University of Macedonia

Master of Science

Department of Applied Informatics

Jason Logo

A configurable graphical environment for Multi-Agent-Systems (MAS) simulation based on JaCaMo framework.

Master thesis

Of

Panagiotis Despotis

Thessaloniki, 06/2018

JASON LOGO: A CONFIGURABLE GRAPHICAL ENVIRONMENT FOR MULTI-AGENT-SYSTEMS (MAS) SIMULATION BASED ON JACAMO FRAMEWORK.

Panagiotis Despotis

Software Engineer Bachelor's degree, TEI of Central Macedonia, 2013

Master Thesis

Submitted for the partial fulfillment of its requirements

POSTGRADUATE TITLE OF EDUCATION IN APPLIED INFORMATICS

Επιβλέπων Καθηγητής
Σακελλαρίου Ηλίας

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26/06/2018

ΣΑΚΕΛΛΑΡΙΟΥ ΗΛΙΑΣ	ΡΕΦΑΝΙΔΗΣ ΙΩΑΝΝΗΣ	ΧΑΤΖΗΓΕΩΡΓΙΟΥ ΑΛΕΞΑΝΔΡΟΣ
.....	.....	.....

ΔΕΣΠΟΤΗΣ ΠΑΝΑΓΙΩΤΗΣ

.....

Περίληψη

Η προσομοίωση είναι η μίμηση μιας διεργασίας του πραγματικού κόσμου ή ενός συστήματος. Οι προσομοιώσεις πολυπρακτορικών συστημάτων (*Multi-Agent systems* - MAS) στοχεύουν στη δοκιμή της συμπεριφοράς και των αποτελεσμάτων των αλληλεπιδράσεων μεταξύ αυτόνομων προγραμμάτων (πρακτόρων) κάτω από ένα κοινό περιβάλλον ως σύστημα. Μια γραφική απεικόνιση του περιβάλλοντος μπορεί να βοηθήσει στην καλύτερη κατανόηση των αποτελεσμάτων μιας προσομοίωσης. Σκοπός της παρούσας εργασίας ήταν η δημιουργία ενός γραφικού διαμορφώσιμου περιβάλλοντος προσομοίωσης για πολυπρακτορικά συστήματα με τη χρήση της γλώσσας Jason (AgentSpeakL) ως κύρια γλώσσα προγραμματισμού των πρακτόρων του συστηματος. Για να το επιτύχουμε, συνδυάσαμε δύο τεχνολογίες. Το JaCaMo και την NetLogo. Το JaCaMo είναι ένα framework που μας παρέχει τη γλώσσα Jason για τον προγραμματισμό των πρακτόρων, το Cartago ως το πλαίσιο για τη δημιουργία αντικειμένων (εργαλείων) που θα χρησιμοποιηθούν από τους πράκτορες της Jason και το Moise, το οποίο χρησιμοποιείται για την οργάνωση, το σχεδιασμό και τη μοντελοποίηση των πρακτόρων. Από την NetLogo δανειστήκαμε την ιδέα ενός περιβάλλοντος προσομοίωσης 2D αποτελούμενου από κελιά "μπαλώματα" στα οποία πάνω από αυτά τα κελία λειτουργούν οι πράκτορες "χελώνες". Επίσης κάποια χαρακτηριστικά από το UI της NetLogo.

Λέξεις κλειδιά: Προσομοίωση, πολυπρακτορικά συστήματα, γραφικό περιβάλλον, απεικόνιση, JaCaMo, Cartago, Jason, NetLogo

Abstract

Simulation is the imitation of the operation of a real-world process or system. *Multi-Agent systems* (MAS) simulations are aiming on testing the behavior and the results of the interactions between autonomous programs (agents) under a common environment as a system. A graphical representation of the environment can help for better understanding the results of a simulation. The aim of the present work was to create a graphical configurable simulation environment for *Multi-Agent-System (MAS)* using the Jason (AgentSpeakL) language as the primary language for programming the agents of the MAS. In order to achieve this we combined two technologies. The JaCaMo framework and the NetLogo. JaCaMo provided us the Jason language for programming the agents, Cartago as the framework for creating artifacts (tools) to be used by the agents in Jason and Moise which it is used for organize, planning and modeling the agents. From the NetLogo we borrowed the idea of a 2D simulation environment consisting of "patches" which on top of them are operating agents "turtles", also some futures from the UI of the NetLogo.

Keywords: Simulation, Multi-Agent systems (MAS), JaCaMo, Graphical environment, Java, Swing

Contents

1 Introduction and Motivation

Simulation of *Multi-agent systems* (MAS) is an important tool in research and development, as it allows on the one hand the modeling of complex systems and, on the other, the extensive testing of the design of a MAS. In addition, it allows the familiarization among the programming languages of a MAS. Furthermore, the graphical representation of a simulation which is our main focus is essential especially when it is used for education purposes for better understanding. NetLogo [1] it is a tool which provides the environment and capabilities for creating a MAS graphical simulation but it lacks in the use of a proper agent-based programming language. On the other hand, there are powerful agent programming languages, such as Jason [2] which lack any simulation visualization facilities. Unfortunately, up to date, there is not a complete set which provides the proper tools for creating a configurable graphical simulation with the use of a proper agent-based language.

NetLogo [1] is a well-suited platform which provides a multi-agent programming language and modeling environment for simulating complex 'emergent phenomena'. Developers (modelers) can program a vast number of independent autonomous entities (agents) which all can operate concurrently, in order to monitor the behaviors between these entities (agents) in a micro-level scope and also to unravel the patterns that are emerged from their interactions over time. It was designed by Uri Wilensky and it is a powerful tool which is used for research and education purposes. In NetLogo [1] agents are called "turtles" and they operate over a grid of "patches". Turtles and patches are programmable agents which all can interact with each other and perform multiple tasks concurrently. Modelers are able to shape the turtles and patches by changing the view, color and size for creating a visualized environment and study mathematical abstractions or play games. NetLogo [1] also includes a third agent type, the "observer". There is only one observer. Observer is able to issue instructions to the turtles and patches.

In view of that, having as reference the NetLogo [1], we created JasonLogo (JLogo) a project which it is developed under JaCaMo [3] platform and Jacamoide [18] (JaCaMo eclipse plugin), aiming to provide the proper set of tools for creating a configurable graphical simulation environment with minimum use of Java code. Adopting the main

concept of NetLogo [1] having “turtles” and “patches” as the main agent types, which in our implementation “turtles” are motion agents and “patches” the stationary one alongside with the “observer” agent which will be the instantiator and the coordinator agent in a simulation. Using Jason [2] as the programming language, which is the implementation of AgentSpeak, the primary language for programming logic-based agents based on BDI [6] architecture, Cartago [4] which is the infrastructure for creating artifacts (computational entities) which can be used by the agents as tools and Moise [7] which it is used for organize, planning and modeling the agents. The three technologies are the JaCaMo [3] platform, and together they aspire to be a practical solution to programming a MAS. Using JaCaMo [3] platform for creating a framework which will provide similar tools like NetLogo [1] will allow the development and study of MAS simulations and also it will contribute significantly to the training of developers who first come into contact with the field of MAS.

The rest of this thesis is organized as follows: Chapter 2 presents the related work, whereas chapter 3 presents the Analysis and design, follows the chapter 4 with the implementation and chapter 5 with example models in JLogo. Finally, chapter 6 presents the conclusions of this thesis.

2 Related Work

In this section we are going to brief present some of the basic ideas and tools which are used for programming and modeling Multi-Agent Systems (MAS) simulations.

2.1 Agents

The definition of an agent is a person *who acts* on behalf of another person or group. An agent can be a person, a machine, a piece of software or a variety of other entities. In abstract an agent is generally regarded to be an autonomous entity that can interact with its environment. In other words, it must be able to perceive its environment through sensors and act upon its environment with effectors.

A software agent is a computer program that *acts* for (behalf) a user or other program in a relationship of agency (person or group). A software agent can interact with other agents or humans like chat-bots. A major category of software agents are the intelligent agents (IA) which have capabilities of process natural language, understanding and speech. In our days a famous example of the use of intelligent agents are the personal assistants (see Siri [14], Google assistant [15], Cortana [16], Alexa [17] etc.) which are development by all the major software companies in the world.

To design a software agent various model types have been introduced. Two major model types are the Agent-based model (ABM) [5] for modeling reactive agents and Belief-desire-intention model (BDI) [6] for modeling conductive agents.

2.1.1 Agent-based model (ABM)

Agent-based model (ABM) [5] is the technique for modeling the logic of an agent. Modelling the logic of the decision make of an agent usually is very simple, it is depending on the current state of the environment, which means that the agents are "reacting" with the environmental changes. ABMs are used for simulate various scenarios of real world for studying and testing their results. In Multi-Agent system simulations there are several autonomous agents operating at the same time concurrently, studying the interactions and the patterns which emerged over time. In a MAS, agents are based on ABMs, they may have the same logic e.g. they have the same

ABM or a different one depending on the agents' role. As successful ABMs are for modeling simulations about a real world process they are lack in social simulations which there is the need for simulating a human like behavior.

2.1.2 Belief-desire-intention model (BDI)

The belief-desire-intention (BDI) [6] model it is used for programming intelligent agents. These cognitive agents are based on a simplified human being behavior. Like humans which are based on their knowledge (which may be false) about the environment to make decisions relative to select what is the best plan to follow for achieving their desires, these agents also have some information of the world as beliefs, depending on these beliefs they try to select the best plans (intentions) to execute to achieved their goals (desires).

Characteristics of a BDI model are the following:

- **Beliefs:** Is the information which the agent knows about the world and itself. A belief is not necessary to be true. Also, the belief base can be constantly updated.
- **Desires**: Is the goals which an agent wants to achieve.
- **Intentions**: Is the plans which agents needs to complete in order to achieve its goals.

2.2 Simulation

The definition of a simulation is the imitation of a process of the real world or a system. Simulations are used for better understanding how a process evolves over time. To be able to simulate something a model is required, in which all the characteristics, operations and behaviors of the system are defined. The evolution of the model over time is described as the simulation of the system.

2.2.1 BDI - ABM simulations

Both BDI and ABM models have their advantages and their disadvantages in simulations, for instance BDI models are very effective on decision making for selecting the right plan to execute depending on their beliefs about the environment and ABMs are very effective on executing it. There are several successful paradigms in BDI - ABM integration. In the work of Luna Ramirez and Maria Fasli [9] we can see the same idea

with our own but with a different approach. They integrate the NetLogo in Jason by using the capability of Jason having a custom environment. They create the environment and keep the simple reactive agents in the level of NetLogo, and the BDI agents in both Jason and NetLogo. A translator is implemented in java converting the Jason literals to NetLogo commands and vice-versa. This approach needs to have a good knowledge of Jason, NetLogo and Java as the simulation runs in the two platforms. Furthermore, a more abstract approach is the work of the work of Singh, D., Padgham, L. and Logan, B. [8], they have created a framework which intends to combine different types of BDI – ABMs platforms. Also, this approach needs to have a good knowledge of the platforms which are combined and Java.

We think that the above approaches are especially complex to be implemented by the students who first come contact with the field of MAS.

2.3 Jason

Jason [2] is the implementation of the AgentSpeak, an abstract computer programming language for developing Multi-agent systems based on the BDI [6] architecture. Jason [2] is used for programming cognitive agents in a multi-agent system. It is developed with java, which that means it can run anywhere that java can. The source files of the agents in Jason are written in the .asl files. In Jason you can declare beliefs, plans and initial goals of an agent in the source file.

An example of the model of an agent in Jason

```
//Belief base
sky(blue).
sun(bright).
//Initial goals
!how_is_the_day
//Plans
+!how_is_the_day: sky(S) & sun (B)
<-
	.print(“The sun is ”,B,“ and the sky is”,S).
```

In the above example the agents have some beliefs about weather condition. The desire of the agent is to print a message about the weather condition. To achieve this calls `+!how_is_the_day` plan.

2.4 CArtAgO

Common Artifact infrastructure for Agents Open environments (CArtAgO) [4] is a framework which provides the infrastructure for creating tools and resources to be used by the agents to achieve their tasks and goals. CArtAgO it is based on Agents & Artifacts meta-model for modeling *Multi-Agent systems*. Artifacts are small programmable pieces of the system. The artifacts can be shared between the agents of the MAS.

In the following we are going to brief explain some of the basic functionalities of a CArtAgO artifact integrated Jason.

