Overview of MATSim-BDI integration

1 Introduction

This project contains two main modules that are written using different frameworks, MATSim and GORITE. The modules together function to introduce taxi like functionality into a MATSim simulation. This functionality is implemented with each module implementing only part of the overall functionality required and then allowing the modules to communicate through a relatively simple and lightweight interface. The BDI module implements reasoning and decision making for agents in the MATSim simulation (taxis) and also for agents not in the simulation (operators). The MATSim module on the other hand simply provides the functionality for an agent to change its plan in real time as the simulation runs.

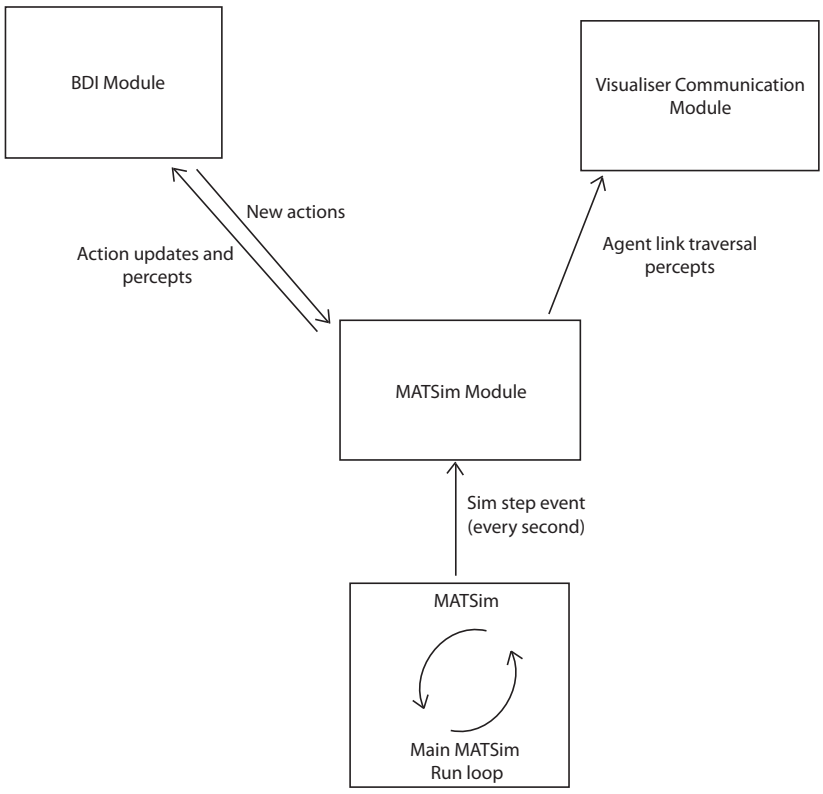

Figure 1: Overview of module organisation and relation to core MATSim simulation

2 The module based architecture

One of the goals of this project was to explore the idea of modules that implement parts of the high level functionality of an agent based system in a way that makes them independent from the overall functionality of the system. In this project only two complicated modules were developed although a third simple module that sends data to a separate visualisation program is also provided. Modules are implemented so that they

conform to the ModuleInterface interface which is defined in the moduleInterface package. This interface is relatively straightforward and uses SimpleMessage (moduleInterface.data.SimpleMessage) objects for communication between modules. SimpleMessage objects do not have any well defined meaning in the abstract, it is left up to the specific modules to use an agreed upon meaning for the messages that they send to each other.

SimpleMessages are basically simple structures that have the following fields:

(int) ID: A unique ID for the message, two messages are considered equal if they have the same ID.

(String) agentID: The ID of the agent that the message concerns.

(String) name: The type of message being sent, the interpretation of this field is left up to the individual modules.

(Object[]) params: Any further parameters that will be needed by the recipient to understand the message.

(Object[]) response: Any data that is generated in response to message.

(emun State) state: The state that the message is in, the meaning of this is again dependant on the modules and the type of message.

