A. Simulation Environment for JAM

A.1 Introduction

The *SEJAM* simulator is a built as a software layer on top of a generic *JAM* node. The simulation layer provides visualisation and an extended simulation API that can be accessed by agents.

There are logical and physical nodes. A *SEJAM* world consists of logical nodes handled by one physical nodes, which can be connected to remote physical nodes via IP links. The simulation world consists of agent processing nodes (logical/virtual JAM nodes) and some communication links (virtual channels) between nodes enabling migration of agents and propagation of signals between nodes. Nodes are placed in a two-dimensional spatial simulation world, and each node has a distinct position.

There are computational and physical agents. They differ in operation behaviour, but both are modelled with AgentJS. Physical agents represent physical entities like, e.g., humans, ants, cars. Physical agents are bound to a node (the agent is only mobile through its node). The default agent type is computational and computational agents represent mobile software that can migrate between logical and physical ndoes.

A meshgrid network arranges nodes in a two-dimensional grid (that can be irregular and incomplete).

A patchgrid network is similar to NetLogo simulation worlds. A patch grid arranges nodes in a two-dimensional grid, too, but with an implicit global virtual communication link that enables migration of phyiscal agents between any of the nodes in the grid. Migration of computational agents (default) and propagation of message signals still require explicit links between nodes.

Communication ports and links have either (static or dynamic) point-to-point or broad/multicast characteristics, i.e., nodes within a specific spatial range can communicate with each other (computational agents).

Nodes (except in mesh- and patch grid worlds) can be moved within the two-dimensional world carrying their agents.

The simulation world coordinates and the relation to directions (e.g., `DIR.NORTH`) is show in Fig. 1.

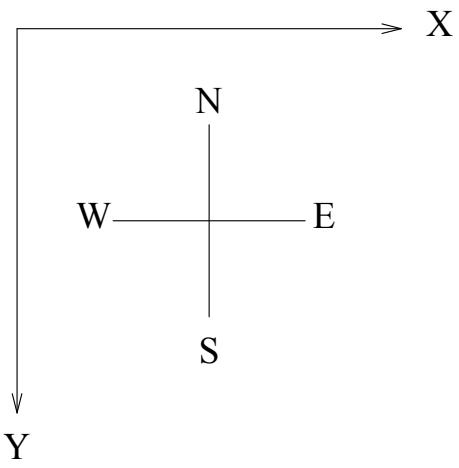

Figure 1. SEJAM2 simulation world coordinates and directions

A.2 Architecture

A.3 Model

The simulation model is defined by a JS object structure containing the following sections:

A.3.1 model

```
type model = {
    name? : string,
    classes : model.classes,
    // synonym for classes entry
    agents : model.classes,
    resources? : model.resource,
    patches? : model.resource,
    parameter? : model.parameter,
    world : model.world,
    physics? : physics,
```

```
}
```

The simulation model is a structure object consisting basically of agent and world descriptor attributes defining agent behaviour and visual properties, and the composition of the simulation world consisting of logical JAM nodes.

A.3.2 model.name

```
type model.name = string
```

The name of the simulation (model).

A.3.3 model.classes

A.3.4 model.agents

```
type model.classes = model.agents = {
    $ac: {
      behaviour: agent constructor function |
                 open(file),
      visual : visual,
      type? : 'computational'|
              'phyiscal'
      on?: { $event : visual },
    }
}
```

Definition of agent classes (behaviour, constructor function) and visual properties used in the simulation. The agent class can be loaded from a file (by using the `open` function, which is not supported in browsers). The event handlers defined in the *on* section can be used to be display visuals on signal delivery to an agent (the visual should define a display timeout).

