SPARK-PL: Agents and Spaces

Alexey Solovyev

Abstract

The main concepts of SPARK/SPARK-PL are introduced in this tutorial.

Table of Contents

1. SPARK Agents and Spaces

Agents in SPARK are any classes derived from the base Agent class. This base Agent class has a very limited functionality by itself. It has three main functions: creating agents, processing agents, and destroying agents.

A space in SPARK is an environment where agents exist and interact with each other. Basic agents cannot be associated with a space. There is another type of agents called space agents which can move inside a space and interact with other agents inside that space. In fact, there can be several spaces in the same model and different space agents (SpaceAgent class) can be located in different spaces. But in the most cases there is only one space which will be often called the default space. More about multiple spaces can be read in another tutorial.

2. Agents in SPARK-PL

Agents in SPARK-PL are declared using the 'agent' keyword. Each agent should be derived from either 'Agent' type or 'SpaceAgent' type (in fact, 'SpaceAgent' is derived from 'Agent'). Agents by themselves are also types (classes) in SPARK-PL.

```
; Declares a new agent type derived from SpaceAgent
agent Agent1 : SpaceAgent

; Declares a new agent type derived from Agent
agent Agent2 : Agent

; Declares a new agent type derived from Agent1
agent Agent3 : Agent1
```

As all types in SPARK-PL, each agent can have a constructor declared as a method with the name 'create'. A constructor is called every time a new agent is created. It is not possible to call a constructor for already existing object. It is often convenient to set up some internal agent's variable inside this constructor (it is especially useful for space agents which can have distinct colors, positions, shapes, and sizes).

```
agent Agent1 : SpaceAgent

var age : number

to create
 age = random 100 + 20
end
```

Behavior of agents is specified in one method called 'step'. This method is called every simulation step for all agents of all types. This method has one argument named 'tick' which counts the number of steps passed since the start of a simulation. Note: it is not possible to assign any value to the 'tick' variable inside 'step' method, but it can be used freely as the right hand side value.

```
agent Agent1 : SpaceAgent

var age : number

to create
 age = random 100 + 20
end

to step [ tick ]
 ; Decrease age by 1 each step after 101 simulation steps
 ; (the first value of 'tick' is always 0)
 if tick > 100
  [ age -= 1 ]
end
```

Actually, it is not required to have a step method for an agent. Moreover, for a step method its argument 'tick' can be omitted because it will be always added automatically.

3. Creating agents and the Observer

To create agents the commands 'create' and 'create-one' are used. All agents in SPARK exist in a special container called the Observer. When a new agent is created it is automatically added to the observer. When an agent is destroyed, it is removed from the observer. The observer can be used also for retrieving all agents of a particular type or for counting agents of a given type. In SPARK-PL it is not required to know anything about the observer because it is never used directly.

The command 'create' has two arguments: a type name and a number of agents to be created. A command 'create-one' has only one argument: a type name (and it creates one agent of the given type).

```
; Create 100 agents of type Agent2
var agents1 = create Agent2 100

; Create 1 agent of type Agent1
var agent2 = create-one Agent1

; Create 1 agent of type Agent1
var agents2 = create Agent1 1
```

The last two commands in the previous example yield different results. The command 'create' always returns an array of agents, meanwhile the command 'create-one' always returns a direct reference to a new agent.

```
; A valid command
(create-one Agent1).age = 100

; An invalid command
;(create Agent1 1).age = 100

; A valid modification
(get (create Agent1 1) 0).age = 100
```

In the example above the command 'get' was used. Recall, that this command has two arguments: an array or a list of any objects and an index, and it returns an entry of an array or a list at the given index.

Often it is required to do something with all agents in a list or an array. One way to do so is to use 'for' loop as follows

```
for i = 0 : count agents1
[
 var agent1 = get agents1 i
 agent1.age = 100
]
```

There is a shorter way to do the same thing. With the command 'ask' you can simply write

```
ask agents1
[
 age = 100
]
```

Recall that the command 'ask' is not just a shortcut for a 'for' loop, it also modifies the 'self' reference, that is the reference to the active object.