Example of a CustomArtifact class.

```
public class CustomArtifact extends Artifact {

	private JasonLogoModel model;
	private String title;

	public void init() {
		this.title = "Default title";
		defineObsProperty("count",0);
	}

	public void init(String title,int count) {
		this.title = title;
		defineObsProperty("count",count);
	}

	@OPERATION
	public void count(int i) {
		ObsProperty belief = getObsProperty("count");
		belief.updateValue(0,belief.intValue(0)+i);
	}
}
```

Artifacts are java classes developed by the modelers to provide the necessary resources or operations to the agents. The basic class which an artifact is extending is the `cartago.Artifact`, there are also other types of Artifacts but in the moment will we stick with the basic one. The initialized methods of an artifacts are the init methods, like

the constructors in the java classes modelers can have as many init methods with different parameters as they need.

Class methods with void return parameter and annotated by @OPERATION are operations which can be called by the agents. Methods parameter corresponds to operations parameters.

Observable property is a tuple set with a functor and multiple parameters. In the above example the observable's property name is "count" and value is a single integer argument. To define an observable property, it is achieved by the defineObsProp method of an artifact. Furthermore, artifact's observable properties are automatically added in agents' belief base when the have focus the artifact. To retrieve an observable property, it is achieved by the getObsProperty("count").

Creation of artifact it is achieve by the use of the makeArtifact operation of the workspace artifact (All agents by default are join the default workspace). The parameters of the makeArtifact operation are

- 1st the name of the artifact
- 2nd the class of the artifact
- 3rd a list of objects which are relative with the init methods
- 4th a feedback parameter which returns the artifact object

```
makeArtifact(c1,"example.artifact.CustomArtifact",["Title",1],W);
focus(W);
makeArtifact(c2,"example.artifact. CustomArtifact ",[],E);
focus(E);
```

An agent can focus an artifact by using the focus operation of the workspace artifact. By focusing an artifact, the agent gets as beliefs all the observable properties of the artifact and can use its operations.

An operation can be called by the method name of the artifact.

E.g. count(**1**)

2.5 JaCaMo approach

JaCaMo [3] is a framework which it is used for Multi-Agent Programming. Combines Jason as the programming language, CArtAgO [4] for programming environment artifacts and Moise [7] for programming Multi-Agent organisations. All these threes technologies compose the JaCaMo.

JaCaMo it based on a specific programming model named JaCa. In JaCa a MAS is designed and programmed as a set of agents which work and cooperate inside a common environment.

JaCaMo has a configuration file which in it, it is defined the initial state of the MAS. Agents, workspaces, artifacts and organisations can pre-defined in the jcm file.

An example of jcm file.

```
mas test {
        agent bob : bobSource.asl {
                focus:wsp.customArtifact
        }
        workspace wsp {
                artifact customArtifact: example.CustomArtifact(“Test”,1)
        }
}
```

In the configuration file a workspace with name “wsp” is created, in the workspace an artifact with name “customArtifact” is created. Also, an agent named bob is created with Jason [2] source file the “bobSource.asl” which has join the workspace “wsp” and focused the artifact “customeArtifact”.

example.CustomArtifact(“Test”,1) The parameters are relative to which init method will called on the instantiation of the artifact.

3 Analysis and Design

JLogo attempts to combine the ease use of NetLogo [1], with the programming power of Jason [2] platform. In order to achieve this, we followed closely the successful NetLogo [1] paradigm in simulation. Thus, it was necessary to implement a grid world, where each cell is a patch location and in these patches the agents can move and operate. Furthermore, JLogo provides a set of sensors and operations which can be used by the agents to "sense" and "alter" the simulation environment. Also provides a graphical environment which except of the simulation view displays information about the simulation configuration, provides graphical components for controlling the simulation speed and when the simulation starts. The graphical environment can be altered by the developers by adding extra graphical components such as buttons, fields etc. and bind them with an internal agent action. Furthermore, the user is able to interact with the simulation view by click in the cells of the grid.

3.1 JLogo Environment

JLogo environment is a grid-based environment consisting of tiles, called patches. A patch is a grid location. In the graphical simulation environment two types of agent are operating, the motion and the stationary. The motion agents can move and navigate over the grid of "patches" in the world. The stationary agents operate on a "patch" location, it cannot move, yet it can interact with other motion and stationary agents. Also, there is a third type of agent the "observer" which is responsible for the instantiation of world and coordinate the other agents.

The size of both the simulation environment and the patches are configurable on the instantiation of the environment by the observer agent. In the simulation view the starting point of any object (patch or agent) is the top left corner including the simulation view, also the orientation is upside down. The view of the motion agent is render in the center of the agent position for better representation on display [Figure 1].

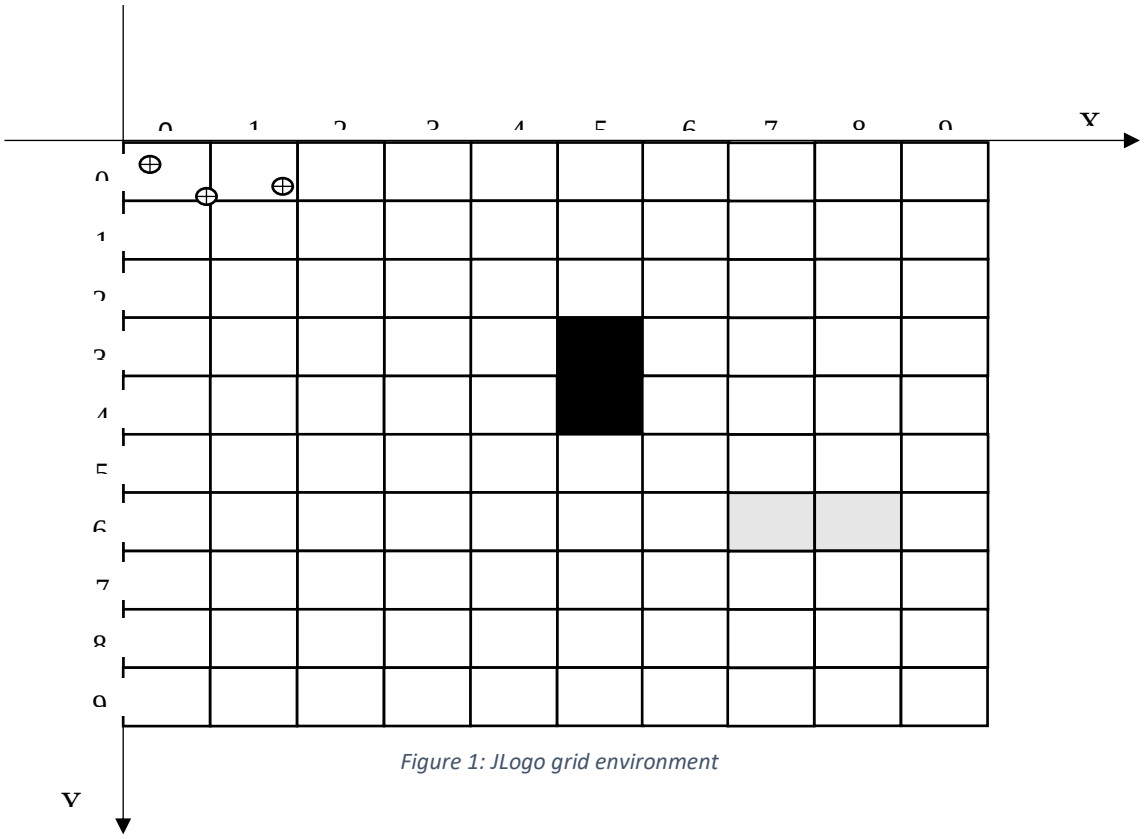

Figure 1: JLogo grid environment

3.1.1 JLogo patch

A "patch" in JLogo environment is a position of the grid model. Patch has two main properties.

- The location in the grid. Is Cartesian coordinate location which every point cannot be less than zero and greater than gridSize [0, gridSize).
- The color of the patch. The default color of a patch is black.

One and only stationary agent should operate in a patch location. On the other hand, there are no restrictions on how many motion agents can be in the patch location. A patch location can be manipulated or "sensed" by other agents through Cartago artifacts and Jason internal actions, as it is going to be discussed later. The main artifact which is responsible for manipulate a patch location is the EnvironmentArtifact.

3.1.2 JLogo agent

In JLogo there are three types of agents. The motion, the stationary and the observer agent.

Motion agent (turtle):

The motion agent ("turtle"), can move and navigate in the simulation environment, interact with other "turtles" and "patches" and share beliefs.

The main properties of a motion agent are the following:

- Id: The id of the agent. Is unique for every agent and it is the same the same with the id that the agent has in the Jason level.
- Location: The absolute pixel location of the agent in the environment.
- Patch Location: The patch location of the agent
- Breed: The type of the agent
- Direction: Direction of the agent in the 2D model in degrees.
- Speed: How many pixels agent will be moved with one step (Default:1)

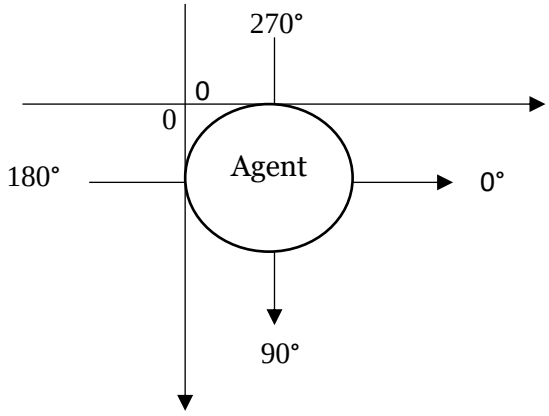

Figure 2: Agent model

Also an agent has display properties as:

- Render size: The render size of the agent
- Render image: An image which is rendered instead of the default shape of the agent
- Render color: The color of the agent
- Visibility: Defines if an agent is it rendered on the simulation view

The position of the agent in the world model is the top left corner and the direction is upside down following Java's swing orientation [Figure 2].

The default render view of a motion agent is the oval shape, but this can change through the EnvironmentArtifact and Interaction artifact assigning an image to the specific agent.

Stationary agent (patch):

The stationary agent is a “patch” agent, which operates over a grid location. Only one “patch” agent can operate at the same time over a grid location. A patch agent cannot move in the environment, however can interact with other motion and stationary agents. Also, can share beliefs, spawn new agents and manipulate the environmental state. A patch agent extends the properties of a patch location by adding an id.

Properties of a “patch” agent.

- Id: The id of the agent is unique and it is the same with the id of agent in the Jason level.

Observer agent:

Observer is the instantiator of the JLogo simulation environment. A single instance of this type of agent should exist in a JLogo simulation. The declaration of the observer it is pre-defined in MAS configuration file (jmc). In most cases it will be responsible to create the initial world environment and spawn the players (agents) of the simulation. Observer does not have a physical location in the world, however can edit the environment, spawn or kill agents and patches, share beliefs and manipulate other agents. Also can play the role of the coordinator between the agents when this is necessary.

3.2 JLogo Tools

To provide agents the capabilities to navigate, manipulate and sense the environment we create the proper tools through Cartago Artifacts and Jason Internal Actions.

3.2.1 JLogo Artifacts

Four basic artifacts have been introduced which can be focused by the agents to achieve their goals depending on their plans and type.

WorldArtifact: It is responsible for creating the model and GUI of JLogo. Also, can “listen” the events of the UI such as button clicks, speed sliders changes etc. A single instance of this artifact should be created in a JLogo simulation. The instantiation and operation of this artifact should be done only by the observer agent.

EnviromentArtifact: Provides the operations for add and remove agents and patches from the environment. It can be focused by any type of agent. Many instances of this artifact can be created depending the concurrent use by the agents.

InteractionArtifact: Provides the operations for the motion agent to be able to navigate in the simulation environment and alter their display properties. Every motion agent must have its own unique interaction artifact.

SenseArtifact: Provides the operations to share beliefs between the agents who have focus the artifact. It can be focused by any type of agent.

Focusing the artifact automatically adds to agents the following beliefs:

- worldSize: The size of the world in pixels E.g. (600).
- gridSize: The grid size of the world in pixels E.g. (100).
- patchSize: The patch size E.g. (6)
- simSpeed: Simulation speed it is updated from the slider on the GUI
- tick: A step counter.

