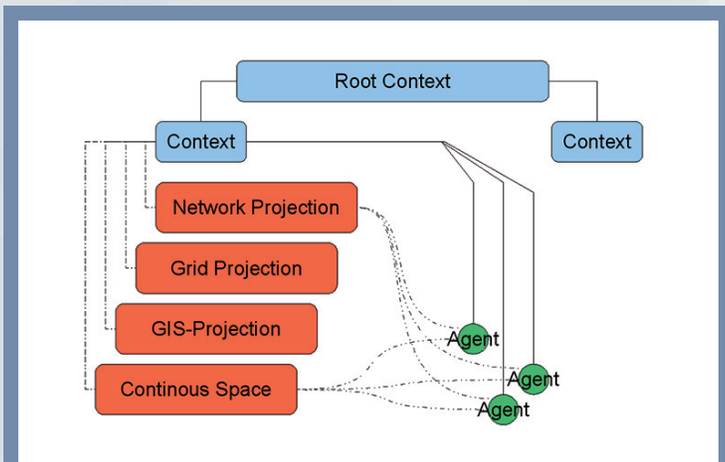
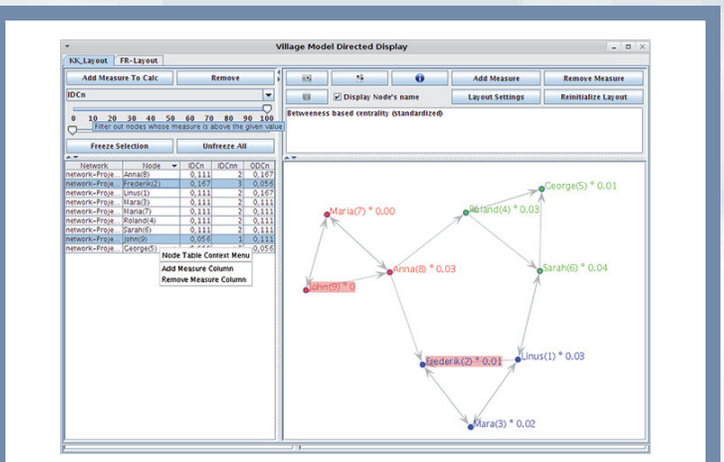
Sascha Holzhauer

Developing a Social Network Analysis and Visualization Module for Repast Models

Center for Environmental Systems Research

CESR-PAPER 4





kassel university press

UNIKASSEL VERSITÄT

CESR – Paper 4

Center for Environmental
System Research

Sascha Holzhauer

Developing a Social Network Analysis and Visualization Module for Repast Models

kassel university press

UNIKASSEL VERSITÄT

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar

ISBN print: 978-3-89958-978-8
ISBN online: 978-3-89958-979-5
URN: urn:nbn:de:0002-9799

Druck und Verarbeitung: Unidruckerei der Universität Kassel
Printed in Germany

Abstract

Dealing with social networks becomes more and more attractive to modelers. This is especially true for the field of agent-based modeling, in which represented actors are connected with others, with which they communicate, exchange goods, or spend their leisure time. While a number of software frameworks exists that ease the development of such models, the support for analyzing and visualizing social networks is still in its infancy.

First, the paper identifies requirements on software that seeks to aid the handling of social networks. In contrast to software in the field of social network analysis (SNA), the dynamics of modeled networks require specific considerations. Changes in the network structure should be perceptible for the user, which makes certain demands on the layout process. Five frameworks and additional libraries are reviewed in order to find an appropriate starting point to implement the requirements worked out, and attention is also drawn to existing models that shall be extended by network features. Repast J is identified as a rich and widespread framework that is intended for social science simulations and also incorporates basic network facilities.

This paper introduces the newly developed software library ReSoNetA (**Re**past **So**cial **Net**work **A**nalysis), which adds network functionality to the Repast J framework. It uses several features of the recently released Repast Simphony framework, and extends and improves network visualization capabilities. ReSoNetA also offers an extensible network measure framework, which enables the user to compute any network measure through GUI elements, use them for analyzing the networks, and also to affect the visualization. Several network measures and network layouts are described in detail.

While ReSoNetA constitutes a valuable base for analysis and visualization of social networks, the library has some potential to be developed further. Recommendations for such extensions are given. For example, providing sophisticated and efficient network layouts that preserve the viewer's mental map, while the network structure changes during simulation is hard to achieve and needs some further investigation. Most of the ReSoNetA features were developed in line with the Repast Simphony architecture, which makes it possible to incorporate them into that recent agent-based modeling framework.

