SPARK-PL: Attributes and User Interface

Alexey Solovyev

Abstract

Table of Contents

1. SPARK-PL Attributes

1.1. Introduction

Attributes in SPARK-PL are special declarations that can be applied to global variables, methods, or types (agents and models). Each attribute specifies a role of a global variable (method, etc.) in the SPARK user interface or the role in SPARK itself. For instance, in the SPARK interface some model variables can be parameters, other variables can be plotted as charts.

All attributes have the following syntax

```
@attribute-name(parameter1 = value1, parameter2 = value2, ...)
global some-variable
; to some-method
; agent some-agent
; model some-model
```

Each attribute is declared immediately before a global variable declaration. Each attribute has name and several parameters. Parameters can have different values: strings or numbers. All parameter values should be constants, it is not possible to use variables in the parameter declaration. Very often, parameters have default values. Parameters with default values can be omitted in the attribute declaration. It is possible to declare several attributes for the same global variable.

```
@attribute1(par = value)
@attribute2(par1 = value1, par2 = value2)
global some-variable
```

1.2. @chart attribute

This attribute tells that the value of a global variable should be plotted. Parameter 'name' with string value specifies the name of a chart in the user interface. The default value equals to the name of a global variable. Parameter 'interval' specifies how often the value of a variable will be read for plotting. The default value is 1.

```
; Plot variable 'data' each 2 simulation steps
@chart(name = "Data", interval = 2)
global data : number

; Plot variable 'data2' each simulation step.
; Chart's name will be "data2".
@chart()
global data2 : number
```

This attribute can be applied only to numerical variables.

1.3. @parameter attribute

This attribute tells that a global variable is a parameter in a SPARK model. It means that its value can be changed during a model simulation process manually by a user. Parameter 'name' gives the name of a parameter in the user interface. The default value is variable's name. Parameter 'min' specifies the minimum value of a parameter (default 0). Parameter 'max' specifies the maximum value of a parameter (default 0). Parameter 'step' specifies the adjustment step for a parameter (default 1). Parameter 'default' specifies the initial value of a parameter.

```
@parameter(name = "Data", default = 50, min = 0, max = 100, step = 2)
global data : number
```

This attribute can also be applied to boolean variables.

```
@parameter(name = "Flag")
global model-flag : boolean
```

Do not initialize global variables which has the parameter attribute. Use 'default' parameter instead. Consider an example

```
@parameter(name = "Data", min = 0, max = 100, step = 2)
global data = 50 : number
```

In this example, the variable 'data' will be always initialized with the value 50 every time the 'setup' method of a model is called. Very often it is not a desired behavior especially when some parameters control the initialization process.

1.4. @dataset parameter

If you want to collect the values of some variable during simulation process, then use '@dataset' attribute. Every global variable with that attribute will be added to the model data set, and this data set can be saved

at any time during simulation process. There is only one parameter for '@dataset' attribute: name of a variable in the data set. As always, the default value is variable's name itself.

```
; Parentheses can be omitted
@chart
@dataset(name = "Data")
global data : number

; Even parameters can be added to the data set.
; In that case all changes of a parameter will be saved.
@parameter
@dataset
global parameter : number
```

1.5. @external attribute

This attribute is applied to methods in a model type. Methods with this attribute will be available in the user interface. That is, if there are methods with the '@external' attribute in a model, then in the user interface a window 'Methods' will appear. This window will contain buttons with names of all methods declared with the '@external' attribute. A user can click on these buttons to call methods manually at any time during (or before) simulation. Only methods without parameters can have this attribute. There is only one parameter 'name' with the default value equals to the method's name.

```
; Only methods inside a model type can be declared
; with the @external attribute
model Model

@external(name = "Do Something")
to do-something
 ; do something
end
```

1.6. @step attribute

The attribute '@step' can be applied to declarations of agent types. It specifies the order in which agents make their steps. It has two parameters: 'priority' and 'time'. Both parameters are optional. The default value of 'priority' is 1000, the default value of time is "1". Note that the priority is a number, and the time is a string.

To understand the '@step' attribute, it is required to understand how agents make their steps in SPARK. Any agent has a type. Agents of the same type always make their steps in the same order they were created. That is, if you have two agents of the same type, then one which was created first is moving first each simulation step. The only question is how agents of distinct types are scheduled to make steps during each simulation step (or during each tick because one simulation step equals to one tick).

