Elevator Controller in Jason

Thoai Nguyen <me@thoaionline.com>

Table of Contents

1. Project Description

Project Purpose

This project aims to develop a basic Agent framework in Jason for controlling elevator systems. The simulations and their benchmark results are handled by an open source elevator simulator <http://elevatorsim.sourceforge.net/>. The project focuses on 3 goals:

1. Building an agent framework in Jason that control an elevator system (i.e. controlling cars) by processing events (percept) then react accordingly.
2. Develop basic algorithms in this agent framework to demonstrate its operation.
3. Extend the elevator simulator so that it can communicate with our agent framework for decision making during simulation scenarios.

Main Features

1. RMI Interface: to connect and handle communication between the system's two components in an object-oriented manner.
2. Jason Environment: acts a "bridge" between the Jason agents and the elevator simulator by forwarding events and decisions between the two.
3. JController: the extension to the original elevator simulator that communicate with the Jason part.
4. Coordinator agent: one agent responsible for creating car driver agents, which then in turn handle movement of the cars.
5. Simple car driver: a dump car driver that implements algorithm of SimpleController.
6. Meta car driver: a car driver that implements algorithm of MetaController.

2. Background

- **IDE**: Eclipse 3.6.0 for Java
- **Eclipse plugins**: Jason for Eclipse
- **Repository**: Subversion
- **Language**: Java 6, Jason <http://jason.sourceforge.net/>

3. System Installation

Component overview

The system consists of two component:

1. **ElevatorClient**: An extended version of the Elevator Simulator (client)
2. **JElevator**: A Jason Agent framework for controlling the elevator (server)

