NetSim: A Social Networks Simulation Tool in R

Christoph Stadtfeld
University of Groningen

Abstract

NetSim is an R package that allows to simulate the co-evolution of social networks and individual attributes. It can be used to study the impact of micro models that describe the behavior of individuals on the macro outcome of social networks. **NetSim** is based on a flexible Markov framework that enables the combination of a variety of different models.

Keywords: social networks, simulation, micro-macro link, stochastic actor-oriented models, R.

1. Introduction

One of the key challenges in today's social networks research is to understand the link between dynamic micro-models that describe behavior of individuals and macro-outcomes that describe social networks as a whole. In this article, I introduce **NetSim** (Stadtfeld 2013), a flexible R package (R Core Team 2013) that allows to combine and simulate a variety of micro-models to research their impact on the dynamic macro-features of social networks.

For illustration of micro-macro-relations in social networks consider two examples:

First, homophily is the tendency of individuals to form social network ties with similar others. Empirical evidence for this micro-behavior has been found on a variety of attributes such as age, gender, ethnicity and opinions (McPherson, Smith-Lovin, and Cook 2001). But under which circumstances does homophily lead to the formation of antagonistic groups (this is the macro outcome)? The outcome of a dynamic homophily process is illustrated in figure 1(a)[1]. The formation of social network ties based on homophilic choices (as well as preference for reciprocity and triadic structures, see Kadushin (2012, pp. 21,24)) leads to a simulated social network in which nodes of the same type (white circle, square, black circle) slightly tend to cluster together.

Second, humans are known to be *influenced* by friends and peers in their social networks. Their habits, beliefs and behavior may change depending on the habits, beliefs and behaviors of those they are connected with in social networks. Under which circumstance, however, do influence micro-mechanisms "go viral" and spread through large social networks (macro outcome)? This second example is illustrated in figure 1(b). In a fixed social network[2] with equally distributed attributes, actors were simulated to influence one another. This leads to a predominance of two attributes (squares, white circles). The corresponding nodes are those

[1] All simulations in this paper are done using the **NetSim** package. Network plots were created with the **igraph** package in R (Csardi and Nepusz 2006).

[2] A regular ring lattice structure with two reciprocal neighbors per actor and some random rewiring; see Watts and Strogatz (1998) and section 4.4

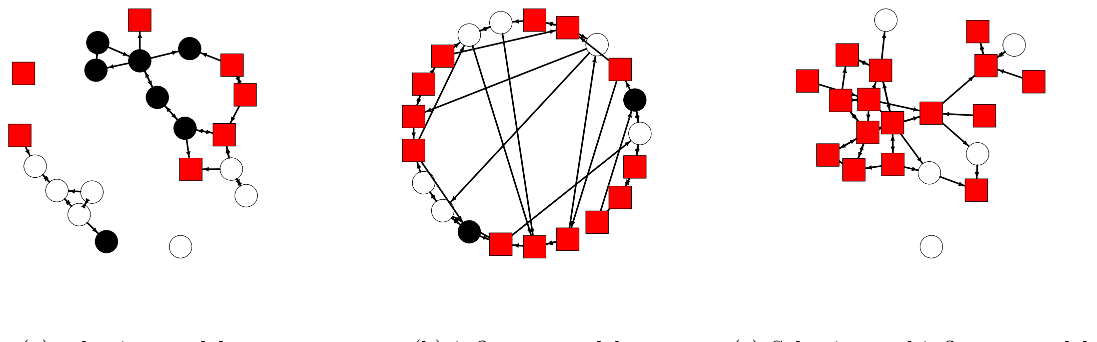

(a) selection model (b) influence model (c) Selection and influence model

Figure 1: Simulated social networks based on different combinations of micro-mechanisms. The three types of nodes (dark circle, square, white circle) indicate different attributes. a) Social network based on homophily (selection), reciprocity, and transitivity. The attributes are fixed. b) Fixed social network with changing attributes based on an influence mechanism. c) Social network based on both selection and influence mechanisms. The simulation of network a) is explained in section 4.1, the simulation of c) in section 4.2.

which are close to one another regarding distance in the network.

A number of publications from a broad variety of disciplines (such as sociology, epidemiology, physics, economics) have examined the link between micro-mechanisms and macro-outcomes in (social) networks by conducting simulation studies (see, for example, Flache and Macy (2011); Morris and Kretzschmar (1997); Watts and Strogatz (1998); Jackson and Rogers (2007)). In most cases these studies focus on one or very few micro-mechanisms. In real social networks, however, it is hard to dissect the role of one particular micro mechanism as a multitude of mechanisms interplay with one another. Homophily and influence effects from the previous two examples (see figures 1(a) and 1(b)) can often hardly be separated. Imagine the case of political opinions: There is a chance that opinions spread through social networks as people may convince their friends to adopt their own opinions. But also, social network ties are influenced by opinion homophily in the first place – people with similar opinions are more likely to become friends.

When simulating the homophily micro-model and the influence micro-model from the previous examples simultaneously, a possible dynamic outcome is shown in figure 1(c). Like in the first model, nodes of the same type are found to cluster together (the squares) whereas – like in the second example – some attributes disappear (black circles completely, white circles almost completely). Depending on the research question, this combined model may provide a much better description of how the underlying micro-mechanisms shape a social network on a macro-level.

A good social networks simulation tool should therefore support the flexible combination of a variety of micro mechanisms. Until now there was not such a flexible and extensible tool tailored to the research domain of social networks. **NetSim** closes this gap by introducing a generic meta-model that is defined as flexible Markov process framework. Dynamic models that are based on different concepts such as Poisson processes, round-based or deterministic time models can now be integrated in one framework and be simulated in one process model.

The central class of models in **NetSim** is the generic class of stochastic actor-oriented models (SAOMs) for social networks and behavioral attributes (Snijders 1996; Snijders, van de Bunt, and Steglich 2010; Steglich, Snijders, and Pearson 2010)). The new Markov framework is constructed as an extension and generalization of this class of models. In **NetSim**, SAOMs can be combined in one simulation with other more specific models such as infection models, network rewiring models, small world models or models for the joining and leaving of nodes. The software package also aims at facilitating the implementation of new models from different research areas.

A generic Markov model is introduced in section 2. The R interface of **NetSim** is introduced in section 3. Some micro-models are listed in section 4 and code examples are given. The implementation and architecture of **NetSim** is discussed in section 5. **NetSim** makes use of the R package **Rcpp** (Eddelbuettel and Francois 2011; Eddelbuettel 2013) to integrate C++ functions in R. Section 6 explains how the **NetSim** framework can be extended with additional models.

2. A generic Markov model

The simulation framework is developed as a generalization and extension of SAOMs for network and behavior change (Snijders *et al.* 2010). These models describe micro changes of social networks or behavioral attributes as discrete individual choices that are embedded in a continuous-time Markov process.

I adopt this idea and define a generic Markov process in which changes may occur continuously over time. Also the state space (which consists of the state of networks, individual and global attributes) may be continuous. However, changes of the state space are discrete and follow up to four generic steps.

1. A number of *time models* determine the time span until the next change takes place. These can be, for example, competing individual Poisson processes or deterministic round-based time models. The minimal realized time span of all time models is chosen and the following corresponding steps are triggered.

2. One or several *time updaters* update the process state based on the time span determined.

3. The selected time model then triggers one or more *change models* which stochastically or deterministically propose a change based on the process state. A change model does not yet change the process state, which allows having several independent changes per simulation step.