Each agent type has a special numerical value: priority. This number can range from 1 to 1000. The priority specifies in which order agents of distinct types perform their actions. If the priority of one type is less (as a number) than the priority of another type, then agents of the former type will move before agents of the later type. If two types have the same priority then the names of types will be compared and the first type in the lexicographic ordering will get the right of first move (agents of that type will move first). As mentioned above, by default the priority is 1000 for all agent types. It means, that if priorities are not

specified explicitly (using the '@step' attribute), then agents make their turns according to the lexicographic ordering of their type names. Consider several examples.

```
@step(priority = 1)
agent Virus : SpaceAgent

@step(priority = 2)
agent Macrophage : SpaceAgent
```

In this example, agents of type 'Virus' will act before agents of type 'Macrophage' because priorities are specified explicitly and 1 < 2.

```
@step(priority = 1)
agent Virus : SpaceAgent

@step(priority = 1)
agent Macrophage : SpaceAgent
```

In this example, agents of type 'Macrophage' will act first because they have the same priority as viruses but in the alphabetical ordering 'Macrophage' goes first.

```
@step(priority = 10)
agent Virus : SpaceAgent

agent Macrophage : SpaceAgent
```

In this example, agents of type 'Virus' will act first because the priority is not specified for macrophages, so they get the lowest priority (1000) by default.

Consider one more example which shows that the order in which agents make their steps can be very important.

```
model TestModel

space GridSpace -10 10 -10 10 false false

to setup
 ; Create several agents of type 'Agent1'
 ask create Agent1 10
 [
  set-random-position

  ; Create agents of type 'Agent2' at the same positions as
  ; agents of type 'Agent1'
  hatch-one Agent2
 ]
end


; Agent1
agent Agent1 : SpaceAgent
```

```
to step
 ; Kill all agents of type 'Agent2' at my position
 kill agents-here Agent2
end


; Agent2
agent Agent2 : SpaceAgent

to step
 ; Kill all agents of type 'Agent1' at my position
 kill agents-here Agent1
end
```

Who will survive in the example above? Agents of type 'Agent1' will survive because they have the right of first move since 'Agent1' < 'Agent2' in the lexicographic ordering. But the order of steps of agents can be easily altered using the '@step' attribute. If we had something like

```
@step(priority = 2)
agent Agent1 : SpaceAgent
;...

@step(priority = 1)
agent Agent2 : SpaceAgent
;...
```

then agents of type 'Agent2' would survive.

Now, let's look at the second parameter of the '@step' attribute which is called 'time'. It was mentioned before, that agents make their steps each tick. In general, it is not true. In fact, agents can skip some ticks or even make several steps during one tick. Ticks measure the number of simulation steps. Each simulation step has a time value which is required to finish a step. This time value is measured in abstract time units. By default, each step requires exactly one time unit. This value can be altered with the '@tick' attribute (see below). All agent types also have time values associated with them. This time value measures the amount of time which should pass between two consecutive steps of any agent of a given type. By default, this time is one unit. Since the default tick time is also one, it follows that by default each agent moves exactly once each tick (the amount of time passed between ticks is 1, and each agent waits for 1 time unit before next step).

The 'time' parameter of the '@step' attributes specifies how long agents of a given type wait before making next steps. This parameter is not an integer number, it is a rational number. Fractional values can be assigned. For instance, the following time values are possible: "1", "1/2", "1/3", "4/3", "5", etc. Note that all values should be quoted. If the tick time is "1" and for some agent type the time value is "1/3", then agents of that type will move three times during each simulation step. If the time value is "3/2" for some other type, then agents of that type will move during second, third, fifth, sixth ticks, etc.

Ticks in SPARK are discrete but time is continuous. Ticks partition the time line into discrete intervals of the same length. Agents always make their steps at specific time points. An agent with time value "1/2" will move at time points "1/2", "1", "3/2", "2", etc. The first two time points are in the first tick interval, the next two points ("3/2" and "2") are in the second tick interval, etc. Consider the following example. Assume that we have two agents of distinct types with time values "1/2" and "1/3". The first agent will act at time points "1/2", "1", "3/2", "2", etc. The second agent will move at time points "1/3", "2/3", "1", "4/3", "5/3", "2", etc. The priorities of agents will be compared only when they move at the same time point, i.e. at the points "1", "2", etc. For all other time points only one agent is moving, so there are no priority conflicts.

