The Computational Platform for DT Models: Version 0.11 Technical Documentation

Steffen Fürst (Global Climate Forum)

July 27, 2020

1 Overview

This text is supplementary material for the Computational Platform for DT models. The platform implements an extended and generalized graph dynamical system. The extensions are added to ease the development of ABM simulations by combining (social) networks with spatial information. This document describes the current implementation of the platform. Additional information about the basic idea behind the platform, an example implementation of a model in form of a tutorial, and the source code of the platform itself can be found at `https://globalclimateforum.org/GCF-Platform-for-DT-Models`. To use the platform for writing own models, the basic idea text combined with the tutorial should be sufficient. Previous, theoretical work on the concept can be found at `https://globalclimateforum.org/2018/09/26/`.

The current version of the platform is only a prototype to demonstrate the concepts, including the parallelisation of the simulation. It is written in R without any consideration of the CPU performance. Other versions in other languages will follow, with the core probably written in C++. This versions will then also clean up the inconsistent naming between the description of the platform and the platform code itself.

2 Important Concepts

This section contains a short wrap-up of important concepts of the platform. But it is advisable to read the basic idea text available at `https://globalclimateforum.org/GCF-Platform-for-DT-Models` before this document.

2.1 Nodes & Edges

Nodes may be of different types, e.g. representing persons, households, firms, etc. and have a state that belongs to a type-specific state space.

Edges between the nodes are directed, if a transition of node a depends on the state of node b, an edge from b to a is needed. Edges may also have different types (e.g. a network for goods exchange may differ from a network for information exchange) and may have a state (e.g., when representing an exchange of goods, an edge's state may report the quantity).

2.2 Node Layers

The nodes of the graph are organized in three different layers. The main interactions between agents are described in the **Interagent Layer**, which in the current implementation is also called "explicit graph layer". The "explicit" here refers to the point that the edges in the graph must be generated explicitly by the model code, while in the other two layers edges are also generated implicitly by the platform itself. Nodes of this layer are also called computational agents.

The second layer is the **Spatial Layer**, which provides a discrete grid. Each element of the grid is also a node in the graph, can have a state, too, and each computational agent can be assigned to

an element of the grid. In particular, elements of the grid may also aggregate information from all agents in the explicit graph "located" there.

The third layer contains nodes that has an edges from and/or to every other node of the graph and is therefore called **Global Layer**. This may be the case, e.g., for model parameters that all (or large sets of) computational agents get as input, for gathering information from them to generate model output for visualisation, and for aggregating information within the model that is passed back to the computational agents.

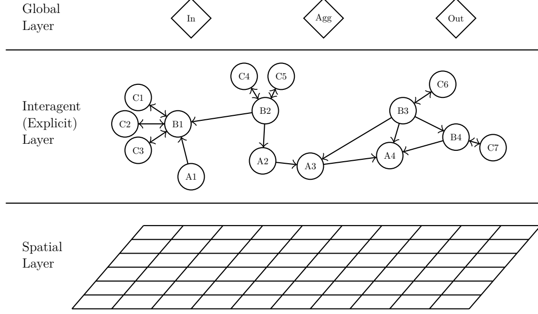

Fig. 1: All nodes of the graph, but only the explicitly given edges are shown

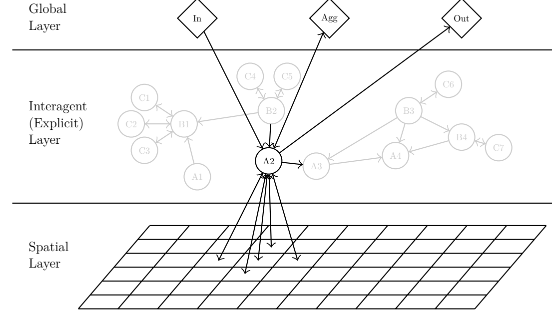

Fig. 2.: Explicit and implicit edges for an example computational agent

2.3 Parallel Execution

A key feature of this platform is that it allows simulations to be run on machines with vastly different computing power, up to high performance computers. Which means that the overall state of the simulation can be splitted into smaller chunks and distributed over different processing elements (PEs). Many design decisions of the platform where taken with this requirement in mind, e.g. that agents a can only access the state of agent b, when an edge from b to a exists.

In the current version the distribution is a virtual one. The implementation uses a single CPU core but for illustrating the principle, the state of the nodes from the interagent layer is distributed over several lists, where each list is representing a PE. The nodes of the global and spatial layer are available on all PEs, in the real parallel implementation this will require additional synchronization, which was omitted in this implementation.

Also the explicit edges and the implizit edges between nodes of the spatial and the interagent layer are virtually distributed. An edge and it's state is available on PE i in the case, that one of the both connected nodes are distributed to PE i, whereby the nodes of the spatial layer are all assigned to PE 1 in the current implementation.

3 Model State Data Structure & Initialization

3.1 Concept

The agents on the platform can be of different types $\delta_a \in \Delta_A$, where Δ_A is the set of all agent types. For each agent type δ_a we have a different set of state variables $x_{\delta_a,1}, \ldots, x_{\delta_a,n}$, where each $x_{\delta_a,i}$ is element of a (potentially) different set of possible states $X_{\delta_a,i}$. So the state space for an agent of type δ_a is $\Theta_{\delta_a} = X_{\delta_a,1} \times \ldots \times X_{\delta_a,n}$.

The agents can be set into relation via a directed multi-graph, whereby the agents are represented by vertices. The edges in this graph can also be associated with states. We construct the state space for the edges in the same way as for the vertices/agents.