4. Each change model is linked to one or more *change updaters* which then update the process state based on the result of the corresponding change model.

An exemplary simulation four-step is shown in figure 2 and discussed below. In the following, I compare the four-step Markov framework to the the SAOM Markov framework and explain how it was extended.

In SAOMs for network changes (or change of individual attributes), the individual decision whether to add, remove or maintain a social networks tie (or to change an individual attribute) is modeled as a multinomial choice model of all discrete change options available. If an actor faces such a choice she can either add a tie to an actor she is not connected to, remove one of her outgoing ties to others or keep the situation unchanged. Each of these decisions will change the characteristics of the actor's local environment. The "utility" of a specific local environment is evaluated by measuring and weighting certain effect, like the number of ties, the degree of reciprocity, or the number of transitive structures the actor is embedded in.

The point in time when such a decision takes place is defined by a Poisson process. Both the Poisson rates and the choice functions can be parameterized and be dependent on the state of the process. In SAOMs, the process state is usually defined by one or several networks networks and one or several actor attributes.

I extend this model in different ways.

First, the state of the Markov process is extended to incorporate global attributes. This allows to simulate the change of global regimes, of certain periods (e.g., weekdays vs. weekend) or to make changes dependent on the process time. By this, the defined simulation models may still be considered a Markov processes but not necessarily a homogeneous one (being independent of the process time).

Second, not only Poisson rates are allowed to determine the moments in which potential changes of the network take place. By extending the process state with the process time, a time model triggering a change could be, for example, a round based time model in which changes happen after a fixed number of time units. The Markov property, stating that change probabilities may only depend on the current process state is not violated. **NetSim** can simulate round-based agent-based models and models depending on other deterministic time spans.

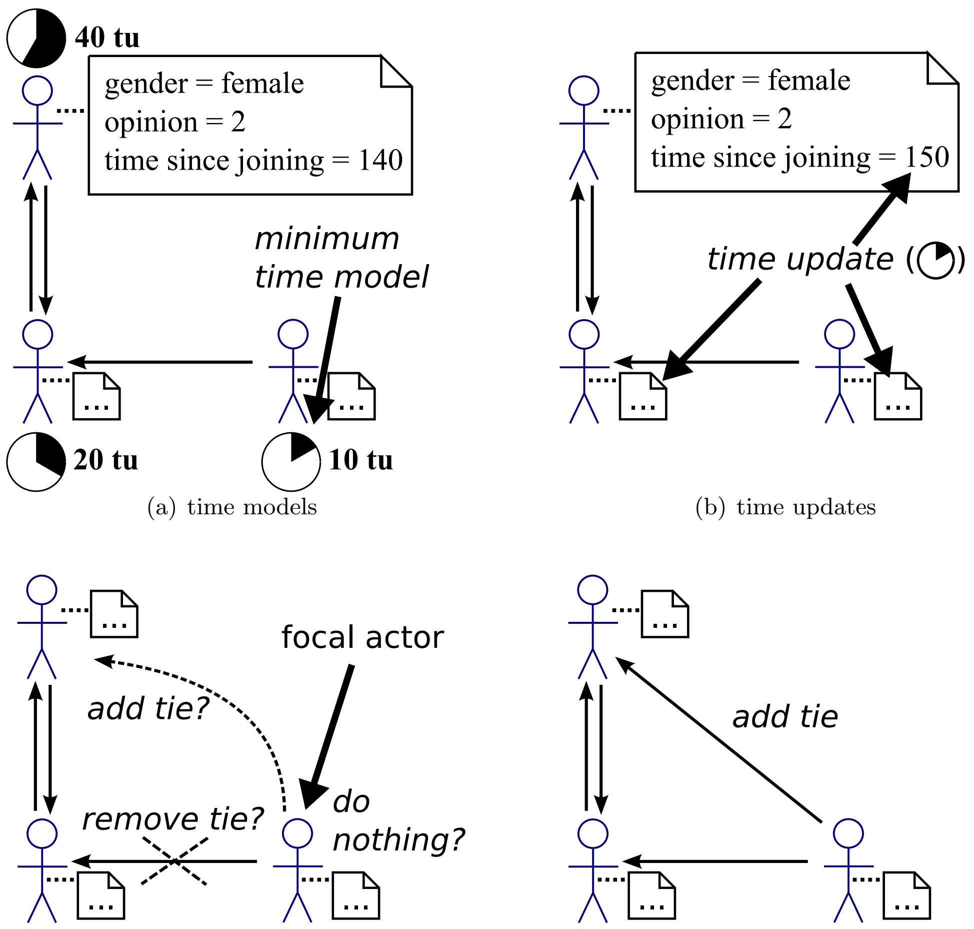
Third, after the determination of points in time at which potential changes take place, an additional time update of the process state is allowed. For example, the choice behavior of an actor may depend on the time since it joined, whereas the Poisson rates triggering this change may not. Also, it might be interesting to model valued network in which the strength of a tie decays over time (see Stadtfeld and Geyer-Schulz (2011)) and thereby influences sub-sequent change models. This can also be realized with a time update model.

Fourth, not only multinomial choice models are allowed as change models but any kind of model that triggers a later update of the process state. In this paper, I discuss models of Jackson and Rogers (Jackson and Rogers 2007) and Watts and Strogatz (Watts and Strogatz 1998) that define alternative micro models how network ties may change in social networks.

Fifth, the change update is separated from the change model to allow the occurrence of multiple changes at the same time which are independent from one another. Also, one change model can thereby trigger multiple updates. This is different in stochastic actor-oriented models in which an update will always swap a tie, or increase or decrease an individual attribute. In **NetSim**, additional change update can be simulated, for example, valued tie updates.

Sixth, it is possible to model heterogeneity of behavior between actors and heterogeneity of behavior over time. Changes of behavior over time can be modeled as a stochastic process and be described by change models and corresponding updates.

Seventh and finally, **NetSim** provides the possibility of simulating the joining and leaving of actors. The behavior of joining actors does not have to be pre-defined but can follow a stochastic process that may depend on the process state at the time of joining. This is enabled by marking the "youngest" node in the process state. Change models may assign a behavior to that node right after joining depending on the process state (e.g., the behavior of others).



(a) time models (b) time updates

(c) change models (d) change updates

Figure 2: An exemplary change in the four-step simulation framework. First, competing time models determine the length of the time span (10 tu = 10 time units). The minimum time span is chosen. Second, time updates are processed (here: an update of individual attributes). Third, the change models that correspond to the chosen time models determine a particular type of change (here: either adding a tie, removing a tie or doing nothing). Fourth, the change updates change the process state based on the result of the change model – a new tie is included.

Figure 2 shows one four-step of an exemplary simulation process. The process state consists of a network with three actors with each actor having three individual attributes (gender, opinion and time since joining). Thin arrows indicate ties in the social network. Attributes values are shown for one actor in the box on top. The actor is female, holds opinion 2 and joined the process state 140 time units ago. The model manager consists of three time models, each one indicating the time until one of the actors reconsiders her network configuration. In figure 2(a) the simulator determines the time until the next change. It is lowest for the actor in the lower right corner (10 time units compared to 20 and 40 time units). This actor is the focal actor of the simulation step. Before she re-considers the network configuration, one time update is applied as shown in figure 2(b). Based on the time span of 10 time units the individual attribute 'time since joining' of each actor is increased by 10. After this, the focal actor re-considers her network configuration based on the now updated process state. This choice is defined as a change model. The actor may remove the tie to the lower-left actor, add a tie to the upper-left actor of keep the network unchanged (figure 2(c)). In this example, she decides to add a tie. As in principle more change models could follow, the update is not yet applied but only in a separate change update step shown in figure 2(d).