The module interface provides three methods for modules to communicate, all of which make use of the SimpleMessage objects to carry the messages and associated data. The first is through the takeControl method which takes as an argument a list of new actions to be performed or updated and a seperate list of percepts to be responded to. This method should be understood as giving control of the execution of the simulation over to the recipient of the call as well as asking that module to perform all the actions provided. These actions may cause the modules state to change and this is the only accepted way send requests to a module that cause a state change.

If an action will not cause a module to change its state then the other two methods of inter-module communication may also be used. The simplest of these is the processSensingMessages method which takes a list of SimpleMessages that can be interpreted by the recipient as queries about its state. These should not cause the module to take any action but simply to look up the requested information and then return it in the list of SimpleMessages returned by this method. The third way in which modules may communicate is by registering for event callbacks from another module. This is done by calling registerEventInterest to notify the recipient that you are interesting in the list of event types provided. The receiving module is then expected to periodically callback to the takeControl method with any new percepts which have occurred.

Each module is required to implement the ModuleInterface and to only communicate with other modules through this interface. They are also expected to only change there state when takeControl has been called. The reason for these constraints is so that the individual modules do not need to know about the larger context in which they are operating. By conforming to these constraints it should be relatively easy to use a module in an application other than the one it was originally written. Some changes may need to be made as the modules currently communicate directly with one another and custom code may need to be added to allow an existing module to communicate with a new module.

3 MATSim module

The MATSim module is contained in the mATSim package and uses the MATSim framework to run a single day simulation based on the configuration file provided. In addition to this it also carried out the additional function of selecting some or all of the standard MATSim agents to be under the control of an external module. At the moment this external module is our BDI module although in theory this could be swapped out with little change to the MATSim module. When these agents are selected the external module is notified of their selection and their preplanned plan in MATSim is wiped so that given no input the agents will do nothing for the entire simulation. The module will also accept driveTo actions for these agents and on receipt of these actions will update the agents plan so that at the earliest available time they will drive from their current location to the one provided in the action. Once an agent arrives at the requested destination an updated action will be sent back to the external module notifying it that the agent has arrived. In addition to this when an agent is close to the destination a percept will also be sent to notify the agent that it is close to the destination.

The module achieves this by receiving a callback for the doSimStep event from MATSim which happens once every second of simulation time, the callback calls the BDIMATSimWithinDayEngine (mATSim.mobsim.BDIMATSimWithinDayEngine). The doSimStep method of this class is the main method in which the functionality of this module is implemented as every second of simulation time it assesses the state of the MATSim simulation and decides if information needs to be sent to the external module. The MATSimModel (mATSim.MATSimModel) class is the class which actually implements the ModuleInterface and so is also important in understanding how this module communicates with others. In addition to these two classes the two classes in the mATSim.controllers package are important both for setup of the module and for receiving MATSim callbacks for events that the module needs to know about. Finally the AgentActivityEventHandler (mATSim.eventHandlers) is used to get callbacks from MATSim when the agents update there position within the MATSim network.