Table of Content

Abbreviations and Synonyms

API	Application Programming Interface
ABM	Agent-Based Modeling
BSD	Berkley Software Distribution (http://www.opensource.org/licenses/bsd-license.php)
FAQ	Frequently Asked Questions
GIS	Geographical Information System
GNU	Gnu is Not Unix
GPL	GNU General Public License (http://www.gnu.org/copyleft/gpl.html)
GUI	Graphical User Interface
IDE	Integrated Development Environment
JUNG	Java Universal Network/Graph Framework
LGPL	GNU Lesser General Public License (http://www.gnu.org/copyleft/lesser.html)
MASON	Multi-Agent-Simulation Of Neighborhood/Networks/...
OS	Operating System
Repast	Recursive Porous Agent-Simulation Toolkit
UML	Unified Modeling Language
URL	Uniform Resource Locator
ZUI	Zoomable User Interface

Definition of Symbols

N	=|G|, the number of vertices within a graph
(v_3,v_1)	edge between v_3 and v_1
dm(G)	diameter of graph G
$C_D(v_i)$	degree-centrality of actor v_i
$C_C(v_i)$	closeness-centrality of actor v_i
$C_B(v_i)$	betweenness-centrality of actor v_i
$P_D(v_i)$	indegree-prestige of actor v_i
$P_P(v_i)$	proximity-prestige of actor v_i
$P_R(v_i)$	rank-prestige of actor v_i

1. Introduction

1.1. Goals and Structure

1.1.1. Goals

The overall purpose of this work is to present software that facilitates the development and usage of computer models which deal with social networks. Incorporating such networks becomes more and more both important and widespread, for example in the fields of integrated assessment (Pahl-Wostl, 2005) or land use (Krebs et al., 2007). On the other hand, toolkits that support modelers in implementation, analysis, and visualization of social networks are still rare. It would be nice not to reinvent the wheel every time a social network model is built, but software that offers most of the commonly needed features is still lacking.

While the need for such software is obvious, it is not clear what such a tool should look like and which features would have to be included. Thus, the first step is to identify the demands on a social network modeling toolkit regarding the representation of networks, their dynamic analysis and the communication of simulation results to the users and the public. Observing the social science field of social network analysis, and the rather mathematical field of graph drawing, in particular for visualization issues, seems promising. Well-grounded results of these sub-disciplines may help network modelers to enhance their work.

A compact software module that builds upon an appropriate agent-based modeling framework and provides the required additional features is considered as a suitable solution. Such a software library should incorporate the most crucial means to analyze and visualize social networks, while further demands are to be identified but need to be left for further development.

1.1.2. Structure

The remainder of this chapter first introduces into social network analysis as a field of social sciences, which forms the basis for any investigation of social networks. Afterwards, a running network example is given that serves for illustration purposes throughout this paper, before a synopsis on network and graph notation follows. The last section “Social Network Modeling” seeks to clear up the role of networks in agent-based modeling. The purpose of chapter two is to find out the demands on modeling social networks in general, and on a software that aims to support this in particular. Therefore, the first two sections present agent-based modeling as an appropriate paradigm for modeling social networks and issues to be considered during the model design. Then, an ideal software framework is outlined that fosters such a design and also minimizes the developer exertion. The third section identifies several aspects regarding network analysis and visualization that the library should assure, before the last section which deals with rather practical issues for the support of existing agent-based models.

Chapter three reviews software that could serve as a foundation for the aspired module. Several well-known multi-agent-based modeling frameworks are listed. They are evaluated according to their user friendliness, scope of features and appropriateness regarding social network modeling.

Chapter four finally sums up the requirements on the new software library. Repast Simphony is described in detail in the second section, since it offers some modern and powerful features,

which are worth to explore. Afterwards, the integration of the library under development and Repast Simphony are delineated. Furthermore, the library's concept and architecture are presented (section four), and its features are listed comprehensively (section five). Methods for analysis and visualization of networks are summarized in section six and seven respectively as far as the new module supports these. The two sections are furthermore intended as a user guide to find appropriate methods for the model at hand, but nevertheless deal with technical aspects that could also help to apply these methods and to interpret their results.