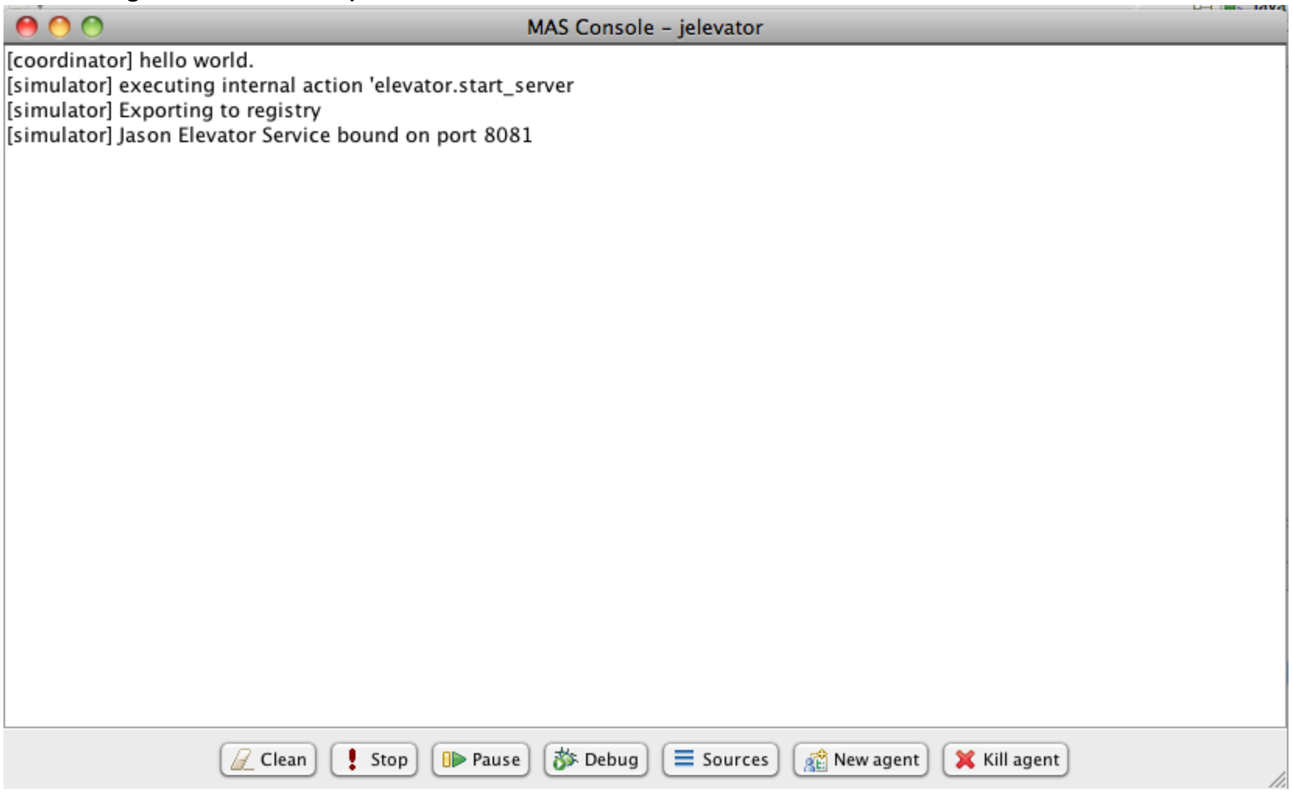
JElevator - The Jason Controller

The server component needs to be started prior to launching the client. It can be executed by either:

1. "Run" the project JElevator from within Eclipse IDE (recommended for development) **or**
2. Execute the project from the command line with the following command from within the JElevator directory
   - java -classpath **/Applications/Jason-1.3.3/./lib/ant-launcher.jar** org.apache.tools.ant.launch.Launcher -e -f bin/build.xml run

- You will need to replace the highlighted text with path to your corresponding path to ant-launcher.jar, which is part of the standard Jason distribution. The example is the default path on an OS X installation.

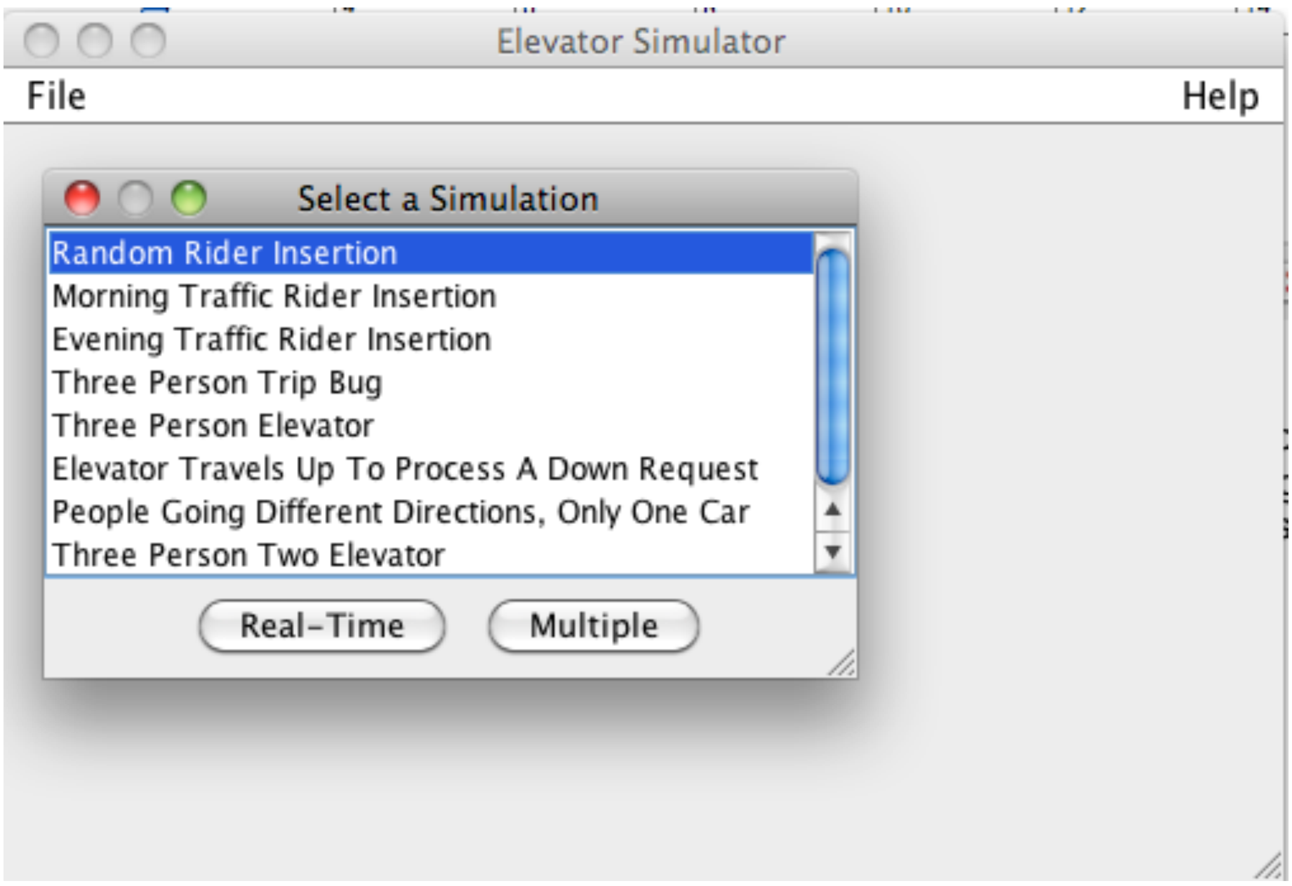
Upon successful launch, the Jason console will show up and report that it is listening on port 8081 (default, configurable in JElevator.mas2j). At this point, you can continue with launching the client component.



```
[coordinator] hello world.
[simulator] executing internal action 'elevator.start_server
[simulator] Exporting to registry
[simulator] Jason Elevator Service bound on port 8081
```