Very often, it is simpler to control the action time of agents explicitly using the current tick value which is passed to the 'step' method of each agent.

1.7. @tick attribute

This attributes specifies the amount of time for each simulation step (tick). The only parameter is 'time'. This attribute should be applied to a model declaration.

```
@tick(time = "1/2")
model TestModel

space GridSpace -5 10 -2 2 true false

to setup
 create-one SlowAgent
 create-one FastAgent
end


; Slow agents will move one time each 4 ticks
@step(time = "2")
agent SlowAgent : SpaceAgent

to create
 color = red
 move-to [-5,1,0]
end

to step
 ; Make one step to the right
 move [1,0,0]
end

; Fast agents will move two times each tick
@step(time = "1/4")
agent FastAgent : SpaceAgent

to create
 color = green
 move-to [-5, -1, 0]
end

to step
 ; Make one step to the right
 move [1, 0, 0]
end
```

1.8. @observer attribute

This attribute is applied to a model declaration. It has two parameters: 'observer' and 'mode'. The 'observer' parameter specifies the name of an observer implementation which will be used for a model. There are several available implementations of observers in SPARK. Right now, there is one fully

supported observer implementation (called "Observer1") and two experimental observers ("Observer2" and "ObserverParallel").

The parameter 'mode' specifies the execution mode for a model. There are two execution modes for the default observer ("Observer1"): "serial" and "concurrent". "ObserverParallel" has an additional mode: "parallel" which is similar to the "concurrent" mode and will not be discussed here (a parallelized version of SPARK is under development).

The default execution mode is the "serial" mode. It is a usual execution mode when all actions of agents have an immediate effect. It means, that, for example, a newly created agent will be immediately added to the list of existing agents and will be visible for all other agents as soon as it was created.

In the "concurrent" mode several actions are not performed immediately, but they are postponed until all agents finish their steps. Actions that are postponed are the following: creation of new agents ('create', 'create-one', 'hatch', 'hatch-one', etc.), deletion of existing agents ('die', 'kill'), changing values of data layers ('add-value', 'add-value-here', 'set-value', 'set-value-here', etc.), moving agents in a space ('move', 'move-to', etc.). The main feature of the "concurrent" mode is that all agents during a simulation step see the same environment (surrounding agents, values of data layers) regardless of the order of their steps. There are several restrictions for using the "concurrent" mode. The basic rule is that agents should not modify values of fields and call methods of other agents directly. Instead, agents need to use data layers and spaces for interacting with each other.

Consider an example.

```
@observer(mode = "concurrent")
model ConcurrentModel

space GridSpace -10 10 -10 10 false false

global data : grid

to setup
 ask create Agent1 10
 [
  set-random-position
  ; Create agents of type 'Agent2' at the same positions
  ; as agents of type 'Agent1'
  hatch-one Agent2
 ]

 data.set-value 1
end

; Agent1
@step(priority = 1)
agent Agent1 : SpaceAgent

to create
 color = red
 radius = 0.2
end

to step
 ; Decrease the data value by 1
```

```
 data.add-value-here -1
end

;Agent2
@step(priority = 2)
agent Agent2 : SpaceAgent

to create
 color = yellow
 radius = 0.5
end

to step [tick]
 ; If the data value is less than 1, then die
 if data.value-here < 1
 [
  die
  exit
 ]

 ; Increase the data value by 1
 data.add-value-here 1
end
```

Run this example, and you see that agents of both types will survive. If you change the execution mode to the "serial" mode, then agents of type 'Agent2' will die because agents of type 'Agent1' move first and they decrease the value of the data to 0. In the "serial" mode agents of type 'Agent2' see this value 0 and die. In the "concurrent" mode the value of the data which agents of type 'Agent2' see is still '1' (not '0') so they survive and increase the value of the data by 1. After both agent types finish their steps the value of the data will be modified (-1 + 1 = 0 so, in fact, values are not changed at all).