Sometimes it is required to have a full control over a model execution process. Assume that you don't want to rely on a step method of agents and instead want to call some agent's methods manually. In this case the 'ask' command is also useful. But first you need to get a list of all agents for which you want to call a specific method. All agents of a given type can be obtained using the 'agents' command. This command has one argument: a type name. It makes sense to process agents manually in either 'begin-step' or 'end-step' methods of a model.

```
model Model

to begin-step
 ; Process all agents of type Agent1 manually
 ask agents Agent1
 [
  do-something
 ]
```

```
end


agent Agent1 : Agent

to do-something
 ; do something
end
```

Often it is required to know the number of agents of a specific type. It is possible to use the 'count' command along with the 'agents' command, but it is more convenient (and efficient) to use a special command 'agents-number'. This command has one argument: a type name. This command is often used to collect statistical data, so it is often used inside the 'end-step' method.

```
to end-step
 total-agent1 = agents-number Agent1

 ; The next command gives the same result
 ; but it is not very efficient
 total-agent1 = count agents Agent1
end
```

To destroy an agent use its method 'die'. Agent can call 'die' by itself (in the 'step' method, for example), or it can call this method on another agent to kill it. There is also a command 'kill' with one argument: a list of agents to be killed.

```
agent Agent1 : Agent

age = random 100

to step
 age -= 1
 if age < 0
  [ die ]
end
```

In general, it is not safe to continue agent's activity after it has died. That is, it is not recommended to have any code after a call of the 'die' method. Because of this, it is often required to use the command 'exit' immediately after the 'die' method.

```
to step
 age -= 1
 if age < 0
 [
  ; equivalent to 'die'
  kill self
  exit
 ]

 ; This method will not be called if the agent is already dead
 do-something
end
```

To be more precise, it is not a requirement to exit from a method after destroying an agent. The only thing which is prohibited to do after this method is to use methods related to a space in which the agent exists (see next sections).

4. Space in SPARK-PL

It is necessary to declare a space in SPARK-PL. A space is declared in a model type with the 'space' keyword

```
model Model

space StandardSpace -10 10 -20 20 true true
```

You can treat the 'space' keyword as a command with 7 arguments. The first argument specifies a type of a space. There are four types now: StandardSpace, GridSpace, StandardSpace3d, and PhysicalSpace2d. The Standard space (including 3d space) is a continuous space where agents can occupy any position. The Grid space is also a continuous space where agents can occupy any position, but it also has a special discrete structure. The Grid space is divided into square cells (1 by 1 each) and all agents in the same cell are considered to be at the same location for some space-related commands ('agents-at', 'agents-here' commands, see the next section for details). The advantage of the Grid space is that some operation work faster there but the results are not accurate. The Grid space is not usually used. The Physical space is not described in this tutorial.

Next four arguments (six arguments for the 3d space) specify the dimension of a space: minimum x coordinate, maximum x coordinate, minimum y coordinate, maximum y coordinate. The last two arguments (three arguments for the 3d space) determine a topology of a space. Each space is a rectangle and the last two arguments specify whether the opposite edges of this rectangle are glued together. In other word, here are all possible combinations of these two last parameters and the description of the resulting topology.

Table 1. SPARK space topologies

false	false	Rectangle
true	false	Vertical cylinder
false	true	Horizontal cylinder
true	true	Torus

Note: in the current SPARK-PL implementation only constants can be used for specifying the parameters of a space.

There are several commands for working with a space. Commands

```
space-xsize
space-ysize
space-xmin
space-xmax
space-ymin
space-ymax
```

are quite intuitive. The size along x axis equals to the difference between the maximum x coordinate and the minimum x coordinate. The same is true for the size along y axis. Other commands work with agents in a space and will be described in the next section.

5. Space agents

Space agents are located in a space, that is, they have a position in a space. A position of a space agent can be always obtained through 'position' field defined in the SpaceAgent type. It has the vector type. It is not possible to assign this field directly, instead the method 'move-to' should be used.

```
agent Agent1 : SpaceAgent

to step
 ; Add [1, 0, 0] to the current position
 move-to position + [1, 0, 0]
end
```

Actually, it is better to use 'move-to' method for assigning a new position of an agent which is not related to the old position. In order to move relatively to the current position (as in the example above) it is better to use the 'move' command which simply adds its argument to the current position.

```
agent Agent1 : SpaceAgent

var velocity = [1, 0, 0]

to step
 move velocity
end
```