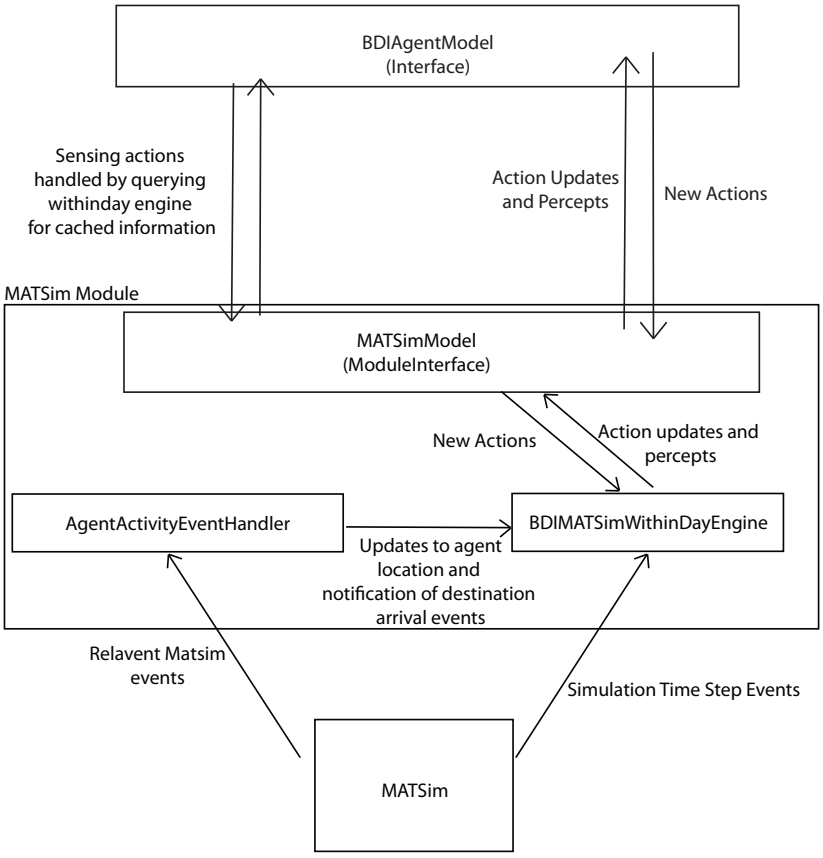

Figure 2: Internal orgainsation of the MATSim module showing core classes

4 BDI module

The BDI module is contained in the bDIPart package and is written with the GORITE framework. The module can be seen as containing two parts, the first part contains the general code which would be useful in another module with a different set of agents that reason differently. The second part is the code that implements the taxi and operator agent functionality and while this code could act as a good starting point in writing new functionality is not very useful outside of the current application. The general code is contained in the bDIPart.general package while the rest of the code is specific to the taxi application.

