



Guide for creating a model-family for MASS/PET

PET model writing manual and tutorial

*Created by Attila Korompai, Vilmos Kozma and Marton David
Ivanyi*

For AITIA International and ELTE-IKKK



2007 October

Contents

1	Introduction	3
1.1	Prerequisites	4
2	The structure of a PET model-family	5
2.1	Agents	5
2.1.1	Agent properties	5
2.1.2	Visitor classes	6
2.2	Visualizations	6
2.2.1	Applet type visualizations	6
2.2.1.1	Implementing an object type visualization applet	7
2.2.1.2	Implementing image type visualization through applet	7
2.2.2	Image type visualizations	8
2.2.3	Link type visualizations	8
2.3	The model-family descriptor xml	8
3	Tutorial: Creating the Hunter game	9
3.1	Introduction of the Hunter game	9
3.2	Software requirements	9
3.3	Creating the model-family	9
3.3.1	Creating the skeleton of the descriptor	9
3.3.2	Creating the space agent	10
3.3.3	Creating the CreateSpaceVisitor class	11
3.3.4	Mapping the space agent	12
3.3.5	Creating the Runner agent	13
3.3.6	Mapping the Runner agent	16
3.3.7	Creating a visualization applet	17
3.3.8	Adding detector functions to agents	18
3.3.9	Mapping the detectors in the model-family descriptor	19
3.3.10	Creating actions	21
3.4	Installing the model-family	23
4	Conclusions	24
5	Appendix A: The model-family descriptor xml	25
5.1	The structure of the descriptor	25
5.1.1	The agent element	25
5.1.1.1	The field element	26
5.1.1.2	The visualization element	26

1 Introduction

1.1 Agent-Based Modeling

Agent-based modeling is a branch of computer simulation. It models the individual, together with its imperfections (e.g., limited cognitive or computational abilities), its idiosyncrasies and personal interactions. The approach builds the model from 'the bottom-up', focusing mostly on micro rules and seeking to understand the emergence of macro behavior. Participatory simulation - a branch of agent-based simulation - is a methodology building on the synergy of human actors and artificial agents, excelling in the training and decision-making support areas. In participatory simulations some agents are controlled by users, while others are software governed.

1.2 MASS

The Multi-Agent Simulation Suite (MASS) is a software package intended to enable modelers to utilize the tools of agent-based simulation in various fields, without having to develop heavy programming skills.

MASS consists of four applications built around a simulation core. The simulation suite has its own core called the Multi-Agent Core (MAC), but it is also able to run on the popular Repast core. Being multi-core enables modelers to verify that results are core-independent, thus we plan to further develop this option. The Functional Agent-Based Language for Simulation (FABLES) is a programming language and its integrated modelling environment specially designed for creating agent-based simulations. The Model Exploration Module (MEME) is a tool that enables orchestrating experiments, managing results and has support for their analysis. The Participatory Extension (PET) is an optional web-based environment for multi-agent and participatory simulations. The fourth element of MASS, the Visualization Package does not translate into a standalone application. It consists of the various implementations of charts and visualizations used in all the other software.

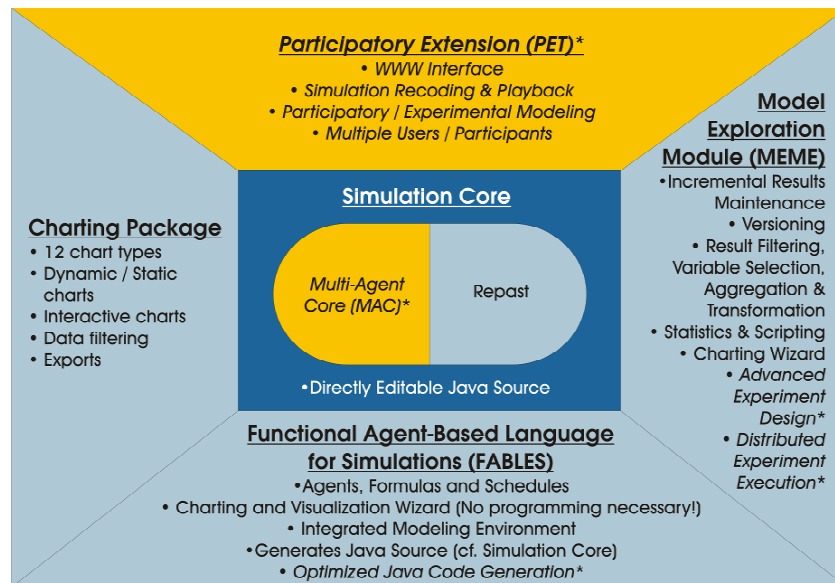


Figure 1 - Multi-Agent Simulation Suite

1.3 Purpose

The goal of this document is to give help for modelers to easily create and install model-families for the Participatory Extension (PET). PET is part of the Multi-Agent Simulation Suite (MASS) and it is a web based multi-user simulation environment, capable of

participatory simulation. It is based either on its native core, the Multi-Agent Core (MAC) or on Repast. The paper goes through the steps of the development of a PET model-family discussing the important issues and highlighting the possibilities of model writers.