ElevatorClient - The simulator

The ElevatorClient is basically an extension to the ElevatorSimulator, that can connect and forward events to JElevator, then receive instructions from Jason agents in JElevator and send them back to the simulator. ElevatorClient can be launched as a regular Java program, either from inside eclipse or directly from the console.


Elevator Simulator
File
Help
Select a Simulation
Random Rider Insertion
Morning Traffic Rider Insertion
Evening Traffic Rider Insertion
Three Person Trip Bug
Three Person Elevator
Elevator Travels Up To Process A Down Request
People Going Different Directions, Only One Car
Three Person Two Elevator
Real-Time
Multiple

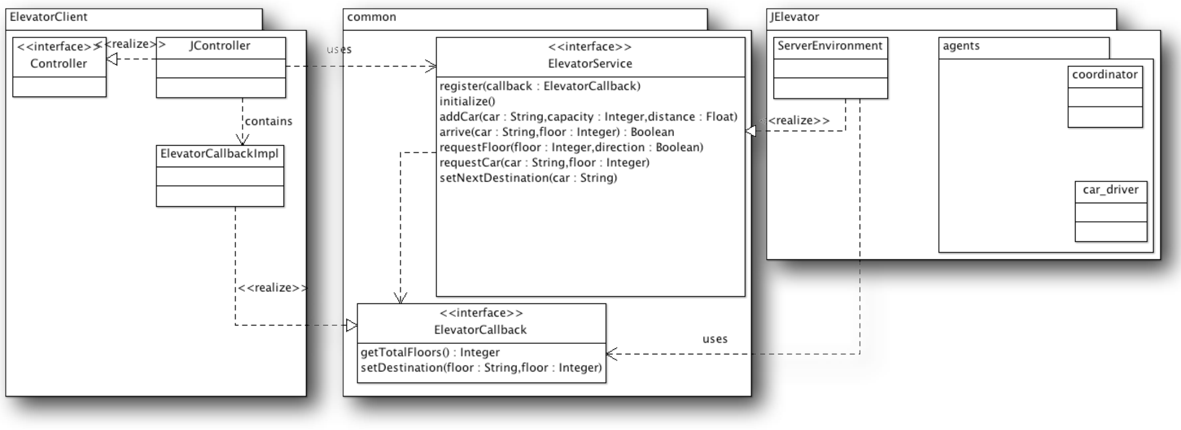
To make use of the extension, select JController as the Controller for the simulation that you want to run. There two built-in controllers (i.e. MetaController and SimpleController) are pure Java controllers that ship in the original simulator whereas JController is the new one with server communication capability.

