudentBehaviour |
| 12 | Student0 | Timetable | Unhappy – responds with slot | AwaitHappyBehaviour |
| 13 | Timetable | Student1 | Informing of timetable slots (1 slot total) | SwapStudentBehaviour |
| 14 | Student1 | Timetable | Informing of interested slots (Student0’s slot) | AwaitTimetableBehaviour |
| 15 | Timetable | Student0 | Ask if Student0 wishes to swap its slot for Student1’s slot | SwapStudentBehaviour |
| 16 | Student0 | Timetable | Agrees to swap for Student1’s slot | AwaitSlotVerifyBehaviour |
| 17 | Timetable | Student1 | Notifies of agreed swap and asks if happy | SwapStudentBehaviour |
| 18 | Student1 | Timetable | Happy – responds and terminates | AwaitHappyBehaviour |
| 19 | Timetable | Student0 | Informing of timetable slots (0 slots total) | SwapStudentBehaviour |
| 20 | Student0 | Timetable | Informing of interested slots | AwaitTimetableBehaviour |
| 21 | Timetable | Student0 | Asking if unhappy or has slots | SwapStudentBehaviour |
| 22 | Student0 | Timetable | Happy – responds and terminates | AwaitHappyBehaviour |

The conversation from lines 9-22 are captured in order in Appendix G. The implementation is faithful to the of the original Communication design.

### Testing

My Test Report will address the following three requirements, across my 4 test cases:

1. Show the number of students in the system and their preferences.
2. Show the number of tutorial groups for each module and their timeslots.
3. Show the number of students each tutorial can hold.

I show the preferences accordingly:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Preference** | **0** | **1** | **2** | **3** | **4** |
| **Colour** |  |  |  |  |  |

If there is a SET ID inside the coloured box, the Student Agent was initialised with that slot.

Note that for each example, the lower the Tutorial Number, the earlier in the week that tutorial is. For example, if SET010101 has a class Monday 9am, Tuesday 9am and Wednesday 9am; the Monday class would be number 0, the Tuesday class number 1 and the Wednesday class number 2.

My test sets are pre-defined. I will use between 2 and 6 ‘buckets’ for my Student Agents, to assign different initial slots and preferences to each. I will define these buckets where appropriate.

#### Test Case 1:

|  |  |  |
| --- | --- | --- |
| **Number of Students** | **Number of Modules** | **Number of Classes** |
| 2 | 1 | 2 |

My first test case was to swap two students between a Monday morning and a Friday afternoon slot. The goal of this test case was to see if two Student Agents would complete a swap of two slots they hated. I set the preferences as following:

##### Student 0 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** | SET010101 |  |  |  |  |  |  |  |  |
| **Tuesday** |  |  |  |  |  |  |  |  |  |
| **Wednesday** |  |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Student 1 Preferences:

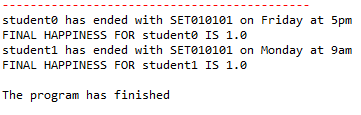
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** |  |  |  |  |  |  |  |  |  |
| **Tuesday** |  |  |  |  |  |  |  |  |  |
| **Wednesday** |  |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  | SET010101 |

##### Expected Output:

Social welfare is 1.

Student 1 ends with Tutorial 0 and Student 0 ends with Tutorial 1.

##### Actual Output:



#### Test Case 2:

My second test case increased the complexity, by increasing the number of classes and students. It’s also going to test how Student Agents react with different levels of slot preferences.

|  |  |  |
| --- | --- | --- |
| **Number of Students** | **Number of Modules** | **Number of Classes** |
| 6 | 1 | 3 |

##### Student Buckets:

|  |  |  |
| --- | --- | --- |
| **Bucket 1:** | **Bucket 2:** | **Bucket 3:** |
| Student 0, Student 3 | Student 1, Student 4 | Student 2, Student 5 |

The goal of this test case is to get Student Agents swapping their slots for a ‘sub-par’ slot, and then getting the Student Agent to swap the slot again for their preferred slot.

##### Bucket 1 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** | SET010101 |  |  |  |  |  |  |  |  |
| **Tuesday** |  |  |  |  |  |  |  |  |  |
| **Wednesday** |  |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Bucket 2 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** |  |  |  |  |  |  |  |  |  |
| **Tuesday** | SET010101 |  |  |  |  |  |  |  |  |
| **Wednesday** |  |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Bucket 3 Preferences:

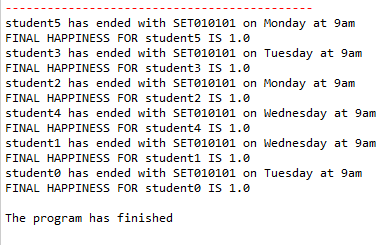
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** |  |  |  |  |  |  |  |  |  |
| **Tuesday** |  |  |  |  |  |  |  |  |  |
| **Wednesday** | SET010101 |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Expected Output:

Social welfare is 1.

Bucket 1 ends with Tutorial 0, Bucket 2 ends with Tutorial 1 and Bucket 3 ends with Tutorial 2.

##### Actual Output:



#### Test Case 3:

My third test case increased the complexity again, by increasing the of students and SETs. It’s going to further test how my system reacts to differing levels of slot preferences and multiple SETs.

|  |  |  |
| --- | --- | --- |
| **Number of Students** | **Number of Modules** | **Number of Classes** |
| 8 | 2 | 2 |

##### Student Buckets:

|  |  |
| --- | --- |
| **Bucket 1:** | **Bucket 2:** |
| Student 0, Student 2, Student 4, Student 6 | Student 1, Student 3, Student 5, Student 7 |

The goal of this test case is to get Student Agents swapping their one of their slots for a ‘decent’ slot and another for a ‘great’ slot of a different SET at a different time.

##### Bucket 1 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** | SET010101 | SET010102 |  |  |  |  |  |  |  |
| **Tuesday** |  |  |  |  |  |  |  |  |  |
| **Wednesday** |  |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Bucket 2 Preferences:

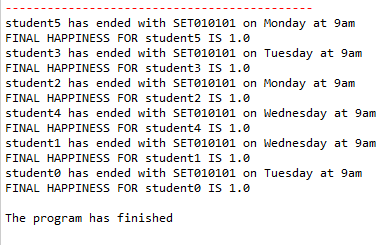
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** |  |  |  |  |  |  |  |  |  |
| **Tuesday** | SET010102 | SET010101 |  |  |  |  |  |  |  |
| **Wednesday** |  |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Expected Output:

Social welfare is ~0.6 (Mean of 1 and 0.25).

Bucket 1 ends with Tuesday morning tutorials and Bucket 2 ends with Monday morning tutorials.

##### Actual Output:



#### Test Case 4:

My fourth test case much further increased the complexity again. I increased the modules, increased the number of classes, and increased the number of students. I further altered the preferences. I tried to get this timetable as close to a ‘regular student’s’ timetable as possible – blocking off unwanted 5pm slots and setting a distribution of preferences.

|  |  |  |
| --- | --- | --- |
| **Number of Students** | **Number of Modules** | **Number of Classes** |
| 24 | 3 | 3 |

##### Student Buckets:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Bucket 1:** | **Bucket 2:** | **Bucket 3:** | **Bucket 4:** | **Bucket 5:** | **Bucket 6:** |
| Student 0  Student 6  Student 12  Student 18 | Student 1  Student 7 Student 13 Student 19 | Student 2  Student 8 Student 14 Student 20 | Student 3  Student 9 Student 15 Student 21 | Student 4  Student 10  Student 16  Student 22 | Student 5  Student 11 Student 17 Student 23 |

The goal of this test case is to get Student Agents swapping their one of their slots for a ‘decent’ slot and another for a ‘great’ slot of a different SET at a different time.