1.4 Prerequisites

The reader of this document is expected to have the following abilities:

- To write and to understand Java code at a basic level,
- Administrator level knowledge of PET is an advantage (see the relevant MASS/PET documentation for the details).

2 The structure of a PET model-family

2.1 Agents

The key parts of PET simulations are the agents. The most important rule for PET agents is that it must be inherited from the class `ai.aitia.mass.base.Agent`. An agent consists of three logical parts:

- **Properties and their accessor methods:** properties are responsible to reflect the agent's actual state. An agent can have built-in and custom properties.
- **Actions:** methods that returns `void` and get no parameters. These methods can be called by either the scheduler or the human controller of the agent through the web-interface.
- **Detectors:** methods returning arbitrary object and getting no parameters. The purpose of these methods is to provide information from themselves and the surrounding environment. This information can be used by the visualizations.

2.1.1 Agent properties

An agent has two kinds of properties:

- **Built-in properties:** In PET all agents must inherit from the class `ai.aitia.mass.base.Agent`. As a result every agent has the same set of built in properties described by the following list. (Note that in normal case a model writer should never deal with the built in properties)
 - `id`: a unique identifier of the agent
 - `type`: the type of the agent
 - `controllable`: a Boolean property indicating whether the agent is controllable or not
 - `visible`: a Boolean property indicating whether the agent is visible or not
 - `visualization`: a string reference for the assigned visualization
 - `detectorsVisible`: indicates whether the detectors of the agent are visible on the user interface
 - `controlled`: a Boolean property indicating whether the agent is controlled or not
 - `defaultAction`: this object holds a callable reference to the agents default action method
 - `timeoutAction`: this object holds a callable reference to the agent's timeout action method. Timeout method calls are triggered by the agent's timeout event. A PET model can be configured to timeout a controlled agent which is not showing activity for a certain period of time. For details on timeout settings consult the PET Admin manual documentation.
 - `actions`: A list which contains all the action method of the agent. Action methods return `void` and get no parameters. These methods can be called by either the scheduler or the human controller of the agent through the web-interface.
 - `detectors`: A list which contains all the detector method of the agent. Detector methods return arbitrary objects and get no parameters. The purpose of these methods is to provide information from themselves and the surrounding environment.
 - `simulation`: the simulation object
- **Custom properties:** In PET every property of the agent which is added by the model writer is ranked as custom property. The accessibility of a custom property

can be defined with the help of the model-family descriptor file (see chapter 2.3 later). By omitting the setter method for a property it will be read-only.

To make a distinction between built-in and custom agents, all the properties and accessor (getters and setters) methods are suffixed with the `_Pet` token. A property of an agent becomes known for PET by mapping it in the model-family descriptor. There are four special built-in property which is virtually mapped in the descriptor. These are: `visible`, `controllable`, `visualization` and `detectorsVisible`. Mapped properties can be managed by the framework and displayed on the web-interface. A mapped property has the following attributes which is defined in the model-family descriptor:

- **Accessibility:** defines the access rule for this property. Possible values are:
 - `no`: the property is not accessible on the web interface
 - `r`: the property is read-only
 - `rw`: the property has read/write access
- **Default value:** the default value for the property
- **Display name:** the name of the property displayed on the web-interface
- **Class:** if the type of property is not primitive or `String` then it is necessary to define the class.
- **Short description:** a short description of the property which is displayed on the web-interface in a tool-tip.
- **Constant list:** it is possible to define a list of named constant values for the property. In this case, on the web-interface the possible values of the property are displayed in a drop-down list.
- **Initialized:** a Boolean attribute indicating whether the value of a certain property can be modified during the simulation. To modify a property the simulation must be in ready state.
- **Visibility:** a Boolean attribute indicating that the property is visible for the user.

2.1.2 Visitor classes

Agent objects can access all other agents with the help of the `IAgent[] getAgents_Pet()` function declared in the `Agent` class or with the help of the `visit()` method of `IAgentVisitor` interface. Some developer believes that the latter approach allows creating cleaner code.

```
public interface IAgentVisitor
{
    public void visit(IAgent agent);
}
```

You can easily perform some operation on each agents (visit the agents) by calling the `visit_Pet(IAgentVisitor visitor)` method of the `Agent` class. Pass an implementation of the `IAgentVisitor` interface to it as parameter.

2.2 Visualizations

PET supports the creation of visualizations for modelers. The next three chapters detail the possibilities.

2.2.1 Applet type visualizations

An applet-type visualization appears in an applet in the browser. The working mechanism is the following: while the simulation is running an applet class which is loaded in the web-browser periodically requests a servlet (a so called detector or image-producer servlet) which responses with a serialized object or with a byte stream of an image respectively. The retrieved object or image provides the necessary information to

draw/redraw the visualization on the surface of the applet. As the above text suggests there are two kinds of visualization applets:

- **Object type**: in this case the applet requests the detector servlet which responds with an object produced by calling one of the detector methods of the visualized agent. The name of the called detector method is included in the http request header. This object should hold all the information which is required to draw/redraw the actual state of the visualization from the point of view of the visualized agent. Note that objects returned by the detector method must implement the `java.io.Serializable` interface.
- **Image type**: in this case the applet calls PET's image-producer servlet which responds with a byte stream of an image. Then this image is drawn on the surface of the applet.