The overall state space of the system at time t is then $\Theta_t = \prod_{\delta_a \in \Delta_A} \Theta_{\delta_a}^{n_{\delta_a,t}} \times \prod_{\delta_e \in \Delta_E} \Theta_{\delta_e}^{n_{\delta_e,t}}$ where Δ_E is the set of all edge types, $n_{\delta_a,t}$ is the number of agents of type δ_a and $n_{\delta_e,t}$ is the number of edges of type δ_e in the system.

To initialze the simulation, we must define all the different types and construct the initial model state. The types of the set Δ_A are therefore assigned to one of the three layers.

3.2 Code

So the initialize function of the platform has the parameters `agentTypes`, `globalTypes`, `rasterTypes` and `edgeTypes`, where `agentTypes` are the types of the nodes of the **Interagent Layer**, `globalTypes` the types of the nodes of the **Global Layer** and `rasterTypes` the types of the nodes of the **Spatial Layer**. The initial state for the **Interagent Layer** can be constructed by the `stateInitFunc`, for the **Global Layer** by the `globalInitFunc` and for the **Spatial Layer** by the `rasterInitFunc`. Please be aware, that the initial state can contain an empty set for some of the types, e.g. when entities of a type only appear later in the simulation.

The `initSimulation` function creates a new R environment, which contains the complete state of the simulation and some functions to access the three different layers. It also incl. the tables that allocate the nodes and edges of the **Interagent Layer** to the different (virtual) PEs.

```
initSimulation <- function(agentTypes,
                           edgeTypes,
                           modelParams,
                           stateInitFunc,
                           globalTypes,
                           globalInitFunc = NULL,
                           rasterTypes = NULL,
                           rasterInitFunc = NULL) {
  sim <- new.env()
  sim$agentTypes <- agentTypes
  sim$edgeTypes <- edgeTypes
  sim$globalTypes <- globalTypes
  sim$rasterTypes <- rasterTypes
  sim$numPEs <- modelParams$numPEs

  modelStateFull <- stateInitFunc(modelParams)
  ## sim$distributed is a list of two vectors. The vectors are representing
  ## the different PEs, so the length of those vectors is equal to sim$numPEs.
  ## The first vector contains the model state of the Interagent Layer distributed
  ## to the different PEs, the second contains for each vector element
  ## (representing a PE) a table that contains for all edges where a node of
  ## this PE is involved the information on which PE the node of the opposite
  ## side is "living".
  sim$distributed <- .mapAgentsToPEs(modelStateFull, agentTypes, sim$numPEs)
  sim$getExplicitLayer <- function() { sim$distributed[[1]] }
  sim$getPEs <- function() { sim$distributed[[2]] }

  if (! is.null(globalInitFunc)) {
    sim$global <- globalInitFunc(modelParams)
  }
  ## we always create automatically the type MODELPARAMS in the Global Layer
  sim$global$MODELPARAMS <- modelParams

  if (! is.null(rasterTypes)) {
    sim$raster <- rasterInitFunc(modelParams)
    for (r in rasterTypes) {
      ## .createRasterLayer modifies the sim environment as a side effect
      ## by adding for each rasterType a table with nodes for this raster and
      ## a table with a PE list
      sim <- .createRasterLayer(sim, r, sim$raster[[r]], 1)
    }
  }

  sim$getGlobalLayer <- function() { sim$global }
  sim$getRasterLayer <- function() { sim$raster }

  ## reduce is a helper function to calculate global values based on the state
  ## of the nodes in the *Interagent Layer*.
  sim$reduce <- function(type, agentFunc, aggregationFunc) {
```

```
    sim$getExplicitLayer() %>% map(type) %>% map(agentFunc) %>% unlist %>%
    ↪  aggregationFunc
  }

  sim
}
```

As we can see, the `initSimulation` function calls two helper functions `.mapAgentsToPEs` and `.createRasterLayer` which are implemented as follows:

```
## The helper function ~.mapAgentsToPEs~ takes a
## non-distributed model state and creates a distributed version.
## See also the comment in initSimulation.
.mapAgentsToPEs <- function(modelState, agentTypes, numPEs) {
  edgeTypes <- modelState %>% names %>% discard(. %in% agentTypes)

  ## Store which agent was moved to which PE
  PElist <- tibble(ID = character(),
                   PE = integer())

  ## Prepare the distributed data structures (the indices are the different PEs)
  modelStateDistributed <- vector("list", numPEs)
  PElists <- vector("list", numPEs)
  for (i in 1:numPEs) {
    PElists[[i]] <- tibble(ID = character(),
                           PE = integer())
  }

  ## Distribute the agents
  for (a in agentTypes) {
    ## We group the agents by their rowid (modulo numPEs), so that each PE gets
    ↪  the
    ## same number (plus/minus one) of agents
    modelState[[a]] %<>% rowid_to_column("PE") %>%
      mutate(PE = ((PE - 1) %% numPEs) + 1)
    PElist %<>% bind_rows(modelState[[a]]) %>% select(ID, PE)
    perPE <- modelState[[a]] %>% split(.$PE)

    for (i in 1:numPEs) {
      modelStateDistributed[[i]][[a]] <- perPE[[i]]
      PElists[[i]] %<>% bind_rows(modelStateDistributed[[i]][[a]]) %>%
        select(ID, PE)
    }
  }

  ## Distribute the edges
  for (e in edgeTypes) {
    if (nrow(modelState[[e]]) > 0) {
      for (i in 1:numPEs) {
        ## Find for each PE the edges where an agent of this PE is involved ..
        fID <- modelState[[e]] %>% filter(fromID %in% PElists[[i]]$ID)
        tID <- modelState[[e]] %>% filter(toID %in% PElists[[i]]$ID)
        ## ... and add those edges to the distributed state
        modelStateDistributed[[i]][[e]] <- bind_rows(fID, tID) %>% unique
        ## Also add the PEs of the agents on the other edge pos to the PElists
        PElists[[i]] %<>% bind_rows(fID %>% transmute(ID = toID) %>%
        ↪  left_join(PElist, by = "ID")) %>% unique
        PElists[[i]] %<>% bind_rows(tID %>% transmute(ID = fromID) %>%
        ↪  left_join(PElist, by = "ID")) %>% unique
      }
    }
  }

  list(modelStateDistributed, PElists)
```

```
}

# create a unique name for the single raster nodes
.createRasterID <- function(raster, x, y) {
  paste0(raster,"x", x,"y", y)
}

# create for a raster a tibble that contains the distribution
# of the nodes to PEs. In the current implementation, we distribute
# all nodes to PE 1, as we don't have a real parallel implemention anyway
.createRasterLayer <- function(sim, rname, rdata, distance) {
  ## create a unique name for the table of PEs for the nodes of a raster
  PEsName <- paste0(rname, "PEs")

  ## Iterate over all elements of the raster and add the
  ## PE information to the sim environment
  for (y in seq(dim(rdata)[1])) {
    for (x in seq(dim(rdata)[2])) {
      id = .createRasterID(rname, x, y)
      sim[[PEsName]] %<>% bind_rows(tibble(ID = id, PE = 1))
    }
  }
  sim
}

## a platform helper function for the initialization, that takes a non-distributed
## model state where the nodes does not have an ID, and add the node tables
## a column "ID", which contains unique IDs for each node
withStringID <- function(modelState) {
  modelState %>%
    imap(~ rowid_to_column(.x, var = "ID") %>% mutate(ID = paste0(.y,ID)))
}
```