3.2.2 JLogo Internal Actions

Internal actions are an interface between the Jason layer and JModel. Most of them are used as 'sensors' operations which can 'read' the environmental state and can be used as guards or operations in the agent's plans. Internal actions are categorized under various prefixes depending the actions that performs. Some internals actions can be operated as **validators** on the agent plans.

Validator: In a validator internal action, which employs the classic logic predicate semantics, i.e. can have its arguments partially instantiated. For example

breedOf (agId,B) is an action that will return the breed of agent (agId) in the "output" parameter B, but if the action is called with a ground value in parameter B, then it will compare if the breed of agent matches the value of B and succeeds or fail respectively.

Internal actions that are related with information regarding an agent are called by adding the 'agent.' *prefix*.

- *breed* (agId, B): Returns the breed of the agent(agId). **[Validator]**

- *color* (agId, C): Returns the color of the agent(agId). **[Validator]**
- *size* (agId, S): Returns the szie of the agent(agId). **[Validator]**
- *visible* (agId, B): Returns the visibility of the agent(agId). **[Validator]**
- *dir (*agId, D): Returns the direction of the agent(agId). **[Validator]**
- *pos (agId, X, Y)*: Returns the pixel location of the agent(agId). **[Validator]**
- *patch* (agId, X, Y): Returns the patch location of the agent(agId). **[Validator]**
- *nextLocation* (agId,X,Y,S): Returns the patch location of an agent if performs a number of steps(S) forward. **[Validator]**
- *hasPatchesAhead(agId,C,W,S):* Returns true if an agent (agId) moving (W) number of steps from its current position and direction a speed (S optional) and step over a patch with the specific color(C). Else returns false **[Validator]**

Internal actions that 'sense' the environment are under the 'ask.' *prefix*.

- *agentsOf (B, L):* Returns all agent ids in a list (L) which are of the specific breed(B)
- *agentsOn (X, Y, L)*: Returns all agent ids in a list(L) which are in a specific patch location[X, Y].
- *countPatchesOf*(C,T): Counts and returns the number of patches of the specific color(C). **[Validator]**
- *nextPatchTo*(X,Y,D,X2,Y2): Returns the neighbor patch location (L2[X2,Y2]) of a current patch location(L1[X,Y]) and direction(D). **[Validator]**
- *firstPatchOf*(X,Y,D,C,X2,Y2): Returns the first patch location(L2[X2,Y2]) with color (C) of a patch location(L1[X,Y]) and direction (D). **[Validator]**
- *inWorld* (x, y): Action succeeds or fails depending of the given patch Location [x, y] if exists in the grid world. **[Validator]**
- *isNeighbor* (X1, Y1, X2 ,Y2): Action succeeds or fails depending if the two given locations(L1[X1, Y1], L2[X2, Y2]) are neighbors. **[Validator]**
- *patchesOf(C,L):* Returns a list(L) of patches of a specific color(C).
- *patchColor(X,Y,C):* Returns or validates the patch color. If you pass feedback parameter returns the patch colors else validates the color of the patch if matches the given one. **[Validator]**

- *patchNeighbour(X,Y,L):* Returns the neighbor patches of the current patch location in a list (does not include diagonal patch locations).
- *patchAgent(X,Y,N):* Returns the patch agent id of a patch location if exists. **[Validator]**
- *patchAgentPos(N,X,Y):* Returns the patch agent position. **[Validator]**

General internal actions that perform a simple action are under the general 'jia.' *prefix*.

- *distanceTo* (x1, y1, x2, y2): Returns the distance between two locations (L1[x1, y1], L2[x2, y2]). **[Validator]**
- *validatePos* (X1, Y1, X2, Y2): Action fails or succeeds depending if the two locations (L1[X1, Y1] , L2[X2, Y2]) are equals. **[Validator]**
- *random* (P,R): Returns a random number (R) between [0,r) range.

Internal actions alongside with the Cartago artifacts operations which we are going to present later in this thesis, can be used by the agents to senses the environment. It should be noted that, Cartago artifact operations cannot be used as guards in Jason agent's plans.

3.3 JLogo User interface

Figure 3 it is shows JLogo user interface after running the JLogo JaCaMo project.

The main features:

- Simulation view displays the graphical simulation
- Simulation speed slider controls the speed of the simulation
- Start button triggers the observer agent start plan.
- Ticks shows the current ticks value
- On the bottom it is displayed information about the graphical

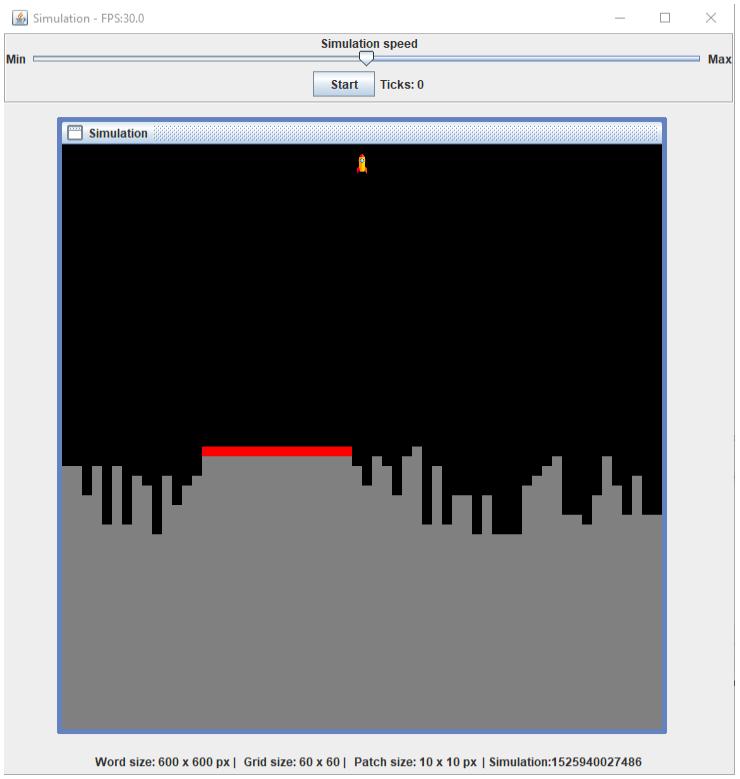

Figure 3: JLogo's interface with lunar landing simulation example

environment such as World size in pixels, Grid size, Patch size in pixels and simulation number.

It should be noted that, the development interface is the Eclipse IDE with the use of Jacamoide plugin.

4 JLogo implementation

JLogo is a JaCaMo [3] project developed under Jacamoide [18] which intends to extend the capabilities of Jason by providing the proper tools in the Jason [2] layer so the agents can interact, sense and navigate in a 2D graphical simulation environment. In order to be able the agents to "sense" and "alter" the environment we created "operations" and "sensors" which are exposed in Jason by Cartago [4] artifacts and Jason [2] internal actions (jia). The environmental state is rendered by the simulation view. The simulation view and the other UI components are rendered in the JLogo GUI window [Figure 4].

Even though Jason [2] provides some components (GridWolrdView, GridWorldModel etc.) for creating graphical simulations, the use of them requires the user to resort to java for their implementation. JLogo aims, for the students / developers who first come into contact with the field of MAS to focus *only* in the Jason [2] (AgentSpeak) tier for studying and creating their MAS simulations providing a configurable graphical environment. To achieve this, we use java's swing for creating the graphical components. As a JaCaMo [3] project developers can use Eclipse IDE alongside with Jacamoide [18] plugin to develop their JLogo simulations. Developers / Students can use JLogo project as their base JaCaMo [3] project for creating 2D graphical

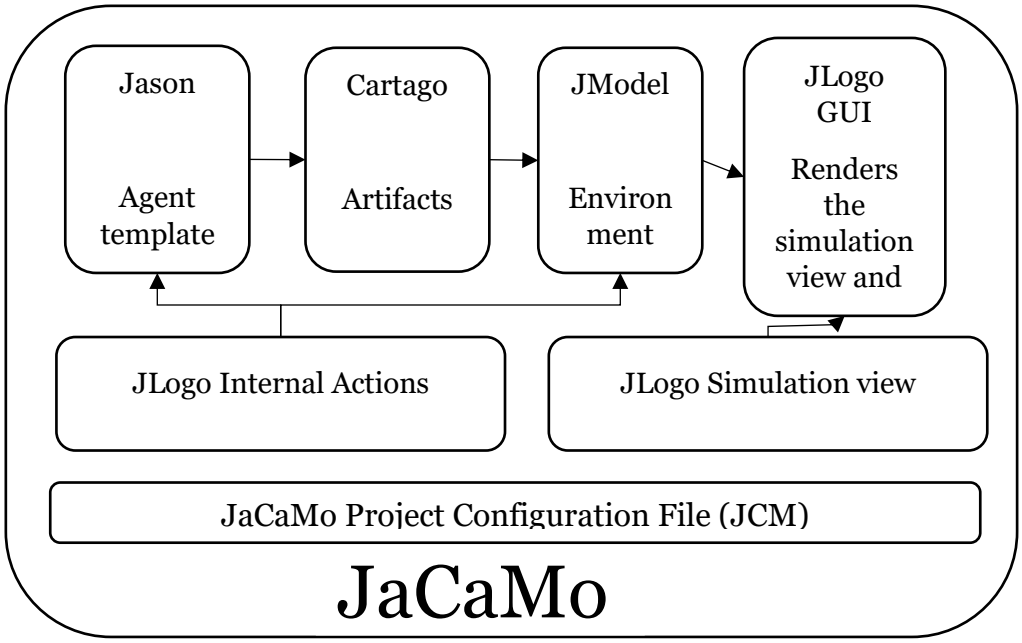

Figure 4: JLogo JaCaMo structure

simulations. They can extend or altered it based on their needs. Automatically a graphical window will appear rendering the simulation.

JLogo Project file (JCM)

In JaCaMo, a project such as the JLogo has a configuration file that defines the initial state of the MAS. In this file, workspaces, agents (Instances, beliefs and focused artifacts/workspaces), artifacts and organisations are declared. In JLogo we create only the operational workspace (jlogo) and the Observer agent which will handle all the other configuration options.

This is a standard JLogo JCM file

```
mas jasonlogo {
        agent observer : observer.asl
        workspace jlogo {}
        class-path:lib
        asl-path: src/agt
                src/agt/inc
}
```

Developers can extend and alter the JLogo JCM at will, based on their MAS simulation needs.

4.1 Jason Layer (Operational agent templates)

In JLogo environment there are three basic roles of agents (observer, motion, stationary) which can be used in the simulation. In the following we present an agent template structure for every type of agent in the JLogo.

4.1.1 Observer Agent

The 'God' of the simulation. It is responsible for instantiating the initial state of the simulation, can create or destroy dynamic agents and patches, sense and alter anything in the simulation environment. One and only instance of this type of agent should exist in the JLogo MAS simulation. The instantiation of the Observer agent is pre-defined in the JLogo JCM file.

The initial plan of the observer is to focus the operational workspace and create the necessary artifacts for the JLogo simulation.

```
/* Initial  Beliefs */
pool(0). //Artifact pool id
ready(false). //Observer ready flag

/* Initial goals */
!createArtifacts
```

The basic artifacts which Observer needs to create for the graphical simulation environment are the following:

- WorldArtifact: Instantiates the JLogo model and JLogo GUI (The graphical representation of the MAS)
- EnvironmentArtifact: Provides the tools to edit the simulation environment (Add or remove agents and patches, alter the patches.
- SenseArtifact: Provides the tools which an agent can use to sense the world and share beliefs between the agents which have focus the artifact.

For example, the following code:

```
+!createArtifacts:true
<-
	joinWorkspace("jlogo",_);
	makeArtifact(world,"jasonlogo.artifact.WorldArtifact",["Sim",600,10],W);
	focus(W);
	makeArtifact(env,"jasonlogo.artifact.EnvironmentArtifact",[],E);
	focus(E);
	makeArtifact(sense,"jasonlogo.artifact.SenseArtifact",[],S);
	focus(S);
.
```

After the instantiation of the mandatory artifacts, observer is ready to create the initial environment state. In this state Observer can define motion agents, patch agents and simple patches.