Please note that not all simulation scenarios support selection of controller. Currently, only the “Random Rider Insertion” is configured to make use of the new Controller.

4. Project Design and Implementation

Class Diagram - RMI Interfaces

(See Elevator_Design.zargo)


ElevatorClient
<<interface>>
Controller
<<realize>>
JController
contains
ElevatorCallbackImpl
<<realize>>
common
uses
<<interface>>
ElevatorService
register(callback : ElevatorCallback)
initialize()
addCar(car : String,capacity : Integer,distance : Float)
arrive(car : String,floor : Integer) : Boolean
requestFloor(floor : Integer,direction : Boolean)
requestCar(car : String,floor : Integer)
setNextDestination(car : String)
<<interface>>
ElevatorCallback
getTotalFloors() : Integer
setDestination(floor : String,floor : Integer)
uses
JElevator
ServerEnvironment
<<realize>>
agents
coordinator
car_driver

ElevatorClient - The simulator

This section explains the basic operation of the elevator simulator that is necessary for understanding the project design, for a deeper understanding of the simulator, please refer to its source code and website.

Basic concepts:

- Elevator: a system consisting of cars, that is responsible for transporting passengers between floors in a building.
- Car: a physical component of the elevator, that can handle a fixed number of passengers, capable of receiving requests to go to specific floors by passengers who already got inside, and moving between floors to transport passenger.
- Building: a structure of floors in continuos order.
- Floor: stopping point for cars, where passengers standby waiting for car(s) . Each floor is identified by a unique positive integer that reflects its position in the building.

Simulation overview

A simulation scenario consists of:

1. Declaration of number of floors, passengers, available cars and their corresponding capacities.
2. A set of events, each of which will occur at a specific time during the simulation.

There are two types of events/requests that can raise during a simulation:

1. Whenever a passenger want to go up/down, provided that they are currently at floor X
   - Parameters: floor, direction
2. Whenever a passenger select their destination floor when they’re inside a car
   - Parameters: car, destination floor

A simulation happens as follow:

```
while (there are pending events OR cars are still moving) {
        send current events to the controller;
        if a car C arrive at floor X
        {
                inform the controller and ask for the car's upcoming direction
                move passengers into the car
                ask the controller to set the car's next destination
        }
}
```

A car is considered "stopped" or not moving when its destination is set to **null**.

All controller will implement the *org.intranet.elevator.model.operate.controller.Controller* interface, and will be assigned to the **building** at the start of the simulation. (See *RandomElevatorSimulator* for an example.)

The JController class

Currently, all the simulation scenarios and controllers are hard coded into the simulator. It is therefore necessary to develop a mechanism to de-couple the simulator and the actual controller logic, so that we can change the algorithm in the Jason agents without further modifying the simulator code. To achieve this, a new controller (i.e. JController) is developed, acting as a bridge between the simulator and the Jason framework. This controller is integrated into the Random Rider Insertion scenario as an option.

The JController uses RMI to communicate with the Jason system. When initialized, it creates a callback stub, which is essentially an alias to a ElevatorCallback object that can communicate natively with the simulator. Afterward, it initializes a connection to the localhost's RMI registry at port 8081 (this registry is created by the Jason framework in advance), and locate an ElevatorService object called "elevator". The callback stub is then registered with the remote "elevator" object so that the Jason framework know where to send response (i.e. instructions) to. From this point forward, all events from the simulation are forwarded to the Jason framework through this "elevator" object. Whenever a decision is made (in the Jason part), its is sent back to the ElevatorCallback object that lies inside JController, then forwarded to the simulator.

On one hand, since the JController deals with native objects of the simulation (e.g. Car, Floor, Building, etc.) whereas the Jason framework only requires basic information to function (Car name, floor number, etc.), all requests to the ElevatorService are simplified to contain only essential information that the Jason controller needs. On the other hand, it is necessary to map the info sent by the Jason system back in to the simulator's original, native Java objects. To achieve this, the JController keeps track of a list of cars added at the beginning of the simulation and use this list to look up for the original car based on its name returned by the Jason controller via ElevatorCallback object.