3.3 Example

To show the internal structure of the `sim` environment[1], in the following we create such an environment with a simple model state, that consists of three agents/nodes in the **Interagent Layer** and a single 2x2 raster as **Spatial Layer**. The **Global Layer** contains a node for the model parameter and defines the type `OUTPUT` for the output of the simulation. As in the initial state we don't have already any output, we do not define any globalInitFunc, the `OUTPUT` node will be created later.

```
initStateFunc <- function(modelParams) {
    list("AGENTS" = tibble(x = 1:3,
                           GRASSx = c(1,2,1),
                           GRASSy = c(3,2,3)),
         "NETWORK" = tibble(fromID = c("AGENTS1", "AGENTS1"),
                            toID =   c("AGENTS2", "AGENTS3")))
}

initRasterFunc <- function(modelParams) {
    list("GRASS" = matrix(1:9, nrow = 3, ncol = 3))
}


sim <- initSimulation(agentTypes = "AGENTS",
                      edgeTypes = "NETWORK",
                      modelParams = tibble(numPEs = 3, rasterGrowth = 2),
                      stateInitFunc = compose(withStringID, initStateFunc),
                      globalTypes = "OUTPUT",
```

[1] What we show here are platform internal details that should never be used directly by the model implementation (and will likly be `private` in the later revisions).

```r
                        rasterTypes = "GRASS",
                        rasterInitFunc = initRasterFunc)
```

3.3.1 Interagent Layer

Lets take a look at the most complex structure that represents the **Interagent Layer**. As mentioned in a comment inside the `initSimulation`, `sim$distributed` contains one vector (`sim$distributed[[1]]`) for the model state of the **Interagent Layer**.

```r
sim$distributed[[1]] ## or sim$getExplicitLayer()
```

```
[[1]]
[[1]]$AGENTS
# A tibble: 1 x 5
     PE ID          x GRASSx GRASSy
  <dbl> <chr>   <int>  <dbl>  <dbl>
1     1 AGENTS1     1      1      3

[[1]]$NETWORK
# A tibble: 2 x 3
  ID       fromID  toID
  <chr>    <chr>   <chr>
1 NETWORK1 AGENTS1 AGENTS2
2 NETWORK2 AGENTS1 AGENTS3


[[2]]
[[2]]$AGENTS
# A tibble: 1 x 5
     PE ID          x GRASSx GRASSy
  <dbl> <chr>   <int>  <dbl>  <dbl>
1     2 AGENTS2     2      2      2

[[2]]$NETWORK
# A tibble: 1 x 3
  ID       fromID  toID
  <chr>    <chr>   <chr>
1 NETWORK1 AGENTS1 AGENTS2


[[3]]
[[3]]$AGENTS
# A tibble: 1 x 5
     PE ID          x GRASSx GRASSy
  <dbl> <chr>   <int>  <dbl>  <dbl>
1     3 AGENTS3     3      1      3

[[3]]$NETWORK
# A tibble: 1 x 3
  ID       fromID  toID
  <chr>    <chr>   <chr>
1 NETWORK2 AGENTS1 AGENTS3
```

We can see, that the state of the layer is distributed to three different PEs, e.g. index `[[2]]` shows the state that is available at PE 2, in this case the state of `AGENT2` and the edge from `AGENT1` to `AGENT2`. We can also check for which nodes the PEs are known on this PE 2:

```
sim$distributed[[2]][[2]] ## or sim$getPEs()[[2]]
```

```
  ID      PE
1 AGENTS2 2
2 AGENTS1 1
```

As there is no node between `AGENT2` (which is the only one assigned to PE 2) and `AGENT3`, PE 2 does not know anything about `AGENT3`. This has also the implication, that in the transition functions, which are described later, `AGENT2` can not create an edge to `AGENT3`.