Internally call signal ‘setup’ is called after the instantiation of WorldArtifact

```
+setup
<-
	!createWorld
.
+!createWorld
```

```
<-
        //Create your initial world state here
        //.
        //World state is ready update the observer ready belief to true
        -+ready(true);
.
```

When the initial environment is created, the observer has completed all the necessary setup work and it is ready and waiting the user to call the 'start' from the interface by click on start button.

```
+start:ready(true)
<-
        .print("Ok everyone GO!");
        .broadcast(achieve,observerReady);
.
```

One mandatory task of the observer is to create the interaction tools for the operational motion agents when they request one.

```
+!requestTool[source(S)] : toolBox(S,T)
<-
        .print("Agent:",S," have requested a tool again!");
        .send(S,achieve,pickupTool(T))
.

+!requestTool[source(S)]
<-
        .print("Agent:",S," have requested a tool");
        !createTool[source(S)]
.
```

These tools are used by the operational agents giving them the capabilities to navigate in the environment.

```
+!createTool[source(S)] : pool(X)
<-
        -+pool(X+1);
        .wait(100);
        .concat("i",X,I);
        makeArtifact(I,"jasonlogo.artifact.InteractionArtifact",[],_);
        +toolBox(S,I);
        .send(S,achieve,pickupTool(I))
.
```

Other major jobs that observer can do is to create and remove agents and patches from the environment.

Spawn an agent: To spawn an agent you need to provide a name (Optional), Breed, agent template path (asl), X patch position, Y patch position, D direction.

```
+!spawnAgent(Agent,Breed,Template,X,Y,D): ready(R) <-
	.print("Creating agent...");
	.create_agent(Agent,Template);
	createAgent(Agent,Breed,X,Y,D);
	if(R) {
		.send(Agent,achieve,requestTool);
	}
.
```

Kill an agent: To remove an agent from the environment you must provide the agent id.

```
+!killAgent(Agent):toolBox(Agent,T)
<-
	.kill_agent(Agent); //Remove agent from MAS
	removeAgent(Agent); //Remove agent from Model
	lookupArtifact(T,S);
	disposeArtifact(S); //Remove agent Interaction artifact
	-toolBox(Agent,T); //Remove artifact from the toolbox
	.print("Remove ",Agent," and all its tools.");
.
+!killAgent(Agent):true
<-
	.kill_agent(Agent)
	removeAgent(Agent);
	.print("Remove ",Agent,".");
.
```

Spawn a patch agent: To spawn agent you must provide patch agent template path (asl), patch X position, patch Y Position and color.

```
+!spawnPatch(Template,X,Y,Color): not ask.patchAgent(X,Y,N) & ready(R)<-
	.create_agent(Patch,Template);
	setPatchColor(X,Y,Color);
	setPatchId(X,Y,Patch);
	if(R) {
		.send(Patch,achieve,prepare);
	}.
.
```

If patch location already has a patch agent on top of it this plan will fail.

```
-!spawnPatch(Template,X,Y,Color)
<-
	.print("Already exist patch agent on X:",X," y:",Y);
.
```

Kill a patch agent: To remove a patch agent from the environment you should know the patch position.

```
+!killPatch(X,Y):ask.patchAgent(X,Y,N) <-
	clearPatch(X,Y);
```

```
        .kill_agent(N);
.
```

Agent manipulate operations

Using the capabilities of Cartago [4] to call artifact operations through Jason without focusing the artifact, the observer can manipulate the other agents. The observer keeps the Interaction Artifacts names of the motion agents in a list. Having this "toolbox" only observer can manipulate the motion agents.

E.g. Observer force agent "bob" to move one step forward (operation "fd").

```
+!moveBobForward:true <-
        ?toolBox(bob,T);
        lookupArtifact(T,A);
        A::fd;
.
```

In the example above:
?toolBox(agentId,T) will return agent's InteractionArtifact name.
lookupArtifact(T,A) will return the InteraxtionsArtifact's instance.
A::operation calls the operation of artifact (A)

4.1.2 Motion Agent

Motion agents can be pre-defined in the JCM file or created dynamically by the Observer or the Patch agents.

The initial plan of the motion agent is to join the workspace and wait for the environment to be instantiated.

```
!join.
+!join : true <-
        joinWorkspace("jlogo",_);
        .print("Hello world. I have join the workspace:jlogo");
.
```

When the observer is ready (has finish the creation of the initial world state) should inform the agents.

```
+!observerReady[source(observer)]:true
<-
        .print("Request interaction tool from observer.");
        .wait(100);
        !requestTool
.
```

When observer is ready the agent requests an interaction tool (artifact) from the observer so can get the capabilities to navigate in the world.

```
+!requestTool:true
<-
	.send(observer,achieve,requestTool);
.
```

Observer informs the agent that the tool is ready. Agent focus the interaction artifact which it was made for it. Every interaction artifact is unique for every motion agent. In this state motion agents can focus other shared artifacts such as the Sense artifact.

```
+!pickupTool(T)[source(observer)] : true
<-
	lookupArtifact(T,I);
	focus(I);
	lookupArtifact(sense,S);
	focus(S);
	!prepare
.
```

After the motion agent has picked up its tools, calls the ready operation of the interaction artifact to get as beliefs its current state. The initial beliefs added to the agent are world grid Size, patch position, pos pixel position, breed, direction.

```
+!prepare:true <-
	.wait(200); //Wait a bit for cartago artifacts to be focused
	ready;
	!prepared
.
```

After this step agent is prepared and ready to achieve his goals and plans.

```
+!prepared:true
<-
	.print("I have picked up my tool and I am prepared for action.");
.
```

From there developers can add agent's goals and plans to start the operation of the motion agent.

4.1.3 Stationary (Patch) Agent

This agent is the 'Operational patch', i.e a location with a behavior that allows to model complex environments. It is placed in a patch location, it is stationary but can interact with the other agents, including spawning new agents and patches.

The initial plan of the patch agent is to join the workspace and wait for the environment to be instantiated.

```
!join.
+!join : true <-
	joinWorkspace("jlogo",_);
	.print("Hello world.I have join the workspace:jlogo");
.
```

When the observer is ready (have finish the initial world state) should inform the agent.

```
+!observerReady[source(observer)]:true
<-
	.print("Observer is ready");
	!prepare
.
```

Patch agent is not a motion agent, so it is not required by the observer to create an interaction artifact. The artifacts which a patch agent needs is the Environment and sense artifacts.

```
+!prepare:true <-
	lookupArtifact(env,E);
	focus(E);
	lookupArtifact(sense,S);
	focus(S);
	!prepared
.

-!prepare:true<-
	.wait(100);
	!prepare;
.
```

After this step the patch is prepared and ready to achieve his goals and plans.

```
+!prepared:true <-
	.print("I am prepared!");
.
```

Patch agents plans for spawning patches and agents is the same as the observer. From there developers can add their goals and plans to start the operation of the agent.

4.2 Cartago layer (JLogo Artifacts)

All capabilities of the agents to navigate, sense and manipulate the environment are achieved through Cartago [4] artifacts which is the main interface between the Jason and JModel.

Four basic artifacts are predefined in the JLogo providing all the mandatory operations which can manipulate the simulation environment. Each artifact is responsible to achieve one purpose only.

4.2.1 World Artifact

World Artifact instantiates the JModel and JView. It is responsible to manipulate the graphical window of the simulation. Should be used only from the "observer" and only one instance should be created in a simulation. The environment initial parameters are passed through the init method which it is used as constructor by the artifact.

@OPERATION **init** constructor of the World Artifact

Parameters:

- title: Graphical simulation window title
- worldSize: Simulation view world size (SxS) in pixels
- patchSize: Patch size in pixels

Example code in Java:

```
/**
 * Artifact constructor
 * @param title Simulation title
 * @param worldSize Simulation world size
 * @param patchSize Simulation patch size
 */
@OPERATION void init(String title,int worldSize,int patchSize) {

        //Create the model
        if(model == null) {
                model = JasonLogoModel.create();
        }
        //Update the world size
        model.updateWorldSize(worldSize, patchSize);

        //Create the view
        view = new JasonLogoGUI(model, title);

        //Hide the setup panel
```

```
            view.getSetupControlPaneL().setVisible(false);

            //Setup callback listeners
            setupViewListeners();

            //Initialize the world
            execInternalOp("initWorld");

    }
```

@OPERATION **simulation** sets the randomizer seed to instantiate the same simulation

- seed: The seed of the randomizer

@OPERATION **purge** clears the agents and patches from the model.

Parameters: No parameters

Also WorldArtifacts “fires” signals in Jason tier as GUI listeners.

- setup: Fires internally when the instantiation of the GUI is completed.
- start: Fires when the start button is clicked
- patchClicked(X,Y): Fires when the mouse clicks on a patch

@OPERATION **enableGrid** Enable grid lines

Parameters: No parameters

@OPERATION **disableGrid** Disable grid lines

Parameters: No parameters

@OPERATION **gridColor** sets the color of the grid

Parameters:

- color The color of the grid

@OPERATION **showMessageDialog** shows a GUI dialog with a message

Parameters:

- message An array of variables that will be parsed as a sentence.

4.2.2 Sense Artifact

Sense Artifact operates as a blackboard. Provides the operations which agents can use for sensing (read) the environment and share (write) their beliefs to all agents which *have focus* the artifact. Can be focused from every type of agent.

@OPERATION **init** Constructor of the Sense artifact declares the belief of gridSize, worldSize, patchSize, simSpeed and tick counter.

Parameters: No parameters

@OPERATION **distanceToAgent** Returns the pixel distance between the operational agent the provided one.

Parameters:

- agentId: The id of the agent will be used to measure.
- distance: FeedBack parameter for returning the distance between the artifact operational agent and the provided one.

Example code in Java:

```
@OPERATION
public double distanceToAgent(String agId) {
        //Get the operational agent id
        String operationalAg = this.getCurrentOpAgentId().getGlobalId();
        //Return the distance of the two agents by consulting the model
        return model.distanceToAgent(operationalAg,agId);
}
```

@OPERATION **addBelief** adds a belief to all operational agents.

Parameters:

- beliefName: The name of the belief
- params: The parameters of the belief.

@OPERATION **removeBelief** removes a belief of all operational agents.

Parameters:

- beliefName: The name of the belief
- params: The parameters of the belief (optional).

@OPERATION **updateBelief** updates a belief

Parameters:

- beliefName: The name of the belief
- params: The parameters of the belief (optional).

@OPERATION **tick** updates tick counter by one. Also updates the tick belief. Every tick update it is delayed by an amount of ms. The amount of the wait time is controlled by the simulation speed slider of the GUI.

Parameters: No parameters

@OPERATION **clearTicks** clears ticks counters. Also updates the tick belief.

Parameters: No parameters

4.2.3 Interaction Artifact

Provides the motion and display capabilities for the agents, gives them the abilities to navigate in the environment by moving, turning etc. and change their appearance by changing the size, view and visibility. This artifact should be unique for each agent (E.g. only one agent should operate the artifact).

@OPERATION init: Is the constructor of the artifact.

Parameters: No parameters

@OPERATION **ready:** Bounds the artifact to the agent. Add to the agent the initial beliefs. (Patch position, pixel position, breed, dir).

Parameters: No parameters

@OPERATION **patch**: Set agent grid position.

Parameters:

- X patch X position
- Y patch Y position

@OPERATION **pos:** Set agent pixel position.

Parameters:

- X pixel X position
- Y pixel Y position

@OPERATION **dir:** Set agent direction.

Parameters:

- D direction [0,360] degrees.

@OPERATION **fd**: Moves the agents one step forward based on the current pixel location and direction

Parameters: No parameters

@OPERATION **bd**: Moves the agent one step backward base on the current pixel location and direction

Parameters: No parameters

@OPERATION **tr:** Turn agent right from his current direction.

E.g. Agent direction 50° tr(20) new agent direction 70°

Parameters:

- degrees: the degrees to turn

@OPERATION **tl:** Turn agent left from his current direction.

E.g. Agent direction 50° tl(20) new agent direction 30°

Parameters:

- degrees: the degrees to turn

@OPERATION **rt** Turns the agent to a random direction

Parameters: No parameters

@OPERATION **speed** set agent's speed

Parameters:

- speed: The new speed of the agent

@OPERATION **speedup** adds speed to the agent's current speed

Parameters:

- speedup: The amount of speed that will add to agent's current speed.

@OPERATION **moveTowardsPos** moves the agent one step towards to the given pixel position

Parameters:

- X: Position of X pixel
- Y: Position of Y pixel

@OPERATION **moveTowardsPatch** moves the agent one step towards to the given patch position

Parameters:

- X: Patch X position
- Y: Patch Y position

@OPERATION **message** sets a text message to an agent to be rendered on the graphical potion of the agent.