3. Using NetSim

3.1. Installation

The installation of the **NetSim** package works straighforwardly via CRAN or R-Forge[3]. The package **Rcpp** for the R/C++ integration needs to be installed as well.

```
R> install.packages(c("NetSim", "Rcpp"))
R> library("NetSim")

Loading required package: Rcpp
```

3.2. Defining a process state

An empty process state can be created with the command

```
R> processState <- create_process_state()
```

Networks are per default created based on (quadratic) R matrixes. An complete network with 5 nodes can be added to the process state as follows:

```
R> nActors <- 5
R> network <- create_network(matrix(1, nActors, nActors))

Warning: 5 ties could not be set.

R> processState <- add_network(processState, network, name = "friendship")
```

Note the warning: By default a network is created with parameters `directed = TRUE`, `reflexive = FALSE`. The matrix has "reflexive" values on the diagonal which are automatically set to 0. This triggers the warning.

The id of the network within the process state may be important when, for example, the network has to be identified as a dependent network in a model. The id can be retrieved as follows:

[3] https://r-forge.r-project.org/projects/netsim/

```
R> get_network_index(processState, name = "friendship")

[1] 0
```

It is 0 in this case as the network with name “friendship” was the first added to the process state. A network can be re-transformed into a matrix object with the following command:

```
R> as.matrix(network)

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    1    1    1
[2,]    1    0    1    1    1
[3,]    1    1    0    1    1
[4,]    1    1    1    0    1
[5,]    1    1    1    1    0
```

Here it can be seen that the diagonal elements were indeed set to 0.

Attribute containers (objects that store exactly one variable per actor in the network) can be defined and included similarly to network objects. Other than networks they are based on R vectors.

```
R> attribute <- create_attribute_container(
      c(rep(0, nActors%/%2), rep(1, nActors - nActors%/%2)))
R> processState <- add_attribute_container(processState, attribute, name = "gender")
R> as.numeric(attribute)

[1] 0 0 1 1 1

R> get_attribute_container_index(processState, name="gender")

[1] 0
```

Finally, global variables may be part of the process state. Once again, the commands are very similar:

```
R> processState <- add_global_attribute(processState, value = 1, name = "timer")
R> get_global_attribute(processState, name="timer")

[1] 1

R> get_global_attribute_index(processState, name="timer")

[1] 0
```

3.3. Defining a model manager

The model manager stores all the four-step model chains of a simulation. An empty model manager can be created with command

```
R> modelManager <- create_model_manager()
```

Four-step model chains with time models, time updaters, change models and change updaters can be added to a model manager with the following commands:

```
R> modelManager <- add_time_model(modelManager, myTimeModel)
R> modelManager <- add_time_updater(modelManager, myTimeUpdater)
R> modelManager <- add_change_model(modelManager, myTimeModel, myChangeModel)
R> modelManager <- add_updater(modelManager, myChangeModel, myUpdater)
```

Note, that it is not necessary to define complete four-step chains. A time model is always necessary but may be followed by either a one or several time updater or one or several change model (or both). Each change model may be followed by a number of updaters. The `my...` variables in the code are to be replaced with the variable names of corresponding models as described in the following sub sections. Models in such a model chain do not need to be initialized with a concrete process state. However, a model may need to be initialized with a pointer to a specific object in the process state. For these model initializations the indexes are needed that were discussed in the previous section.

The model manager above would be simulated as follows: `myTimeModel` determines the time span until the next change depending on the process state. Based on the time span, the process state is then updated with `myTimeUpdater`. Based on the updated process state `myChangeModel` determines a particular change which is applied to the process state by `myUpdater`.

Time Models

Round-based time models are deterministic. They trigger sub-sequent models after a fixed time-interval, for example, one time unit. Therefore, it needs to be linked to a timer (which is part of the process state) by passing a timer index. The following model starts after the third time unit. So, a sub-sequent model will be triggered when the timer is $4, 5$ and so on until the simulation stops. It has to be made sure that the timer is updated properly.

```
R> create_round_based_time_model(timerIndex,
                 intervalLength = 1.0, startTime = 3.0)
```

A round-based time model is implemented in section 4.3.

A Poisson model with a fixed parameter (the standard in Snijders network and behavior models) can be defined as follows:

```
R> create_poisson_model(param = 1)
```

It can be defined as an individual model by defining one Poisson model per actor by, for example, using a `for` loop. This is demonstrated in section 4.1. However, a Poisson model with fixed parameter could also trigger global changes in the process state.

An individual Poisson model that can be changed over time can be defined by the following function:

```
R> create_attribute_poisson_model(attributeIndex)
```

Here, the parameter is saved in an attribute container which may be object to change. For For each actor a value is stored. A value of zero indicates that the corresponding actor is not active at all (infinite waiting time between sub-sequent events). In fact, the time model is not individualized by definition but only returns a time span realization of several concurring Poisson processes. The actual selection of the corresponding actor then takes place in a subsequent change model or time updater and is also to be based on the values in the attribute container. This is demonstrated in section 4.5.

Time Updaters

NetSim implements one time updater so far. It adds a time span to a global timer variable. The parameter is a pointer to this global timer variable. Note, that a global timer requires to be updated after each of multiple concurring time models. This is demonstrated in section 4.5.

```
R> create_timer_updater(timerIndex)
```

Change models

The focal change model of **NetSim** are the network change and behavior change models defined by Snijders and colleagues (Snijders *et al.* 2010; Steglich *et al.* 2010). They are applied in sections 4.1 and 4.2. For one focal actor and a pre-defined set of effects, these models can be defined using the following to commands.

```
R> create_multinomial_choice_network_change_model(
                 focalActorIndex, networkIndex, effectContainer, updater)
R> create_multinomial_choice_behavior_change_model(
                 focalActorIndex, attributeIndex, effectContainer)
```

Both models are defined for a particular actor (`focalActor`) and a particular dependent network (`networkIndex`) or dependent attribute (`attributeIndex`). They make use of an effect container object that can be used for several actors at the same time. The definition of an effect container is demonstrated in sections 4.1 and 4.2. The `updater` parameter defines a change that is considered by the focal actor after choosing a particular tie. By default, actors consider swapping a tie – to remove a tie if it exists or adding a tie if id does not. This default parameter is in line with the definition of SAOMs for network change. The considered changes of behavior models are to increase an attribute, to decrease it or to keep it unchanged.

An effect container with a reciprocity effect of 2.0 can, for example, be defined as follows.

```
R> effectContainer <- create_effect_container()
R> effectContainer <- add_to_effect_container(
                 effectContainer,
                 create_effect("recip", networkIndex),
                 2.0)
```

In this case it is measured on the network indicated by the id `networkIndex`. The name `"recip"` of the effect is chosen according to the effect naming in Ripley, Snijders, and Preciado Lopez (2011), the estimation software manual for SAOMs.