3.3.2 Spatial Layer

The **Spatial Layer** is not virtually distributed to different PEs, as we currently also plan to implement it later in a way that all PEs knows the complete state of the **Spatial Layer**, assuming that the data size of this layer is considerably smaller then the of the `Interagent Layer`.[2] So when we take a look at the **Spatial Layer** we see only our small 2x2 matrix that we created in the `initRasterFunc`:

```
sim$raster ## or sim$getRasterLayer()
```

```
$GRASS
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

But the `.createRasterLayer` function also added anadditional tables per raster to the `sim` environment, which is automatically named. This table contains the information, which PE calculates an update for those nodes.

```
sim$GRASSPEs
```

```
  ID        PE
1 GRASSx1y1 1
2 GRASSx2y1 1
3 GRASSx3y1 1
4 GRASSx1y2 1
5 GRASSx2y2 1
6 GRASSx3y2 1
7 GRASSx1y3 1
8 GRASSx2y3 1
9 GRASSx3y3 1
```

In the current implementation this isn't used and all PEs are set to 1. In later revisions, different distribution schemas will be available. The corresponding nodes incl. there state are only constructed temporarly on demand.

[2] But in a later revisions of the platform, it should be possible to select between a distributed and non-distributed version, so that the modeller can check which fits/perform better to her model implementation. One advantage of the architecture of this platform is, that those decisions are transparent to the model code itself.

3.3.3 Global Layer

The **Global Layer** is simple and just contains a node (or an empty set) for each `globalType`. Like for the rasters, this value is available on all virtual PEs. Helper functions like `sim$reduce` will be available to guarantee the synchronization of the state between the different PEs. As we havn't create our `OUTPUT` node yet, the **Global Layer** consists only of the model parameters, which where added automatically by the `initSimulation` function.

```
sim$global
```

```
$MODELPARAMS
# A tibble: 1 x 2
  numPEs rasterGrowth
   <dbl>        <dbl>
1      3            2
```

4 State Transition

For each layer of the graph the platform provides a function that can be called to calculate a new state for all the nodes of the corresponding layer.

4.1 Interagent Layer

4.1.1 Concept

Let Ψ be the set of all agents of the **Interagent Layer**. Each agent has a transition function (or a set of transition functions) that construct a set of agents and edges, so we have:

$$\begin{aligned} t_\psi \quad &: \quad \Theta_t \to \Theta_{\psi,t} \\ &\text{where } \Theta_{\psi,t} = \prod_{\delta_a \in \Delta_A} \Theta_{\delta_a}^{n_{\phi,\delta_a,t}} \times \prod_{\delta_e \in \Delta_E} \Theta_{\delta_e}^{n_{\phi,\delta_e,t}} \end{aligned} \tag{1}$$

where $n_{\phi,\delta,t}$ is the number of agents/edges that the agent ϕ has constructed in her transition function. We can then combine individual transition functions to a comprehensive transition function by concatenating the constructed elements.

$$t = (t_{\psi_1}, \ldots, t_{\psi_{n_t}}) : \Theta_t \to \Theta_{t+1} \tag{2}$$

So, in contrast to the usual object-oriented ABM frameworks, the state of objects is not modified by the transition function, but completely recreated by the transition function.

4.1.2 Parallel Execution

The already mentioned the system described above should be distributed among different processing elements (PE) in such a way that each PE only receives a subset of the overall system state. Therefore, a transition function is always connected to a set of networks formed by the edges of a type $\delta_e \in \delta_E$. This can be also a raster from the **Spatial Layer**, in this case implicit edges will be added from the node corresponding to the position of the agent and also the Moore neighborhood.

For a single agent ψ only the state of agents $\hat{\psi}_i$ is available for which an edge $(\hat{\psi}_i, \psi)$ exists in one of the associated networks of the transition function. So each PE does only know some part of the overall model state, namely:

- The state of the agents on this PE.

- The edges of the graph, where an agent is on tail or head position.
- The nodes of the **Global Layer** and the **Spatial Layer**.
- While a transition function is called: The state of agents on the tail position of the network associated with the transition function

To apply a transistion function, the following steps are taken:

- First we must ensure that for all explicit edges of the networks associated with the transition function, the state of agents on the tail position is known on the PE of the agents on the head position.
- We also check which agents of an edge of PE x are on another PE y, and send those edges to this PE.
- In the case that one of the associated networks is a raster, create temporarly nodes for the position of the agent and for the Moore neighborhood.
- Then the transition function is applied
- We construct the new model state by gathering the returned lists of agents and edges.

4.1.3 Code

To modify the state of the agents and edges in the **Interagent Layer** we call the platform function `applyTransition`. Thereby the parameter `networks` determines which networks and rasters are associated with this transition function. The state of nodes and edges of the **Interagent Layer** which have a type listed in `invariantTypes` are retained and cannot be changed by the agents in this transition. And `distance` is the Moore Distance used for the temporary construction of raster nodes as explained in section 4.2.