2.2.1.1 Implementing an object type visualization applet

To implement an object type visualization applet, the model writer has to create a class which is inherited from the class `ai.aitia.mass.web.applet.DetectorApplet`. This class is an applet and is doing everything that the system can automate. In fact the modeler only has to override the `paintObject(Graphics g, Object obj)`. Overriding the `init()` method is optional however can be useful to perform initialization related tasks. The first parameter of the `paintObject()` is the `Graphics` object on which the drawing is performed and the second is the object holding the information which is necessary for drawing.

2.2.1.2 Implementing image type visualization through applet

To implement image type visualization, rather than writing an applet the modeler has to create an image producer class. This class generates the image which is displayed in the detector applet or directly on the page (see chapter 2.2.2) automatically. The model-family descriptor holds the required information to prepare the applet. Implementing an image producer is not difficult because the model writer only has to write a class which implements the `ai.aitia.mass.visu.ImageProducer` interface. The life time of an image producer object is equal to the life time of the simulation. Here is the code of the interface:

```
public interface ImageProducer {

    public void init(Simulation sim, IAgent agent);

    public void drawImage(Graphics2D g2d);

    public void stop();

    public int getImageWidth();

    public int getImageHeight();

}
```

Let's examine the functions one-by-one:

- `init(Simulation sim, IAgent agent)`: this function initializes the image producer. Put all the initialization code here.
- `drawImage(Graphics2D g2d)`: draw the image here onto the graphics object that we got as parameter.
- `stop()`: This function is called by PET when the simulation is stopped.
- `getImageWidth()`: returns the width of the produced image

- `getImageHeight()`: returns the height of the produced image

2.2.2 Image type visualizations

This visualization type is appropriate when the modeler wants to display still image in the browser. This means that the visualization is a simple image in the web-browser and is static. The implementation has the same concept as it was written in the previous chapter. The only difference occurs in the implementation of the model-family descriptor.

2.2.3 Link type visualizations

If you choose to implement this visualization type you will be able to implement anything within the limitation of common web interface technologies like Flash, Applet, Ajax, etc. Link-type visualization is rendered in the web-browser as a link and it can target any pages on the internet. The targeted page may launch an applet, flash object or itself is an Ajax application and capable to call the detector functions of agents on the server side.

2.3 The model-family descriptor xml

With the help of a properly installed descriptor file the PET system is able to recognize and collect information on model-families that are installed in the system. Each model-family has its own descriptor file. The descriptor file holds all the necessary information that PET requires including: agent mappings, meta-information on properties, visualizations etc. For detailed description of the model-family descriptor read Appendix A.

3 Tutorial: Creating the Hunter game

3.1 Introduction of the Hunter game

In the Hunter game there are two kinds of agents. Some of them are 'hunters', while others are 'preys'. They are 'living' in a two-dimensional space. As the names imply, hunters try to catch preys, while preys try to avoid them. When a hunter catches a prey the prey disappears and reappears at a random place in the space.

3.2 Software requirements

- Installed JDK 5.0 or later
- Installed and working PET application

3.3 Creating the model-family

3.3.1 Creating the skeleton of the descriptor

In the first step create the skeleton of the model-family descriptor. Create a new file and name it to `hunter.xml`. Copy the following xml code into it:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE model-family SYSTEM "mac_model_family.dtd">

<model-family>

  <family-name>Hunter</family-name>
  <simulation-class>ai.aitia.mass.base.SimulatedRealTimeSimulation</simulation-
class>
  <image>../images/hunter/tiger.jpg</image>

  <description>Hunter simulation</description>

  <agents>
  </agents>
</model-family>
```

In the above xml we defined that the name of the model-family is **Hunter** and the image associated with the model-family is `tiger.jpg` located in PET application's `WEB-INF/./images/hunter` directory. The `tiger.jpg` image must be copied into its place. Copy it now. The short description of the model-family is **Hunter simulation**. This text appears as a tool-tip in the web interface. Yet the `agents` element is empty because we didn't create agents.

We have two kinds of agents:

- The "space" is represented by the `space` agent type in the simulation. We need only one instance from this type. It contains the positions of the other agent type instances: the Runners.
- The **Runner** agent type represents the hunters and preys. There are multiple instances of this type in the simulation.

3.3.2 Creating the space agent

Having a separate agent that represents space is quite useful. Although hunters and preys could contain all the data needed to perform their actions, a Space agent will be a good place to configure global variables like the dimensions of the space or the maximum number of agents in the space.

It is also a good place to gather information about all agents (to create a global view for example).

Another advantage is that you can define Space agent as a member variable (see later) of the Hunter and the Prey agents. The framework supports this kind of hierarchical dependency (Hunter & Prey cannot be instantiated unless a Space has been associated with them), this provides a good way to ensure that a specific Agent exists without writing any code!