Parameters:

- Message: The message to be displayed

@OPERATION **size** sets the size of the agent

Parameters:

- size: the size of the agent

@OPERATION **color** sets the color of the agent

Parameters:

- color: the color of the agent

@OPERATION **image** sets the image of the agent from assets

Parameters:

- image: the image name

@OPERATION **hide** make the agent hidden (Still can interact with other agents but it won't render)

Parameter: No parameters

@OPERATION **show** make the agent visible (Still can interact with other agents but it won't render)

Parameter: No parameters

4.2.4 Environment Artifact

Provides the operations for manipulate the environment. Adds or removes agents and patches.

@OPERATION **setPatchColor** Set patch color

Parameters:

- X patch X position
- Y patch Y position
- C patch color

@OPERATION **setPatchRGBColor** Set patch rgba color.

- red from [0,255]
- green from [0,255]
- blue form [0,255]
- a from [0,255]

@OPERATION **setPatchId** set patch id for patch agents

- X patch X position

- Y patch Y position
- ID the id of the patch

@OPERATION **clearPatch** Clear patch color and patch agent (if there one which operates on the patch)

Parameters:

- X patch X position
- Y patch Y position
- agentId Feedback parameter which return the patch agent id [Optional]

@OPERATION **setPatchText** sets a text in a patch location.

- X patch X position
- Y patch Y position
- Message text of the patch

@OPERATION **createAgent** adds a new agent in the JLogo model

Parameters:

- agent id Agent id
- breed Agent breed
- x Patch x position
- y Patch y position
- d Direction

@OPERATION **removeAgent** removes an agent from the model

Parameters:

- agent id The id of the agent

@OPERATION **setBreedColor** sets current breed agents a color

Parameters:

- breed The breed name

- color The color

@OPERATION **setBreedSize** sets current breed agents a size

- breed the breed name
- size The breed size

@OPERATION **setBreedImage** sets current breed agents an image from assets folder

- breed the breed name
- image the image name

@OPERATION **setAgentColor** sets agent's color

- agentId the agent id
- color the color name

@OPERATION **setAgentSize** sets agent's size

- agentId the agent id
- size the size in pixels

@OPERATION **setAgentImage** sets agent's color

- agentId the agent id
- image the image name

@OPERATION **colorPatches** Colors the patches of the area with a specific color

- X of first point
- Y of first point
- X2 of second point
- Y2 of second point
- Floor percent of patches that will be colored (1 = all patches) (0 = none)
- ColorName the name of the color

@OPERATION **addWall** paints the patches in a straight line. The line should be vertical or horizontal.

➔ x1 = x2 vertical line

➔ y1 = y2 horizontal line

➔ x1 = x2 and y1 = y2 or x1 != x2 and y2 !=y2 (does nothing)

Parameters:

- c Color of the wall
- x1 X axis of starting point
- y1 Y axis of starting point
- x2 X axis of end point
- y2 Y axis of end point

@OPERATION setPatchImage sets patch image

Parameters:

- x location of patch
- y location of patch
- image Image name from assets forlder

4.3 JLogo Model

JLogo model is the heart of the simulation. All information about the graphical simulation is stored in the model. Agents can 'read' and 'write' the model through Cartago artifacts and Jason internal actions. The model is a single instance object which can accessed in the context of the JLogo by the class path.

4.4 JLogo GUI

JLogo is using Java's swing for creating its graphical components. All components are defined on the instantiation of the JasonLogoGUI which is made in WorldArtifact. Also is responsible for triggering the render of the simulation view. The default refresh rate of the simulation view is the 30 fps.

4.5 JLogo Simulation View

JLogoView is the simulation window. It is based on java's swing panel and renders the state of the JLogo model. It is used by the JLogoGUI for updating the simulation view.

5. Example Models in JLogo

We are going to demonstrate the use of JLogo through some example Models for better understanding. So, for this purpose we have create four simulations:

- Lunar Lander: An arcade game
- Burning forest: A fire spreading simulation
- Drone waiter: A simulation about drone waiters
- Tic-Tac-Toe: A game playing against a bot

5.1 Lunar Lander

We are going to implement a famous arcade game the **Lunar Lander**, in this game the spaceship which is the main "player" is trying to land safely in a harsh environment avoiding cliffs, finding the platform and landing safely on it with a proper speed and angle.

The required entities are:

- A moon environment
- A spaceship
- A platform to land.

As it has been mentioned in JLogo "observer" agent is responsible to instantiate the initial simulation state.

In the following the detailed steps are presented.

jasonlogo.jcm

In the configuration file of JLogo (jasonlogo.jcm) is defined the initial state of the MAS. The observer agent and the operational workspace (jlogo) are pre-defined.

```
mas jasonlogo {

   agent observer : lunarLander.asl
   workspace jlogo {}
   class-path:lib
   asl-path: src/agt
         src/agt/inc

}
```

lunarLander.asl (observer)

For the lunar lander example, the observer has two beliefs. How large will be the landing platform (platformSize) and how smooth will be the environment (smooth).

For instance:

```
smooth(22).
platformSize(5).
```

The observer create the world artifact with title “Lunar lander”, simulation view 600x600 pixels and patch size 10px. This will result having a grid world of 60x60.

```
+!createArtifacts:true
<-
	...
	makeArtifact(world,"jasonlogo.artifact.WorldArtifact",["Lunar lander",600,10],W);
	...
.
```

In the createWorld plan, the observer is going to create the environment based on his beliefs about how smooth the environment will be and the size the platform.

```
+!createWorld:gridSize(G) & smooth(B) & platformSize(PS)
<-
	jia.random(G-PS,P);
	jia.random(G/2-B,R);

	for(.range(I,0,G-1)){
		if(I>=P & I<P+PS) {
			addWall(gray,I,G-B-R,I,G);
			setPatch(I,G-B-R,red);

		} else {
			jia.random(G/2-B,N);
			addWall(gray,I,G-B-N,I,G);
		}
	}
	!spawnSpaceShip;
	-+ready(true);
.
```

To create a rough environment simulating that of the moon, cliffs are added by using addWall operation of EnvironmentArtifact for every line of the grid.

How steep the cliffs will be depends on the value of the smooth belief. Thus smooth (0), will result a very steep environment or smooth (gridSize/2-1), will result a flat environment. So, the value should be between 0 and gridSize/2-1.

After creating the lunar environment and the platform, the observer spawns the spaceship agent.

```
+!spawnSpaceShip:gridSize(G)
<-
        setBreedColor(ship,blue);
        setBreedSize(ship,20);
        setBreedImage(ship,"rocket.png");
        !spawnAgent(ship,ship,"ship.asl",G/2,2,270);
.
```

The observer can create an agent by using the plan !spawnAgent. To create a spaceship agent observer creates an agent with source of “ship.asl” and breed “ship”.

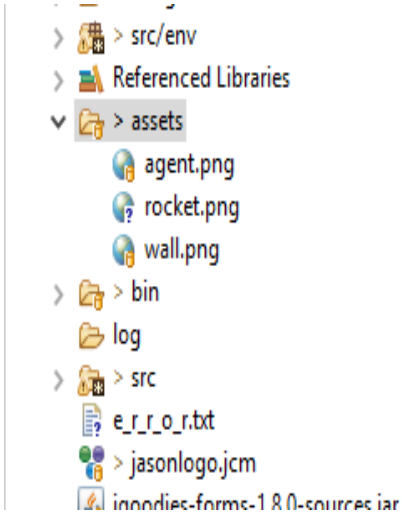

Figure 5: Assets folder for images.

Additionally the observer can set breed’s view by setting the color (setBreedColor), size (setBreedSize) and image (setBreedImage). It should be noted that all images should be placed in the assets folder [Figure 5].

Upon completion of the createWorld plan the UI should be look like the one depicted in [Figure 6].

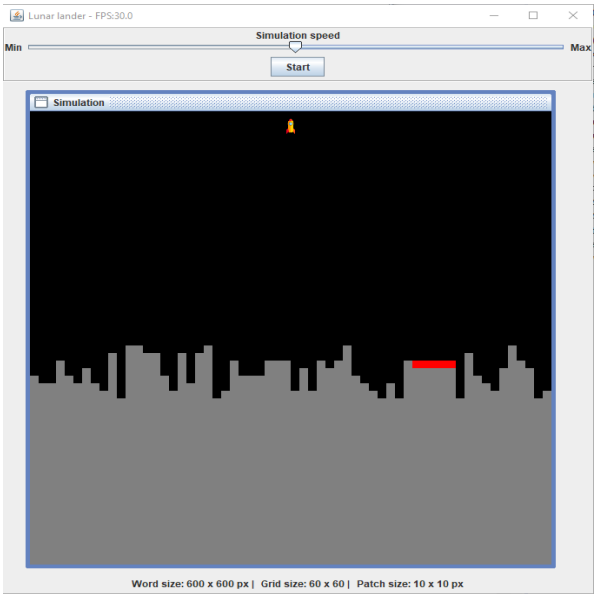

Figure 6: Lunar Lander simulation view.

To inform the agent to start its execution, the user should click the “start” button.

ship.asl

The Ship agent’s main goal is to land successfully on the platform avoiding the steep environment of the moon. To achieve this goal, the ship agent has two main plans. The first plan handles the positioning over the landing platform with correct direction and the second plan concerns landing to the platform with a proper speed.

The initial beliefs of the ship agent are:

```
maxSpeed(3).
gravitySpeed(-2).
```

where maxSpeed defines the max positive speed of the ship and gravitySpeed defines the max negative speed of the ship.

I.e. Positive speed is the speed which the ship moves towards its direction. Negative speed moves the ship in opposite direction from its current direction and zero speed means that the ship is not moving at all in the simulation environment.

When the ship agent is prepared, some initial parameters and beliefs are set.

```
 +!prepared: true
<-
        speed(-1);
        !sense_platform;
        message("Going to platform");
        !goto_platform;
.
```

Setting the ship speed to -1 simulates the initial “gravity” effect to the ship’s movement. The achievement of the goal !sense_platform will add the platform’s position as an agent’s belief.

```
+!sense_platform:jia.patchesOf(red,L)
<-
        for(.member(X,L)) {
                .nth(0,X,I);
                .nth(1,X,J);
                +platform(I,J);
        }
.
```

After setting initial “gravity speed” and sensing the platform underneath, the ship agent is ready to start the execution of the plan !goto_platform. The behavior of the ship in order to reach the correct location is determined by a set of reactive plans that handle different situations in the environment. In what follows, each such reactive plan is explained in more detail.

If the ship’s patch location is on a gray patch, it means that the ship has crashed on the lunar terrain [Figure 7].

```
+!goto_platform:patch(X,Y) & jia.patchColor(X,Y,gray)
<-
        message("Crashed on X:",X," - Y:",Y);
        .send(observer,achieve,done(false));

.
```

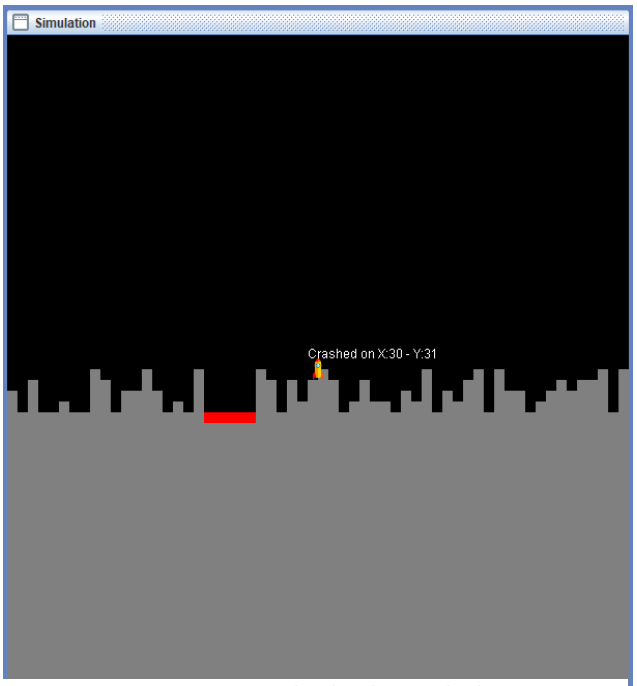

Figure 7: Lunar lander ship crashed.