An optional type attribute can classify the agent in two different super classes:

1. Computational agents (default) → agents represent mobile software processes → ABC class
2. Physical (or behavioural) agents → agents represent physical entities (like humans, although, being virtual in the simulation) → ABM class

A.3.5 model.resources

Passive agents with a visual shape and an optional set of parameters.

```
type model.resources = {
    $rc : {
      visual : visual,
      parameter : {}
    }
}
```

A.3.6 model.patches

```
type model.patches = {
   $pc : {
     visual : visual,
     parameter : {}
    }
}
```

Set of patch classes (node visuals). Patch parameter can be accessed by agents via the tuple space (node installs tuple provider and consumer functions) or by the `ask.patches`procedure.

A.3.7 model.parameter

```
type model.parameter = {
  $param : number|string|array|{}
}
```

Definition of a set of simulation parameters that can be changed by run-time configuration.

A.3.8 model.world

```
type model.world = {
    init? : {
        agents? : {
          $ac : function (nodeid:string,position) →
                {level:number,args:[]|{}|null}
        },
        nodes? : function (node),
        resources? : function (resource),
        patches? : function (resource),
        world? : {},
        physics? : function (physics),
    },
    start?: function (env),
    stop? : function (env),
    data? : {},
    map? : function () → simobj [],
    nodes? : { $nc: function (x,y,id)
                    → node(simobj) },
    resources? : { $rc: function () → simobj },
    meshgrid? : meshgrid,
    patchgrid? : patchgrid,
}
```

Definition of the simulation world.

The initialisation section provides functions that create agents on specific nodes (returning the individual parameter set of an agent), initialises nodes, resources, patches, and an optional additional physical MBP world.

The `model.world.nodes` and `model.world.resources` entries specify simulation object constructor functions returning a simulation object descriptor. The constructor functions can alternatively defined in the `model.nodes` and `model.resources` sections, respectively.

The *map* entry defines a function that returns a set of simulation objects at the beginning of the simulation (creation of the simulation world).

A.3.9 type agent

```
function $name ($p1,$p2,..) {
    this.$v=init;
    ..
    this.act = {
      $activity: function (){..}
    }
    this.trans = {
      $activity: string|identifier|function ()
                                    → string,
    }
    this.on = {
      $handler: function () {},
      $signal:  function () {},
      ..
    }
    this.next = $activity;
}
```

Definition of the agent behaviour (based on the ATG model) with an agent object constructor function.

A.3.10 type node

```
type node = {
    id:string,
    x:number,
    y:number,
    visual: visual,
    ports? : {
      $id: comobj
    }
}
```

A node simulation object.

A.3.11 type visual

```
type visual = {
    shape : 'circle'|'rect'|'triangle'|'icon',
    icon? : string,
    label? : { text : string, fontSize : number },
    width : number,
    height : number,
    fill? : {
      color : string,
      opacity : number [0..1]
    },
    line? : {
      width? : number,
      color? : string,
    },
    x? : number is relative position in parent frame,
    y? : number is relative position in parent frame,
    time? : number is a timeout for a visual,
}
```

This is the visual definition type of an agent, node, or resource (link)..

A.3.12 type simobj

```
type simobj = {
    id: string,
    x: number,
    y: number,
    ports? : { $id : comtype },
    gps? : {latitude,longitude,height},
    visual : visual
}
```

Simulation object descriptor (position and visual geometry) with optional communication ports (type node only).

A.3.13 type comobj

```
type comobj = {
```

```
    type : 'multicast'|'physical'|'unicast',
    status : function (nodes:string [])
             → boolean|string [],
    ip? : string,           // type='physical',
    to? : string,           // ..
    proto? : 'udp'|'http', // ..
    visual : visual,
}
```

Communication port object descriptor.