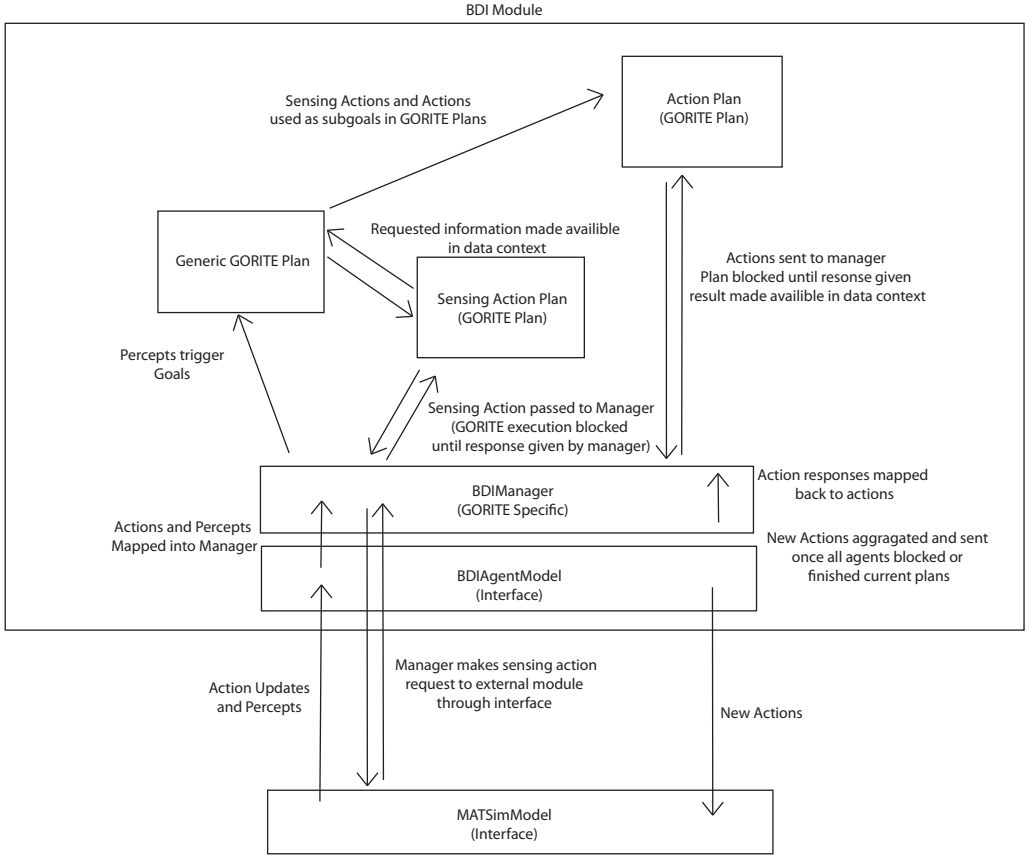

Figure 3: Internal orgainsation of the BDI module showing core classes

In this application the operator agents generate jobs which are basically start and end points in the network. The taxi agents then select jobs that they would like to perform based on distance to the start location, they then post a request for the selected job. The operator agent is then able to review all new requests and either approves or denies the requests. If a taxi agent's request is approved then it will carry out the job before starting the process again, if it is denied then it will select a new job and submit a new request. To achieve this functionality the agents carry out their reasoning in isolation and communicate with the rest of the simulation through the use of ActionGoal which are plans that hook into the messaging machinery for communication with other modules.

ActionGoal and SensingActionGoal are two classes defined in the general package that provide an easy way to send requests either to other BDI agents within the module or to other modules. Both of these classes take responsibility for taking a message from the agent and converting it into a message suitable for transfer between modules. These classes also pause the agents execution and only resume it when a response has been generated to the original message, they will resume for any response and it is left up to the agent to review the given response and decide whether to continue execution or to block again and wait for another response. To facilitate this functionality and to allow all the agents to be run in a separate thread a BDIManager should also be used. The following is a part of a plan that shows how to use either of these plan types, both are used in the same way although they way they communicate with other modules is different.

```
Plan driveToPlan = new Plan("DriveToLocation");

driveToPlan.setSubGoals(new Goal[]{
	new Goal("make a request"){
		public States execute(Instance instance){

			instance.setValue(ActionGoal.ACTION_NAME,"driveTo");
			instance.setValue(ActionGoal.ACTION_ARGS, new Object[]
			    {agent.getName(),agent.getDestinationId()});
			instance.setValue(ActionGoal.ACTION_STORE,"action response");

			return States.PASSED;
		}
	},
	new BDIGoal(ActionGoal.ACTION_GOAL),//SensingActonGoal can be used here instead
	    and will function in almost the same way
	new Goal("check the message"){
		public States execute(Instance instance)
		{
			Action act = (Action)instance.getValue("action response");
			if(act == null){
			    return States.FAILED;
			}
			if(act.getState() == SimpleMessage.State.PASSED)
			{
				return States.PASSED;
			}
			return States.FAILED;
		}
	}
});
```

There are a few things to note about this code, first is that ActionGoals are called like any other BDIGoal in GORITE, you simply create a BDIGoal with the name ActionGoal.ACTION_GOAL. This is will execute the ActionPlan which handles all the functionality and is included by the BDIAgent in it's list of plans so if this is subclassed then everything will work strait away. To allow the GORITE execution to be blocked correctly the arguments that the plan takes need to be stored in the data context this is done by setting a few variables that the plan then uses. By setting ActionGoal.ACTION_STORE the action that was returned from the external module can then by retrieved from the data context and evaluated.

The BDIManager is also defined in the general package and provides functionality that the ActionGoal uses to allow message passing to occur. In addition to this the BDIManager also takes responsibility for executing the agent plans and providing information about whether there are active agents or if they have all finished. To allow the manager to execute the agents plans the performGoal(BDIAgent,Goal) method must be invoked for each agent with a root goal. This registers the agent for execution on the manager's internal thread and also allows it to process the messages that need to be send to other modules by the agent. These classes should be useful for writing new BDI modules that use different types of agents that still need to provide communication between the agents and other modules.