A Jackson and Rogers change model describes a mechanism how actors who are joining a network link with others. Is is further discussed in section 4.3 and also applied in section 4.5. It can be defined using the following command:

```
R> create_jackson_rogers_change_model(networkIndex, pLinkToParentNode = 1.0,
                  pLinkToNeigborNode = 1.0, nParentNodes = 1, nNeighborNodes = 1)
```

The five parameters describe the ID of the dependent network, and the four parameters p_r, p_n, m_r, m_n in Jackson and Rogers (2007, p.894).

A change model according to Watts' and Strogatz' idea of random network rewiring is discussed in section 4.4. It can be created with the following command:

```
R> create_watts_strogatz_change_model(networkIndex)
```

An alternative way of defining a Snijders network change model is to use a dynamic parameter set. Its usage is demonstrated in section 4.5.

```
R> create_attribute_multinomial_choice_network_change_model(
                  networkIndex, poissonAttributeIndex, updater)
```

Change Updaters

When attaching a change updater to a change model it has to be made sure that it can actually interpret the model result provided by the change model. A change model determining a set of actors cannot, for example, trigger an update of a global variable.

The Snijders network and behavior change models trigger a tie swap and an attribute update, respectively. These change updates can be defined by the following two functions (that are applied in sections 4.1 and 4.2).

```
R> create_tie_swap_updater(networkIndex)
R> create_actor_attribute_set_updater(attributeIndex, actorIndex)
```

The latest actor included in the process state can be accessed by change updaters. The following three change updater functions refer to an inclusion of new actors and are applied in sections 4.4 and 4.5.

An actor can be added to the process state with the following function.

```
R> create_add_actor_updater()
```

A new actor can be connected to a set of nodes provided by a change model.

```
R> create_add_ties_from_newborn_actor_updater(networkIndex)
```

Further, a fixed attribute value can be set for it in an attribute container. Note that the following method does not use result provided by a change model.

```
R> create_set_attribute_of_newborn_actor_updater(attributeIndex, value)
```

If a change model returns a set of four actor indexes, a tie rewire updater can be applied. A tie between the first two actors is deleted and a tie is created between actors three and four.

```
R> create_rewire_tie_updater(networkIndex)
```

3.4. Running a simulation

After the definition of a `modelManager` and a `processState` object as discussed in the previous two sections, running a simulation is straightforward:

```
R> timeSpan <- 20
R> simulator <- create_simulator(processState, modelManager, timeSpan,
      verbose = TRUE, debug = FALSE)
R> simulate(simulator)

Starting simulation. Length = 20, Time = 0
Progress (in 10%) [==========] done.
Iteration steps: 1218
Simulation time: 0.21s
Time per iteration: 0.000172s
```

Here, a time span of 20 time units is simulated. The parameter `verbose = TRUE` generates some basic information output including a simulation progress bar as shown above[4]. The parameter `debug = TRUE` would generate a very detailed output for some of the models. For estimations longer than a few iteration steps the usage of this parameter is not recommended as it significantly slows down the simulation process.

[4]In the **NetSim** version 0.9, the progress bar is not displayed on Window computers.

4. Simulation of exemplary micro models

4.1. Snijders network model

In sections 1 and 2, I briefly explained stochastic actor-oriented models (SAOMs) for social network change (Snijders 1996). Here, I discuss the simulation code that was used to generate the network shown in figure 1(a). The simulation model describes the behavior of 21 actors having an individual attributes with values 1, 2 or 3 each. The actors are not connected in the first place. However, individual Poisson processes trigger actor-oriented reconsiderations of local network configurations: Actors favor reciprocal ties, transitive clustering, and to be connected to similar actors. The number of ties is somewhat limited as it is connected with some costs in the individual utility functions.

The process state with one network and one attribute is defined as follows:

```
R> nActors <- 21
R> mat <- matrix(0, nActors, nActors)
R> att <- c(rep(0, nActors/3),
         rep(1, nActors/3),
         rep(2, nActors/3))

R> network <- create_network(mat)
R> attributeContainer <- create_scale_attribute_container(att,
      min=0, max=3, by = 1)

R> processState <- create_process_state()
R> processState <- add_network(processState, network, name = "network")
R> processState <- add_attribute_container(processState, attributeContainer,
      name = "attribute")

R> networkIndex <- get_network_index(processState)
R> attributeIndex <- get_attribute_container_index(processState)
```

Now an object capturing network effects and their corresponding parameters is defined. This object is later shared between all actors. Later, I discuss a model in which actors do not have homogeneous behavioral patterns. The effect names and effect definitions are those defined in the SIENA[5] manual (Ripley *et al.* 2011, sec. 12.1). The following code defines a behavioral model of actors with homophilic preferences plus preferences for low-density networks, reciprocity and transitivity.

```
R> effectContainerHomophily <- create_effect_container()
R> effectContainerHomophily <- add_to_effect_container(
                                effectContainerHomophily,
                                create_effect("density", networkIndex),
                                -2.5)
R> effectContainerHomophily <- add_to_effect_container(
```

[5]RSiena is a software package for the simulation-based estimation of SAOMs in longitudinal data sets.

```
                              effectContainerHomophily,
                              create_effect("recip", networkIndex),
                              2.5)
R> effectContainerHomophily <- add_to_effect_container(
                              effectContainerHomophily,
                              create_effect("transTrip", networkIndex),
                              0.5)
R> effectContainerHomophily <- add_to_effect_container(
                              effectContainerHomophily,
                              create_effect("cycle3", networkIndex),
                              -0.5)
R> effectContainerHomophily <- add_to_effect_container(
                              effectContainerHomophily,
                              create_effect("simX",
                                            attributeIndex,
                                            networkIndex,
                                            0.5),
                              1.0)
```

Then, a model manager can be defined in which the action of each actor followes a Poisson model. A parameter value 40 is chosen, which means that each actor is expected to re-consider her local network configuration 40 times per time unit. In the `for` loop the above defined preferences are assigned to each actor.

```
R> modelManager <- create_model_manager()

R> for (i in c(0 : (nActors - 1) ) ){
    poissonParameter <- 40
    poissonModel <- create_poisson_model(poissonParameter)
    saomHomophilyModel <- create_multinomial_choice_network_change_model(
      i,
      networkIndex,
      effectContainerHomophily
      )
    tieSwapUpdater <- create_tie_swap_updater(networkIndex)

    modelManager <<- add_time_model(modelManager,
                                    poissonModel)
    modelManager <<- add_change_model(modelManager,
                                      poissonModel,
                                      saomHomophilyModel)
    modelManager <<- add_updater(modelManager,
                                 saomHomophilyModel,
                                 tieSwapUpdater)
  }
```

The visualized network in figure 1(a) is based on a simulation with length 1, so every actor changes its network configuration about 40 times, starting with an empty network. The simulation is started with the commands shown in section 3.4. Results can be shown and further processed (e.g., visualized) with the commands introduced in section 3.2.

4.2. Snijders et al. network and behavior model

Here, I demonstrate how to specify a combined homophily and influence SAOM in **NetSim**. SAOMs for networks and attribute change were introduced by Steglich *et al.* (2010). A visualization of one simulation run is shown in figure 1(c). The skeleton of the code is similar to the code in section 4.1. However, additionally to a network change model, I assign an attribute change model to each actor.