```
applyTransition <- function(sim,
                            transitionFunction,
                            networks = NULL,
                            invariantTypes = NULL,
                            distance = 1) {
  ## ensure that the edges are known
  sim$distributed <- .distributeNetworkState(sim)

  ## in the case that the network is a raster, we must add the
  ## PEs of the nodes of this raster to the known PEs table
  for (n in networks) {
    PEsName <- paste0(n, "PEs")
    if (! is.null(sim[[PEsName]])) {
      for (i in 1:sim$numPEs) {
        sim$distributed[[2]][[i]] %<>% bind_rows(sim[[PEsName]])
      }
    }
  }

  modelStateDistributed <- sim$getExplicitLayer()
  ## The .enhanceAgentStates function add that the state of agents
  ## at the tail position of one network to the corresponding PE.
  if (! is.null(networks)) {
    enhancedModelStateD <- .enhanceAgentStates(sim, networks)
  } else {
    enhancedModelStateD <- sim$getExplicitLayer()
  }

  ## Now we simulate the parallel processing of the transition
  ## function by iterating of the different distributed model state
```

```
  for (i in 1:sim$numPEs) {
    ## we first store the state of the agents or edges which
    ## are invariant in this transition
    invariantState <- invariantTypes %>%
        map(~ modelStateDistributed[[i]][[.x]]) %>%
        set_names(invariantTypes)

    ## call all agents to apply their transitionFunction
    modelStateDistributed[[i]] <- sim$agentTypes %>%
      map(~ .transform(sim,
                       .x,
                       modelStateDistributed[[i]][[.x]],
                       enhancedModelStateD[[i]],
                       transitionFunction,
                       networks,
                       distance)) %>%
      .consolidate(sim)

    ## for the invariant state, add this to the state constructed
    ## by the agents (in the case, that the agents also returned
    ## entities of an invariantType, those will be overwritten)
    for (n in invariantTypes)
        modelStateDistributed[[i]][[n]] <- invariantState[[n]]
  }
  sim$distributed[[1]] <- modelStateDistributed

  ## allow method chaining by returning the sim environment
  sim
}

}
```

As mentioned above, we must first ensure that all explicit edges of the edge types given in the parameter `network`, the state of agents on the tail position is known on the PE of the agents on the head position. This is done in the function `.distributeNetworkState`:

```
## ~.distributeNetworkState~ takes the result of a transition
## function, and sends the edges to the PEs of the agents on the head
## or tail position.
.distributeNetworkState <- function(sim) {
  modelStateDistributed <- sim$getExplicitLayer()
  PElists <- sim$getPEs()
  modelStateEnhanced <- modelStateDistributed
  ## create new empty lists
  newPElists <- PElists %>% map(~ .x[0,])

  for (i in 1:sim$numPEs) {
    for (e in sim$edgeTypes) {
      ## We allow the creation of Networks while the simulation is running
      if (! is.null(modelStateDistributed[[i]][[e]]) &&
          nrow(modelStateDistributed[[i]][[e]]) > 0) {
        ## Create two version of the edge tibble, ~matchFrom~ is extended with
        ## the PEs of the tail, ~matchTo~ with the PEs of the head agents
        matchFrom <- inner_join(modelStateDistributed[[i]][[e]],
                                PElists[[i]], by = c("fromID" = "ID"))
        matchTo <- inner_join(modelStateDistributed[[i]][[e]],
                              PElists[[i]], by = c("toID" = "ID"))
        ## sendNetworkTable is a tibble: fromID toID PE
        sendNetworkTable <- bind_rows(matchFrom, matchTo)
        ## sendPElist is a tibble: ID, PE, toPE
        sendPElist <- bind_rows(sendNetworkTable %>% select(ID = fromID,
                                                           toPE = PE),
                                sendNetworkTable %>% select(ID = toID,
```

```
                                                          toPE = PE)) %>%
          inner_join(PElists[[i]], by = "ID")

        ## Use the send... tibbles to distribute the edges to the PEs
        ## and also to send the (agentID, PE) information to those PEs
        for (j in 1:sim$numPEs) {
          if (i != j) {
            modelStateEnhanced[[j]][[e]] %<>%
              bind_rows(sendNetworkTable %>% filter(PE == j) %>% select(-PE)) %>%
              unique
          }
          newPElists[[j]] %<>%
            bind_rows(sendPElist %>% filter(toPE == j) %>% select(-toPE)) %>%
            unique
        }
      }
    }
  }

  list(modelStateEnhanced, newPElists)
}
```

As an agent should have the opportuniy to add an edge to a node of a raster, we then also add the PEs of those nodes to the list of known PEs. This is done in the second block of the `applyTransition` function.

The third block calls `.enhanceAgentStates` in the case, that the transition function was associated with at least on network or raster. This fullfills the requirement, that the state of agents on the tail position of an edge in the `networks` parameter list is available on the PE at the head position of this edge.

```
## The ~.enhanceAgentStates~ function ensures that the state of
## agents at the tail position of the network associated with the next
## transition function is available on the PEs of the agents at the
## head position of the edges. This function is called before the
## transition function itself, which then gets the returned extendend
## model state.
.enhanceAgentStates <- function(sim, edgeTypes) {
  modelStateDistributed <- sim$getExplicitLayer()
  PElists <- sim$getPEs()
  modelStateEnhanced <- modelStateDistributed
  for (e in edgeTypes) {
    for (i in 1:sim$numPEs) {
      ## We allow the creation of Networks while the simulation is running
      if (! is.null(modelStateDistributed[[i]][[e]]) &&
          nrow(modelStateDistributed[[i]][[e]]) > 0) {
        ## agentPEmap is a tibble(fromID, toID, PE of toID agent)
        agentPEmap <- modelStateDistributed[[i]][[e]] %>%
          select(fromID, toID) %>%
          left_join(PElists[[i]], by = c("toID" = "ID")) %>%
          unique

        for (a in sim$agentTypes) {
          ## iterate over the other PEs
          for (j in (1:sim$numPEs %>% discard(. == i))) {
            ## find the agents that must be send to other PEs ...
            agentsIDtoSend <- agentPEmap %>% filter(PE == j) %>% select(fromID)
            agentsToSend <- agentsIDtoSend %>%
              inner_join(modelStateDistributed[[i]][[a]], by = c("fromID" = "ID"))
              ↪  %>%
              unique %>%
              rename(ID = fromID)
            ## ... and add them to the state of this other PE
            modelStateEnhanced[[j]][[a]] %<>% bind_rows(agentsToSend)
```