##### Bucket 1 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** |  |  |  |  |  |  |  |  |  |
| **Tuesday** | SET010101 |  | SET010103 |  |  |  |  |  |  |
| **Wednesday** |  | SET010102 |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Bucket 2 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** |  |  |  |  |  |  |  |  |  |
| **Tuesday** |  | SET010102 |  |  |  |  |  |  |  |
| **Wednesday** | SET010101 |  | SET010103 |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Bucket 3 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** | SET010101 | SET010102 |  |  |  |  |  |  |  |
| **Tuesday** |  |  |  |  |  |  |  |  |  |
| **Wednesday** |  |  | SET010103 |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Bucket 4 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** |  |  | SET010103 |  |  |  |  |  |  |
| **Tuesday** |  |  |  |  |  |  |  |  |  |
| **Wednesday** | SET010101 | SET010102 |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Bucket 5 Preferences:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** | SET010101 | SET010102 | SET010103 |  |  |  |  |  |  |
| **Tuesday** |  |  |  |  |  |  |  |  |  |
| **Wednesday** |  |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

##### Bucket 6 Preferences:

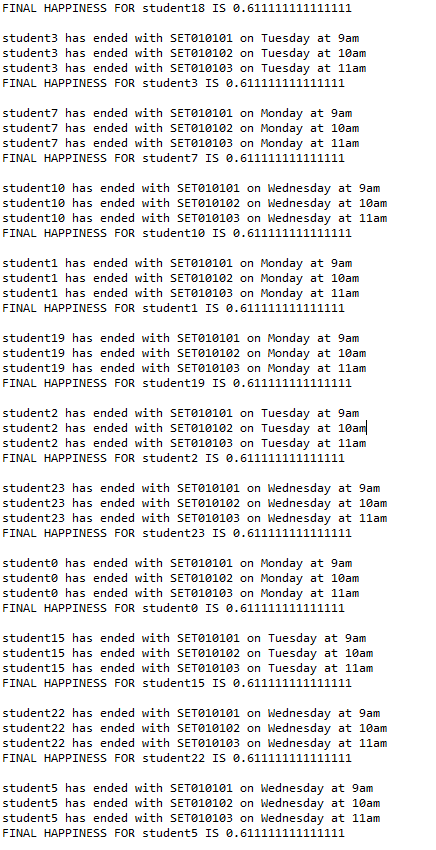
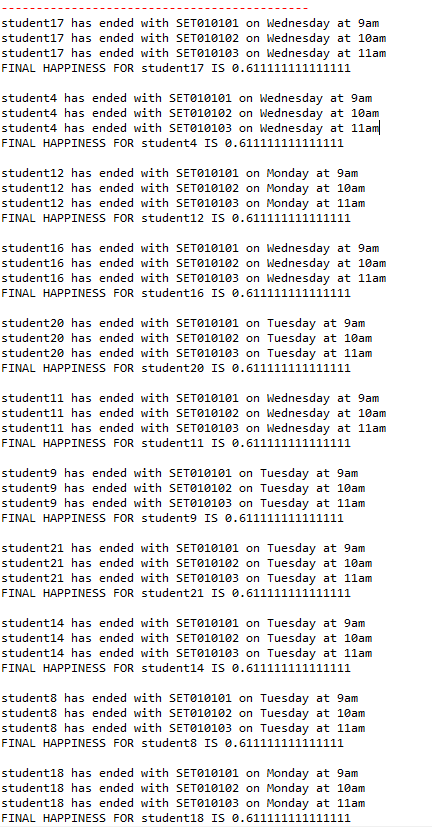
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **9am** | **10am** | **11am** | **12pm** | **1pm** | **2pm** | **3pm** | **4pm** | **5pm** |
| **Monday** |  |  |  |  |  |  |  |  |  |
| **Tuesday** | SET010101 | SET010102 | SET010103 |  |  |  |  |  |  |
| **Wednesday** |  |  |  |  |  |  |  |  |  |
| **Thursday** |  |  |  |  |  |  |  |  |  |
| **Friday** |  |  |  |  |  |  |  |  |  |

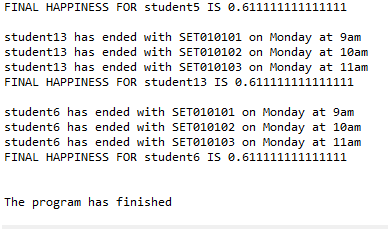
##### Expected Output:

Social welfare is ~0.6 (Mean of 1, 0.5 and 0.25).

Students end up with their preferred slots.

##### Actual Output:





### Evaluation and Future Work:

My Evaluation Report will address the following three requirements:

1. How effectively the system will interact and scale with difficulty
2. The advantages and disadvantages of a multi-agent systems approach to this problem
3. Considering 1 and 2, suggest and justify an improvement

#### Scalability and Efficiency of the System:

The approach I have taken scales well when additional agents are added. If additional agents were added to the system, the only increase would be the number of Student Agents the Timetable Agent would need to interact with. Since the Timetable Agent iteratively interacts with Student Agents, there would be no fundamental change as more Student Agents are added.

The system has also been shown to work with additional tutorials and classes, and I expect this to scale efficiently (at least to a number of modules that’s sensible).

#### Advantages and Disadvantages of the System

An advantage of a multi-agent systems approach to the problem is that Student preferences can be fully accounted for when slots are given to Student Agents. As each Student Agent is representative of one student, each student’s requirements and preferences can be fully considered. In theory you could survey students to find their preferred timeslots and then have the multi-agent system act this out.

Another advantage of a multi-agent systems approach is that student preferences can remain hidden from the Timetable Agent. Students may not want their availability known and this approach allows students to hide their preferences.

A disadvantage of a multi-agent systems approach is that it can be hard to differentiate between hard and soft constraints within the preferences for students. If some slots are subjectively better than others (Wednesday 2pm vs Friday 5pm), a lot of students may wish for the former slot and mark their availability as ‘not available’ for the latter. This could leave the slot being assigned to students who cannot attend the slot.

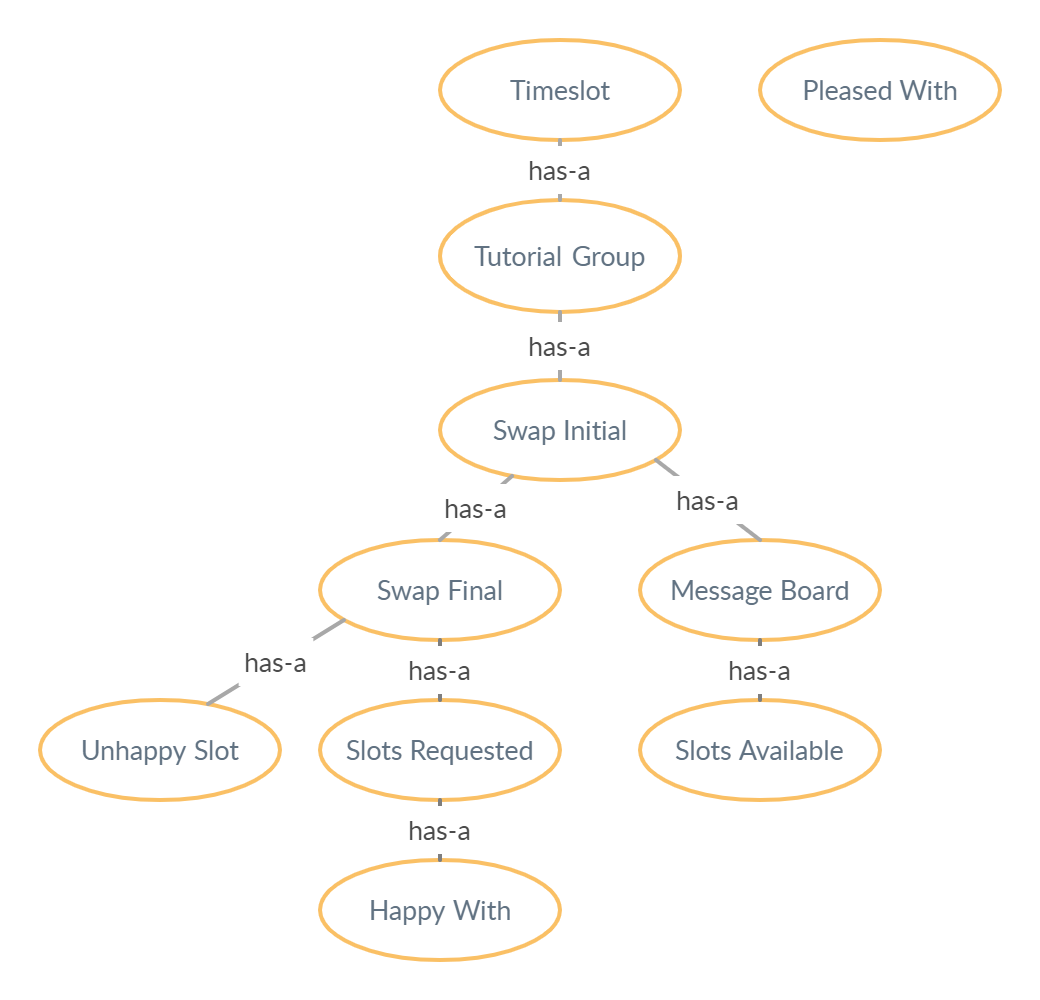
#### Improvement

An improvement I would make to the system would be to allow the posting of multiple slots by Student Agents at once. This would allow slots to be put on the message board quicker and allow them to be swapped earlier on (since slots have been revealed to potential requesters).