An effect container can be defined that is again shared by all actors:

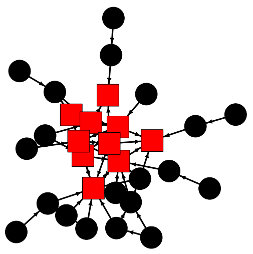
```
R> effectContainerInfluence <- create_effect_container()
R> effectContainerInfluence <- add_to_effect_container(
                   effectContainerInfluence,
                   create_effect("linear",
                                 attributeIndex),
                   0.0)
R> effectContainerInfluence <- add_to_effect_container(
                   effectContainerInfluence,
                   create_effect("quad",
                                 attributeIndex),
                   0.0)
R> effectContainerInfluence <- add_to_effect_container(
                   effectContainerInfluence,
                   create_effect("totSim",
                                 attributeIndex,
                                 networkIndex, 10/18),
                   2.0)
```

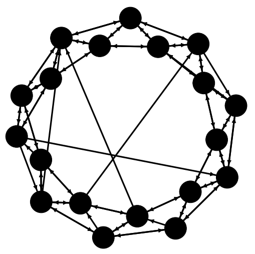
The effect names and effect definitions are taken from Ripley *et al.* (2011, sec. 12.2). The behavior model can be interpreted as follows: Actors have no generic proponsity to increase or decrease their behavioral variable (`"linear"` and `"quad"` effect). However, they have a strong preference to adapt their individual behavior so that they become similar to the actors they are connected with (`"totsim"` effect).

The model manager can be defined as follows. Note, that in the `for` loop the individual Poisson models, the behavioral SAOMs and the updaters are assigned to each of the actors.

```
R> for (i in c(0 : (nActors - 1))){
          poissonParameterInfluence <- 5
          poissonModelInfluence <- create_poisson_model(
            poissonParameterInfluence)

          behaviorSaom <- create_multinomial_choice_behavior_change_model(
                              i,
                              attributeIndex,
                              effectContainerInfluence
```


(a) Jackson & Rogers model


(b) Watts & Strogatz model

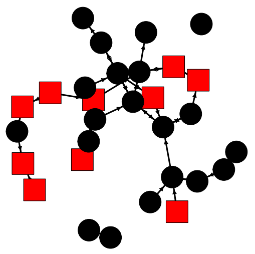

(c) Combination of a Jackson & Rogers model and a Snijders network model

Figure 3: Plots of outcomes of the simulation models described in sections 4.3, 4.4 and 4.5

```
            )

            setAttributeUpdater <- create_actor_attribute_set_updater(
              attributeIndex, i)

            modelManager <<- add_time_model(modelManager,
                                            poissonModelInfluence)
            modelManager <<- add_change_model(modelManager,
                                              poissonModelInfluence,
                                              behaviorSaom)
            modelManager <<- add_updater(modelManager,
                                         behaviorSaom,
                                         setAttributeUpdater)
    }
```

The model manager and the process state are equal to the ones in section 4.1. The Poisson rate of the behavior change model is set to 5, which is less than the network change rate of 40. This is in line with typical empirical SIENA estimates.

4.3. Jackson and Rogers' friends of friends model

Jackson and Rogers (2007) introduced a dynamic network model that describes the "birth" of new nodes in social networks. Once a new node enters the process state it randomly chooses a set of m_r "parent nodes" to which it connects with a probability p_k. Also it randomly chooses a set of m_n neighbors of these parent nodes. To each of these nodes it links with a probability p_n. A nice feature of Jackson and Rogers birth micro model is that it generates social networks with certain macro features that are empirically known as to be typical for social networks: Jackson and Rogers mention a "high level of clustering, a diameter that is smaller than that of a random graph, and [a] negative clustering-degree relationship" (Jackson and Rogers 2007, p.893).

In the following it is described how a round-based Jackson & Rogers (JR) model can be defined in **NetSim**. A network of 10 actors is created and random ties are included with probability 0.3 (the expected density is 30%). The process state consists of this network and a global timer variable. In the previous examples in which the time models were Poisson processes, no timer was needed as the time until the next change could be determined independent from the process time.

```
R> nActors <- 10
R> mat <- matrix(0, nActors, nActors)

R> processState <- create_process_state()

R> network <- create_network(mat)
R> network <- add_random_ties_to_network(network, 0.3)

R> processState <- add_network(processState, network, name = "friendship")
R> networkIndex <- get_network_index(processState, name = "friendship")

R> processState <- add_global_attribute(processState, value = 0, name = "timer")
R> timerIndex <- get_global_attribute_index(processState, "timer")
```

The model manager is defined as follows:

```
R> modelManager <- create_model_manager()

R> roundBasedTimeModel <- create_round_based_time_model(timerIndex,
                          intervalLength = 1.0)
R> timerUpdater <- create_timer_updater(timerIndex)
R> jrModel <- create_jackson_rogers_change_model(networkIndex,
                          pLinkToParentNode = 1.0, pLinkToNeigborNode = 0.5,
                          nParentNodes = 1, nNeighborNodes = 3)
R> addTiesToNewActorUpdater <- create_add_ties_from_newborn_actor_updater(
                          networkIndex)
R> addActorUpdater <- create_add_actor_updater()

R> modelManager <- add_time_model(modelManager, roundBasedTimeModel)
R> modelManager <- add_time_updater(modelManager, timerUpdater)
R> modelManager <- add_change_model(modelManager, roundBasedTimeModel, jrModel)
R> modelManager <- add_updater(modelManager, jrModel, addActorUpdater)
R> modelManager <- add_updater(modelManager, jrModel, addTiesToNewActorUpdater)
```

After each time unit, changes take place. First, the timer is updated to the current time. Then the JR model determines a set of nodes: One parent node is chosen first and up to three of its neighbors are added to the result set with probability 0.5 each. After this change model, a new node is added to the process state (automatically updating all networks, attribute container objects and global variables). This node (the newest born node in the process state) is then connected to all nodes determined by the change model. A plot of a simulation run is shown in figure 3(a). The squares are the 10 initial network nodes that are connected with random ties. The black circles are the subsequently joining nodes. It can be seen that the model generates a certain degree of triadic structures.

4.4. Watts and Strogatz small world model

The 'small world' network models by Watts and Strogatz (1998) do not claim to describe a realistic micro behavior. Rather these models discuss the generation of certain macro features of social networks using a straightforward micro mechanism based on random "rewiring" of nodes. **NetSim** allows to simulate the rewiring of nodes in a regular lattice to generate so called small-world networks. These networks can, for example, be used as a starting points for other simulations or to add a certain level of randomness to a simulation model. This randomness could be interpreted as generating weak ties according to the work of Granovetter (1973).

The process state of this model can be defined similar to the model in section 4.3. Only instead of a randomized graph, a regular ring lattice structure with 19 nodes is defined as the starting point:

```
R> nActors <- 19
R> mat <- matrix(0, nActors, nActors)

R> network <- create_network(mat)
R> network <- add_ring_lattice_to_network(network, 4)

R> processState <- create_process_state()

R> processState <- add_network(processState, network, name="friendship")
R> networkIndex <- get_network_index(processState, "friendship")

R> processState <- add_global_attribute(processState, value = 0, name = "timer")
R> timerIndex <- get_global_attribute_index(processState, "timer")
```

To rewire some of the ties in a dynamic simulation process, the following model manager is defined:

```
R> modelManager <- create_model_manager()

R> roundBasedTimeModel <- create_round_based_time_model(timerIndex)
R> wsModel <- create_watts_strogatz_change_model(networkIndex)
R> rewireUpdater <- create_rewire_tie_updater(networkIndex)
R> timerUpdater <- create_timer_updater(timerIndex)

R> modelManager <- add_time_model(modelManager, roundBasedTimeModel)
R> modelManager <- add_time_updater(modelManager, timerUpdater)
R> modelManager <- add_change_model(modelManager, roundBasedTimeModel, wsModel)
R> modelManager <- add_updater(modelManager, wsModel, rewireUpdater)
```

After each time unit a tie is randomly rewired. A simulation of length six would then generate a graph as the one in figure 3(b). 19 nodes are embedded in a regular ring lattice in which each node started with four reciprocal neighbors. Six ties, however, have been randomly rewired and give the network the specific features of a 'small world' network: It has a high level of local clustering but rather short distances between nodes.