JElevator - the Jason Elevator Controller Framework

Overview

The JElevator framework consists of two major components:

1. The environment:
    - Receive events and generate corresponding percepts
    - Perform actions as specified by the agents
2. The agents:
    - React to percepts provided by the environment.

The Environment

There are two Environment-s defined in this release:
- ElevatorEnvironment (predicated): an environment that communicates directly with the built-in simulator (used for testing purpose only).
- ServerEnvironment (default, recommended): an environment that communicates with a separate simulator via RMI.

Only one environment is active at a time and . The ElevatorEnvironment was initially developed to interact directly with a simulator built into the Jason framework, for testing purpose. This environment requires the built-in simulator to be launched by executing the elevator.run internal action from within the "simulator" agent. This environment configuration is no longer supported.

The ServerEnvironment is the current default environment implementation that interacts with both the external simulator (i.e. ElevatorClient) and the agents. By definition, this environment is capable of generating percepts and receive actions from the agents. It is already equipped with percept and action handling capabilities by extending Jason's Environment class.

In order for the ServerEnvironment object to communicate with the ElevatorClient, the environment instance is exported to the RMI registry through the *elevator.start_server* internal action (executed by the simulator agent). Upon successful registration of the environment object, the JController can interact remotely with it as if the environment object is on the same system as the JController.

Agent Design

The agent framework consists of the following agent types:
1. Simulator agent: responsible for either launching the built-in simulator or publishing the environment object to the RMI registry.
2. Coordinator: responsible for creating car_driver agents and coordinate their work.
3. Car driver: each agent of this type is responsible for controlling of a specific car.

Since this release focuses on creating a working agent framework for controlling elevator rather than implementing complicated algorithms and inter-agent interactions, the majority of the car controlling logic lies in car_driver agents and the coordinator agent only creates these car drivers when needed.

Events and percepts

When an event is perceived by the environment, a percept is forwarded to the agent system. For the ServerEnvironment, the events (see ElevatorService interface) and their corresponding percepts are as follows:

Event	Percept	Receiving agent
Initialize	-started	all agents
Add car	-+Car(CarName)	all agent
Arrive	-+at(Floor)	corresponding car driver
Request car	+request(Floor,Direction)	all agents
Request floor	+request(Floor)	corresponding car driver
Set next destination	n/a	n/a