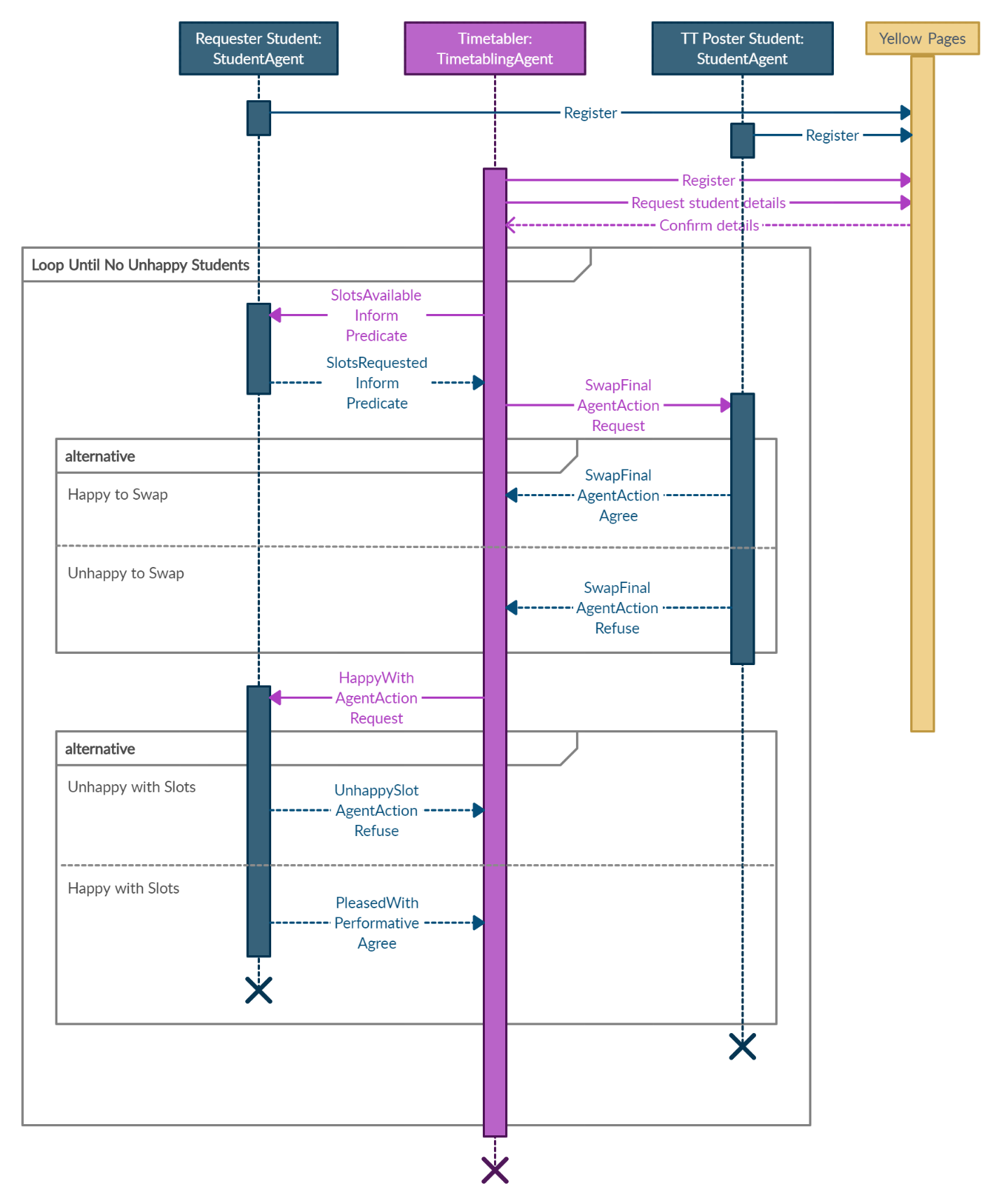
I would advocate for a system where the Timetable Agent would select slots on behalf of students, and then allow students to swap the slots with each other if they didn’t suit their own preferences (similar to the current system).

### Appendix:

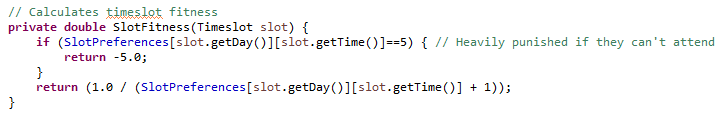
#### Appendix A – Ontology Association



#### Appendix B – Message Sequence Diagram



#### Appendix C – Student Agent Preference Calculation



#### Appendix D – Selecting and Requesting ‘Interested Slots’ from the Timetable Agent



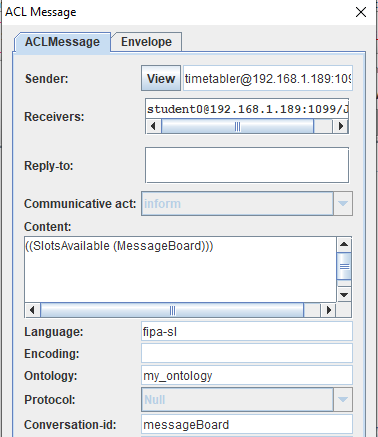
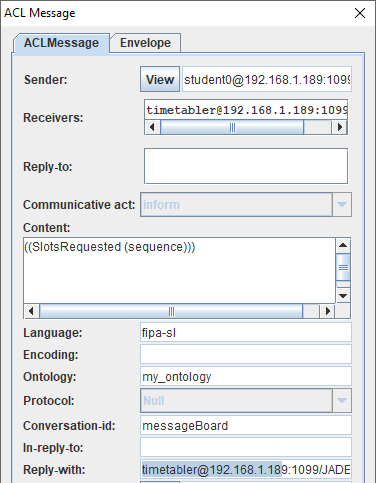
#### Appendix E – Agreeing to or Refusing ‘Requested Slots’ from the Timetable Agent

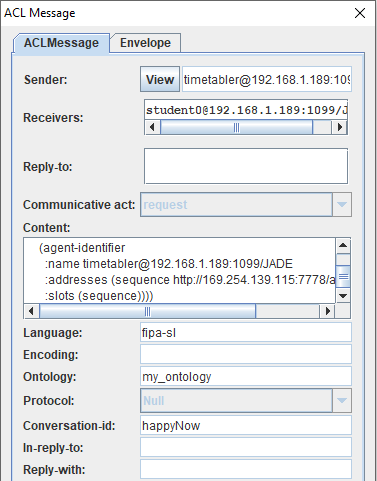
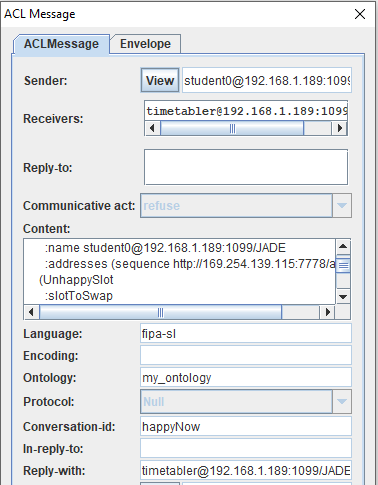


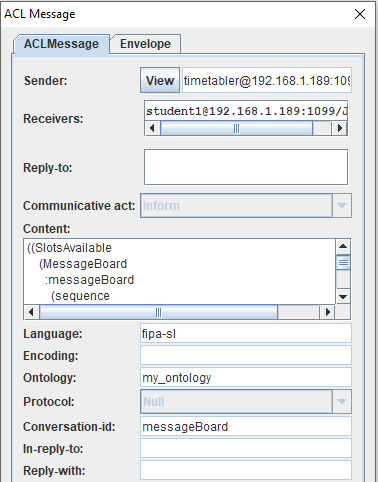
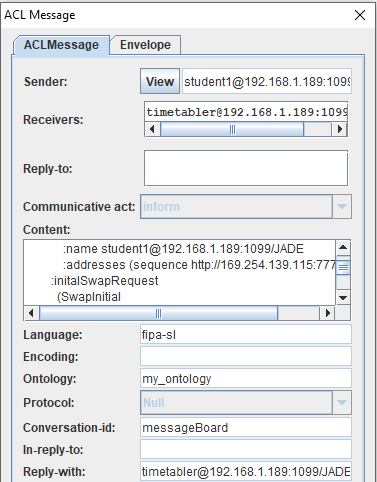
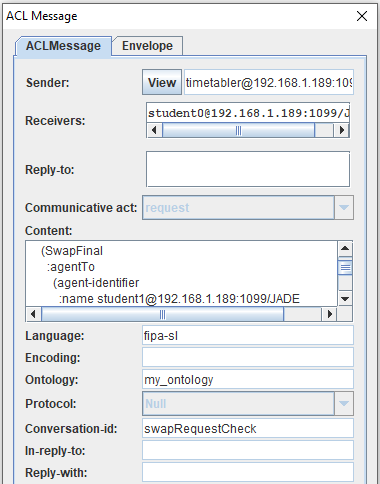
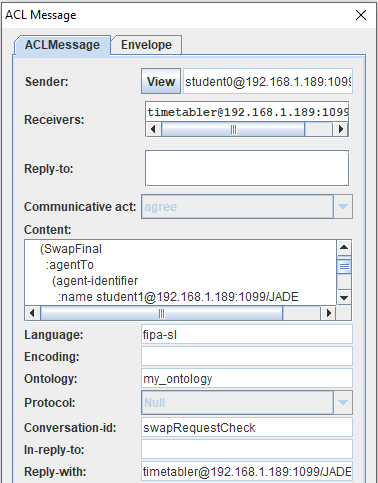
#### Appendix F – Checking new Slots to Post