A.3.14 type meshgrid

```
type meshgrid = {
      rows : number,  // y-axis
      cols : number,  // x-axis
      levels? : number,  // z.axis
      // Positions of z-levels on 2d plane
      matrix? :[[z1_x,z1_y],[z2_x,z2_y],[z3_x,z3_y]],
      node : {
        // Node ressource visual object
        visual : visual,
        // Node filter creating irregular meshgrids
        filter? : function (pos:number []) → boolean
      },
      // Link port connectors
      port? : {    // Link port connectors
        type : 'unicast',
        place : function (node)
                → {x:number,y:number,id:DIR} [],
        visual : visual
      },
      // Connections between nodes
      // with virtual port conectors
      link? : {
        type : 'unicast',
        connect? : function (node1,node2,port1,port2)
                   → boolean,
        visual : visual
      },
```

```
      // Create some custom nodes dynamically
      // with a constructor function defined below
      map? : function (model) -> [],
      // Some custom node constructor functions
      nodes : {
        $nodeclass : function (x:number,y:number)
                     → simobj
      }
}
```

Defines a one, two, or three-dimensional meshgrid network of nodes connecting neighbouring nodes.

A.3.15 type patchgrid

This is a classical simulation world derived from the *NetLogo* simulation world model used for simulating social and technical systems. In a patch grid the world is divided into patches.

If the floating parameter is not set or false, each patch is related to a logical JAM node. Physical agents can occupy a patch via the logical JAM node. They can migrate from one patch to any other patch by a number of steps and a movement angle (delta vector) immediately. A node is connected to all other nodes in the grid by a virtual link. Each node provides connectivity for physical and computational agents which differ. Computational agents cannot migrate to any node in one step. They have to use other virtual links providing connectivity to some (neighbouring) nodes, e.g., simulating a short range mobile link.

If the floating parameter is true, only a visual grid with patch resources is displayed. In this case, a physical agent in the simulation is a mobile tuple <logical node,behavioural agent>.

```
type patchgrid = {
    rows : number,
    columns : number,
    // geometrical width and height of patch in pixel
    width : number,
    height : number,
    // default visual for a patch field, can be changed
```

```
    visual? : visual,
    floating? : boolean
}
```

A.3.16 type physics

```
type physics  = {
  scenes:{
    plate:function plate(world,settings)
          → mbp | open(file)
  }
}
type mbp = {
    masses : mass [],
    loadinggs : [],
    map : function (number []) → mass,
    read: function (parameter) → string,
    write: function (parameter,value)
}
```

Includes a physical model definition (MBP).

A.3.17 type resource

```
type resource = {
    parameter: {},
    visual : visual
}
```

A.4 API

The simulator provides some additional and extended AIOS APIs for agents.

A.4.1 Extension: net

Simulation functions for physical (behavioural) agents derived from NetLogo simulation model is provided by the *net* extension related to a patch grid world. Furthermore, agent grouping is supported.

- *net.ask*
- *net.create*
- *net.forward*
- *net.get*
- *net.turn*
- *net.group*
- *net.set*
- *net.setxy*

net.ask

```
function (what:string, who:string|string
[]|number|number []|bbox, callback?, remote?) →
*[]
```

The *ask* function (derived from NetLogo simulation model) can be used to apply a function to a set of simulation objects, i.e., agents (physical only), patches, resources, nodes. The simulation object passed to the callback function depends on the type selection (*what*) and can be modified by the callback function.

Note that the objects are not protected like agents executed in their own context, and the callback function can invalidate the object, e.g., an (other) agent or a logical *JAM* node. If the simulation objects asked for agents and the *remote* flag is set, the callback function is executed in this agent context asynchronously, otherwise in the context of the caller agent synchronously. In this case, a second argument is providing the node object for the agent. The function returns the requested set of simulation objects.