When the ship is over the landing platform and its direction is vertical up, then can start the execution of the landing plan [Figure 8].

```
+!goto_platform:patch(X,_) & platform(X,_) & dir(270)
<-
        message("Landing");
        !land;
.
```

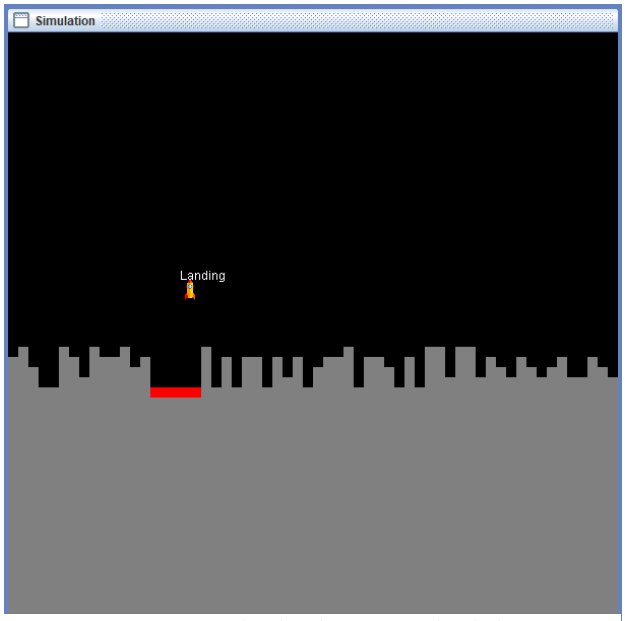

Figure 8 Lunar lander ship starting land plan.

If the ship is over the landing platform but its direction is not vertical up, then an attempt is made to correct the ship’s direction.

```
+!goto_platform:patch(X,_) & platform(X,_) & dir(D)
```

```
<-
        if(D < 270 & D >=90) {
                tr(1);
        } else {
                if(D > 270 | D <90) {
                        tl(1);
                }
        }
        !gravity_pull;
        !goto_platform
.
```

The following plan is activated when the ship is too close to cliffs, and turns the ship away from the cliffs.

```
+!goto_platform: .my_name(N) & patchSize(PS) & jia.patchesAheadOf(N,gray,PS*2)
<-
        tl(45);
        !thrust(2);
        !goto_platform
.
```

The following reactive plan is activated if the ship is neither over the platform nor close to cliffs. We simple let the gravity do its work.

```
+!goto_platform
<-
        !gravity_pull;
        !goto_platform
.
```

The execution of the “land” plan starts when the ship agent is over the platform and its direction is vertical up [Figure 8]. The land plan caters for landing the ship safely on the platform with a proper speed. In our simulation the safe landing speed threshold is set to -0.5. Speed less than -0.5 will result a crash for the ship [Figure 9].

If the ship’s patch location is red and its speed is less than the safe speed limit for landing means that the ship’s crashed.

```
+!land:patch(X,Y) & jia.patchColor(X,Y,red) & speed(S) & S < -0.5
<-
        message("Crashed on X:",X," - Y:",Y);
        .send(observer,achieve,done(false));
.
```

If the ship patch position is on a red patch and its speed is greater than -0.5 means that the ship has landed successfully [Figure 10].

```
+!land:patch(X,Y) & jia.patchColor(X,Y,red) & speed(S) & S >= -0.5
```

```
<-
        message("I have landed successfully!");
        .send(observer,achieve,done(true));
.
```

The following reactive plan is activated if the ship speed is less than -0.5. To avoid the ship crash the specific plan caters to raise the speed above the crash limit.

```
+!land:speed(S) & S <= -0.5
<-
        !thrust(0.1)
        !land;
.
```

The following reactive plan is activated when there is no other plan in landing process executing. This applies the gravity force on the ship's speed.

```
+!land
<-
        !gravity_pull;
        !land;
.
```

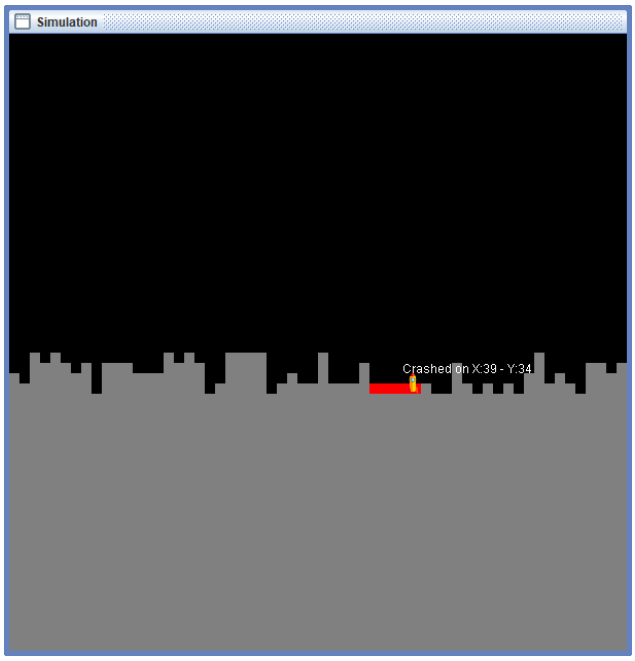

Figure 9: Ship crashed on landing

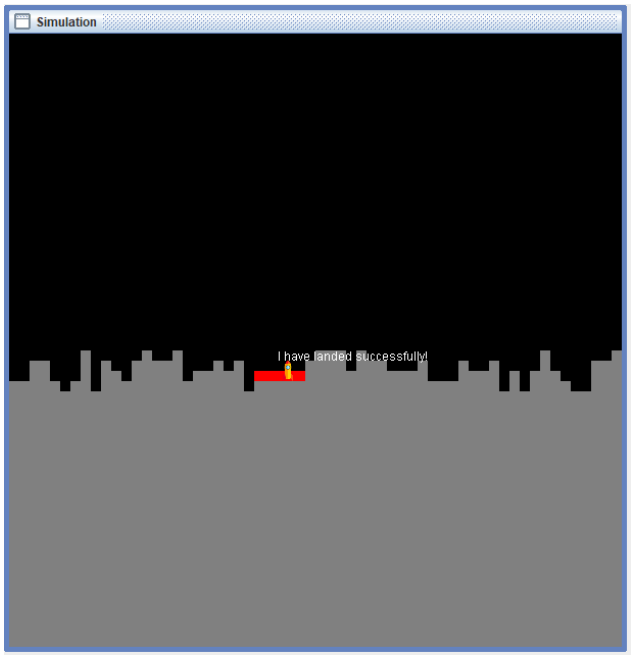

Figure 10:Ship landed successfully

The thrust plan adds an amount (A) of speed in ships current speed.

```
+!thrust(A):speed(S)
<-
        if(S < 1) {
                speedup(A);
        }
.
```

The following reactive plan is activated when the next calculated ship's patch location is outside of the top of the simulation view. Does not let the ship to go outside of the moon surface.

```
+!gravity_pull:.my_name(N) & jia.nextLocationOf(N,X,Y) & Y = -1
<-
	speed(0);
	tr(10);
.
```

This specific plan applies a little negative speed in ship's current speed simulating moon's gravity.

```
+!gravity_pull:speed(S) & gravitySpeed(GS) & dir(D) & simSpeed(Sim)
<-
	if(S > GS){
		speedup(-0.01);
	}
	if(S<0) {
		if(D < 270 & D >=90) {
			tr(1);
		} else {
			if(D > 270 | D <90) {
				tl(1);
			}
		}
	}
	tick;
	fd;
	.wait(Sim);
.
```

After a number of cycles the ship agent should have landed successfully over the platform with a proper speed and direction avoiding in the way any cliff.

5.2 Burning forest

In this example we are going to simulate how a fire can spread through a forest depending on the density of the trees.

In order to model the example, we need to implement:

- A forest
- A fire

The only acting agent in the simulation is the observer, which it is going to create the forest and start a fire in it.

In the following the detailed steps are presented.

The observer agent is pre-defined in the jcm file.

```
mas jasonlogo {
        …
   agent observer : burningForest.asl
        …
}
```

burningForest.asl (observer)

Observer creates the forest. The density of trees in the forest is set by the density belief.

```
density(0.7).
```

In order to simulating a forest, a random number of patches must be painted green. The random factor which depends if a patch will be painted green is the density belief. To achieve this, the use of the operation colorPatches of EnvironmentArtifact comes in handy which paints an area of patches with a specific color and a random factor.

```
+!createWorld:gridSize(S) & density(D)
<-
        colorPatches(0,0,S-1,S-1,1-D,green);
        ask.countPatchesOf(green,T);
        +totalTrees(T);
        !setFire;
        -+ready(true);
.
```

To set the initial fire observer paints the green patches (trees) in the first line as red (burning trees).

```
+!setFire: gridSize(S) & ask.patchesOf(green,L,0,0,S-1)
<-
        .print("Setting fire!");
        for(.member(X,L)) {
                .nth(0,X,I);
                .nth(1,X,J);
                setPatchColor(I,J,red);
        }       .
```

After the completion of the createWorld plan, the simulation view should look like the one depicted in [Figure 11]

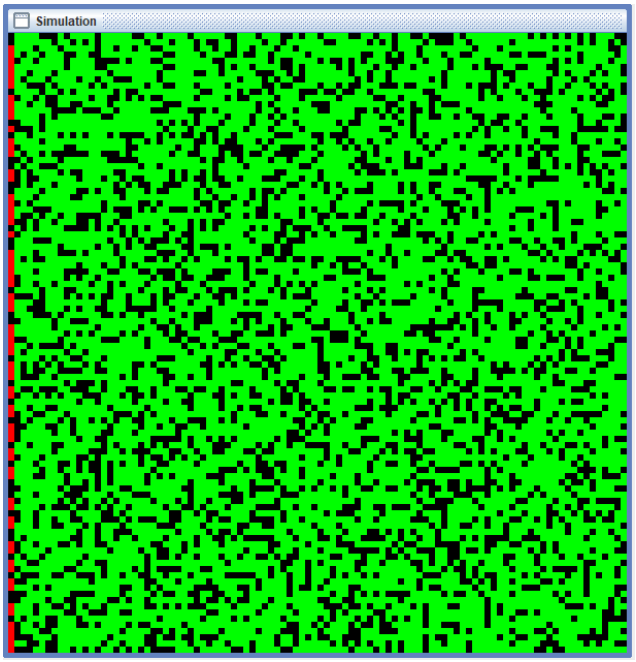

Figure 11: Burning forest initial state

To start the simulation, the user has to press the start button and let the fire progressing.

```
+start:ready(true)
<-
	.print("Ok everyone GO!");
	.broadcast(achieve,observerReady);
	!burn_forest;
.
```

Fire is spreading by having every tree set fire its neighbor trees. In every execution of the plan, all the burning trees (red patches) set fire to their neighbor trees and they get burned completed (changing their color to brown).

```
+!burn_forest:ask.patchesOf(red,L)& simSpeed(S) & .length(L,N) & N >0
<-
	for(.member(X,L)) {
		.nth(0,X,I);
		.nth(1,X,J);
		setPatchRGBColor(I,J,255,0,0,50);
		!burn_tree(I,J);
	}
	tick;
	.wait(S);
	!burn_forest;
.
```

If burn forest plan fails that means that there are no other trees to be burned. The observer prints the results of the simulation in the MAS console.

```
-!burn_forest:ask.countPatchesOf(green,T) & totalTrees(TT)
<-
        .print("Burned ",TT-T," of ",TT,". Forest burned:",100-(T*100)/TT);
.
```

The following plan is the one which spreads the fire of a burning tree to its neighbor trees. Neighbors are defined the top left right and bottom patches of the current patch with color green. To get the neighbor trees of a current tree, the internal action ask.patchNeighbour(X,Y,L,green) which returns the top, right, left and bottom patches of a current patch location with a specific color does the job.

```
+!burn_tree(X,Y):ask.patchNeighbour(X,Y,L,green)
<-

        for(.member(P,L)) {
                .nth(0,P,I);
                .nth(1,P,J);
                setPatchColor(I,J,red);
        }
.
```

When the simulation is completed observer prints the following message in the console: [Observer] Burned 6835 of 6983. Forest burned: 97.88