The last chapter comprises the conclusions and discusses the achievements and indicators of this work. It finally identifies issues that seem to be worth dealing with in future.

1.2. Social Network Analysis (SNA)

Social networks matter. The relations between actors that they maintain or that are imposed on them expand or limit their potentialities in acting. For instance, an actor's social network influences its opportunities to communicate, to exchange goods or to ask somebody for help. An actor may be part of many networks that consist of friends, relatives, neighbors, teammates, colleagues or acquaintances. In each of these networks it occupies a certain position, endowed for example with a particular amount of power or palsy.

Thus, in order to gain insight into the individual's opinion and the way he acts within society it is important to consider its links to other individuals who are part of that society (Friedkin, 1998).

1.2.1. The Subject of Social Network Analysis

In considering the others in an actor's context SNA fills a gap between under socialized theories like rational choice which are focused on the micro layer with rather solitary individuals, and over socialized concepts like structural functionalism that belongs to macro sociology. Structural functionalism doesn't consider relations among individuals because they are supposed to be incidental (Granovetter, 1985).

A theory that tries to elude these shortcomings is the structural theory of action, which was developed by the network analyst Burt. On the macro layer society forms a relational social structure that imposes positions on actors. This social structure and the agents' interests, which are again formed by society, influence the individual's action. This action takes back effect on the macro layer and may also alter relations of social structure (Burt, 1982).

Mark Granovetter clearly postulated that one must consider the networks among actors in analyzing their behavior. He defines the actor's social embeddedness as an individual's context that is shaped significantly by their relations to others. This embeddedness puts into perspective the actor's self interest on the one hand, and the validity of rather strict social norms, role prescriptions and hierarchies on the other (Granovetter, 1985).

The social network approach can finally help to explain certain phenomena of the 20th century which the prevailing concept of roles alone was lacking: To understand why managers share decision making with workers and why annoyed patients take action against doctors it is required to consider relations among actors and their positions within society (Galaskiewics and Wasserman, 1993).

One of the main questions in social network analysis is, what impact network structure has on other social phenomena and how this impact might be measured. Social network analysis has focused its attention on the way network structures relate to social structures, and on how the positions of different individuals affect their opportunities as well as their decisions. To answer these questions, it describes, visualizes, and statistically models the relations between actors of a network. In detail, goals of SNA are identification of important actors[1], crucial links, actors who possess a certain role like a bottleneck in communication, and cohesive subgroups. Furthermore, the investigation of network characteristics like density of actor connections is of interest. Thus, social network analysis takes place at three levels: the actor level, the subgroup level and the network level.

1.3. A Running Example

Introducing into the notation of graphs and networks and explaining all network related concepts such as measures or visualization algorithms is much easier with an example at hand. The following network of an extremely small village community as depicted in figure 1.1 shall guide the reader through this work.

The network consists of nine actors, three of these belonging to an environmental organization (Linus, Frederik and Mara), three belonging to local firms (Roland, George and Sarah) and the remaining three are local politicians (Maria, Anna and John[2]).