Actions

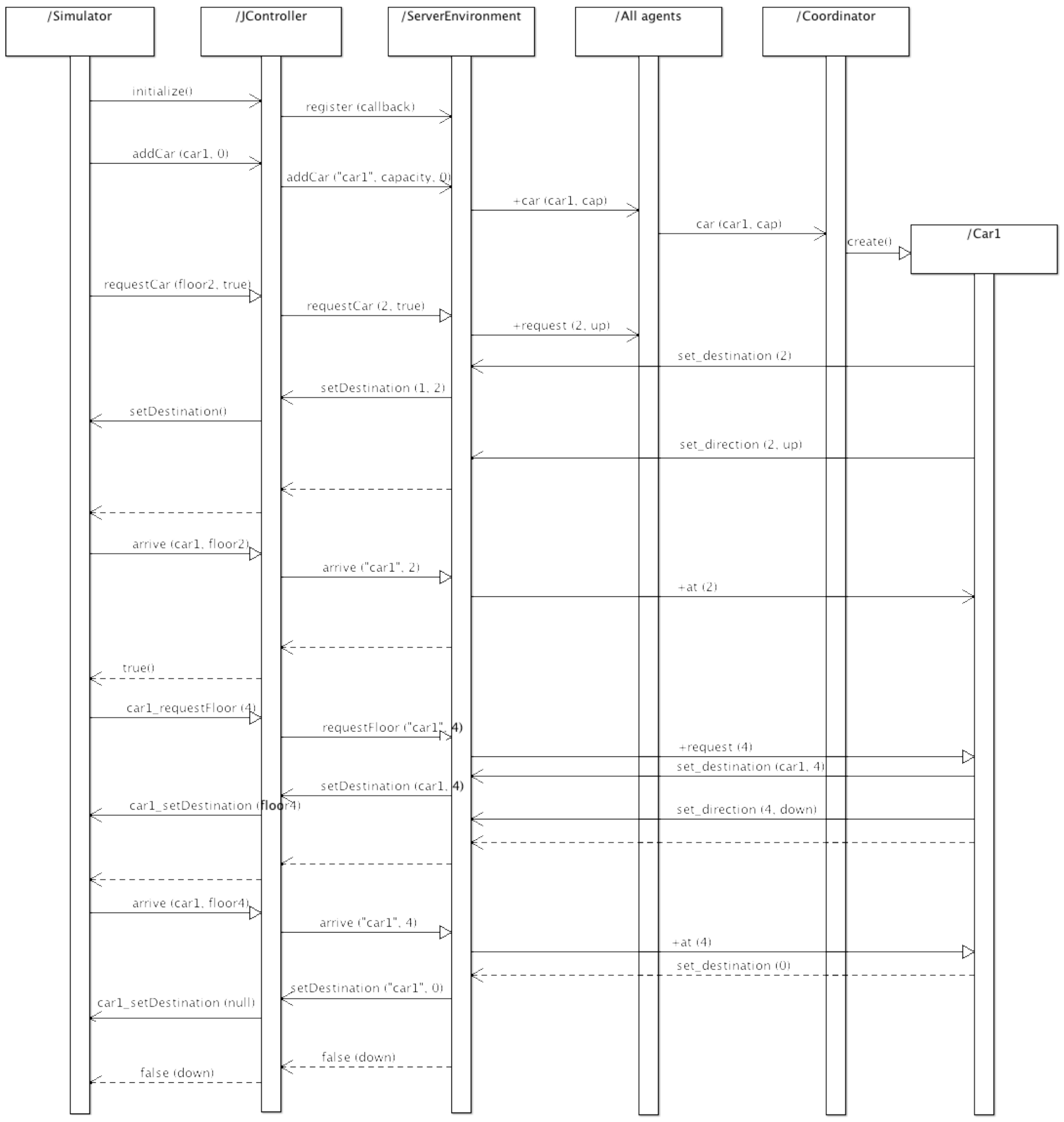
The car drivers are capable of reacting to the environment's percepts with the following actions.

Action	Description
set_destination	Set the car's next destination (floor)
set_direction	Inform the environment whether the car will go up or down when it arrives at a specific floor.

Agent implementation

Please refer to the agent in *car_driver.asl* and *simple_driver.asl* for specific implementations of car controlling algorithms.

Example: Interaction between components in a simple simulation where there is only one passenger going from floor 2 to floor 4.



5. Discussion

Issues and difficulties

Synchronization mechanism

Because the simulator is synchronous while the Jason agent framework operate asynchronously, the destination and directions for each car is stored in the environment object so that it can response to the simulator as soon as these information is requested (e.g. when arrive method is called, requiring the direction to be returned immediately). The environment objects also waits for a timeout (500ms) to ensure that no further actions are performed before it send a response to the client code.

This timeout mechanism is implement in the static class StepHandler, which basically contains a lock with a condition triggered whenever an action is executed by any agent. The step method of this class will wait until an timeout occurs (i.e. no action performed after a specific amount of time), ensuring that all agents finished their reasoning cycles and delivered necessary actions.
This approach has a drawback, that is it’s hard to determine the minimal necessary timeout for the agents to deliver all actions whilst a large timeout would slow down the program significantly.

Multiple mode

The original simulator has a “multiple” mode that generate benchmark data based on different run with various parameters such as number of passenger, number of cars and their capacities, etc. This mode is not yet supported in this release of the system because the framework is not informed when an simulation ends.

We tried to detect the end of simulation and do necessary clean up. However, since the new simulation will occurs immediately after the current simulation, there are still pending actions and belief sets that has not been cleaned up properly, leaving the agent system in an illegal state. This is not the case in the real-time mode when user stop and start new simulation manually, leaving plenty of time for the agent framework to reset.

Future work (suggestion)

1. Support for multiple mode
    1. Implement a mechanism to synchronize actions on the agent framework and the JController
2. Smarter coordinator agent and new algorithms