4.5. Combining different models

As a final example, I present how to combine different models. Figure 3(c) shows the outcome of a simulation combining a round-based Jackson & Rogers tie inclusion model and a Poisson-based SAOM for network change. As in figure 3(a) the squares represent ten nodes which are initially part of the process state. After each of 20 subsequent simulation time units a new circular node is included. All nodes change their network configuration following a SAOM. However, the SAOM is specified differently for (intial) quadratic and (new) circular nodes. All nodes have a Poisson parameter of 2.0, an out-degree effect of -2.0 and a reciprocity effect of 2.0. So, they consider changing their network configuration about twice per time unit, they have a limited out-degree and prefer reciprocal network configurations. Additionally, the behavior of circular nodes is specified with a transitive triplets effect of 0.5 and a cycle3 effect of -0.2. This means that these nodes have a preference for being embedded in denser local clusters. This leads to the result that the initial quadratic nodes (although being central in the beginning – see figure 3(a)) are subsequently moved to the periphery of the network due to their lacking preference for clustering.

Other than in previous examples the behavior is now heterogeneous. Therefore, additional attribute containers are defined as part of the process state that describe individual models. The process state is defined as follows:

```
R> nActors <- 10
R> timeSpan <- 20
R> mat <- matrix(0, nActors, nActors)

R> processState <- create_process_state()

R> network <- create_network(mat)
R> network <- add_random_ties_to_network(network, 0.3)

R> processState <- add_network(processState, network, name="friendship")
R> networkIndex <- get_network_index(processState, "friendship")

R> processState <- add_global_attribute(processState, value = 0, name = "timer")
R> timerIndex <- get_global_attribute_index(processState, "timer")

R> processState <- add_attribute_container(processState,
     create_attribute_container(rep(3.0, nActors)), name = "poissonParameter")
R> processState <- add_attribute_container(processState,
     create_attribute_container(rep(-2.0, nActors)), name = "density")
R> processState <- add_attribute_container(processState,
     create_attribute_container(rep(2.0, nActors)), name = "reciprocity")
```

```
R> processState <- add_attribute_container(processState,
      create_attribute_container(rep(0.0, nActors)), name = "transitivity")
R> processState <- add_attribute_container(processState,
      create_attribute_container(rep(0.0, nActors)), name = "cycle3")

R> poissonAttributeIndex <- get_attribute_container_index(processState,
      "poissonParameter")
R> densityAttributeIndex <- get_attribute_container_index(processState,
      "density")
R> reciprocityAttributeIndex <- get_attribute_container_index(processState,
      "reciprocity")
R> transitivityAttributeIndex <- get_attribute_container_index(processState,
      "transitivity")
R> threeCycleAttributeIndex <- get_attribute_container_index(processState,
      "cycle3")
```

One can see that the SAOM parameters are stored in attribute containers. This allows to dynamically set and update the parameters over time. In this example it is necessary, as the number of included nodes cannot be determined when defining the process state – it depends in this round-based design on the length of the simulation.

Time models, time updaters, change models and change updaters are defined as follows:

```
R> poissonAttributeModel <- create_attribute_poisson_model(poissonAttributeIndex)
R> roundBasedTimeModel <- create_round_based_time_model(timerIndex)

R> jrModel <- create_jackson_rogers_change_model(networkIndex,
                       pLinkToParentNode = 1.0, pLinkToNeigborNode = 0.5,
                       nParentNodes = 1, nNeighborNodes = 3)

R> addTiesToNewActorUpdater <- create_add_ties_from_newborn_actor_updater(
      networkIndex)
R> addActorUpdater <- create_add_actor_updater()
R> addPoissonValueUpdater <- create_set_attribute_of_newborn_actor_updater(
      poissonAttributeIndex, 3.0)
R> addDensityValueUpdater <- create_set_attribute_of_newborn_actor_updater(
      densityAttributeIndex, -2.0)
R> addReciprocityValueUpdater <- create_set_attribute_of_newborn_actor_updater(
      reciprocityAttributeIndex, 2.0)
```

Note that the following two effects differ from the initial actors for which parameters of zero were set (indifference about triadic structures).

```
R> addTransitivityValueUpdater <- create_set_attribute_of_newborn_actor_updater(
      transitivityAttributeIndex, 0.5)
R> addThreeCycleValueUpdater <- create_set_attribute_of_newborn_actor_updater(
      threeCycleAttributeIndex, -0.2)
```

```
R> timerUpdater <- create_timer_updater(timerIndex)
R> tieSwapUpdater <- create_tie_swap_updater(networkIndex)

R> saom <- create_attribute_multinomial_choice_network_change_model(
     networkIndex, poissonAttributeIndex, tieSwapUpdater)
R> saom <- add_effect(saom, create_effect("density", networkIndex),
     densityAttributeIndex)
R> saom <- add_effect(saom, create_effect("recip", networkIndex),
     reciprocityAttributeIndex)
R> saom <- add_effect(saom, create_effect("transTrip", networkIndex),
     transitivityAttributeIndex)
R> saom <- add_effect(saom, create_effect("cycle3", networkIndex),
     threeCycleAttributeIndex)
```

The model manager incorporates two time models and corresponding chains of time updaters (equivalent in both chains), change models and updaters. One chain is the round-based inclusion of new nodes (and the setting of new ties and attribute container variables), the second one is the Poisson process determining the SAOM for network change.

```
R> modelManager <- create_model_manager()

R> modelManager <- add_time_model(modelManager, roundBasedTimeModel)
R> modelManager <- add_time_updater(modelManager, timerUpdater)
R> modelManager <- add_change_model(modelManager, roundBasedTimeModel, jrModel)
R> modelManager <- add_updater(modelManager, jrModel, addActorUpdater)
R> modelManager <- add_updater(modelManager, jrModel, addTiesToNewActorUpdater)
R> modelManager <- add_updater(modelManager, jrModel, addPoissonValueUpdater)
R> modelManager <- add_updater(modelManager, jrModel, addDensityValueUpdater)
R> modelManager <- add_updater(modelManager, jrModel, addReciprocityValueUpdater)
R> modelManager <- add_updater(modelManager, jrModel, addTransitivityValueUpdater)
R> modelManager <- add_updater(modelManager, jrModel, addThreeCycleValueUpdater)

R> modelManager <- add_time_model(modelManager, poissonAttributeModel)
R> modelManager <- add_change_model(modelManager, poissonAttributeModel, saom)
R> modelManager <- add_updater(modelManager, saom, tieSwapUpdater)
```