When a new space agent is created its default position is the origin (if there is no origin in a space, that is, no (0,0) point, then the default position depends on the topology). Agents can be assigned random positions in a space using a 'set-random-position' command. Or they can be moved to some calculated positions.

```
to setup
 ; Assign to newly created agents random positions
 ask create Agent1 100
 [
  set-random-position
 ]

 var i = 0

 ; Place new agents in a circular pattern
 ask create Agent1 10
 [
  move-to vector-in-direction 10 i
  i += 30
 ]
end
```

Very often some space agents create another space agents in the same position where they are located. This can be implemented as follows

```
agent Agent1 : SpaceAgent
```

```
agent Agent2 : SpaceAgent

to step
 ask create Agent1 2
 [
  ; 'myself' refers to the agent which executes
  ; the 'ask' command.
  ; Without 'myself' the position of a newly created
  ; agent would be used which is not our goal.
  move-to myself.position
 ]
end
```

There are methods 'hatch' and 'hatch-one' of SpaceAgent type which can do the same thing.

```
agent Agent1 : SpaceAgent

agent Agent2 : SpaceAgent

to step
 hatch Agent1 2
end
```

Space agents can be visualized. It is possible to customize a color of each agent using its 'color' field. By default, all space agents are black. Colors are treated as vectors with RGB components in the interval [0,1]. There are several predefined color constants like 'red', 'green', 'blue', 'white', etc.

```
agent Agent1 : SpaceAgent

var age = 100


to create
 color = red
end


to step
 age -= 1
 if age < 0
  [ color = orange ]
end
```

Besides color, it is also possible to specify agent's shape. In the current SPARK-PL implementation there is a significant restriction for changing a standard circular shape. Only agents derived from SpaceAgent can do this (it is quite natural). Moreover, a shape can be changed only inside a constructor as the first command.

```
agent Agent2 : SpaceAgent

to create
```

```
 ; Should be the first command
 super 0.5 square
end
```

The 'super' command has two arguments: a radius of an agent which determines a size of an agent and a shape. Shapes are special constants with names. Available names are: 'circle' (a standard shape), 'square', 'square2', 'torus'. The difference between 'square' and 'square2' shapes is that 'square' affects only visual appearance of an agent meanwhile 'square2' also changes the algorithm for determining if the agent collides with other agents or not.

All space agents can have different sizes. A size of a space agent is determined by the 'radius' field which can be read or assigned. In the Grid space the sizes of agents do not play any role except for visualization purposes. In the Standard space sizes play an additional role which is explained below.

With the 'agents' command, it is possible to get all agents of a specific type but very often it is required to get agents only in a specific region of a space. There are several commands to do so. First of all, there are commands 'agents-at' and 'all-agents-at'. The former has three arguments: a type name of agents to be retrieved, a point (vector) specifying a center, and a radius of a circular region in which agents are queried. The later has only two arguments: a center and a radius of a circular region. The command 'agents-at' returns a list of all agents of a specific type which intersect with a specific circular region. The command 'all-agents-at' returns all agents in a circular region. In fact, in the Grid space this commands completely ignores the radius of a circular region and return agents in a cell of the Grid space corresponding to a given point (center). It is the main difference between Standard and Grid spaces.

Sometimes it is necessary to get agents intersecting (or at the same position) with an active agent. Then it is possible to use commands 'agents-here' and 'all-agents-here'. The command 'agents-here' has one argument: a type name. The command 'all-agents-here' does not have arguments at all.

```
agent Agent1 : SpaceAgent

to step
 ; Returns all agents here, including the calling agent itself.
 ; Equivalent to
 ; var a = all-agents-at position radius
 var a = all-agents-here

 ask a
 [
  color = green
 ]
end

agent Agent2 : SpaceAgent

to step
 ; Equivalent to
 ; var agents1 = agents-at Agent1 position radius
 var agents1 = agents-here Agent1

 kill agents1

 ; Changes the radius of all agents of type Agent2
 ; at the current position (including the calling agent)
 ask agents-here Agent2
```

```
 [
  radius = 0.5
 ]
end
```

Another thing to know about space agents is how to measure the distance between two of them. Of course, it is always possible to compute the difference between their positions but it will not give a good result in the case of a topology different from a rectangle. There is a command 'distance' with two arguments: two references to space agents. This command returns a vector which represents a component-wise distance between two agents. If you need to know the absolute value of the distance between two agents then use vector's 'length' method to get it.

```
var d = distance agent1 agent2
var l = d.length
```