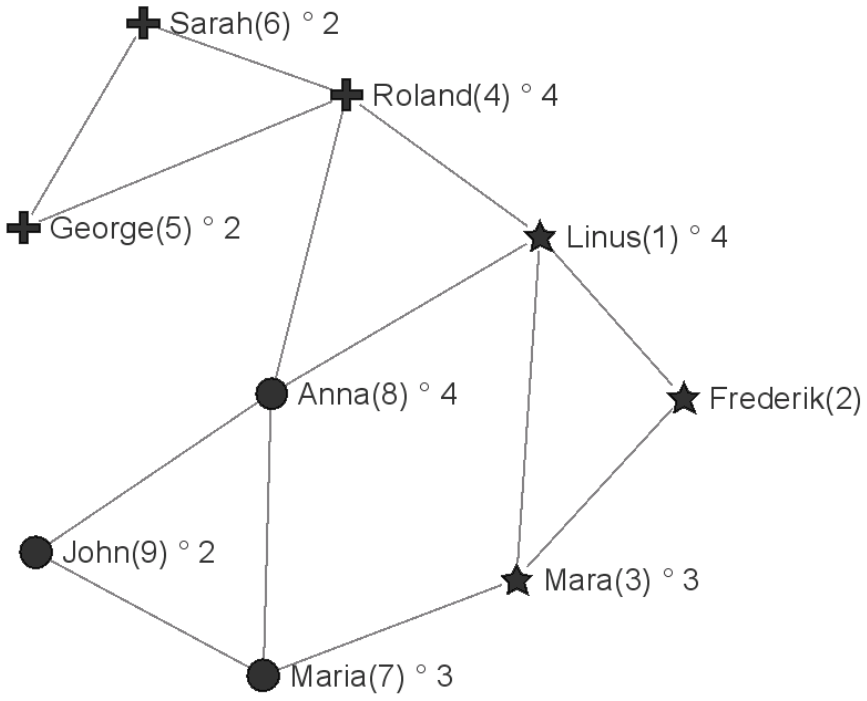


Figure 1.1: *Running example network of actors and their relations depicted as a graph. Numbers next to the actors' names indicate their degree. This illustration was created using ReSoNetA.*

[1]Of course, the term *important* is not a sufficient description. Rather there is a number of different measures for importance which are discussed in section 4.6.

[2]For sure, all names are choosen incidentally. Any associations to real persons are by chance.

Furthermore there are some special persons that should be introduced: Anna is the village's mayor and is therefore continuously seeking voter support. She maintains a relationship with Frederik, with whom she plays badminton and Roland, who is her brother. Anna is also supported by her fellow party members Maria and John. Linus is the president of the environmentalist organization and Sarah's brother-in-law.

To get a better image it might help to consider the following situation: Roland is planning to build a huge shopping center outside the village. The building site that center shall be built on compromises an important swampland habitat and numerous rare trees. Of course, the environmentalists disagree with his plans and try to convince the village's population of the bad consequences. George went to school with Roland and supports him since he, as a building contractor, will benefit from Roland's proposal. Sarah works in the tourism industry and is always looking for additional shopping possibilities.

There are a number of purposes for which this network may be investigated, for instance to advise the actors. At first, one might want to know how high ranking Roland's position is within the network to realize his plans. Furthermore, it is important for Anna to get an idea of how close the actors are to Linus as head of the environmentalists and to Roland in order to decide whether she should support Roland's plans or not.

1.4. Network Notation and its Semantics

SNA uses methods from graph theory to analyze and visualize networks. Most of the notation is also borrowed from this mathematical sub-discipline and is introduced in this section. The symbols are mainly used in describing network measures in section 4.6. There is also a number of different network types and network representations that need to be distinguished.

There are some more concepts and notions like graph permutations or minor graphs that are not described here since these are not dealt with in this paper. However, for a more comprehensive but short introduction into graph notation refer to Butts (2008).

1.4.1. Graph Notation

A network shown as a graph comprises nodes (also called vertices) that represent actors and edges (also referred to as relations) between these nodes that indicate relations among the actors. Figure 1.1 depicts the village inhabitants as nodes and their communication channels as edges. In the following v_i refers to a node of that network, where i is substituted by the actor's number. For instance, Frederik is referred to by v_2. The whole set of nodes is denoted by V(G), the whole set of edges by E(G), where G stands for the graph under examination. An edge is referred to as a pair of nodes, (v_i,v_j). Thus, the relation between Mara and Linus is written (v_3,v_1).