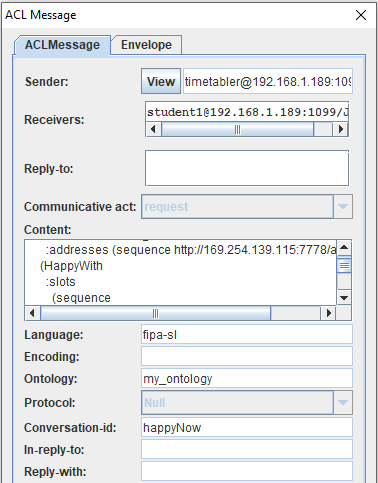
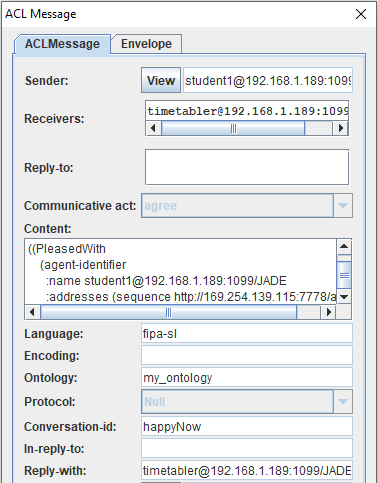
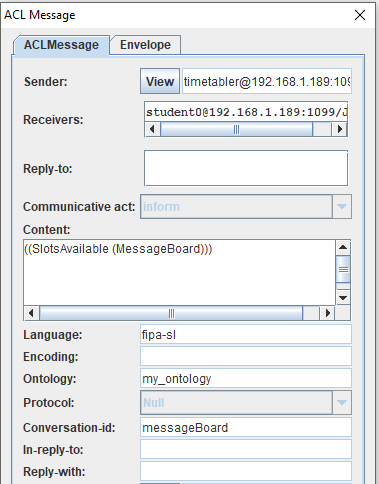
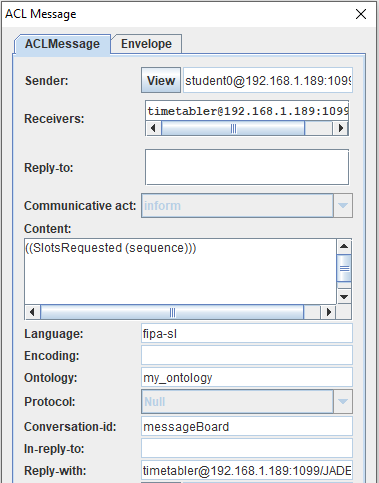


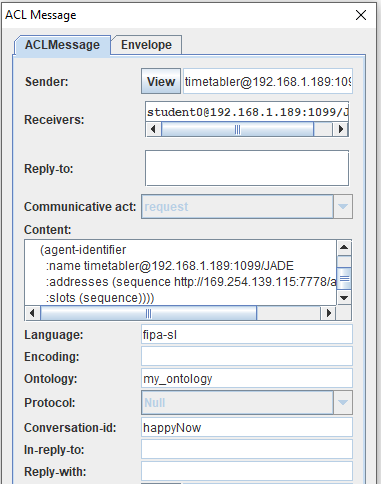
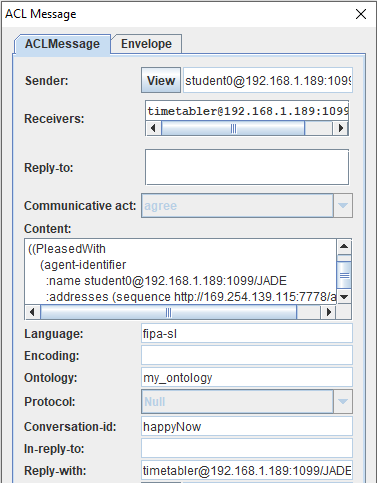
#### Appendix G – Example Conversations for Initial Example

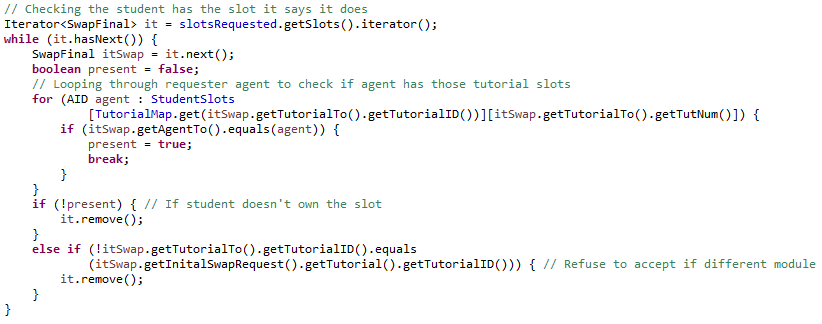
 

#### Appendix H – Denying Slot Request if Invalid



#### Appendix I – Updating the Requester Agent and Poster Agent’s slots



#### Appendix J – Checking the Poster Agent Owns the Posted Slot

#### Appendix K – The Student Agent Code

package timetable;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import jade.content.Concept;

import jade.content.ContentElement;

import jade.content.lang.Codec;

import jade.content.lang.Codec.CodecException;

import jade.content.lang.sl.SLCodec;

import jade.content.onto.Ontology;

import jade.content.onto.OntologyException;

import jade.content.onto.basic.Action;

import jade.core.AID;

import jade.core.Agent;

import jade.core.behaviours.CyclicBehaviour;

import jade.domain.DFService;

import jade.domain.FIPAException;

import jade.domain.FIPAAgentManagement.DFAgentDescription;

import jade.domain.FIPAAgentManagement.ServiceDescription;

import jade.lang.acl.ACLMessage;

import jade.lang.acl.MessageTemplate;

import timetable\_ontology.TimetableOntology;

import timetable\_ontology.elements.HappyWith;

import timetable\_ontology.elements.MessageBoard;

import timetable\_ontology.elements.PleasedWith;

import timetable\_ontology.elements.SlotsAvailable;

import timetable\_ontology.elements.SlotsRequested;

import timetable\_ontology.elements.SwapFinal;

import timetable\_ontology.elements.SwapInitial;

import timetable\_ontology.elements.Timeslot;

import timetable\_ontology.elements.TutorialGroup;

import timetable\_ontology.elements.UnhappySlot;

// Student Agent

public class StudentAgent extends Agent{

private Codec codec = new SLCodec();

private Ontology ontology = TimetableOntology.getInstance();

MessageBoard slotsMessageBoard;

private int[][] SlotPreferences;

private TutorialGroup[] CurrentTutorials;

private double meanHappinessTarget = 1.50; // Decreases slowly over time to allow the Agent to be 'Happy' later in the program

private double minMeanHappinessTarget = 0.5; // Minimum meanHappinessTarget

private HashMap<Integer, String> dayHM = new HashMap<>();

private HashMap<Integer, String> timeHM = new HashMap<>();

@Override

protected void setup() {

register();

initHMs();

getContentManager().registerLanguage(codec);

getContentManager().registerOntology(ontology);

Object[] arguments = getArguments();

SlotPreferences = (int[][]) arguments[0];

CurrentTutorials = (TutorialGroup[]) arguments[1];

addBehaviour(new AwaitTimetableBehaviour()); // Message board received by the Timetable Agent

addBehaviour(new AwaitSlotVerifyBehaviour()); // Agrees / Refuses Slot Request confirmations from the Timetable Agent

addBehaviour(new AwaitHappyBehaviour()); // Updates slots confirmed by the Timetable Agent and then confirms if happy or have a slot to give

}

// Waiting for the message board to be passed from the Timetable Agent

public class AwaitTimetableBehaviour extends CyclicBehaviour {

@Override

public void action() {

ACLMessage msg = myAgent.receive(MessageTemplate.MatchConversationId("messageBoard"));

if (msg != null) {

try {

//System.out.println(getAID().getLocalName() + " - Initiating timetable check"); //print out the message content in SL

ContentElement ce = getContentManager().extractContent(msg);

if (ce instanceof SlotsAvailable) {

SlotsAvailable available = (SlotsAvailable) ce;

slotsMessageBoard = available.getBoard();

List<SwapInitial> messageBoard = slotsMessageBoard.getMessageBoard();

// Checking for messages in the message board

if (messageBoard.size() > 0) {

SwapInitial[] interestedSlots = new SwapInitial[CurrentTutorials.length]; // Hold interested slots for each tutorial group

// Fill the 'interested slots'

for (int i = 0; i < interestedSlots.length; i++) {

for (SwapInitial boardSwap : messageBoard) {

if (boardSwap.getTutorial().getTutorialID().equals(CurrentTutorials[i].getTutorialID())) { // If matching tutorial

if (interestedSlots[i] == null) {

if (SlotFitness(boardSwap.getTutorial().getTimeslot()) >

SlotFitness(CurrentTutorials[i].getTimeslot())) { // If no current 'better slot'

interestedSlots[i] = boardSwap;

}

}else if (SlotFitness(boardSwap.getTutorial().getTimeslot()) >

SlotFitness(interestedSlots[i].getTutorial().getTimeslot())){ // Better than 'best' slot

interestedSlots[i] = boardSwap;

}

}

}

}

SlotsRequested wishedSlots = new SlotsRequested();

SwapFinal swapRequest;