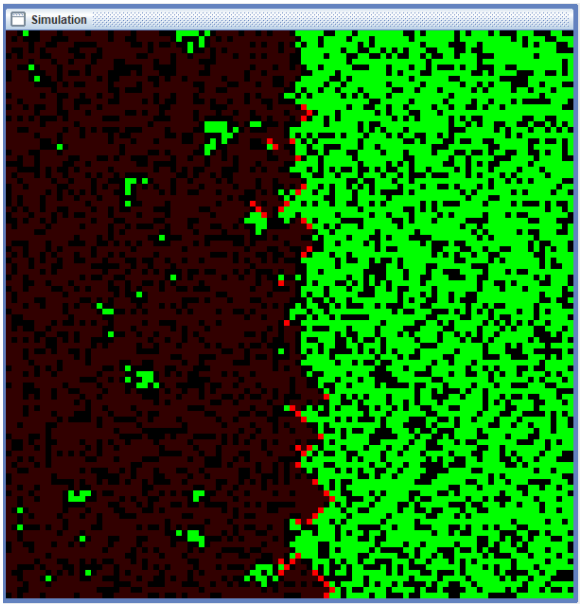

Figure 12: Fire progressing

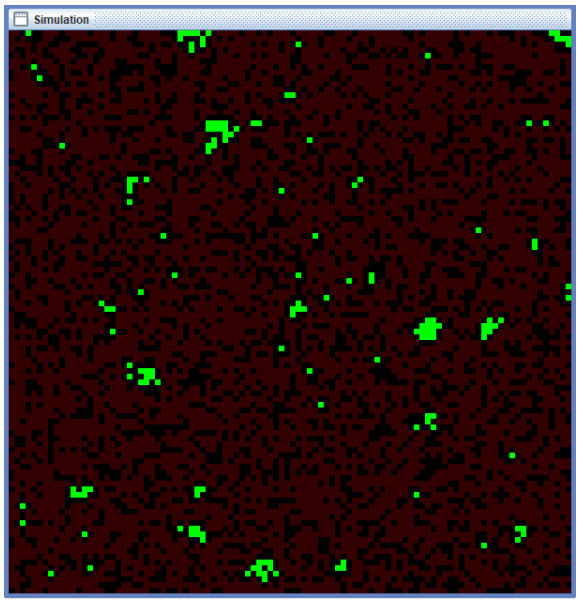

Figure 13: Forest burned

5.3 The Drone waiter

In the future, it is highly possible drones to be used as the waiters replacing humans in the task. In this example we are going to see how well drones can handle this job by simulating a store environment.

In order to model the example, we need to implement:

- A store
- A number of customers
- Machines which produce products
- A drone waiter

The role of the store will be played by the observer.

```
mas jasonlogo {
        ...
        agent observer : store.asl
        ...
}
```

store.asl (Obsever)

The store is responsible to setup the product machines, the drone waiter and let the customers to enter and exit (Spawn and remove customer agents) [Figure 14].

The beliefs of the store are the following:

- maxCustomers: the max number of customers entering the store
- Customers: a counter of the customers that already had enter the store
- CustomerPool: a counter to generate customer agent's ids (for internal use)

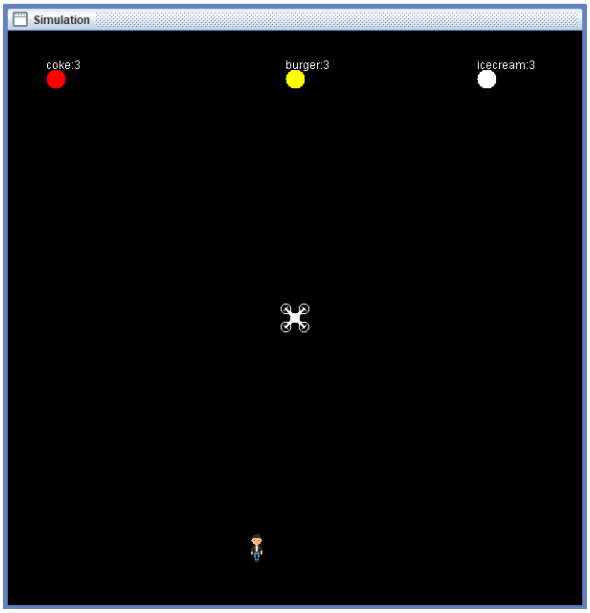

Figure 14: Initial state of the model

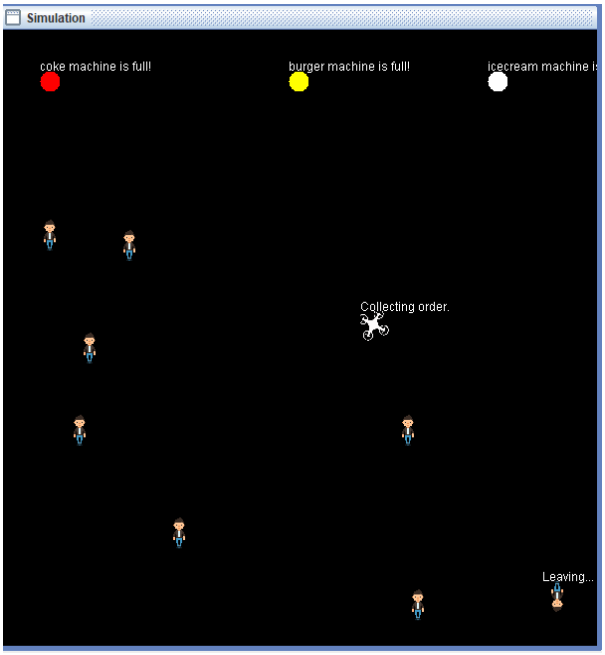

Figure 15: drone serving

The main plans of the store are the following:

In createWolrd plan observer will create the product machines and the drone waiter.

```
+!createWorld
<-
        !spawnAgent(cokeMachine,cokeMachine,"machine.asl",5,5,0);
        setBreedColor(cokeMachine,red);
        setBreedSize(cokeMachine,20);
        !spawnAgent(burgerMachine,burgerMachine,"machine.asl",30,5,0);
        setBreedColor(burgerMachine,yellow);
        setBreedSize(burgerMachine,20);
        !spawnAgent(icecreamMachine,icecreamMachine,"machine.asl",50,5,0);
        setBreedColor(icecreamMachine,white);
        setBreedSize(icecreamMachine,20);

        !spawnAgent(drone,drone,"drone.asl",30,30,270);
        setBreedColor(drone,blue);
        setBreedImage(drone,"drone.png");
        setBreedSize(drone,30);

        setBreedColor(customer,green);
        setBreedImage(customer,"customer.png");
        setBreedSize(customer,30);

        -+ready(true);
.
```

When the simulation is starts, the store allows customers to enter, spawning a new customer agent every few seconds.

```
+start:ready(true)
<-
	.print("Ok everyone GO!");
	.broadcast(achieve,observerReady);
	!spawnCustomers;
.
```

The following plan spawns a new customer every few seconds. As the max number of customers limit defined by maxCustomers belief does not has reached, the plan is executed retrospectively.

```
+!spawnCustomers: maxCustomers(MC) & customers(C) & C < MC
<-
	!spawnCustomer
	.wait(5000);
	!spawnCustomers
.
```

The following plan is activated when the max number of customers limit has reached. The store prints a generic messages and does not continue to spawn any new customers.

```
+!spawnCustomers:  maxCustomers(MC) & customers(C) & C == MC
<-
	.print("Shop closed!");
.
```

The following plan, is the one which creates a new customer in a random position in the simulated store environment.

```
+!spawnCustomer: customerPool(P) & gridSize(GS) & customers(C)
<-
	-+customers(C+1);
	-+customerPool(P+1);
	.concat("customer",P,N);
	jia.random(GS-6,X);
	jia.random(GS-10,Y);
	!spawnAgent(N,customer,"customer.asl",X+3,Y+10,270);
.
```

The store is also responsible for letting the customers to leave the shop. When a customer informs the shop that it is leaving, the store removes the specific customer from the simulation.

```
+!leaving[source(S)]
<-
	!killAgent(S);
.
```

machine.asl

The machine agent is responsible to provide the products to the drone and refill its resources.

The beliefs of a machine are:

- Products: The number of current products
- maxProducts: the number of the max capacity of products
- delay: interval of the generation of new product

The main plans of the machine are to provide a product to the drone when it is request one and refill its resources.

The fillProduct plan is executed constantly by the machine, generating a new product. Activates as long as the max capacity of the products does not have reached.

```
+!fillProduct:products(P) & maxProducts(MP) & P < MP & delay(D) & productName(N)
<-
	message(N,":",P+1);
	-+products(P+1);
	.wait(D);
	!fillProduct
.
```

The specific fillProduct plan is executed, when the max capacity have reached. Wait a small amount of time and executes the plan again.

```
+!fillProduct:productName(N)
<-
	message(N," machine is full!");
	.wait(5000);
	!fillProduct
.
```

This plan is activated when the drone requests a product of the machine, but the machine does not have any product to provide. Informs the drone for the unavailability of the product.

```
+!requestProduct[source(S)]:products(P) & P=0 & productName(N)
<-
	.send(S,achieve,notAvaible(N));
	.print("No product!");
.
```

This plan is activated when the drone requests a product of the machine and the machine has available products to provide.

```
+!requestProduct[source(S)]:products(P) & P>0 & productName(N)
<-
```

```
        message(N,":",P-1);
        -+products(P-1);
        .send(S,achieve,pickProduct(N));
        .print("Drone took product!");
.
```

customer.asl

Upon entering the store, a customer decides what he wants to eat and waits for the drone to take the order. When the order is delivered the customer eats and leaves the shop.

The beliefs of a customer are:

- availableDesires A list of products that he can choose.
- Ordered: If has given the order to the drone

The main plans of the customer are:

Create a list with products that he wants to ear, with at least one product is picked from the available

```
+!iWant:avaibleDesires(L) & .length(L,AL)
<-
        for(.member(X,L)){
                jia.random(2,N);
                if(N==0) {
                        +desires(X);
                }
        }
        .findall(X,desires(X),D);
        .length(D,ND);
        if(ND == 0) {
                jia.random(AL,NN);
                .nth(NN,L,IT);
                +desires(IT);
        }
.
```

The specific plan activates when the drone request the order from the customer. The customer gives “send” the order to the drone.

```
+!order[source(drone)]:.findall(X,desires(X),L)
<-
        -+ordered(true);
        .send(drone,achieve,customerOrder(L));
.
```

The specific plan activates when the drone delivers the order to the customer. The customer “eats” the order and leaves the some after some time.

```
+!eat[source(drone)]:pos(X,_) & worldSize(W)
<-
	message("Eating...");
	.wait(1000);
	message("Leaving...");
	dir(90);
	!gotoPosition(X,W);
	.send(observer,achieve,leaving);
.
```

drone.asl

Drone is responsible to select a customer, go to him, take the order, collect the order and serve it to the customer. This process is repeated as long as there are customers in the store.

The beliefs of the drone are the following:

- served([]) A list of the customers that already had served
- orderToCollect([]). A list of an order for collection
- bench(coke,cokeMachine). The operating machines of the store
- bench(burger,burgerMachine).
- bench(icecream,icecreamMachine).