```
        }
      }
    }
  }
}

  modelStateEnhanced
}
```

The last block of the `applyTransistion` calls for every type of `agentTypes` the `.transform` function:

```
## ~.transform~ takes one agent type and iterates over the agents of
## this type, calling for each agent one of the transition
## functions. Inside of ~.transform~ we use the ~limitState~ function
## to ensure, that only the state of agents in the tail position of
## the associated network is accessable.
.transform <- function(sim,
                       type,
                       agents,
                       modelState,
                       transitionFunc,
                       networks,
                       distance = 1) {
  ## create the raster nodes that are embedded in the part of the model
  ## state that is visible for the agent
  createLocationNodes <- function(agent) {
    rasterTables <- NULL
    for (n in networks) {
      if (n %in% sim$rasterTypes) {
        ## in the case that the network is a raster, we must create
        ## the explicit network first
        rasterTables[[n]] <- NULL
        r <- sim$raster[[n]]

        x <- agent[[paste0(n,"x")]]
        ## it's optional to assign agents to a raster, so only
        ## continue, if this is really the case
        if (! is.null(x)) {
          y <- agent[[paste0(n,"y")]]
          ## calculate the area of the Moore neighborhood (with borders)
          fromx <- max(x - distance, 1)
          tox <- min(x + distance, dim(r)[2])
          fromy <- max(y - distance, 1)
          toy <- min(y + distance, dim(r)[1])

          ## iterate over this area and create nodes the nodes
          for (x in seq(fromx, tox)) {
            for (y in seq(fromy, toy)) {
              id = .createRasterID(n, x, y)
              rasterTables[[n]] %<>% bind_rows(tibble(ID = id,
                                                      x = x,
                                                      y = y,
                                                      value = r[[y, x]]))
            }
          }
        }
      }
    }
    rasterTables
  }

  ## ensure, that only the state of agents in the tail position of the
  ## associated network is accessable.
```

```
  limitState <- function(agent) {
    agentTables <- NULL
    if (! is.null(networks)) {
      for (n in networks) {
        ## a network name can be also a raster, but the raster nodes
        ## are allready handled in createLocationNodes above
        if (!(n %in% sim$rasterTypes)) {
          incoming <- modelState[[n]] %>% filter(toID == agent$ID)

          if (incoming %>% length > 0) # check that we have any edge
            ## filter the agents, so that only agents which are
            ## in the ~incoming~ tibble are part of the ~agentTables~
            agentTables <- sim$agentTypes %>%
              map(~ bind_rows(agentTables[[.x]],
                              modelState[[.x]] %>% filter(ID %in%
                              ↪   incoming$fromID))) %>%
              setNames(sim$agentTypes)
        }
      }
    }

    ## filter the network to edges where the agent is on tail or head position
    availableEdgeTypes <- sim$edgeTypes %>% keep(~ .x %in% networks)
    edgeTables <- availableEdgeTypes %>%
      map(~ modelState[[.x]] %>%
            (function(x) { if (is.null(x)  nrow(x) == 0) NULL else
                                             { x %>% filter(agent$ID == fromID |
                                                            agent$ID == toID)}})) %>%
      setNames(availableEdgeTypes)

    ## create the overall available modelstate for the agent by concatenating
    ## the filtered nodes and edges from the Interagent Layer, the created nodes
    ## from the Spatial Layer and all nodes from the Global Layer
    c(agentTables, edgeTables, createLocationNodes(agent), sim$global)
  }

  ## iterate over all agents and call their transition function with the part
  ## of the model state, that is accessable by the agent
  agents %>%
    transpose %>%
    map(~ transitionFunc(as_tibble(.), type, limitState(.x)))
}
```

As each agent return a seperate list of agents and edges, those must be combined into a single table per type. This is done in the `.consolidate` function:

```
## ~.consolidate~ gets the collected output of ~.transform~ from all
## agents in the system, so ~newModelState~ is a list (with the agent
## types as keys) of lists (with the agents as keys) of tables (the
## tables returned by a transition function), and reduces this to a
## single table per type.
.consolidate <- function(newModelState, sim, oldModelState) {
  flattenLists <- newModelState %>% flatten

  c(sim$agentTypes, sim$edgeTypes) %>%
    map(~ { flattenLists %>% map(.x) %>% bind_rows }) %>%
    setNames(c(sim$agentTypes, sim$edgeTypes))
}
```

The parameter `transitionFunction` is the `applyTransition` function is itself a function that has the form `function(agent, type, modelState)`. This function is called for every agent in the **Interagent Layer**. The function parameter `agent` contains the state of a single agent with the type `type`. The parameter `modelState` contains the part of the overall model state that is accessable for the agent.

4.1.4 Example

As an example for the `applyTransition` function, we create a transition function, that does nothing but printing the accessable model state for the agent with the id `AGENTS3`.