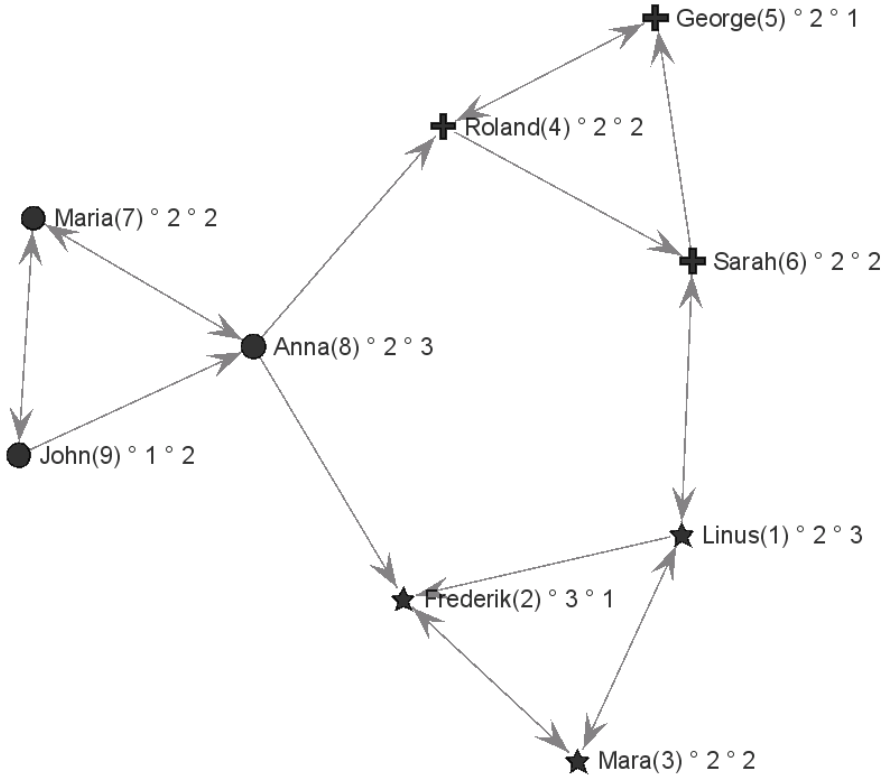


***Figure 1.2**: The running example network with directed relations showing indegree and outdegree (in that order) next to the node's name*

A network and therefore a graph may be undirected or directed, which indicates the meaning of its edges. In the case of an undirected graph, the edge between Linus and Mara indicates that information flows in both directions, which means that $(v_1,v_3)=(v_3,v_1)$ holds. For a directed graph, (v_1,v_3) stands for Mara getting informed by Linus but not necessarily vice versa. The distinction between sender and receiver is therefore significant in these networks. An arrow that specifies the direction often depicts such relations as shown in figure 1.2.

In some cases one assigns values to edges, which are called weights. Depending on the ties' meanings these values might represent the frequency of meetings between the actors represented by the end nodes of that particular edge, or a kind of importance of that relation with respect to financial support.

There are some concepts that describe sequences of vertices and edges. A walk is an any desired way through the graph that starts and ends at some node. The number of edges along the walk is its length. Walks that use every edge only once are called trials, for instance Frederik > Anna > John > Maria > Anna. Those trials which visit every node only once are called path (e.g. Frederik > Anna > Maria > John). Paths whose start and end vertices are the same are naturally called cycles (Anna > John > Maria > Anna). The shortest path between two nodes is referred to as geodesic, and more than one geodesic may exist (Wasserman and Faust, 1994).

Regarding a pair of vertices (v_i,v_j) one might identify the type of connectedness. This is easy for undirected graphs, since in this case two vertices are connected whenever there is a path between them. For directed networks a couple of graded types exist. A pair is said to be strongly connected if there is a directed path in both directions, from v_i to v_j and vice verse, that not

necessarily involves the same node(s) inbetween[3](Mara, Sarah). If such a path occurs merely in one direction the pair (v_i,v_j) is denoted as unilaterally connected (Frederik and Anna). Weakly connected pairs of vertices need to be joined by a sequence of nodes that are connected by edges of arbitrary directions (Roland and Frederik would be weakly connected if there was no tie between Linus and Sarah). If a certain connection type is true for all possible pairs in a graph or subgraph it can be said to be (weakly/bilaterally/strongly) connected.

All nodes that are (unilaterally) connected to a certain vertex - often denoted as ego - by a single tie belong to its neighborhood $N(v_i)$ ($N(v_8)=\{v_2,v_4,v_7,v_9\}$). For directed networks also all vertices (denoted as alter) that hold a relation to the ego are defined as in-neighbors $N^-(v_i)$ ($N^-(v_8)=\{v_7,v_9\}$) and nodes that hold a relation from ego are defined as out-neighbors $N^+(v_i)$ ($N^+(v_8)=\{v_2,v_4,v_7\}$).

In describing networks there are some basic measures. The size of a graph G which is the number of vertices included is written by n=|G|. The number of edges a node v_i is connected to is denoted as degree $d(v_i)$. As shown in figure 1.1 Linus has a degree of 4 since he is connected to Anna, Frederik, Mara and Roland. For directed networks one distinguishes between indegree and outdegree. In the directed example Frederik has an indegree $d_I(v_2)$ of 3 because he gets information from Anna, Linus and Mara, but an outdegree $d_O(v_2)$ of just 1 since he merely talks to Mara. The diameter of a graph is the maximal shortest path between any pair of vertices.