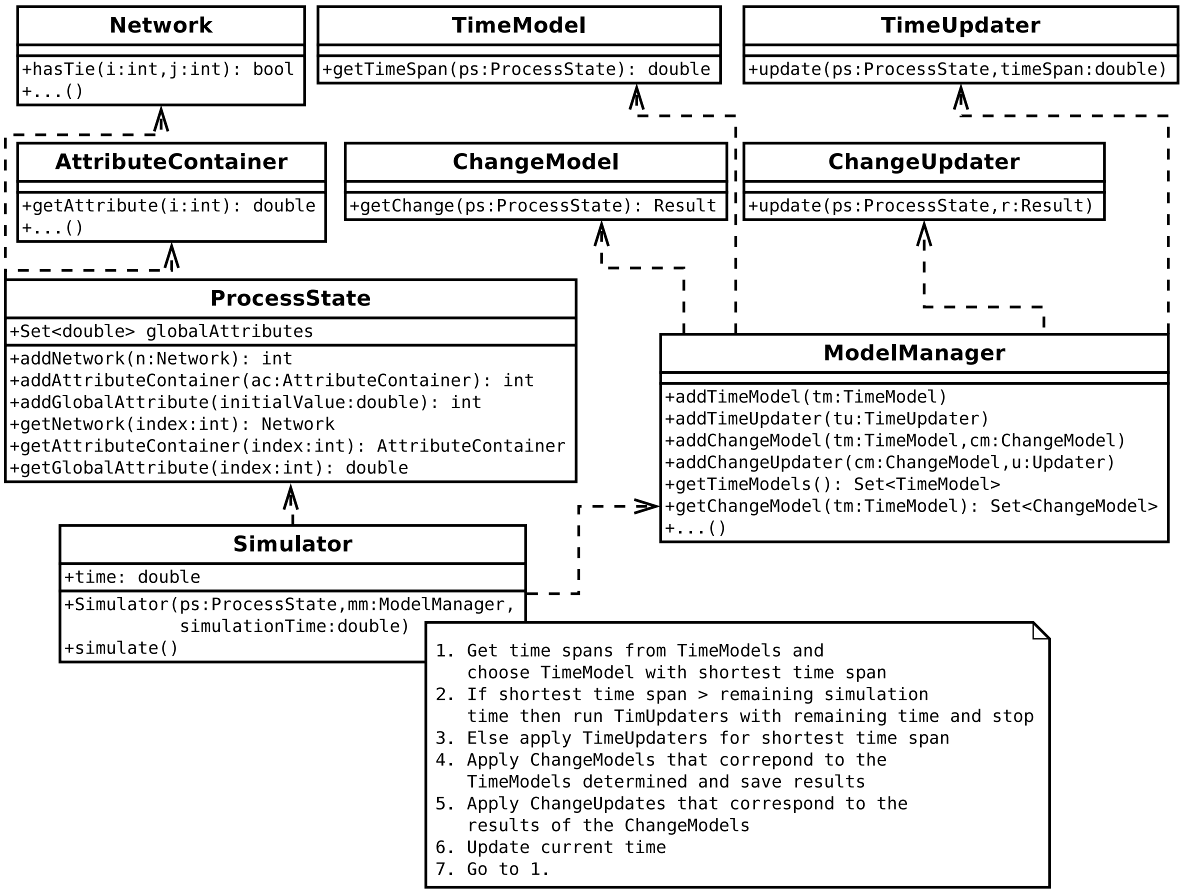
5. Implementation and architecture

The simulation framework is developed in C++. The interface to R is realized with the **Rcpp** package described in Eddelbuettel and Francois (2011); Eddelbuettel (2013). In the following, I describe the class model. To extend the simulation framework with own models it is essential to understand its basic functionality. Figure 5 shows a simplified overview of the class structure in **NetSim**.

On the left side one can see the definition of a process state (`ProcessState`). It may consist of several `Network` objects and `AttributeContainer` objects. Additionally, it represents a set of `globalAttributes` such as a process timer. The `ProcessState` interface allows to add new networks, attribute containers and global attributes. Also, one can get objects that are parts of the process state using a pointer.

On the right side I show the interface of the `ModelManager` class. Objects of type `TimeModel`, `TimeUpdater`, `ChangeModel` and `ChangeUpdater` can be added. Also, a set of the included `TimeModel` and `ChangeModel` objects can be retrieved.

On the lower side it can be seen how the `Simulator` class relates to the `ProcessState` and the `ModelManager` class. It consists of only two functions: a constructor that receives one object of type `ProcessState` and `ModelManager` each and the simulation time span. The second function starts one simulation run. The logic of the `simulate` function is shown in pseudocode on the lower right.


Network
+hasTie(i:int,j:int): bool
+...()
TimeModel
+getTimeSpan(ps:ProcessState): double
TimeUpdater
+update(ps:ProcessState,timeSpan:double)
AttributeContainer
+getAttribute(i:int): double
+...()
ChangeModel
+getChange(ps:ProcessState): Result
ChangeUpdater
+update(ps:ProcessState,r:Result)
ProcessState
+Set<double> globalAttributes
+addNetwork(n:Network): int
+addAttributeContainer(ac:AttributeContainer): int
+addGlobalAttribute(initialValue:double): int
+getNetwork(index:int): Network
+getAttributeContainer(index:int): AttributeContainer
+getGlobalAttribute(index:int): double
ModelManager
+addTimeModel(tm:TimeModel)
+addTimeUpdater(tu:TimeUpdater)
+addChangeModel(tm:TimeModel,cm:ChangeModel)
+addChangeUpdater(cm:ChangeModel,u:Updater)
+getTimeModels(): Set<TimeModel>
+getChangeModel(tm:TimeModel): Set<ChangeModel>
+...()
Simulator
+time: double
+Simulator(ps:ProcessState,mm:ModelManager,
simulationTime:double)
+simulate()

```
1. Get time spans from TimeModels and
   choose TimeModel with shortest time span
2. If shortest time span > remaining simulation
   time then run TimUpdaters with remaining time and stop
3. Else apply TimeUpdaters for shortest time span
4. Apply ChangeModels that correpond to the
   TimeModels determined and save results
5. Apply ChangeUpdates that correspond to the
   results of the ChangeModels
6. Update current time
7. Go to 1.
```

Figure 4: Simplified class diagram of the **NetSim** architecture with pseudo code of the simulation function.

6. Extending NetSim: Implementing new models

The implementation of new models works in a straightforward way in **NetSim**.

Imagine the case of a model that is supposed to decrease the density of a network in a simulation by removing a random tie at certain points in time. This potential removal happens round-based, for example, once after each time unit. This usage of this time model has been demonstrated in section 4.3. Following the four-step design introduced in section 2, I illustrate a potential implementation of, first, a round-based time model, second, a change model determining a random tie and third, a tie removal change updater. A time updater is not necessary for the model chain of this example.