The main plans of the drone are:

Select a customer to serve. The customer must have not already get served, in the shop maybe customers who have already served but does not have left the shop yet.

```
+!getNextCustomer: ask.agentsOf(customer,L) & served(SL) & .difference(L,SL,CL) & .length(CL,N) & N
>0
<-
	.wait(300);
	message("Going to customer.");
	.member(C,CL);
	agent.pos(C,X,Y);
	-+nowServing(C,X,Y);
	!serveCustomer;
.
```

The following reactive plan is activated when all customers in the shop have served. The drone flies to the center of the store.

```
+!getNextCustomer: ask.agentsOf(customer,L) & served(SL) & .difference(L,SL,CL) & .length(CL,N) & N
== 0
<-
```

```
        !goto_center
        message("Waiting for customers...");
        .wait(500);
        !getNextCustomer
.
```

The following reactive plan is activated when the drone is flying to the center of the shop and a new customer enters the shop.

```
+!goto_center: ask.agentsOf(customer,L) & served(SL) & .difference(L,SL,CL) & .length(CL,N) & N >0
<-
        .print("Going to customer");
.
```

As long there are no new customers to serve and the drone is not in the center of the store, the drone flies towards the center. This plan is executed repeatedly.

```
+!goto_center:gridSize(G) & not patch(30,30) & simSpeed(S)
<-
        moveTowardsPatch(30,30);
        tick;
        .wait(S);
        !goto_center;
.
```

The following reactive plan is activated when the drone is in the center of the shop.

```
+!goto_center:gridSize(G) & patch(30,30)
<-
        .print("Went to center");
.
```

The following reactive plan is activated when the drone has select a customer to serve. Flies to customer's position and asks the order from the customer.

```
+!serveCustomer:nowServing(C,X,Y)
<-
        //Go to customer position
        !gotoPosition(X,Y);
        ?pos(MX,MY);
        agent.pos(C,MX,MY);
        //Get customer order
        .send(C,achieve,order);
.
```

This plan flies the drone to a specific location. It is executed repeatedly as long as the drone does not have reached the location.

```
+!gotoPosition(X,Y): not pos(X,Y) & simSpeed(S)
<-
        moveTowardsPos(X,Y);
        tick;
        .wait(S);
        !gotoPosition(X,Y);
.
```

The following reactive plan is activated when the drone has reach the specific location.

```
+!gotoPosition(X,Y):pos(X,Y)
<-
	.print("Went to:(",X,",",Y,")");
.
```

This plan is activated when a customer has send the order to the drone. When the order is received, the drone starts to collect the order.

```
+!customerOrder(L)[source(S)]:.length(L,N) & N >0
<-
	.print("Customer:",S," order:",L);
	message("Collecting order.");
	-+orderToCollect(L);
	!collectOrder;
.
```

This plan is activated when a customer has send an empty order.

```
+!customerOrder(L)[source(S)]:.length(L,0)
<-
	.print("Customer:",S," didn't wont anyting");
.
```

The following plan collects the order of a customer. The drone picks an item from the order lists and collects it. This plan is executed as long as there are items in the order list.

```
+!collectOrder: orderToCollect(L) & .length(L,N) & N >0 & .member(M,L) & bench(M,B) &
agent.pos(B,X,Y)
<-
	//Go to machine
	!gotoPosition(X,Y);
	//Get product of the machine
	.send(B,achieve,requestProduct);
.
```

The following reactive plan is activated when there are no other items in the order list to collect. The drone flies to the customer to serve the order.

```
+!collectOrder: orderToCollect(L) & .length(L,N) & N == 0 & nowServing(C,X,Y) & served(SL)
<-
	.print("Order collected");
	message("Serve order");
	!gotoPosition(X,Y);
	.send(C,achieve,eat);
	.concat([C],SL,NL);
	-+served(NL);
	-nowServing(C,X,Y);
	!getNextCustomer;
.
```

This plan is activated when the requested product from a machine is unavailable. The drone waits a little time and tries again to collect the order.

```
+!notAvaible(N)[source(S)]
<-
	.print("Product not");
	.wait(500);
	!collectOrder
.
```

This plan is activated when the machine provides the requested product to the drone.

```
+!pickProduct(N)[source(S)]:orderToCollect(L)
<-
	.delete(N,L,NL);
	-+orderToCollect(NL);
	!collectOrder;
.
```

The simulation ends when the max number of customers have enter the shop and all have served and left the store.

[observer] Shop closed!

5.4 Tic-Tac-Toe

In this example we are demonstrating how the user can interact with the grid during the execution of the simulation by implemented a famous game **Tic-tac-toe**. To model the specific simulation an AI bot (agent) is playing against the user.

The required entities are:

- A board
- A bot

The observer will create the game board and will be the AI bot which it is going to play against the user.

tic-tac-toe.asl

Observer beliefs:

- play(false) Indicates if the game is on.

The game board will be a 3x3 grid.

```
+!createArtifacts:true
```

```
<-
        ...
        joinWorkspace("jlogo",_);

        makeArtifact(world,"jasonlogo.artifact.WorldArtifact",["Simulation",600,200],W);
        focus(W);

        ...
.
```

The specific plan activates when the user hits the start button. Adds as beliefs all the available locations of the grid which an observer can play. Also updates the play which indicates that the game has started.

```
+start:ready(true) & gridSize(G)
<-
        enableGrid;
        for(.range(I,0,G-1)) {
                for(.range(J,0,G-1)) {
                        +available(I,J);
                }
        }
        -+play(true);
.
```

The operation enableGrid of WorldArtifact will display the grid lines in the simulation view.

The patchClicked reactive plan is a signal, it is activated when the user clicks in a cell (patch) of the grid and the play belief is false.

```
+patchClicked(X,Y): play(false)
<-
        showMessageDialog("Hit start button");
.
```

The patchClicked reactive plan is a signal is activated when a user clicks in a cell (patch) of the grid, the game is on and the specific clicked location is available for select.

```
+patchClicked(X,Y): available(X,Y) & play(true)
<-
        +position(X,Y,"X");
        -available(X,Y);
        setPatchColor(X,Y,red);
        setPatchText(X,Y,"X");
        !haveWin("X");
        !play;
.
```

The patchClicked reactive plan is a signal is activated when a user clicks in a cell (patch) of the grid, the game is on and the specific clicked location is not available for select [Figure 16].

```
+patchClicked(X,Y): not available(X,Y) & play(true)
<-
	showMessageDialog("X:",X," - Y:",Y," is already marked!");
.
```

The operation showMessageDialog of WorldArtiact displays a java swing dialog to the GUI view of JLogo.

The haveWin plan it is executed when there is a win combination relative to the players marked cells [Figure 17].

```
+!haveWin(T): play(true) &
(.findall(_,position(0,Y,T),L) & .length(L,3)) |
(.findall(_,position(1,Y,T),LL) & .length(LL,3)) |
(.findall(_,position(2,Y,T),LLL) & .length(LLL,3)) |
(.findall(_,position(X,0,T),IL) & .length(IL,3)) |
(.findall(_,position(X,1,T),IIL) & .length(IIL,3)) |
(.findall(_,position(X,2,T),III) & .length(III,3)) |
(.findall(_,position(X,X,T),DDD) & .length(DDD,3)) |
(.findall(_,(position(0,2,T) | position(1,1,T) | position(2,0,T)),DDDD) & .length(DDDD,3))
<-
	showMessageDialog(T," has won!");
	-+play(false);
.
```

If haveWin plan it is failed means that there is not winner yet.

```
-!haveWin(T):play(true)
<-
	.print("No winner yet");
.
```

The reactive play plan is the AI mind of selecting a cell to be played and marked. In the current implementation the agent selects one of the available grid locations.

```
+!play:available(X,Y) & play(true)
<-
	.print(L);
	.wait(300);
	+position(X,Y,"O");
	-available(X,Y);
	setPatchColor(X,Y,blue);
	setPatchText(X,Y,"O");
	!haveWin("O");
.
```

If there are not available locations to be played by the AI means that the game is draw.

```
+!play: not available(X,Y) & play(true)
```

<-
showMessageDialog("Draw");
.

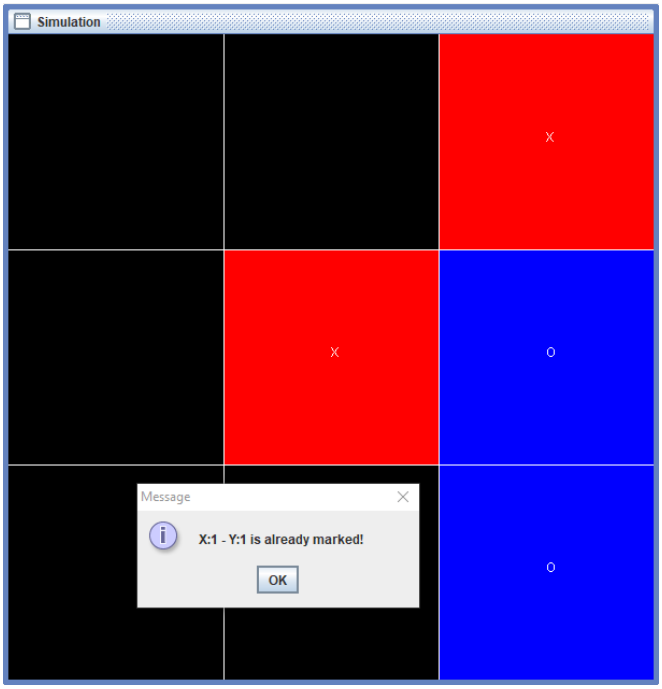

Figure 16: Tic Tac Toe board

Figure 17: Player won

6. Conclusion and Discussion

As far as we known, JLogo is the first attempt to provide a *configurable* graphical environment and an infrastructure for creating 2D graphical simulations using JaCaMo [3] platform. As an infrastructure provides a capable set of instructions for programming a 2D graphical environment as we have demonstrate on model examples. However, as a configurable environment it is not possible to achieve it without the use of Java.

On the other hand, as a template project JLogo can easily edited as all the source code it is available in the project depending the developer's needs. In addition the use of the JLogo is not strict meaning that the developers can write their own artifacts and internal actions.

In the current state JLogo, is recommended to be used only for education purposes as it is not very stable yet. It will help students who first come into contact with the field of MAS using Jason as programming language to create their own models and display the simulation into a 2D graphical environment for better understanding the results.

6.1 Limitations

Although JLogo does not have any restriction on how many agents (motion and stationary) can operate on the same time, there are a performance issues on simulations with hundreds of agents. There are two major reasons for this. The first is the concurrent operation of a shared artifact by multiple agents. Sometimes Cartago [4] struggles to make the artifact available when a vast number of multiple agents request to focus it and operate it. The second reason is the concurrent modification of the model by the agents with the use of the Artifacts and the internal actions. The latter occurs more rarely.

Another limitation of the JLogo is the configuration of the simulation GUI. In this state JLogo is not very flexible on adding and removing GUI components such as buttons which could help the users to interact with the simulation on demand. To achieve that, the use of Java code is required. Furthermore the customization of the

environment is limited in the configuration of world size, patch size and the speed of the tick step.

6.2 Future extensions

In future updates JLogo should become more stable platform especially on simulations with hundreds of agents operating concurrently and extend the current set of instructions will help developers to modeling more complex and detailed simulations.

In current state JLogo is a JaCaMo [3] project developed under the Jacamoide [18]. This limits the JLogo as a configurable graphical environment for developing graphical simulations. In future updates JLogo should became a separated IDE by creating a new plugin for the Eclipse [19] or a complete new platform which will provide a more flexible environment, for editing the GUI of the JLogo simulations by adding UI components such as buttons, sliders etc. with drag and drop operations for ease of use. Also providing more options for displaying the results such as plots will help the users on better understanding the results of their simulations.

Another helpful addition would be to add a command line in the GUI providing the users the ability to pass execute commands to the observer agent like NetLogo [1].

References

1. NetLogo: Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.
2. Bordini, R. H., Hübner, J. F., and Wooldridge, M. 2007. Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons, Ltd.
3. Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci, Andrea Santi ,Multi-agent oriented programming with JaCaMo, Science of Computer Programming, 2013.
4. Alessandro Ricci, Mirko Viroli, Andrea Omicini. “Give Agents their Artifacts”: The A&A Approach for Engineering Working Environments in MAS. 6th International Joint Conference “Autonomous Agents & Multi-Agent Systems” (AAMAS 2007), 14-18 May 2007
5. Wilensky U., and William Rand W. An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo, MIT Press, 2015.
6. Anand S. Rao, 1996. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. Proceedings of Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96).
7. Jomi F. Hübner, Olivier Boissier, Rosine Kitio, Alessandro Ricci , Instrumenting multi-agent organisations with organisational artifacts and agents, Autonomous Agents and Multi-Agent Systems, May 2010, Volume 20, Issue 3, pp 369-400
8. Singh, D., Padgham, L., Logan, B. (2016). Integrating BDI agents with agent-based simulation platforms In: Autonomous Agents and Multi-Agent Systems,30,1050 – 1071
9. W. A. L. Ramirez and M. Fasli, "Integrating NetLogo and Jason: A disaster-rescue simulation," 2017 9th Computer Science and Electronic Engineering (CEEC), Colchester, 2017, pp. 213-218.

Internet sources

10. The Jacamo Project http://jacamo.sourceforge.net/

11. The NetLogo Platform: https://ccl.northwestern.edu/netlogo/

12. The Jason platform: http://jason.sourceforge.net/

13. The Cartago platform: http://cartago.sourceforge.net/

14. Siri wiki: https://en.wikipedia.org/wiki/Siri/

15. Google assistant: https://assistant.google.com/

16. Cortana wiki: https://en.wikipedia.org/wiki/Cortana/

17. Alexa wiki: https://en.wikipedia.org/wiki/Amazon_Alexa/

18. Jacamoide: http://jacamo.sourceforge.net/eclipseplugin/tutorial/

19. Eclipse IDE: http://www.eclipse.org/