The type of simulation object is selected by the *what* argument:

- `agent(s)` returns agents within a radius, at a patch position, or in a specific direction → `{agent:string, class:string, pos:{x,y}} []`

- `agent(s)-<class>` restricted to a specific agent class → `{agent:string, pos:{x,y}} []`
- `node(s)` returns nodes within a radius, at a patch position, or in a specific direction → `{node:string, class:string, pos:{x,y}} []`
- `node(s)-<class>` restricted to a specific node class
- `patch(es)` returns patches → `{x,y,..} [][] | {x,y,..} [] | {x,y,..}`
- `resource(s)` returns resources within a radius or specific direction with object descriptors → `{resource:string, distance:number, class:string, data:*, x, y, w, h} []`
- `resource(s)-<class>` restricted to a specific class → `{id:string, distance:number, data:*, x, y, w, h} []`
- `children` returns agent group children → `string []`
- `parent` returns agent group parent → `string`
- `bbox` returns bounding box of agent (including groups) or resource coverage in the patch world → `{x,y,w,h}`
- `distance` returns the distance to nearest object or objects in a specific direction, within a bounding box, or relative to a specific object (objects: agents, resources) → `{distance:number, objects:{} []}`.
- `distance-<class>` returns the distance to the nearest object of a specific class (in general `agent` or `resource`, or a specific agent or resource class)

The *who* parameter commonly specifies the objects (of the type class) or a spatial region to be searched. Agents of a specific class can be selected by the *what* argument, a specific agent can be selected by the *who* argument by passing the agent identifier to the *who* parameter (or an regular expression for finding a set of agents, alternatively).

For a radius search around the current agent position the radius *r* (in patch units) has to be passed to the *who* paramter. A specific patch world position (of a patch, node, or resource) is referenced by an array `[x,y]`. The current position is referenced by a `null` argument. A search for all ob-

jects of a type and class is performed with a `*` argument.

A patch search returns a two-dimensional array within a radius region (except if `patch(es)-array` was selected), a two- or one-dimensional array on a bounding box search, and one patch object if a position was selected. A patch search returning all patches (`*` argument) returns the original two-dimensional patch array.

A search in a rectangular region is performed with a bounding box `{x,y,w,h}` or `{x0,x0,y1,y1}`) in absolute path world coordinates, a bounding box heading towards a specific region `{dir,distance?,spread?}` including the current position at the outer side, or relative to the current position `{dx,dy,w,h}`. The bounding box has to be passed to the *who* parameter. In case of a patch search within a two-dimensional region, the returned object set is organised in a two dimensional array (rows by columns) mapping the search region, otherwise a vector array is returned.

A search (e.g., distance) towards a specific direction can be limited to a maximal distance by passing a `{dir,distance:number}` argument to the *who* parameter.

```
set=net.ask('agents',4)
// [{agent:"tyroilla",class:"ac",pos:{x:3,y:2}}]
set=net.ask('distance',DIR.NORTH)
// [{agent:"tyroilla",distance:2,
//   class:"ac",pos:{x:3,y:2}},..]
set=net.ask('resources-street',{x:2,y:2,w:4,h:4})
// [{resource:"street1",class:"street",
//   x:3,y:3,w:10,h:3},..]
patch=net.ask('patch',[2,3])
// {x:2,y:3,data:..}
```

Example 1.

net.create

```
function (what:string, num:number, callback?) →
string []
```

Creates a number of simulation objects, i.e., agents, nodes, resources (derived from *NetLogo* simulation model). The optional callback function can be used to initialise the object. In the case of agents, the *callback handler is executed in the agent context* asynchronously and can be used

to position (physical agent) in the world or to transfer an agent to a destination node.

The type of simulation object is selected by the *what* argument:

- `agent(s)`
- `agent(s)-<class>`
- `node(s)`
- `node(s)-<class>`
- `resource(s)`
- `resource(s)-<class>`

net.forward

```
function (delta?:number)
```

Moves agent and its node in the current direction (agent heading) by *delta* places (approximately if heading angle is not orthogonal). Group formations are moved together. Patch grid boundaries are checked. A movement beyond the patch grid world coordinates is discarded (including attached children).

net.get

```
function (p:string) → *
```

Returns a physical agent parameter. Valid simulation object parameters are currently *color* and *shape*, *heading*. The *heading* variable holds the current forward direction of the agent.