6.1. Implementing a round-based time model

A new time model can be embedded in the files `src/model/TimeModel.{h/cpp}`. In the same files the base class `TimeModel` is implemented. Complex models are recommended to be stored in separate files. The interface in the header file can be defined as

```
class RoundBasedTimeModel : public TimeModel{

public:
        RoundBasedTimeModel(size_t timerIndex, double intervalLength);
        double getTimeSpan(ProcessState * processState);

private:
        size_t _timerIndex;
        double _intervalLength;
};
```

with the following implementation of the two methods in the `.cpp` file:

```
RoundBasedTimeModel::RoundBasedTimeModel(size_t timerIndex,
                double intervalLength) {
        _intervalLength = intervalLength;
        _timerIndex = timerIndex;
}

double RoundBasedTimeModel::getTimeSpan(ProcessState* processState) {
        double time = processState->getGlobalAttribute(_timerIndex);

        double timeSincePrevious = fmod(time, _intervalLength);
        return _intervalLength - timeSincePrevious;
        }
}
```

To embed a round-based model in a Markov framework a time counter has to be included in the process state. It is a global variable of which the index has to be known by the time model (`_timerIndex`). The interval length (`_intervalLength`) is the length of an interval between two subsequent events. If a round-based model is supposed to make a chance after each time unit it is initialized with 1. The base class `TimeModel` of class `RoundBasedTimeModel` requires the implementation of a function with the signature `double getTimeSpan(ProcessState * processState)`. A slightly extended version of the round-based time model is part of the **NetSim** implementation. It was used in sections 4.3, 4.4 and 4.5.

6.2. Implementing a change model selecting a random tie

Simple change models can be embedded in the files `src/model/ChangeModel.{h/cpp}`. The `ChangeModel` base class can also be found there. For rather complex models I again recommend separate files. Change models have to implement a function with signature `ModelResult * getChange(ProcessState * processState)`. Our random tie choice time model has the following interface:

```
class ChooseRandomTieChangeModel : public ChangeModel{
public:
        ChooseRandomTieChangeModel(size_t networkIndex);
        ModelResult * getChange(ProcessState * processState);
private:
        size_t _networkIndex;
};
```

The constructor only sets the local variable `size_t _networkIndex` that indicates the id of the focal network in the process state. The `getChange` function can be implemented as follows:

```
ModelResult * ChooseRandomTieChangeModel::getChange(
        ProcessState* processState) {

        MemoryOneModeNetwork * network = dynamic_cast<MemoryOneModeNetwork *>(
                        processState->getNetwork(_networkIndex));

        std::pair<int, int> tie = NetworkUtils::getRandomTie(network);

        return new TieModelResult(tie.first, tie.second);
}
```

First, the function gets the focal network from the process state object and casts it to a one mode networks (note that I did not implement any exception handling here). The class `NetworkUtils` provides several utility function and I make use of its random tie selection method. This utility function makes use of a shared random number generator. The function returns a pointer to a new `ModelResult` object, in this case a more specific one of type `TieModelResult`. An updater linked to this change model has to be able to handle such an object.

6.3. Implementing a tie removal change updater

Simple change updater classes can be implemented in the files `src/model/Updater.{h/cpp}`. The `Updater` base class can also be found there. The tie removal updater discussed has the following interface:

```
class RemoveTieUpdater : public Updater{
public:
        RemoveTieUpdater(size_t networkIndex);
        void update(ProcessState* processState, ModelResult* result);
        void undoUpdate(ProcessState* processState, ModelResult* result);

private:
        size_t _networkIndex;
};
```

The functions `update` and `undoUpdate` must be implemented by sub classes of the abstract `Updater` class. The constructor – again – only initializes local variables. The `update` function can be implemented as follows (the `undoUpdate` function can be implemented similarly):

```
void RemoveTieUpdater::update(ProcessState* processState, ModelResult* result) {

        TieModelResult * tieResult = dynamic_cast<TieModelResult *>(
                result);

        Network * network = processState->getNetwork(_networkIndex);

        network->removeTie(tieResult->getActorIndex1(),
                        tieResult->getActorIndex2());
}
```

Here, I again perform an (unchecked) cast to the expected result object type and use the `Network` function `removeTie` to remove the tie determined by the change model.

6.4. Implementing the R/C++ interface with Rcpp

The last step to get the new model working within **NetSim** is to define the interface between R and C++ using the straightforward **Rcpp** package. Example files can be found in the folders `src/rwrapper` (R wrapper code in C++) and `R/` (R method stubs).

7. Conclusions

This paper introduces **NetSim**, a new and flexible tool to simulate social network dynamics. Researchers can make use of the package to study the link between dynamic micro models and macro outcomes in social networks. The key strength of **NetSim** is that it defines a generic Markov process which allows to combine different types of models from a number of disciplines. Round-based models used in economics can be simulated together with stochastic actor-oriented models that are widely applied in the social sciences. Infection models from epidemiology can be combined with dynamic network change models from physics. The flexibility of the framework allows a new type of simulation studies that take a multitude of micro mechanisms into account. We demonstrated a model in which actors joining a community would behave different from actors being part of the community from the very beginning. One can go even further and specify models in which actors change their behavior over time, for example, to adapt to changes in their (social) environment. **NetSim** is designed to be easily extendible with additional models. Exploring the space of potential applications will be a matter of future research.

References

Csardi G, Nepusz T (2006). "The igraph Software Package for Complex Network Research." *InterJournal*, **Complex Systems**, 1695.

Eddelbuettel D (2013). *Seamless R and C++ Integration with Rcpp*. Springer-Verlag, New York. ISBN 978-1-4614-6867-7.

Eddelbuettel D, Francois R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18.

Flache A, Macy M (2011). "Local Convergence and Global Diversity: from Interpersonal to Social Influence." *Journal of Conflict Resolution*, **55**(6), 970–995.

Granovetter MS (1973). "The Strength of Weak Ties." *American Journal of Sociology*, **78**(6), 1360–1380.

Jackson MO, Rogers BW (2007). "Meeting Strangers and Friends of Friends: How Random are Social Networks?" *American Economic Review*, **97**(3), 890–915.

Kadushin C (2012). *Understanding Social Networks. Theories, Concepts and Findings*. Oxford University Press, New York.

McPherson JM, Smith-Lovin L, Cook JM (2001). "Birds of a Feather: Homophily in Social Networks." *Annual Review of Sociology*, **27**, 415–444.

Morris M, Kretzschmar M (1997). "Concurrent Partnerships and the Spread of HIV." *AIDS*, **11**(5), 641–648.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org`.

Ripley RM, Snijders TAB, Preciado Lopez P (2011). *Manual for SIENA 4.0*. Nuffield College and Department of Statistics, University of Oxford. Version: May, 1 2011.

Snijders TAB (1996). "Stochastic Actor-oriented Models for Network Change." *The Journal of Mathematical Sociology*, **21**(1), 149–172.

Snijders TAB, van de Bunt GG, Steglich CEG (2010). "Introduction to Stochastic Actor-based Models for Network Dynamics." *Social Networks*, **32**(1), 44– 60. Dynamics of Social Networks.

Stadtfeld C (2013). "NetSim: A Social Networks Simulation Tool in R." (project website), URL `http://www.christoph-stadtfeld.com/netsim/`.

Stadtfeld C, Geyer-Schulz A (2011). "Analyzing Event Stream Dynamics in Two Mode Networks: An Exploratory Analysis of Private Communication in a Question and Answer Community." *Social Networks*, **33**(4), 258–272.

Steglich C, Snijders TAB, Pearson M (2010). "Dynamic Networks and Behavior: Separating Selection from Influence." *Sociological Methodology*, **40**(1), 329–393.

Watts DJ, Strogatz SH (1998). “Collective Dynamics of ’Small-World’ Networks.” *Nature*, **393**(6684), 440–442.

Affiliation:

Christoph Stadtfeld
Department of Sociology / ICS
University of Groningen
The Netherlands
E-mail: c.stadtfeld@rug.nl
URL: http://www.christoph-stadtfeld.com