ArrayList<SwapFinal> swapsRequested = new ArrayList<>();

// Checking interested slots

for (SwapInitial swap : interestedSlots){

if (swap != null) {

swapRequest = new SwapFinal();

swapRequest.setInitalSwapRequest(swap);

swapRequest.setAgentTo(getAID());

for (TutorialGroup currentTutorial : CurrentTutorials) {

if (currentTutorial.getTutorialID().equals(swap.getTutorial().getTutorialID())) {

swapRequest.setTutorialTo(currentTutorial); // Get the current tutorial

break;

}

}

swapsRequested.add(swapRequest);

}

}

wishedSlots.setSlots(swapsRequested);

ACLMessage reply = msg.createReply();

reply.setPerformative(ACLMessage.INFORM);

try {

getContentManager().fillContent(reply, wishedSlots);

send(reply); // >0 wished slots

}

catch (Exception e) {

e.printStackTrace();

}

}

else { // If message board has no slots

SlotsRequested wishedSlots = new SlotsRequested();

wishedSlots.setSlots(new ArrayList<>());

ACLMessage reply = msg.createReply(); // 0 wished slots

reply.setPerformative(ACLMessage.INFORM);

try {

getContentManager().fillContent(reply, wishedSlots);

send(reply);

}

catch (Exception e) {

e.printStackTrace();

}

}

}

}

catch(Exception e) {

e.printStackTrace();

}

}

else {

block();

}

}

}

// Responds to SwapFinal requests

public class AwaitSlotVerifyBehaviour extends CyclicBehaviour {

@Override

public void action() {

MessageTemplate mt = MessageTemplate.MatchConversationId("swapRequestCheck");

ACLMessage msg = myAgent.receive(mt);

if (msg != null) {

try {

//System.out.println(getAID().getLocalName() + " - Checking slot request");

ContentElement ce = getContentManager().extractContent(msg);

if (ce instanceof Action) {

Concept action = ((Action)ce).getAction();

if (action instanceof SwapFinal) {

SwapFinal swapRequest = (SwapFinal) action;

ACLMessage reply = msg.createReply();

getContentManager().fillContent(reply, swapRequest);

int count = 0;

// Checking if they want the tutorial

for (TutorialGroup tutorial : CurrentTutorials) {

if (tutorial.getTutorialID().equals(swapRequest.getTutorialTo().getTutorialID())) { // Checking same tutorial

if (SlotFitness(tutorial.getTimeslot()) <=

SlotFitness(swapRequest.getTutorialTo().getTimeslot())) { // If indifferent or prefers slot

reply.setPerformative(ACLMessage.AGREE); // Agree because want to swap slot

CurrentTutorials[count] = swapRequest.getTutorialTo();

break;

}

else {

reply.setPerformative(ACLMessage.REFUSE); // Refuse because not interested

break;

}

}

else {

reply.setPerformative(ACLMessage.REFUSE); // Refuse as not the same tutorial ID

}

count++;

}

Action actionReply = new Action();

actionReply.setAction(swapRequest);

actionReply.setActor(getAID());

try {

getContentManager().fillContent(reply, actionReply);

send(reply);

}

catch(CodecException codecE) {

codecE.printStackTrace();

}

catch(OntologyException oe) {

oe.printStackTrace();

}

}

}

}

catch (CodecException ce) {

ce.printStackTrace();

}

catch (OntologyException oe) {

oe.printStackTrace();

}

}

else {

block();

}

}

}

// Sends worst slot to Timetable Agent OR check AGENT IS HAPPY HERE

public class AwaitHappyBehaviour extends CyclicBehaviour {

@Override

public void action() {

MessageTemplate mt = MessageTemplate.MatchConversationId("happyNow");

ACLMessage msg = myAgent.receive(mt);

if (msg != null) {

//System.out.println(getAID().getLocalName() + " - Happy? or Post worst slot");

try {

ContentElement ce = getContentManager().extractContent(msg);

if (ce instanceof Action) {

Concept action = ((Action)ce).getAction();

if (action instanceof HappyWith) {

HappyWith happy = ((HappyWith)action);

for (SwapFinal slot : happy.getSlots()) {

for (int i = 0; i < CurrentTutorials.length; i++) {

if (slot.getTutorialTo().getTutorialID().equals(CurrentTutorials[i].getTutorialID())) {

CurrentTutorials[i] = slot.getInitalSwapRequest().getTutorial();

break;

}

}

}

double meanHappiness = 0;

// Get Mean unhappiness across slots

for (int i = 0; i < CurrentTutorials.length; i++) {

meanHappiness += SlotFitness(CurrentTutorials[i].getTimeslot());

}

meanHappiness /= CurrentTutorials.length;

// Reducing happiness required each run

if (meanHappinessTarget > minMeanHappinessTarget) {

meanHappinessTarget -= 0.02;

}

// If high enough overall happiness -- 0 or 1 slots accepted

if (meanHappiness <= meanHappinessTarget) {

// Checking slots that haven't been advertised yet

List<Integer> slotsNotAdvertised = new ArrayList<>();

// Getting current slots not on messageboard

for (int i = 0; i < CurrentTutorials.length; i++) {

boolean advertisedAlready = false;

for (SwapInitial item : slotsMessageBoard.getMessageBoard()) {

if (item.getAgentFrom().equals(getAID()) &&

item.getTutorial().getTutorialID().equals(CurrentTutorials[i].getTutorialID())) {

advertisedAlready = true;

continue;

}

}

if (!advertisedAlready) {

if (SlotFitness(CurrentTutorials[i].getTimeslot()) < 1.0) { // Don't post slot if happy already

slotsNotAdvertised.add(i);

}

}

}

int worstTutorial = -1; // Worst tutorial is slotsNotAdvertised positional reference

// Comparing slots - if all slots posted (or no slots student wishes to post) then nothing

if (slotsNotAdvertised.size() > 0) {

worstTutorial = 0;

// Select worst slot not currently on timetable

for (int i = 1; i < slotsNotAdvertised.size(); i++) {

if (SlotFitness(CurrentTutorials[slotsNotAdvertised.get(i)].getTimeslot()) <

SlotFitness(CurrentTutorials[slotsNotAdvertised.get(worstTutorial)].getTimeslot())) {

worstTutorial = i;

}

}

}

UnhappySlot slot = new UnhappySlot();

// Add slot found (if found) to 'Unhappy Slot' posting

if (worstTutorial != -1) {

SwapInitial si = new SwapInitial();

si.setAgentFrom(myAgent.getAID());

si.setTutorial(CurrentTutorials[slotsNotAdvertised.get(worstTutorial)]);

slot.setSlotToSwap(si);

}

Action newAction = new Action();

newAction.setActor(getAID());

newAction.setAction(slot);

ACLMessage reply = msg.createReply();

reply.setPerformative(ACLMessage.REFUSE);

try {

getContentManager().fillContent(reply, newAction);

send(reply);

}

catch(CodecException codecE) {

codecE.printStackTrace();

}

catch(OntologyException oe) {

oe.printStackTrace();

}

}else {

for (TutorialGroup tg : CurrentTutorials) {

System.out.println(getAID().getLocalName() + " has ended with " + tg.getTutorialID() + " on " + dayHM.get(tg.getTimeslot().getDay()) + " at " + timeHM.get(tg.getTimeslot().getTime()));

}

ACLMessage reply = msg.createReply();

PleasedWith pleased = new PleasedWith();

pleased.setStudent(getAID());

reply.setPerformative(ACLMessage.AGREE);

try {

getContentManager().fillContent(reply, pleased);

send(reply);

System.out.println("FINAL HAPPINESS FOR " + getAID().getLocalName() + " IS " + meanHappiness);

System.out.println("");

doDelete();

}

catch(CodecException codecE) {

codecE.printStackTrace();

}

catch(OntologyException oe) {

oe.printStackTrace();

}

}

}

}

}

catch (CodecException ce) {

ce.printStackTrace();

}

catch (OntologyException oe) {

oe.printStackTrace();

}

}

else {

block();

}

}

}

// Calculates timeslot fitness

private double SlotFitness(Timeslot slot) {

if (SlotPreferences[slot.getDay()][slot.getTime()]==5) { // Heavily punished if they can't attend

return -5.0;

}

return (1.0 / (SlotPreferences[slot.getDay()][slot.getTime()] + 1));

}

// Register with the DFD

protected void register() {

DFAgentDescription dfd = new DFAgentDescription ();

dfd.setName(getAID());

ServiceDescription sd = new ServiceDescription();

sd.setType("student-agent");

sd.setName( getLocalName() + "-student-agent");

dfd.addServices(sd);

try{

DFService.register(this , dfd);

}

catch(FIPAException e) {

e.printStackTrace();

}

}

private void initHMs() {

dayHM.put(0, "Monday");

dayHM.put(1, "Tuesday");

dayHM.put(2, "Wednesday");

dayHM.put(3, "Thursday");

dayHM.put(4, "Friday");

dayHM.put(5, "Saturday");

dayHM.put(6, "Sunday");

timeHM.put(0, "9am");

timeHM.put(1, "10am");

timeHM.put(2, "11am");

timeHM.put(3, "12pm");

timeHM.put(4, "1pm");

timeHM.put(5, "2pm");

timeHM.put(6, "3pm");

timeHM.put(7, "4pm");

timeHM.put(8, "5pm");