```
public class Space extends Agent
{
    public static final int EMPTY = 0;
    public static final int HUNTER = 1;
    public static final int PREY = 2;

    private int height;
    private int width;
    private int[][] space;

    @Override
    public void init_Pet() {
        space = new int[width][height];
    }

    @Override
    public void onStep_Pet() throws StepFailedException {
        space = new int[width][height];
        CreateSpaceVisitor visitor = new CreateSpaceVisitor(space);
        visit_Pet(visitor);
    }

    @Override
    public void onTimeout_Pet() {
    }

    @Override
    public void onTake_Pet() {
    }

    @Override
    public void onRelease_Pet() {
    }
}
```

```
@Override
public void onRemove_Pet() {
}

public int getWidth() {
    return width;
}

public void setWidth(int width) {
    this.width = width;
}

public int getHeight() {
    return height;
}

public void setHeight(int height) {
    this.height = height;
}
}
```

In the Agent class the abstract methods are as follows: `init_Pet()`, `onStep_Pet()`, `onTimeout_Pet()`, `onTake_Pet()`, `onRelease_Pet()`, `onRemove_Pet()`. Valid agents must give an implementation for each of the above methods. In our case we write code only into the `init_Pet()`, `onStep_Pet()` methods. As you can see in the code above the `onStep_Pet()` method uses a `CreateSpaceVisitor` class. The next chapter introduces this class.

3.3.3 Creating the CreateSpaceVisitor class

This class implements the `IAgentVisitor` interface. It is used by the space agent to extract the coordinate information from runner agents. You can read chapter 2.1.2 as reminder.

```
public class CreateSpaceVisitor implements IAgentVisitor
{
    int[][] space;

    public CreateSpaceVisitor(int[][] space)
    {
        this.space = space;
    }

    public void visit(IAgent agent) {
        if (agent instanceof Runner) {
            Runner runner = (Runner)agent;

```

```

        boolean hunter = runner.isType();
        int x = runner.getX();
        int y = runner.getY();

        if (hunter) {
            space[x][y] = Space.HUNTER;
        } else if (space[x][y] != Space.HUNTER) {
            space[x][y] = Space.PREY;
        }
    }
}
}

```

3.3.4 Mapping the space agent

With the following descriptor fragment we declare that our space agent has 2 properties (height, width) that need to be managed by the framework. It also provides a display name for them with the `<displayName>` element. However this is not compulsory, if not specified then the displayed name is the field name. An image is assigned to the agent as well. Copy the image `jungle.jpg` into the `WEB-INF/./images/hunter` directory like you did it with the image assigned to the model-family.

```

<agent name="Space" class="ai.aitia.mass.demos.hunter.Space">
  <fields>
    <field name="height" access="no">
      <default>300</default>
      <displayName>Space height</displayName>
      <shortDescription>Height of the Space</shortDescription>
    </field>
    <field name="width" access="no">
      <default>300</default>
      <displayName>Space width</displayName>
      <shortDescription>Width of the Space</shortDescription>
    </field>
  </fields>
  <actions />
  <detectors />
  <image>../images/hunter/jungle.jpg</image>
  <visualizations/>
  <description>Space agent</description>
</agent>

```

As a result of the declaration, the space agent appears on the "Edit model" page in the list of agents. On the "Add new agent" page we can see the declared properties. See Figure 1 and Figure 2.

⌘ Agents

Type	Class	Number	Actions
Space agent	ai.aitia.mass.demos.hunter.Space	1	

Figure 1 Space agent on "Edit model" page

⌘ Add new agent - Space agent

Property	Class	Value	Visibility	Initialized
Controllable		<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Detectors visible		<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Space height	java.lang.Integer	300	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Space width	java.lang.Integer	300	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Visible	java.lang.Boolean	<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Visualization	java.lang.String	None	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false

Declared properties Add 1 piece(s)... Add

Figure 2 Declared properties of space agent

3.3.5 Creating the Runner agent

Since Hunter and Prey agents basically do the same thing (they identify the closest enemy and move) we need to write only one class. The value of a Boolean property of the Runner class named `type`, determine that a certain agent is hunter or prey.

```
public class Runner extends Agent
{
    private boolean type; // true - hunter ; false - predator
    private int x;
    private int y;
    private int skip;
    private Space space;

    @Override
    public void init_Pet() {
        newRandomPlace();
    }

    @Override
    public void onStep_Pet() throws StepFailedException {
        if (getTick_Pet() % (skip + 1) != 0) return;

        Runner runner = getClosest();

        if (runner != null) {
            int diff = 1;
            if (!type) diff = -1;
        }
    }
}
```

```
        int oldX = x;
        int oldY = y;

        int otherX = runner.getX();
        int otherY = runner.getY();

        if (type && !runner.type && x == otherX && y == otherY)
            runner.newRandomPlace();

        if (oldX > otherX)
            x = oldX - diff;
        else if (oldX < otherX) x = oldX + diff;

        if (oldY > otherY)
            y = oldY - diff;
        else if (oldY < otherY) y = oldY + diff;

        if (y < 0) y = 0;
        if (y >= space.getHeight()) y = space.getHeight() - 1;

        if (x < 0) x = 0;
        if (x >= space.getWidth()) x = space.getWidth() - 1;

        if (type && !runner.type && x == otherX && y == otherY)
            runner.newRandomPlace();
    }
}

private void newRandomPlace() {
    x = getRandom_Pet().nextInt(space.getWidth());
    y = getRandom_Pet().nextInt(space.getHeight());
}