```
showModelState <- function(agent, type, modelState) {
  if (agent$ID == "AGENTS3") {
    print(modelState)
  }
  list()
}

applyTransition(sim, showModelState, c("NETWORK", "GRASS"))
```

```
$AGENTS
# A tibble: 1 x 5
     PE ID          x GRASSx GRASSy
  <dbl> <chr>   <int>  <dbl>  <dbl>
1     1 AGENTS1     1      1      3

$NETWORK
# A tibble: 1 x 3
  ID       fromID  toID
  <chr>    <chr>   <chr>
1 NETWORK2 AGENTS1 AGENTS3

$GRASS
# A tibble: 4 x 4
  ID            x     y value
  <chr>     <int> <int> <int>
1 GRASSx1y2     1     2     2
2 GRASSx1y3     1     3     3
3 GRASSx2y2     2     2     5
4 GRASSx2y3     2     3     6

$MODELPARAMS
# A tibble: 1 x 2
  numPEs rasterGrowth
   <dbl>        <dbl>
1      3            2
```

We can see, that this agent can access the state of `AGENT1` as there is a edge in `NETWORK` from this agent to her. Also only the part of the raster in his neighborhood is accessable by `AGENT3`.

We can also see, that we don't have any agents and edges left in the **Interagent Layer** after this transition, as the transition function has only returned an empty list:

```
sim$getExplicitLayer()
```

```
[[1]]
[[1]]$AGENTS
# A tibble: 0 x 0

[[1]]$NETWORK
# A tibble: 0 x 0
```

```
[[2]]
[[2]]$AGENTS
# A tibble: 0 x 0

[[2]]$NETWORK
# A tibble: 0 x 0


[[3]]
[[3]]$AGENTS
# A tibble: 0 x 0

[[3]]$NETWORK
# A tibble: 0 x 0
```

4.2 Raster

4.2.1 Concept

The state transition of the **Spatial Layer** is much easier than that of the **Interagent Layer**. The transition function is called for each node/position of a single raster, and as with the transition function of the **Interagent Layer**, the model state known in the transition function is limited to parts of the entire model state. This time we can associate the transition function with agent types of the **Interagent Layer** and with edge types. For the agents of an associated agent type, we check their positions on the `raster`, which must be part of the agent state. The state of agents with the same position as the raster nodes are then available to the raster node. Edges are handled in the same way as the **Interagent Layer** state transition, an edge is visible if the raster node is part of the edge.

The transition function must return a list that must have an entry `VALUE` containing the new value for the grid position. Optionally, the list can also contain arbitrary edges of arbitrary edge types. In the case that an edge is returned for an edge type δ_e, the complete network of that type is created by the sum of all edges returned. But unlike the transition function in the **Interagent Layer**, networks that are invariant for this transition function do not need to be reconstructed by the transition function itself.

4.2.2 Code

```
applyRasterTransition <- function(sim, raster, transitionFunction,
                                  agentTypes = NULL,
                                  networks = NULL) {
  el <- sim$getExplicitLayer()
  ## generate the column names for the raster position in the
  ## agent state
  xName <- paste0(raster, "x")
  yName <- paste0(raster, "y")

  ## update the value of the node at the x,y position as a side effect and
  ## return the edges constructed by the transition function
  updatePos <- function(x, y) {
    ## create a temporary node for this raster position
    nodeName <- .createRasterID(raster, x, y)
    node <- tibble(ID = nodeName, x = x, y = y, value = sim$raster[[raster]][[y,
    ↪  x]])

    ## gather all the agent of the given agentTypes that are located
    ## on this position
    agentTables <- NULL
```

```
    for (t in agentTypes) {
      for (i in 1:sim$numPEs) {
        agentTables[[t]] %<>%
          bind_rows(el[[i]][[t]] %>% filter(get(xName) == !!x, get(yName) == !!y))
      }
    }

    ## gather all edges of the given networks which have the raster node on
    ## the head or tail position
    edgeTables <- NULL
    for (t in networks) {
      for (i in 1:sim$numPEs) {
        if (! is.null(el[[i]][[t]])) {
          fromNode <- el[[i]][[t]] %>% filter(fromID == !!nodeName)
          toNode <- el[[i]][[t]] %>% filter(toID == !!nodeName)
          edgeTables[[t]] %<>% bind_rows(fromNode)
          edgeTables[[t]] %<>% bind_rows(toNode)
          ## in the case, that the raster node is on the head position, we
          ## must also add the PE of the node on the tail position to the
          ## list of known PEs
          if (nrow(toNode) > 0)
            sim$distributed[[2]][[1]] %<>%
              bind_rows(toNode %>% transmute(ID = fromID, PE = i)) %>% unique
        }
      }
    }

    ## call the transition function with the gathers agents, edges and the
    ## global layer
    r <- transitionFunction(node, c(agentTables, edgeTables, sim$global))
    ## update the value
    sim$raster[[raster]][[y, x]] <- r[["VALUE"]]
    ## and remove the value from the list, so that we return only the edges
    r[["VALUE"]] <- NULL
    r
  }

  ## the updatePos function can return edges, we gather them in the
  ## combined list
  combined <- list()
  ## iterate over all positions
  for (y in seq(dim(sim$raster[[raster]])[1])) {
    for (x in seq(dim(sim$raster[[raster]])[2])) {
      ## updatePos can return lists with edges of arbitrary networks
      network <- updatePos(x, y)
      ## check for which networks edges was in the returned list
      networkTypes <- names(network)
      ## and add those to the combined
      combined <- networkTypes %>%
                    map(~ bind_rows(combined[[.x]], network[[.x]])) %>%
                    setNames(networkTypes)
    }
  }
  ## clear all networks for which we got at least on edge returned
  ## by the transition function
  for (n in names(combined))
    sim$distributed[[1]][[1]][[n]] <- NULL
  ## and add the new networks instead
  sim$distributed[[1]][[1]] %<>% append(combined)

  ## allow method chaining by returning the sim environment
  sim
}
```

4.2.3 Example

Again, we create a transition function that outputs the retrievable model state, this time for the raster node `GRASSx1y3`. But since the method `applyRasterTransition` is not part of the tutorial mentioned at the beginning, we will also show a possible useful transition function this time.

Let us assume that the `AGENTS` are sheep that want to eat grass. In the case that more than one sheep is at the same place, they have to share the amount of grass that this node holds. If there is no sheep on this node, the grass grows `MODELPARAMS$grassGrowth` units, otherwise the amount of grass is set to 0 because it was eaten by the sheep(s).

Since the sheep state cannot be changed in the transition function itself, edges to the sheep are generated. The edges have as state the amount of grass the sheep has eaten.