}

}

#### Appendix L – The Timetable Agent Code

**package** timetable;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.List;

**import** timetable\_ontology.TimetableOntology;

**import** timetable\_ontology.elements.HappyWith;

**import** timetable\_ontology.elements.MessageBoard;

**import** timetable\_ontology.elements.PleasedWith;

**import** timetable\_ontology.elements.SlotsAvailable;

**import** timetable\_ontology.elements.SlotsRequested;

**import** timetable\_ontology.elements.SwapFinal;

**import** timetable\_ontology.elements.SwapInitial;

**import** timetable\_ontology.elements.TutorialGroup;

**import** timetable\_ontology.elements.UnhappySlot;

**import** jade.content.Concept;

**import** jade.content.ContentElement;

**import** jade.content.lang.Codec;

**import** jade.content.lang.Codec.CodecException;

**import** jade.content.lang.sl.SLCodec;

**import** jade.content.onto.Ontology;

**import** jade.content.onto.OntologyException;

**import** jade.content.onto.basic.Action;

**import** jade.core.AID;

**import** jade.core.Agent;

**import** jade.core.behaviours.Behaviour;

**import** jade.core.behaviours.OneShotBehaviour;

**import** jade.domain.DFService;

**import** jade.domain.FIPAException;

**import** jade.domain.FIPAAgentManagement.DFAgentDescription;

**import** jade.domain.FIPAAgentManagement.ServiceDescription;

**import** jade.lang.acl.ACLMessage;

**import** jade.lang.acl.MessageTemplate;

// Timetabling agent

**public** **class** TimetablingAgent **extends** Agent{

**private** Codec codec = **new** SLCodec();

**private** Ontology ontology = TimetableOntology.*getInstance*();

**private** MessageBoard messageBoard = **new** MessageBoard(); // WHEN MESSAGEBOARD PASSED - NEED TO TAKE ALL ELEMENTS OF THIS BOARD AND ADD THEM TO A NEW BOARD

// Tracks Student Slots

**private** ArrayList<AID>[][] StudentSlots;

// Maps

**private** HashMap<String, Integer> TutorialMap = **new** HashMap<>();

// Keep the 'happy students' since you could run over them in theory to check if there's a few unhappy agents left..

// ..may find an even better slot

**private** List<AID> happyAgents = **new** ArrayList<AID>(); // StudentAgents 'happy' with their tutorial assignment

**private** List<AID> unhappyAgents = **new** ArrayList<AID>(); // StudentAgents 'unhappy' with their tutorial assignment

@SuppressWarnings("unchecked")

@Override

**protected** **void** setup() {

register();

getContentManager().registerLanguage(codec);

getContentManager().registerOntology(ontology);

Object[] arguments = getArguments();

List<String> studentString = (List<String>) arguments[0];

List<TutorialGroup[]> tutorialList = (List<TutorialGroup[]>) arguments[1];

doWait(5000); // Wait for student agents to load

addBehaviour(**new** FindStudentsBehaviour(**this**)); // Finds the students

addBehaviour(**new** SetupTTBehaviour(studentString, tutorialList));

addBehaviour(**new** SwapStudentSlotBehaviour()); // Runs the Student Swapping Behaviour

}

// Runs the Student Swapping Behaviour for the whole agent

**public** **class** SwapStudentSlotBehaviour **extends** Behaviour {

**private** AID currentAgent;

**private** **int** studentIndex = 0;

**private** **int** step = 0;

**private** SlotsRequested slotsRequested = **new** SlotsRequested(); // Slots requested by StudentAgents

**private** SlotsRequested slotsConfirmed = **new** SlotsRequested(); // Slots confirmed by TimetablingAgent

**private** SlotsRequested slotsDenied = **new** SlotsRequested(); // Slots denied by StudentAgents

@Override

**public** **void** action() {

**switch**(step) {

**case** 0:

currentAgent = unhappyAgents.get(studentIndex);

addBehaviour(**new** InitiateStudentBehaviour(currentAgent)); // Notifies the swapperAgent they're 'live'

step++;

**break**;

**case** 1:

MessageTemplate swapConfirmMt = MessageTemplate.*and*(MessageTemplate.*MatchConversationId*("messageBoard")

,MessageTemplate.*MatchSender*(currentAgent)); // Listens for request from 'live' agent

ACLMessage msg = myAgent.receive(swapConfirmMt);

**if** (msg != **null**) {

**try** {

ContentElement ce = getContentManager().extractContent(msg);

**if** (ce **instanceof** SlotsRequested) {

slotsRequested = ((SlotsRequested)ce);

step++;

// Checking the student has the slot it says it does

Iterator<SwapFinal> it = slotsRequested.getSlots().iterator();

**while** (it.hasNext()) {

SwapFinal itSwap = it.next();

**boolean** present = **false**;

// Looping through requester agent to check if agent has those tutorial slots

**for** (AID agent : StudentSlots

[TutorialMap.get(itSwap.getTutorialTo().getTutorialID())][itSwap.getTutorialTo().getTutNum()]) {

**if** (itSwap.getAgentTo().equals(agent)) {

present = **true**;

**break**;

}

}

**if** (!present) { // If student doesn't own the slot

it.remove();

}

**else** **if** (!itSwap.getTutorialTo().getTutorialID().equals

(itSwap.getInitalSwapRequest().getTutorial().getTutorialID())) { // Refuse to accept if different module

it.remove();

}

}

}

}

**catch** (Exception e) {

e.printStackTrace();

}

}

**else** {

block();

}

**break**;

**case** 2:

slotsConfirmed.setSlots(**new** ArrayList<>()); // Setting slotsConfirmed slots

slotsDenied.setSlots(**new** ArrayList<>());

// This allows the Timetable Agent to only ask for slots if slots have been selected

// (Reduced message passing)

**if** (slotsRequested.getSlots().size() == 0) { // Go to step requesting slots if no slots selected

step=4;

}

**else** {

addBehaviour(**new** VerifySlotBehaviour(slotsRequested)); // Verifies slot with swappedAgent

step++;

}

**break**;

**case** 3:

MessageTemplate swapRequestFinalMt = MessageTemplate.*MatchConversationId*("swapRequestCheck"); // Listens for response from swappedAgent

msg = myAgent.receive(swapRequestFinalMt);

**if** (msg != **null**) { // Check for swapped slots and un-swapped slots

**try** {

ContentElement ce = getContentManager().extractContent(msg);

**if** (ce **instanceof** Action) {

Concept action = ((Action)ce).getAction();

**if** (ce **instanceof** Action) {

SwapFinal swapRequest = (SwapFinal) action;

**if** (msg.getPerformative() == ACLMessage.***AGREE***) {

slotsConfirmed.addSlot(swapRequest);

}**else** {

slotsDenied.addSlot(swapRequest);

}

}

}

**if** ((slotsConfirmed.getSlots().size() + slotsDenied.getSlots().size()) == slotsRequested.getSlots().size()) { // Increase step if received all replies

// Updating the slots for the Requester Agent and the Poster Agent

**for** (SwapFinal swap : slotsConfirmed.getSlots()) {

// Update requester agent to the new slot

Iterator<AID> it = StudentSlots[TutorialMap.get(swap.getTutorialTo().getTutorialID())]

[swap.getTutorialTo().getTutNum()].iterator();

**while** (it.hasNext()) {

AID aidRem = it.next();

**if** (aidRem.equals(swap.getAgentTo())) {

it.remove();

**break**;

}

}

StudentSlots[TutorialMap.get(swap.getTutorialTo().getTutorialID())]

[swap.getTutorialTo().getTutNum()].add(swap.getInitalSwapRequest().getAgentFrom());

// Update the initial agent on the timetable to the agreed slot

Iterator<AID> receiverit = StudentSlots[TutorialMap.get(swap.getInitalSwapRequest().getTutorial().getTutorialID())]

[swap.getInitalSwapRequest().getTutorial().getTutNum()].iterator();

**while** (receiverit.hasNext()) {

AID aidRem = receiverit.next();

**if** (aidRem.equals(swap.getInitalSwapRequest().getAgentFrom())) {

receiverit.remove();

**break**;

}

}

StudentSlots[TutorialMap.get(swap.getInitalSwapRequest().getTutorial().getTutorialID())]

[swap.getInitalSwapRequest().getTutorial().getTutNum()].add(swap.getAgentTo());

}

step++;

}

}

**catch** (Exception e) {

e.printStackTrace();

}

}

**else** {

block();

}

**break**;

**case** 4:

addBehaviour(**new** InitiateStudentHappyBehaviour(slotsConfirmed, currentAgent)); // Initiates response verification with swapperAgent

step++;