net.group.add

```
function (parent:agentid, childs:[ag1,ag2,..],
align:dir)
```

Adds a set of physical agents including the logical node (children) to a parent agent and node and attaches the added agents visually object to the parent node container with the given alignment mode (direction): `DIR.NORTH`, `DIR.SOUTH`, `DIR.WEST`, `DIR.EAST`, `DIR.NW`, `DIR.NE`, `DIR.SW`, `DIR.SE`. Physical agents can be anytime grouped without prior declaration. Groups can be nested, i.e., a child can be a parent node of another group.

net.group.rem

```
function (parent:agentid, childs:[ag1,..])
```

Removes child nodes from a parent agent group container.

net.set

```
function (p:string, v:*)
```

Sets a physical agent variable. Valid simulation object parameters are currently *color* and *shape*, *heading*.

net.setxy

```
function (x:number, y:number)
```

Moves object to new position (*x*,*y*).

net.turn

```
function (angle:number|dir)
```

Turns agent heading by a delta angle (in degree) or sets the heading to a specific direction. An angle of 0 is related to the North direction, 90 degree to the East direction, and so on.

A.4.2 Extension: simu

Generic simulator interface.

simu.changeVisual

```
function (id:string,visual)
```

Changes the visual property of a simulation object.

simu.clear

```
function (msg:boolean, log:boolean)
```

Clears the system and (agent) message windows.

simu.createNode

```
function (nodeclass:string|function, arg1,
arg2,...) → nodeid
```

Creates a new simulation node (logical JAM node). The first argument either specifies the name of a simulation node constructor function defined in the `model.world.nodes` section of the simulation model.or a function returning an object of type *simobj*.

simu.createOn

```
fucntion (nodeid:string, ac:string, args:{}|[],
level:number) → agentid
```

Create a new agent of specified class on given node. The agent constructor function must be already compiled (from the simulation model section *classes*)

simu.createResource

```
function (id:string, resclass:string, visual,
paramaeters?:{}) → resid
```

Creates a new resource. A resource is a passive agent with an optional set of parameter variables.

simu.deleteResource

```
function (id:string)
```

Deletes and removes a resource from the simulation world.

simu.deleteNode

```
function (nodeid:string)
```

Deletes a simulation node (logical JAM node).

simu.event.add

```
function ()
```

simu.event.get

```
function () → []
```

simu.getStats

```
function (target:'node'|undefined, arg:string)-> {}
```

simu.getSteps

`function () → number`

simu.inspect

`function (object|array|*,..) → string`

simu.message

`function (string)`

Print a pop-up message.

simu.model

`function () → model`

Returns the simulation model descriptor object.

simu.move

`function (id:string, dx, dy, and, andmore)`

Relative movement of a simulation object in the 2D simulation world.

simu.moveTo

`function (id:string, x, y, and, andmore)`

Absolute movement of a simulation object in the 2D simulation world.

simu.network

`{x:number, y:number, z:number}`

simu.options

`{..}`

simu.parameter

`function (parameter:string) → *|model.parameter`

Returns simulation model parameter (*model.parameter*, if defined). If argument is undefined the entire model parameter object is returned, otherwise a specific parameter.

simu.print

```
function (object|array|*)
```

Pretty printer for objects and arrays. Used for inspection only.

simu.sprint

```
function (object|array|*) → string
```

Pretty printer for objects and arrays. Used for inspection only. Returns string.

simu.start

```
function (steps:number|undefined)
```

Start the simulation.

simu.stat

```
function (p,v)
```

simu.stop

```
function () → number
```

Stop the simulation and returns the number of simulated steps.

simu.time

```
function () → number
```

Get the current simulation world time (includes lag correction and is not the host system time)

simu.untis

```
??
```

Simulation world units

simu.Vec3

```
function (x,y,z) → CANNON.vec3
```

Returns a vector for physical simulation (*CANNON* MBP).