Sometimes it is useful to consider only a certain area of a graph, which is then called subgraph. S={Anna, John, Maria} would be such a subgraph ($S \subseteq G$). To S also belong all the edges between all vertices that are part of S. If everybody is connected immediately to everybody else within the subgroup, and this is maximal in the sense that there is no further node in the rest of the graph that could be added without relaxing the conditions, it is called a clique ({Frederik, Linus, Mara} is an example of a clique). If the nodes are connected by paths of at maximum p edges the subgraph is a p-clique ({Anna, Frederik, Linus, Mara, Sarah} is an example of a 3-clique). P-cliques whose diameter is also less or equal to p are denoted as p-clans[4] ({Anna, Frederik, John, Maria, Roland}), and p-clubs are subparts whose diameter may not exceed p but don't need to be cliques[5] ({Anna, Frederik, Linus, Sarah}).

K-plexes denote parts of graphs in which any vertex is connected to at least n-k nodes within that part (if Linus was connected to Anna, {Anna, Frederik, Linus Mara} would be a 1-plex). Therefore, regarding only connections of the subgraph every node of a k-plex has a minimum degree of n-k. K-cores are finally defined as (n-k)-plexes, i.e. the degree of their members needs to be k (Wasserman and Faust, 1994).

[3]Is this the case the pair is called *recursively connected*.

[4]Practically that means that a geodesic between two nodes of a clique may not require a node that is outside the clique.

[5]That is, they possibly were not maximal under clique conditions.

1.4.2. Types of Networks

One of the most important distinctions between networks was already discussed during the previous subsection: that of directed and undirected networks. However, there are a number of additional expressions that describe certain types of networks useful for further discussion.

Egocentric networks have an ego node as their center and consider merely vertices that are connected to ego. They are widespread in SNA since they are quite easy to record by interviewing egos (Jansen, 2006). Ego specifies so-called alteri to which she is connected to, all relations among alteri that are known to her, and also the subjective strength of all these connections. Sometimes this data is striven to verify by asking the alteri named by ego.

Simple graphs have no loops, i.e. self-choices, and at maximum one relation between any pair. Networks that are not compliant with these restrictions are referred to as complex graphs. In particular, networks that allow for more than one relation between two nodes are called multi-graphs. This concept is helpful if one considers several kinds of relations at the same time. In the example, one might split the acquaintance relations into their origin like being neighbors, relatives or playing sports together. In case Linus and Anna are not only neighbors but also relatives, they would be connected by two ties. However, multigraphs are hard to analyze and to handle for computations, so they are often considered as several networks with the same set of vertices laid one upon the other.

If there are different kinds of nodes the network is called multi-mode network. The most common multi-mode network is that of actors and events, in which every actor is connected to the event she visited. It is also called affiliation-network. These affiliation-networks may be viewed as hyper-graphs that consist of objects called points in the context of hyper-graphs, and sets of these points that are called edges, since all objects belonging to a certain edge-set are considered as connected. Thus, all actors that visited a specific event would represent an edge. If any edge contains only two points the graph would be simple again.

Complete networks denote a structure in which every node is connected to every other vertex. Thus it also fulfills the requirement of a clique. Hence, the number of edges |E| is n(n-1) for directed and n(n-1)/2 for undirected networks, and every node has a degree of n-1.

1.4.3. Network Representations

There is a number of ways to represent and store network data. An intuitive representation is the graph structure as already described in section 1.4.1. This is also a useful concept for agent-based modeling that normally uses object oriented programming languages. Nodes are represented by objects that hold their relations to other nodes, and edges are usually objects by themselves. One could ask an edge which its end nodes are, or question a node about its relations. However, for storing and mathematical calculations this representation is less appropriate, since it is less efficient. Matrix notations like an adjacency matrix are common for these purposes. Nodes are listed both in the columns and rows. The strength of the relation from the row vertex to the column vertex is assigned to any field in the matrix, and zero indicates that there is no edge between the particular nodes[6]. Of course, for undirected networks the

[6]In case there are no weights for edges, “1” indicates that the edge exists.

matrix is symmetric. The several relations of a multi-graph are often represented as an array of values for each pair of vertices in matrix notation. This is also referred to as a three dimensional matrix. An adjacency matrix of the directed running example is depicted in table 1.1.