**break**;

**case** 5:

**boolean** happy = **false**;

**boolean** done = **false**;

MessageTemplate SlotRequestMt = MessageTemplate.*and*

(MessageTemplate.*MatchConversationId*("happyNow"),MessageTemplate.*MatchSender*(currentAgent));

msg = myAgent.receive(SlotRequestMt);

**if** (msg != **null**) {

**try** {

ContentElement ce = getContentManager().extractContent(msg);

**if** (ce **instanceof** Action) {

Concept action = ((Action)ce).getAction();

**if** (action **instanceof** UnhappySlot) {

UnhappySlot slot = (UnhappySlot) action;

SwapInitial initSwap = slot.getSlotToSwap();

**if** (initSwap != **null**) {

**boolean** present = **false**;

// Checking if duplicate slot

**for** (SwapInitial checkSlot : messageBoard.getMessageBoard()) {

**if** (checkSlot.getTutorial().getTutorialID().equals

(initSwap.getTutorial().getTutorialID()) && checkSlot.getAgentFrom().equals(initSwap.getAgentFrom())) {

present = **true**;

}

}

// Checking if timetabler has student as owning slot

**boolean** found = **false**;

**for** (AID student : StudentSlots

[TutorialMap.get(initSwap.getTutorial().getTutorialID())][initSwap.getTutorial().getTutNum()]) {

**if** (student.equals(initSwap.getAgentFrom())) {

found = **true**;

**break**;

}

}

// If not found, student not present

**if** (!found) {

present = **false**;

}

**if** (!present) {

messageBoard.addToMessageBoard(initSwap);

}

}

done = **true**;

step = 0;

}

}

**else** **if** (ce **instanceof** PleasedWith) { // Agent is pleased with slot

happy = **true**;

done = **true**;

step = 0;

}

}

**catch** (Exception e) {

e.printStackTrace();

}

}

**else** {

block();

}

// If happy, add to happier list increment student index

**if** (done) {

**if** (happy) {

// Remove agent's slots from the timetable if they're happy

List<Integer> slotLocs = **new** ArrayList<>();

**for** (**int** i = 0; i < messageBoard.getMessageBoard().size(); i++) {

**if** (messageBoard.getMessageBoard().get(i).getAgentFrom().equals(currentAgent)) {

slotLocs.add(i);

}

}

// Remove slot locations from the message board

// and then negatively increment all slot locations

**for** (**int** i = 0; i < slotLocs.size(); i++) {

messageBoard.removeFromMessageBoard(slotLocs.get(i));

**for** (**int** j = i+1; j < slotLocs.size(); j++) {

slotLocs.set(j, slotLocs.get(j)-1);

}

}

happyAgents.add(currentAgent);

unhappyAgents.remove(currentAgent); // Remove agent

}

**if** (studentIndex >= (unhappyAgents.size()-1)) { // -1 because used for indexing

studentIndex = 0;

}**else** {

studentIndex++;

}

}

**break**;

}

}

@Override

**public** **boolean** done() {

**if** (unhappyAgents.size() > 0) {

**return** **false**;

}

System.***out***.println("");

System.***out***.println("The program has finished");

doDelete();

**return** **true**;

}

@Override

**public** **void** reset() {

System.***out***.println("Resetting program");

**super**.reset();

step = 0;

}

}

// Notifies the swapperAgent they're 'live'

// Receives a done message if students are finished their activity

**public** **class** InitiateStudentBehaviour **extends** OneShotBehaviour {

**private** AID studentAid;

// Called at the moment the behaviour is added

**public** InitiateStudentBehaviour (AID studentAgent) {

**this**.studentAid = studentAgent;

}

@Override

**public** **void** action() {

List<SwapInitial> siList = **new** ArrayList<SwapInitial>();

**for** (SwapInitial m : messageBoard.getMessageBoard()) {

siList.add(m);

}

MessageBoard mb = **new** MessageBoard();

mb.setMessageBoard(siList);

SlotsAvailable available = **new** SlotsAvailable();

available.setBoard(mb);

ACLMessage msg = **new** ACLMessage(ACLMessage.***INFORM***);

msg.addReceiver(studentAid);

msg.setConversationId("messageBoard");

msg.setLanguage(codec.getName());

msg.setOntology(ontology.getName());

**try** {

getContentManager().fillContent(msg, available);

send(msg);

}

**catch** (CodecException ce) {

ce.printStackTrace();

}

**catch** (OntologyException oe) {

oe.printStackTrace();

}

}

}

// Confirms the validity of the slots requested with other agents

**public** **class** VerifySlotBehaviour **extends** OneShotBehaviour {

SlotsRequested slotsRequested;

**public** VerifySlotBehaviour(SlotsRequested slotsRequested) {

**this**.slotsRequested = slotsRequested;

}

@Override

**public** **void** action() {

**for** (SwapFinal swap : slotsRequested.getSlots()) {

ACLMessage msg = **new** ACLMessage(ACLMessage.***REQUEST***);

msg.setLanguage(codec.getName());

msg.setOntology(ontology.getName());

msg.addReceiver(swap.getInitalSwapRequest().getAgentFrom());

msg.setConversationId("swapRequestCheck");

Action action = **new** Action();

action.setAction(swap);

action.setActor(getAID()); // set itself as the actor - since it's requesting swap

**try** {

getContentManager().fillContent(msg, action);

send(msg);

}

**catch**(CodecException codecE) {

codecE.printStackTrace();

}

**catch**(OntologyException oe) {

oe.printStackTrace();

}

}

}

}

// Initiates response verification with the swapperAgent

**public** **class** InitiateStudentHappyBehaviour **extends** OneShotBehaviour {

SlotsRequested slotsConfirmed;

AID receiver;

**public** InitiateStudentHappyBehaviour(SlotsRequested slotsConfirmed, AID agentTo) {

**this**.slotsConfirmed = slotsConfirmed;

**this**.receiver = agentTo;

}

@Override

**public** **void** action() {

**try** {

HappyWith happy = **new** HappyWith();

happy.setSlots(**new** ArrayList<>());

// Make sure slots are removed from the message board if they've been swapped to another agent

**for** (SwapFinal slot : slotsConfirmed.getSlots()) {

happy.addSlots(slot);

**for** (**int** i = 0; i < messageBoard.getMessageBoard().size(); i++) {

**if** (slot.getInitalSwapRequest().getAgentFrom().equals(messageBoard.getMessageBoard().get(i).getAgentFrom()) &&

slot.getInitalSwapRequest().getTutorial().getTutorialID().equals(messageBoard.getMessageBoard().get(i).getTutorial().getTutorialID())) {

messageBoard.getMessageBoard().remove(i);

**break**;

}

}

}

ACLMessage msg = **new** ACLMessage(ACLMessage.***REQUEST***);

msg.setLanguage(codec.getName());

msg.setOntology(ontology.getName());

msg.addReceiver(receiver);

msg.setConversationId("happyNow");

Action action = **new** Action();

action.setAction(happy);

action.setActor(getAID());

getContentManager().fillContent(msg, action);

send(msg);

}

**catch**(CodecException codecE) {

codecE.printStackTrace();

}

**catch**(OntologyException oe) {

oe.printStackTrace();

}

}

}

// Finds the Student Agents and adds them to the UnhappyAgents list

**public** **class** FindStudentsBehaviour **extends** OneShotBehaviour {

**public** FindStudentsBehaviour(Agent a) {

**super**(a);

}

@Override

**public** **void** action() {

DFAgentDescription studentTemplate = **new** DFAgentDescription();

ServiceDescription sd = **new** ServiceDescription();

sd.setType("student-agent");

studentTemplate.addServices(sd);

**try**{

unhappyAgents.clear();

DFAgentDescription[] agentsType1 = DFService.*search*(myAgent,studentTemplate);

**for**(**int** i=0; i<agentsType1.length; i++){

unhappyAgents.add(agentsType1[i].getName()); // this is the AID

}

}

**catch**(FIPAException e) {

e.printStackTrace();

}

}

}

// Sets up the timetable agent with a list of all the modules

**public** **class** SetupTTBehaviour **extends** OneShotBehaviour {

List<String> students;

List<TutorialGroup[]> tutorials;

**public** SetupTTBehaviour(List<String> students, List<TutorialGroup[]> tutorials) {

**this**.students = students;

**this**.tutorials = tutorials;

}

@SuppressWarnings("unchecked")

@Override

**public** **void** action() {

// Gets the maximum number of tutorials and tutorial groups

**int** maxTuts = 0;

List<AID> tempAIDs = **new** ArrayList<>();

List<String> tempTuts = **new** ArrayList<>();

// Getting Student AIDs in order

**for** (String student : students) {

**for** (AID agent : unhappyAgents) {

**if** (agent.getName().equals(student)) {

tempAIDs.add(agent);

**break**;

}

}

}

// Getting the Unique Tutorials in a sorted ArrayList

**for** (TutorialGroup[] group : tutorials) {

**for** (TutorialGroup tutorial : group) {

**if** (tutorial.getTutNum() > maxTuts) {

maxTuts = tutorial.getTutNum();

}

**if** (!tempTuts.contains(tutorial.getTutorialID())) {

tempTuts.add(tutorial.getTutorialID());