@Override
public void onTimeout_Pet() {
}

@Override
public void onTake_Pet() {
}
```

```
@Override
public void onRelease_Pet() {
}

@Override
public void onRemove_Pet() {
}

private Runner getClosest() {
    ClosestEnemyVisitor visitor = new ClosestEnemyVisitor(this);
    visit_Pet(visitor);
    return visitor.getClosestEnemy();
}

public boolean isType() {
    return type;
}

public void setType(boolean type) {
    this.type = type;
}

public int getX() {
    return x;
}

public void setX(int x) {
    this.x = x;
}

public int getY() {
    return y;
}

public void setY(int y) {
    this.y = y;
}

public int getSkip() {
    return skip;
}

public void setSkip(int skip) {
```

```

        this.skip = skip;
    }

    public Space getSpace() {
        return space;
    }

    public void setSpace(Space space) {
        this.space = space;
    }
}

```

The `init_Pet()`, `onStep_Pet()`, `onTimeout_Pet()`, `onTake_Pet()`, `onRelease_Pet()` and `onRemove_Pet()` methods declared as abstract in the `Agent` class (from which we derive this class) so we must give an implementation for them. The `onStep_Pet()` is called by the framework in each turn of the simulation when the agent is not controlled. The `init_Pet()` method is called only once, when the simulation is initialized.

3.3.6 Mapping the Runner agent

Now we declare the `Runner` agent in the descriptor file:

```

<agent name="Runner" class="ai.aitia.mass.demos.hunter.Runner">
  <fields>
    <field name="type" access="r">
      <constants>
        <constant name="Hunter" value="true" />
        <constant name="Prey" value="false" />
      </constants>
    </field>
    <field name="x" access="r" />
    <field name="y" access="r" />
    <field name="skip" access="r" />
    <field name="space" access="no" />
  </fields>
  <actions/>
  <detectors/>
  <image>../images/hunter/paw.gif</image>
  <visualizations/>
  <description>Hunter and prey agents</description>
</agent>

```

In the above xml fragment we declare five fields and assign an image to this agent type. So copy the `paw.gif` image into the `WEB-INF/./images/hunter` directory. The display name for the agent is "Hunter and prey agents". We define constant values for the `type` property to transform its Boolean value to human readable.

The result can be seen on the "Edit model" and "Add new agent" pages as showed by Figure 3 and Figure 4.

Type	Class	Number	Actions
Space agent	ai.aitia.mass.demos.hunter.Space	0	[Icons]
Hunter and prey agents	ai.aitia.mass.demos.hunter.Runner	0	[Icons]

Figure 3 the Runner agent type

⌘ Add new agent - Hunter and prey agents

Property	Class	Value	Visibility	Initialized
Controllable	java.lang.Boolean	<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Detectors visible	java.lang.Boolean	<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
skip	java.lang.Integer	0	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
space	ai.aitia.mass.demos.hunter.Space	279	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
type	java.lang.Boolean	Prey	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Visible		<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Visualization		None	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
x	java.lang.Integer	0	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
y	java.lang.Integer	0	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false

Add piece(s)...

Figure 4 properties of the Runner agent

3.3.7 Creating a visualization applet

In this chapter we create an object type visualization applet. For technical details read chapter 2.2.1.

Create the `SpaceApplet` class:

```
public class SpaceApplet extends DetectorApplet
{
    private Graphics bufferGraphics;
    private Image offscreen;
    private int width;
    private int height;

    @Override
    public void init() {
        super.init();
        width = getWidth();
        height = getHeight();
        offscreen = createImage(width + 2, height + 2);
        bufferGraphics = offscreen.getGraphics();
    }

    @Override
    public void paintObject(Graphics g, Object obj) {
```

```

Object[] data = (Object[])obj;
int runnerX = ((int[])data[0])[0];
int runnerY = ((int[])data[0])[1];

int[][] map = (int[][])data[1];

bufferGraphics.setColor(Color.BLACK);
bufferGraphics.fillRect(0, 0, width + 2, height + 2);

for (int x = 0; x < map.length; x++) {
    int[] column = map[x];

    for (int y = 0; y < column.length; y++) {
        switch (map[x][y]) {
            case Space.HUNTER:
                bufferGraphics.setColor(Color.RED);
                bufferGraphics.fillRect(x, y, 3, 3);
                break;
            case Space.PREY:
                bufferGraphics.setColor(Color.WHITE);
                bufferGraphics.fillRect(x, y, 3, 3);
                break;
        }
    }
}
if (runnerX > 0 && runnerY > 0) {
    bufferGraphics.setColor(Color.YELLOW);
    bufferGraphics.fillRect(runnerX, runnerY, 3, 3);
}
g.drawImage(offscreen, 0, 0, this);
}
}

```

This class is derived from `DetectorApplet` class so the communication part is done by the ancestor and we only need to concentrate on drawing. This is done in the `paintObject(Graphics g, Object obj)` method and is based on the data contained by the `obj` parameter. This object is returned by detector functions of our agents. So we need to add a detector function to our agent classes.

3.3.8 Adding detector functions to agents

Add the following code fragment to the `Space` agent:

```

public Object[] getSpaceAndID() {
    Object[] data = new Object[2];
    data[0] = new int[] { -1, -1 };
    data[1] = space;
    return data;
}

```

```
}

```

And the following to `Runner` agent:

```
public Object getSpaceAndID() {
    Object[] data = space.getSpaceAndID();
    data[0] = new int[] { x, y };
    return data;
}
```

These are the detector function. As we discussed in earlier chapters, detector functions are typically called by applets through the network and applets draw the visualization based on the returned data. Note that we will create only one visualization applet for both `Space` and `Runner` agent. This enforces the design of the returned data structure to be common between `Space` and `Runner` agents. In the above code snippets the return value is a one dimensional array with two elements. The first element holds the coordinates of the visualized agent. In the case of `space` class these are `{-1,-1}` indicating that the value is returned by the `space` agent so no `Runner` object is highlighted. In the second case, the coordinates of the `Runner` agent is returned. In this case the applet highlights this agent to indicate which `Runner` is visualized. The second element of the returned array is the `space` map (the `space` object). The `space` agent's `space` object is a two dimensional integer array sized for the width and height of the `space` holding the actual coordinates of `Runners`.

3.3.9 Mapping the detectors in the model-family descriptor

Replace the line:

```
<detectors/>
```

with the following xml fragment in `hunter.xml` in both the `Space` agent and `Runner` agent section:

```
<detectors>
  <detector name="getSpaceAndID" />
</detectors>
```

The above change makes the detector functions (of both `Space` and `Runner`) manageable for PET.

Replace the line:

```
<visualizations/>
```

with the following xml fragment in `space` agent's section:

```
<visualizations>
  <visualization codebase="../applet" archive="sim.jar"
    code="ai.aitia.mass.demos.hunter.applet.SpaceApplet">
    <type>applet</type>
    <name>HunterSpace</name>
    <display-name>Hunter Space</display-name>
    <detector-name>getSpaceAndID</detector-name>
    <action-frame>false</action-frame>
    <width type="property">width</width>
    <height type="property">length</height>
    <paint-mode>paint_object</paint-mode>
    <refresh-rate>200</refresh-rate>
```

```
</visualization>
</visualizations>
```

In the **Runner** agent’s section replace the line:

```
<visualizations/>
```

with the following xml fragment:

```
<visualizations>
  <visualization codebase="../applet" archive="sim.jar"
    code="ai.aitia.mass.demos.hunter.applet.SpaceApplet">
    <type>applet</type>
    <name>RunnerVisualization</name>
    <display-name>Hunter Space for Runner</display-name>
    <detector-name>getSpaceAndID</detector-name>
    <action-frame>true</action-frame>
    <width type="property">width</width>
    <height type="property">length</height>
    <paint-mode>paint_object</paint-mode>
    <refresh-rate>200</refresh-rate>
  </visualization>
</visualizations>
```

With the above fragments we declare visualizations for both **Space** and **Runner** agents. Now we can assign these visualizations to our agents the admin side. See Figure 5 which shows the “Add new agent” page.

Models | Simulations | User management | User Interface | Logout US

↳ Edit model (Don't Be a Prey!) | List agents | Groups | **New agent (Type: Hunter and prey agents)**

⚙ Add new agent - Hunter and prey agents

Property	Class	Value	Visibility	Initialized
Controllable	java.lang.Boolean	<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Detectors visible	java.lang.Boolean	<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
skip	java.lang.Integer	<input type="text" value="0"/>	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
space	ai.aitia.mass.demos.hunter.applet.Space	<input type="text" value="279"/>	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
type		Prey	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Visible	java.lang.Boolean	<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Visualization	java.lang.String	Hunter Space for Runner	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
x	java.lang.Integer	<input type="text" value="0"/>	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
y	java.lang.Integer	<input type="text" value="0"/>	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false

Add piece(s)...

Figure 5 Selecting visualization

After setting the visualization we can build and run a simulation. See Figure 6.

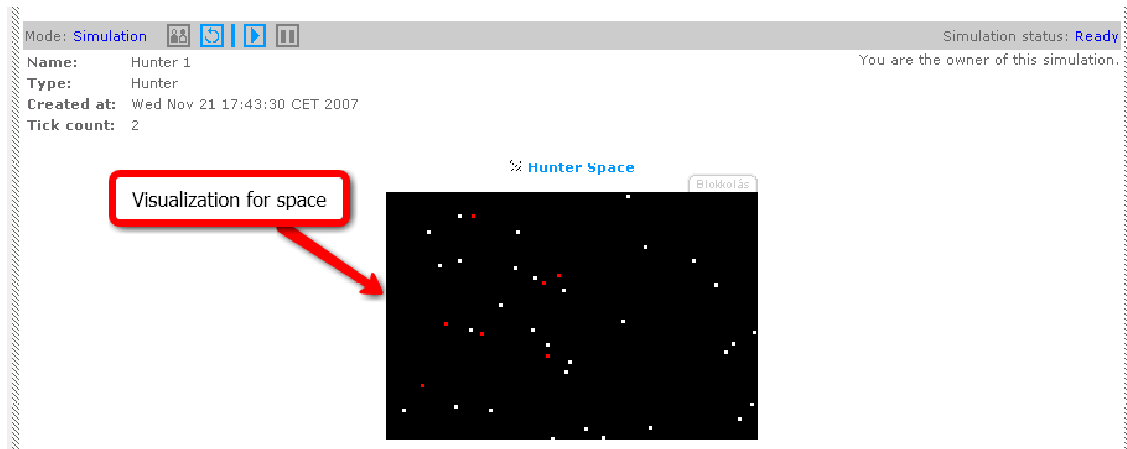


Figure 6 Visualization for Space agent

3.3.10 Creating actions

The PET framework allows the user to take the control over agents provided that they are marked as controllable. Marking an agent controllable is supported automatically by the framework. However to really perform operations on agents through the web-interface, action methods must be added to our agent classes and they must be declared as an action in the descriptor file.

Extend the `Runner` class with the following methods:

```
private void move(int diffX, int diffY) {
    int newX = x + diffX;
    int newY = y + diffY;

    if (newX < 0) newX = 0;
    if (newX > space.getWidth() - 1) newX = space.getWidth() - 1;
    if (newY < 0) newY = 0;
    if (newY > space.getHeight() - 1) newY = space.getHeight() - 1;

    x = newX;
    y = newY;
}

public void moveLeft() {
    move(-1, 0);
}

public void moveLeftUp() {
    move(-1, -1);
}

public void moveUp() {
    move(0, -1);
}
```

```
public void moveUpRight() {
    move(1, -1);
}

public void moveRight() {
    move(1, 0);
}

public void moveRightDown() {
    move(1, 1);
}

public void moveDown() {
    move(0, 1);
}

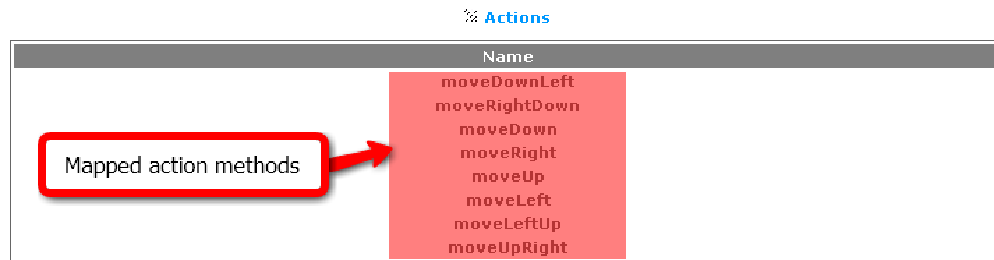
public void moveDownLeft() {
    move(-1, 1);
}
```

The `move()` method does the actual move on the agent. This method is called by the `moveXYZ()` methods, the action methods. Each action method moves the agent to a specific direction by one coordinate.

Now map the action methods in the descriptor:

```
<agent name="Runner" class="ai.aitia.mass.demos.hunter.Runner">
    ...
    <actions>
        <action name="moveLeft" />
        <action name="moveLeftUp" />
        <action name="moveUp" />
        <action name="moveUpRight" />
        <action name="moveRight" />
        <action name="moveRightDown" />
        <action name="moveDown" />
        <action name="moveDownLeft" />
    </actions>
    ...
</agent>
```

As a result, on the agent controlling page we see a box enlisting the mapped action methods. See Figure 7. By clicking on the rendered links the controlled agent will step one in the certain direction.

Figure 7 Control box for `Runner` agent

3.4 Installing the model-family

- Place the model family descriptor (`hunter.xml`) in the `WEB-INF/config/descriptors/mac` directory.
- Compile all the java code and create a jar file from it. Copy the jar file in the `WEB-INF/lib` directory.
- Create a Jar file which is downloaded by the browser to load the visualization applet. Include only the `SpaceApplet`, `DetectorApplet` and `Space` classes only. Copy the newly created file into the PET application's `WEB-INF/./applet` folder.

Finally you must restart the PET application. Restart either the whole application server or just the PET application. This is done according to the used Java Servlet Container in which PET runs. If you installed the system from the windows installation distribution then you can do it by opening the **Start Menu -> Control Panel -> Administrative Tools -> Services** panel. Here, right click on the **Apache Tomcat** entry and select **restart**.

Now you can write your own model-family.

4 Conclusions

PET is an agent based modeling framework providing infrastructure for modelers to develop participatory simulations and run them in web-based environment. This document goes through the steps of writing a native PET model-family, explains the related concepts and working mechanisms. The paper also contains a tutorial helping modelers to easily create their first PET model-family.

5 Appendix A: The model-family descriptor xml

With the help of a properly installed descriptor file the PET system is able to recognize and collect information on MAC model-families that are installed in the system. Each model-family has its own descriptor file. The general requirements of the descriptor are listed below:

- The name part of the file is arbitrary but the extension must be "xml".
- The installed file must be placed in the `WEB-INF/descriptors/mac/` directory of the application PET.
- The content of the file must follow the rules defined in the `WEB-INF/descriptors/mac/mac_model_family.dtd` file.

Now let's examine the structure which is defined by the above mentioned XML Document Type Definition file.

5.1 The structure of the descriptor

The name of root element is `model-family`. Its child elements are:

- `family-name`: the name of the model-family
- `simulation-class`: the class name of the used simulation type. Currently the system supports three types:
 - `ai.aitia.mass.base.SimulatedRealTimeSimulation`
 - `ai.aitia.mass.base.GameSimulation`
 - `ai.aitia.mass.base.RealTimeSimulation`
- `image`: path to the image that is assigned to the simulation model. The path is relative from the `WEB-INF` directory
- `description`: the description of the model-family
- `agents`: contains a list of `agent` elements. The `agent` element is discussed deeply in Appendix A.

5.1.1 The agent element

Each `agent` element defines an agent type. The attributes of the element are:

- `class`: The name of class of the agent
- `name`: The reference name of this agent type

The child elements of the `agent` element are:

- `fields`: Contains a list of `field` elements. Each `field` element describes a property of the given agent type. The next chapter discusses the `field` element in detail.
- `actions`: Contains a list of `action` elements. An `action` element refers to a method of the agent. An `action` element has empty body and a single `name` attribute. The referred method must have `void` return value and must get no parameters. These methods called by the user through the web interface.
- `detectors`: Contains a list of `detector` elements. A `detector` element has empty body and a single `name` attribute. The name must be a function name of the agent. The function must have `not void` return value and must get no parameters. Detector methods are used by agents to collect information from other agents. They are typically used by visualizations.

- **image**: The path of image associated to this agent. The path is relative to the **WEB-INF** directory.
- **visualizations**: Enumerates the visualization definitions associated to this agent. This element contains a list of **visualization** elements. For more information on this element see Appendix A.
- **description**: The display name of the agent type.

5.1.1.1 The field element

Each **field** element describes an agent property. It has two attributes:

- **name**: The name of the property
- **access**: Defines the permissions on this property. Possible values and their meanings are:
 - **r**: The property is read only
 - **rw**: The property has read/write access
 - **no**: The property is not visible for any agents of this type
- **class**: This attribute is optional. Defines the class of the property.

The **field** element has the following child elements:

- **displayName**: The display name of this property
- **shortDescription**: A short description for this property
- **constants**: Contains a list of **constant** elements. A **constant** element defines a named value which is useful when we want to display human readable text instead of numbers or text tokens. A good example is when a property has 3 different states and the states are expressed with numbers. For example: 0=cold, 1=tepid, 2=hot. The element has two attributes:
 - **name**: The displayed text which should be human readable.
 - **value**: The assigned value.

5.1.1.2 The visualization element

The **visualization** element has four attributes:

- **codebase**: Defines the path to the jar file that contains the visualization applet. Applies only for applet type visualization (see later).
- **archive**: The name of the jar file that contains the visualization applet. Applies only for applet type visualization (see later).
- **code**: The name of applet class. Applies only for applet type visualization (see later).
- **href**: The href of the image-producer servlet. Applies only for link type visualization (see later).

The child elements are the following:

- **type**: defines the type of the visualization. There are three visualization types:
 - **applet**: The visualization is produced by an applet written by the model writer. The applet class must be inherited from the class `ai.aitia.mass.web.applet.DetectorApplet`
 - **link**: In this case instead of a visualization a link is rendered on the screen and its target is the real visualization. By clicking this link the a new window should popup with the visualization.
 - **image**: In this case the visualization is a still image.
- **name**: the reference name of the visualization
- **display-name**: The display name of the visualization

- **detector-name**: The name of the detector method. This element is required only when the value of the **type** element is **applet** and the value of **paint-mode** (see below) is **paint_object**.
- **image-name**: A unique id of the image. This element is required when the value of the **type** element is **applet** and the value of **paint-mode** (see below) is **paint_image**.
- **width**: The width of the applet. Required when the visualization **type** is **applet**.
- **height**: The height of the applet. Required when the visualization **type** is **applet**.
- **paint-mode**: Defines the working mechanism of the referenced applet. Required when the visualization **type** is **applet**. There are two possible values:
 - **paint_object**: in this case the applet calls the detector method and refreshes its area upon the information that the retrieved object holds.
 - **paint_image**: in this case the applet calls the image producer servlet. The image producer servlet is responsible to produce the image of the visualization with the help of an image producer class. The image producer class is coded by the model writer and must implement the `ai.aitia.mass.visu.ImageProducer` interface. (See below)
- **refresh-rate**: The refresh rate of the visualization applet. Required when the visualization **type** is **applet**.
- **imageproducer-class-name**: The name of class which implements the `ai.aitia.mass.visu.ImageProducer` interface.