```
sim <- initSimulation(agentTypes = "AGENTS",
                      edgeTypes = "FEED",
                      modelParams = tibble(numPEs = 3, grassGrowth = 2),
                      stateInitFunc = compose(withStringID, initStateFunc),
                      globalTypes = "OUTPUT",
                      rasterTypes = "GRASS",
                      rasterInitFunc = initRasterFunc)

updateRaster <- function(rasterNode, modelState) {
   if (rasterNode$ID == "GRASSx1y3") {
     print(modelState)
   }
   list("VALUE" = if (nrow(modelState$AGENTS) > 0) 0
                  else rasterNode$value + modelState$MODELPARAMS$grassGrowth,
        "FEED" = modelState$AGENTS %>%
                   transmute(fromID = rasterNode$ID,
                             toID = ID,
                             value = rasterNode$value / nrow(modelState$AGENTS)))
}

sim %>% applyRasterTransition("GRASS", updateRaster, "AGENTS")
```

```
$AGENTS
# A tibble: 2 x 5
     PE ID           x GRASSx GRASSy
  <dbl> <chr>    <int>  <dbl>  <dbl>
1     1 AGENTS1      1      1      3
2     3 AGENTS3      3      1      3

$MODELPARAMS
# A tibble: 1 x 2
  numPEs grassGrowth
   <dbl>       <dbl>
1      3           2
```

As we can see, our example node `GRASSx1y3` can only see the two `AGENTS` which have the model state `GRASSx == 1` and `GRASSy == 3`. And, as usual, additionally the **Global Layer**. Let's have a look at the raster:

```
sim$raster$GRASS
```

```
     [,1] [,2] [,3]
[1,] 3    6     9
[2,] 4    0    10
[3,] 0    8    11
```

The amount of grass developed as expected. Finally, we also check the **Interagent Layer**. We are mainly interested in the `FEED` edges, and since the nodes of the grass layer are all mapped to PE 1, it is sufficient to check the part of the **Interagent Layer** on PE 1:

```
sim$getExplicitLayer()[[1]]
```

```
$AGENTS
# A tibble: 1 x 5
     PE ID          x GRASSx GRASSy
  <dbl> <chr>   <int>  <dbl>  <dbl>
1     1 AGENTS1     1      1      3

$NETWORK
# A tibble: 2 x 3
  ID       fromID  toID
  <chr>    <chr>   <chr>
1 NETWORK1 AGENTS1 AGENTS2
2 NETWORK2 AGENTS1 AGENTS3

$FEED
# A tibble: 3 x 3
  fromID     toID    value
  <chr>      <chr>   <dbl>
1 GRASSx2y2 AGENTS2   5
2 GRASSx1y3 AGENTS1   1.5
3 GRASSx1y3 AGENTS3   1.5
```

4.3 Global

4.3.1 Concept

The **Global Layer** contains nodes that should be available on all PEs. Also The transition function for the nodes of the **Global Layer** should have access to all nodes.

For such operations it is therefore very obvious to use MPI functions like MPI_ALLREDUCE and to provide wrappers for them. The current implementation of the platform has an example of such a wrapper, namely the function `reduce`, which is added to the `sim` environment in an `initSimulation` call:

```
## reduce is a helper function to calculate global values based on the state
## of the nodes in the *Interagent Layer*.
sim$reduce <- function(type, agentFunc, aggregationFunc) {
  sim$getExplicitLayer() %>% map(type) %>% map(agentFunc) %>% unlist %>%
  ↪  aggregationFunc
}
```

4.3.2 Code

The transition function for a global node only receives the global node and the current state of the simulation and must return the new state of the global node. For the platform itself, the state of global nodes is transparent, so that the state can also be a table with a changing number of rows, e.g. with one row for each time step.

The `updateGlobal` function exists mainly to hide the internal data structure of the sim environment:

```
updateGlobal <- function(sim, type, func) {
  sim$global[[type]] <- func(sim$global[[type]], sim)

  ## allow method chaining by returning the sim environment
  sim
}
```

4.3.3 Example

As an example we create a transition function, that calculate the sum of all `x` values of all `AGENTS` and add this sum as a new row to a tibble[3]:

```
observeSumX <- function(current, sim) {
    current %<>% bind_rows(tibble(sumX = sim$reduce("AGENTS", "x", sum)))
}

updateGlobal(sim, "OUTPUT", observeSumX)
```

As we can see, the **Global Layer** has now also a new node `OUTPUT`, which contains the sum.

```
sim$getGlobalLayer()
```

```
$MODELPARAMS
# A tibble: 1 x 2
  numPEs grassGrowth
   <dbl>       <dbl>
1      3           2

$OUTPUT
# A tibble: 1 x 1
   sumX
  <int>
1     6
```

We can call the transistion function repeatedly. Since the state of the agents did not changed between thw two calls, we get additional rows with the same value:

```
updateGlobal(sim, "OUTPUT", observeSumX)
sim$getGlobalLayer()
```

```
$MODELPARAMS
# A tibble: 1 x 2
  numPEs grassGrowth
   <dbl>       <dbl>
1      3           2

$OUTPUT
# A tibble: 2 x 1
   sumX
  <int>
1     6
2     6
```

[3]When we call updateGlobal for the first time, `current` equals `NULL`, and `bind_rows` creates a new tibble.