A.4.3 Database

There are two data base modes supported:

1. Internal embedded SQL data base mode with local file system access
2. External SQL data base (already running) that is accessed via a proprietary communication protocol and socket/http communication.

The first mode supports synchronous operations, whereas the second mode only supports asynchronous operations using a callback function. The first mode returns SQL request-response objects:

```
type sqlresponse = {
    code:number,
    ok:boolean|undefined,
    statement:string,
    result: []|{}|*|undefined
}
```

simu.db.init

`function (path:string, channel:number|undefined, callback?) → status`

Opens or creates a new SQL data base. Two modes are supported: (1) Embedded SQL and local file system mode (*channel* argument must be undefined) or (2) External SQL data base that is accessed via a proprietary communication protocol and socket/http communication.

simu.db.createMatrix

`function (path:string, matname:string, header:[], callback?) → status`

Creates a new matrix in the data base referenced by the path argument. The header argument specifies the type interface of a row, i.e., `['$name:$type']` or `[$type]`. `$type` is either an SQL `integer` or `varchar(n)` type.

simu.db.createTable

`function (path:string, matname:string, header:{}, callback?) → status`

Creates a new table in the data base referenced by the path argument. The header argument specifies the type interface of a row, i.e., { `$name:$type` }. `$type` is either an SQL `integer` or `varchar(n)` type.

simu.db.drop

`function (path:string, table:string, callback?) → status`

Deletes a table (or matrix) in the data base referenced by the path argument.

simu.db.exec

`function (path:string, cmd:string, callback?) → status`

Executes a SQL statement on the data base referenced by the path argument.

simu.db.get

`function (path, callback?) → status`

??

simu.db.insert

`function (path,table:string,row,callback?) -> status`

Inserts a row in the given table and data base.

simu.db.insertMatrix

`function (path, matrix:string, row, callback?) → status`

Inserts a matrix into the data base.

simu.db.readMatrix

`function (path,matrix:string,callback) → status`

Reads an entire matrix.

simu.db.select

```
function (path,table:string, vars:[], cond:string,
callback?) → status
```

Data base select operation

db.writeMatrix

```
function (path:string, matrixname:string,
matrix:[][], callback?) → status
```

Writes an entire matrix to the given data base.

A.4.4 CSV

simu.csv.read

```
function (file:string, callback, verbose?) →
status
```

Asynchronous reads and parses a CSV file and passes the content as a matrix to the callback function

simu.csv.write

```
function (file:string, header:string [], data:[][],
callback:function, verbose?) → status
```

Writes a matrix to a file in CSV format.

A.4.5 Physics

simu.phy.get

```
function (id:string) → *
```

Returns a physical simulation object

simu.phy.refresh

```
function ()
```

Refreshs the physical simulation

simu.phy.step

```
fucntion (n:number, callback)
```

Performs (n) physical and computational simulation step(s)

simu.phy.stepPhyOnly

```
function (n:number, callback)
```

Performs (n) physical simulation step(s)

A.5 Networking

A world consists of a set of logical (virtual) JAM nodes can communicate via virtual channels with each other. Selected logical nodes can be connected to the outside world via physical IP links.

A.5.1 Comparison of NetLogo and SEJAM Simulations

The well known and established *NetLogo* simulator is (commonly) used for the ABM domain only. In contrast, the *SEJAM* simulator is primarily used for the ABC domain, although any physical simulation can be transformed in a computational task implementing the behaviour of a physical entity. The fusion of real worlds deploying computational agents (that interact with machines and humans) with virtual simulation models requires a mapping methodology to be able to simulate physical agents (i.e., artifacts of real entities) using computational agents.

Computational agents as mobile software processes require a connected communication network of host platforms (computers) to migrate along a path AB. A human being, in contrast, do not require such a digital transport network. In a simulation world, a physical (pure behavioural) agent can overcome arbitrary distances in one step (in principle).