}

}

}

maxTuts++; // Since the tutorial numbers are 0 upwards

Collections.*sort*(tempTuts);

// Adding tutorials to the Tutorial HashMap

**for** (**int** i = 0; i < tempTuts.size(); i++) {

TutorialMap.put(tempTuts.get(i), i);

}

StudentSlots = **new** ArrayList[TutorialMap.size()][maxTuts]; // Initialising the student slots

**for** (**int** i = 0; i < TutorialMap.size(); i++) {

**for** (**int** j = 0; j < maxTuts; j++) {

StudentSlots[i][j] = **new** ArrayList<AID>(); // Initialise the ArrayList correctly for each element

}

}

**int** loopVal = 0;

// Adding students to the correct tutorial group

**for** (**int** i = 0; i < tempAIDs.size(); i++){

**for** (**int** j = 0; j < TutorialMap.size(); j++) { // SET ID HashMap (e.g. SET010101, SET010102)

// Adds Students to the 2d array ArrayList based on tutorial and class number (allows tracking of students in tutorials)

StudentSlots[TutorialMap.get(tutorials.get(loopVal)[j].getTutorialID())][tutorials.get(loopVal)[j].getTutNum()].add(tempAIDs.get(i));

}

// LoopVal tracks the correct arguments (e.g. 9 students created with 3 arguments)

**if** (loopVal >= tutorials.size()-1) {

loopVal = 0;

}**else** {

loopVal++;

}

}

}

}

// Register with the DFD

**protected** **void** register() {

DFAgentDescription dfd = **new** DFAgentDescription ();

dfd.setName(getAID());

ServiceDescription sd = **new** ServiceDescription();

sd.setType("timetabling-agent");

sd.setName( getLocalName() + "-timetabling-agent");

dfd.addServices(sd);

**try**{

DFService.*register*(**this** , dfd);

}

**catch**(FIPAException e) {

e.printStackTrace() ;

}

}

}

#### Appendix M – Ontology Code

##### HappyWith Code

package timetable\_ontology.elements;

import java.util.ArrayList;

import jade.content.AgentAction;

import jade.content.onto.annotations.Slot;

public class HappyWith implements AgentAction{

private ArrayList<SwapFinal> slots;

@Slot (mandatory = true)

public ArrayList<SwapFinal> getSlots() {

return slots;

}

public void setSlots(ArrayList<SwapFinal> slots) {

this.slots = slots;

}

public void addSlots(SwapFinal slots) {

this.slots.add(slots);

}

}

##### MessageBoard Code

package timetable\_ontology.elements;

import java.util.ArrayList;

import java.util.List;

import jade.content.Concept;

import jade.content.onto.annotations.Slot;

public class MessageBoard implements Concept {

private List<SwapInitial> messageBoard = new ArrayList<SwapInitial>();

public void setMessageBoard(List<SwapInitial> message) {

messageBoard = message;

}

public List<SwapInitial> getMessageBoard() {

return messageBoard;

}

public void addToMessageBoard(SwapInitial message) {

this.messageBoard.add(message);

}

public void removeFromMessageBoard(int index) {

messageBoard.remove(index);

}

}

##### PleasedWith Code

package timetable\_ontology.elements;

import java.util.List;

import jade.content.Predicate;

import jade.core.AID;

public class PleasedWith implements Predicate{

private AID student;

public AID getStudent() {

return student;

}

public void setStudent(AID student) {

this.student = student;

}

}

##### SlotsAvailable Code

package timetable\_ontology.elements;

import jade.content.Predicate;

import jade.content.onto.annotations.Slot;

import jade.core.AID;

public class SlotsAvailable implements Predicate {

private MessageBoard board;

###### 

@Slot (mandatory = true)

public MessageBoard getBoard() {

return board;

}

public void setBoard(MessageBoard board) {

this.board = board;

}

}

##### SlotsRequested Code

###### package timetable\_ontology.elements;

###### import java.util.ArrayList;

###### import jade.content.Predicate;

###### import jade.content.onto.annotations.Slot;

###### public class SlotsRequested implements Predicate {

###### private ArrayList<SwapFinal> slots;

###### 

###### @Slot (mandatory = true)

###### public ArrayList<SwapFinal> getSlots() {

###### return slots;

###### }

###### 

###### public void setSlots(ArrayList<SwapFinal> slots) {

###### this.slots = slots;

###### }

###### 

###### public void addSlot(SwapFinal slot) {

###### this.slots.add(slot);

###### }

###### }

##### SwapFinal Code

###### package timetable\_ontology.elements;

###### import jade.content.AgentAction;

###### import jade.content.onto.annotations.Slot;

###### import jade.core.AID;

###### public class SwapFinal implements AgentAction{

###### 

###### private SwapInitial swapInitial; // Initial slot on the timetable

###### private AID agentTo; // Agent requesting tutorial (REQUESTER)

###### private TutorialGroup tutorialTo;

###### 

###### @Slot (mandatory = true)

###### public AID getAgentTo() {

###### return agentTo;

###### }

###### public void setAgentTo(AID agentTo) {

###### this.agentTo = agentTo;

###### }

###### 

###### @Slot (mandatory = true)

###### public SwapInitial getInitalSwapRequest() {

###### return swapInitial;

###### }

###### public void setInitalSwapRequest(SwapInitial initalRequest) {

###### this.swapInitial = initalRequest;

###### }

###### 

###### @Slot (mandatory = true)

###### public TutorialGroup getTutorialTo() {

###### return tutorialTo;

###### }

###### public void setTutorialTo(TutorialGroup tutorialTo) {

###### this.tutorialTo = tutorialTo;

###### }

###### }

##### SwapInitial Code

###### package timetable\_ontology.elements;

###### import jade.content.AgentAction;

###### import jade.content.onto.annotations.Slot;

###### import jade.core.AID;

###### // MessageBoard is an ArrayList of SwapInitials

###### // Requester details always listed in SwapFinal

###### public class SwapInitial implements AgentAction {

###### private TutorialGroup tutorialFrom;

###### private AID agentFrom; // Agent receiving request

###### 

###### @Slot (mandatory = true)

###### public AID getAgentFrom() {

###### return agentFrom;

###### }

###### public void setAgentFrom(AID agentFrom) {

###### this.agentFrom = agentFrom;

###### }

###### 

###### @Slot (mandatory = true)

###### public TutorialGroup getTutorial() {

###### return tutorialFrom;

###### }

###### public void setTutorial(TutorialGroup tutorial) {

###### this.tutorialFrom = tutorial;

###### }

###### }

##### Timeslot Code

###### package timetable\_ontology.elements;

###### import jade.content.Concept;

###### import jade.content.onto.annotations.Slot;

###### public class Timeslot implements Concept {

###### 

###### public Timeslot() {

###### }

###### 

###### // Used by main

###### public Timeslot(int day, int time) {

###### this.day = day;

###### this.time = time;

###### }

###### 

###### private int day;

###### private int time;

###### 

###### @Slot (mandatory = true)

###### public int getDay() {

###### return day;

###### }

###### 

###### public void setDay(int day) {

###### this.day = day;

###### }

###### 

###### @Slot (mandatory = true)

###### public int getTime() {

###### return time;

###### }

###### 

###### public void setTime(int time) {

###### this.time = time;

###### }

###### }

##### UnhappySlot Code

###### **package** timetable\_ontology.elements;

###### **import** jade.content.AgentAction;

###### **public** **class** UnhappySlot **implements** AgentAction {

###### 

###### **private** SwapInitial slotToSwap;

###### **public** SwapInitial getSlotToSwap() {

###### **return** slotToSwap;

###### }

###### **public** **void** setSlotToSwap(SwapInitial slotToSwap) {

###### **this**.slotToSwap = slotToSwap;

###### }

###### }

##### TutorialGroup Code

###### package timetable\_ontology.elements;

###### import java.util.ArrayList;

###### import jade.content.Concept;

###### import jade.content.onto.annotations.AggregateSlot;

###### import jade.content.onto.annotations.Slot;

###### import jade.core.AID;

###### public class TutorialGroup implements Concept {

###### public TutorialGroup() {

###### }

###### 

###### // Used by main

###### public TutorialGroup(Timeslot slot, String tutorialID, int tutNum) {

###### this.slot = slot;

###### this.tutorialID = tutorialID;

###### this.setTutNum(tutNum);

###### }

###### 

###### private Timeslot slot;

###### private String tutorialID; // The set code (e.g. SET10101)

###### private int classSize;

###### private int tutNum;

###### @Slot (mandatory = true)

###### public Timeslot getTimeslot() {

###### return slot;

###### }

###### public void setTimeslot(Timeslot slot) {

###### this.slot = slot;

###### }

###### 

###### @Slot (mandatory = true)

###### public String getTutorialID() {

###### return tutorialID;

###### }

###### public void setTutorialID(String tutorialID) {

###### this.tutorialID = tutorialID;

###### }

###### @Slot (mandatory = true)

###### public int getTutNum() {

###### return tutNum;

###### }

###### public void setTutNum(int tutNum) {

###### this.tutNum = tutNum;

###### }

###### }