***Table 1.1**: Adjacency matrix of the directed running example*

	Anna	Frederik	George	John	Linus	Mara	Maria	Roland	Sarah
Anna	0	1	0	0	0	0	1	1	0
Frederik	0	0	0	0	0	1	0	0	0
George	0	0	0	0	0	0	0	1	0
John	1	0	0	0	0	0	1	0	0
Linus	0	1	0	0	0	1	0	0	1
Mara	0	1	0	0	1	0	0	0	0
Maria	1	0	0	1	0	0	0	0	0
Roland	0	0	1	0	0	0	0	0	1
Sarah	0	0	1	0	1	0	0	0	0

1.5. Social Network Modeling

The term social network modeling refers to the field of simulation modeling that explicitly takes into consideration the social networks among represented agents. This section is intended to clarify why considering ties among agents is meaningful.

1.5.1. The Need for Modeling Network Relations

As discussed in section 1.2.1 relations among actors are crucial since they influence their behavior. Of course, this also applies to social network modeling where developers seek to represent the actor's behavior more or less accurately depending on a certain model purpose. Yet there are some methodical reasons in detail.

The ability to distinguish, identify, and address others is claimed to be important in modeling socially intelligent entities (Edmonds, 1998). Discrimination of individuals is crucial to treat heterogeneous partners differently. For instance, for an agent to choose profitably from various recommendations it is beneficial to know which partner is the most reliable. In order to forward a certain message from a friend to a neighbor one needs to identify and address the correct neighbor. Furthermore, it is important to have a genuine impression of interaction partners regarding their properties, their power and their goals to anticipate their behavior towards selecting the right action for oneself.

Edmonds and his colleagues do agent-based modeling[7] and seek to represent the nature and development of the agents' internal model in a way that achieves such aspects of social intelligence. Such a development needs to be open-ended, so as not to restrict the emergence of

[7]A thorough introduction into agent-based modeling and its benefits for social network modeling is presented in chapter 2.1.

possible behavior. The authors apply an adapted technique of genetic programming to generate both actions and models that evaluate their outcome in order to choose the best action.

Such a sophisticated concept of agents' internal model may not be appropriate for any kind of agent-based models. Simulations that are quite special in the agents' situation of acting and span only a short time horizon may not require a complicated mechanism of internal model evolution as is desirable for example for testing hypotheses for the emergence of norms in social science simulation. Nevertheless, Edmonds shows that social networks in which an entity is embedded are essential towards representing socially intelligent individuals. Insight into network structure is necessary in order to decide whom to ask whether a certain person is reliable, for instance. To inform a particular individual, one need to know who of ones neighbor is connected to that person, and to estimate an individual's power her position in the network is valuable information.

Moss and Edmonds (2005) especially recommend the use of socially embedded agent based modeling when clusters of volatility occur in macro level data, for instance in demands for water, that statistical models lack to predict or even fail to explain. They state that individuals are not only meta-stable entities regarding their behavior in a way that tilt effects occur, but are also influenced by other individuals they are related to. These effects may also sum up if, for instance, an actor's behavior depends on the fact, whether the number of friends who are connected to her and exhibit the same behavior or have similar attitudes exceeds a certain threshold.

If sufficiently fine-grained statistical data shows leptokurtosis[8] or clusters of volatility, these actor properties are likely to be important and often preclude the application of fixed statistical distributions on populations. The role of social interaction then needs to be considered explicitly. Therefore, in order to achieve valid micro-level results, Moss and Edmonds claim a close interaction between observations of social behavior in real societies and a conceptual development of social agents in modeling. This also refers to the social network in which the agents are embedded. Thus, expert knowledge and/or empirical studies about relations among individuals need to be incorporated into the agent-based model. As Moss and Edmonds argue, these models then serve as generators of data that indicates the kind of volatility as observed. Furthermore, these models are then accessible to cross-validation: At the micro-level the agent's social behavior may be validated by experts, whereas at the macro level the model produces aggregated data that may be compared to empirical findings. In any case agent-based models help to investigate the kind of macro data the micro-level processes might be responsible for.

In the field of resource management, it became clear that solutions based on market equilibrium assumptions too often fail, and the human dimension needs to be considered in order to achieve flexible and sustainable resource management regimes. Agent-based modeling makes it possible to represent behavior and interaction of stakeholders and to assess these in terms of plausibility.

[8]A leptokurtic distribution is identified by a higher, thin peak of the actual frequency distribution with respect to the corresponding normal distribution. This goes hand in hand with numerous significant values relatively far from the mean, i.e higher probability than a normally distributed variable of extreme values, which form a so-called fat tail.