To combine social interaction and computational simulation models, the *SEJAM* simulation model was extended by two super classes of agents: (1) Computational (2) Behavioural/Physical agents. A behavioural agent consists of a <*mobile node, physical agent*> tuple. The behavioural agent is bound to this node and can change its position only by moving the node carrying the agent. In contrast, computational agents are mobile and can

hop from one platform to another if there is a communication link between both platforms by performing process serialisation and deserialisation. The link can be virtual (inside the simulation world) or physical connecting a logical node of the simulation world with another *JAM* platform in the real world (e.g., a smartphone) via the Internet.

The mapping of *NetLogo* model constructs and statements on the *SEJAM* simulation model is shown in the following table. There are two approaches implementing the *NetLogo* patch grid world in the *SEJAM* simulation world: (1) Floating grid with mobile nodes carrying immobile agents representing turtles (active agents) and resources representing patches (2) Static grid with immobile nodes and mobile agents (turtles) representing patches. Only the first approach is considered (although both can be implemented with *SEJAM*). In *NetLogo*, the agent behaviour is entirely controlled and executed by a global observer (centralised macro control), whereas in *SEJAM* the agent behaviour is controlled and executed by each individual agent (decentralised micro control).

NetLogo	SEJAM
Entity Turtle	Active agent: mobile logical JAM node linked with physical behavioural JAM agent, Passive agent: Resource
Entity Patch	Resource
Entity World	JAM world consisting of one physical JAM node and multiple mobile logical JAM nodes (on patch positions), patch grid
Observer	World agent on dedicated JAM world node controlling simulation
`turtles-own [x,y]`	Agent body variables `this.x, this.y`
`patches-own`	Patch resource parameter set

NetLogo	SEJAM
`forward N, rotate dang`	Moving of physical agent with its node inside the simulation world
`ask turtles`	`net.ask('agents',..)` simulation interface for physical agents
`ask patches`	`net.ask('patches',..)` simulation interface for physical agents
`globals`	Global model parameter, JAM world agent and world node

Table 1. Comparison of the NetLogo simulation model with SEJAM model using floating patch grid approach

```
to setup
  ask patch x y [
    sprout 1 [
      set color brown
      set shape "circle"
      set size 10
      stamp
    ]
  ] function world () {
  this.act = {
    init: function () {
      net.ask ('patch', [x,y],
        function (patch) {
         net.set({
           color: 'brown',
           shape: 'circle',
           width: 10,
           height: 10
         })
      })
    }
```

```
  } }
```

Modification of patches in NetLogo and SEJAM with NetLogo compatibility layer

```
// 0. NetLogo
bread [agents agent]
ask [                ask id []  ask id-here []
    set data 0
    set more 0
    rotate a
    forward n
]
// 1. Observer control
function world () {
  simu.bread(['agent'])
  this.act = {
    function act1 () {
      net.ask('agents','*'|id|null,
               function (ag) {
          // Executed in ag Ctx
                 this.data = 0
                 this.more = 0
                 net.rotate(a)
                 net.forward(n)
                 },true)
   }
}
// 2. Agent control
function agent() {
  this.act = {
    function actx () {
        this.data = 0
        this.more = 0
        net.rotate(a)
        net.forward(n)
    }
  }
}
```

Relation of NetLogo turtle modification and AgentJS behaviour. Either the

agent itself modifies its visual, position, and body data, or the world agent modifies a set of agents via the simulation interface.

```
model = {
    agents : {
      $name : {
        behaviour : function ac () {},
        visual : visual,
        type : 'physical'|'computational'
      }
    },
    resources : {
       $name : {
           visual : visual,
           parameter : {}
       }
    },
    nodes : {
       $name : {
         visual : visual,
         parameter ; {}
       }
    }
    world : {
        patchgrid : {
            rows: number,
            columns: number,
            ..,
            visual? : visual,
        }
    }
}
```

The entire simulation model structure (visual parameters define the visual shape of simulation objects)

A.6 Release Information

Author: Stefan Bosse
Revision: 8.7.2019