Incorporating interaction is important since the individual's decision process is partly based on information provided by others. It is obvious that information flow in a social network with only a few connections between the actors is less than in a rather dense network under the assumption that interactions occur equally often. If information spreads quicker reaction time on external effects is shorter, and the early action may result in quite different outcomes. Thus, in contrast to economic models based on market equilibrium information flows are local, costly and time consuming, and all these factors are influenced by the network structure among involved individuals (Pahl-Wostl, 2005).

A sound representation of decision-making is even more crucial since the agent-based models in turn are used to support social learning processes, as which the decision making is perceived, and facilitate a shared problem perception, provide understanding for the system's complexity, and identify new strategies to choose from. Knowledge about attitudes, expectations, and power of the other stakeholders is required to develop promising, collective strategies.

The strength of agent-based modeling to consider social processes and interaction among decision makers is also recognized by many land-use modelers (Matthews et al., 2007). Some models analyze interactions between land-managers or farmers, incorporate these influences into decision making, and link such micro-scale processes to macro-scale.

Epidemiologists found that network structure has severe effects on the spread of diseases. For instance, there are significant differences between random and small-world networks regarding their epidemic threshold of infected individuals and their vulnerability against targeted attacks. It could be shown that, during the 2001 British foot-and-mouth disease epidemic, the removal of any one of three important vertices in the small world network of livestock markets and farmers could have prevented over 80% of the infected premises (Lem, 2006).

2. Demands in Social Network Modeling

The introduction exposed the matter of social network analysis and motivations to represent networks in social models. The subject of this chapter is to identify the needs and requirements on both, the modeling concept and software that should help scientists in developing an appropriate model. This is an important step towards achieving the aim of this work to provide ideas and means for social network modeling.

2.1. Model Design

As already stated in sub-section 1.5.1, the most plausible method for representing a population of interacting entities is agent based modeling, which enables practitioners to model heterogeneous entities with individual properties and features. This paradigm has its origin in distributed artificial intelligence. Here, a multitude of interrelated agents are used to fulfill certain tasks such as collecting information in the world wide web about a certain topic or looking for the cheapest offer of a special product.

Software agents for simulation of social matters have some categorical features and properties, which are more or less crucial for living like humans within a society (Gilbert and Troitzsch, 1999):

- Agents have potentially uncertain beliefs about their environment which they base their actions upon. These beliefs, which are often gained by the agent's perception of its environment, need to be represented by the agents, using predicate logic for instance.
- Given their set of beliefs, agents may use inference to derive additional assumptions about their surroundings.
- As agents act independently from any central entity and purposefully on their own they require individual goals that might be as general as survival or more specific like finding a friend for playing cards. Goals might be even defined by emotions like happiness, if the model shall integrate these.
- To finally achieve their goals, agents need to define sub-goals, identify prerequisites for their execution, and schedule the sub-goals in a certain order. These tasks are subject of their planner.
- The interaction among agents is crucial in order to exchange information, negotiate contracts or deal with each other. For all of this a common language is needed. Depending on the model purpose, this language might be very simple and contain only a few words or very sophisticated with the ability to be extended by the agents.
- In order to interact with others, an agent also requires a social model that includes information on existing relationships with others or experiences regarding the other agents' characteristics.

2.1.1. The Social Model

Of course, this work especially deals with the agents' social model that represents their perspective on the social network. Gilbert and Troitzsch (1999) note that it is important to differentiate between the agent's social model and the social network as the developer models it:

While the modeler might survey the whole network, the agents are not aware of relations between nodes that are strange to them. To fall back on the running example of figure 1.2, given that each person only knows about its neighborhood and the amount of their relations, with respect to their degree Frederik considers Linus as important while John considers Anna as meaningful.

Agent-based modeling as implemented by an object-oriented programming language is advantageous in that the agents may span the overall network directly: Each agent object represents a node in the network that stores links to all other agents it is connected to. Each agent only knows about its direct ties and needs to ask these in order to get to know individuals that are farther away. The modeler needs to define the agent's range of vision or alternatively give each agent the ability to decide about which relations it is willing to tell whom and which it will keep secret.

2.1.2. Modeling Network Dynamics

The dynamics of social networks are another important factor to consider in designing the model. One might impose a certain network structure upon the agents as identified by interviews, for instance. However, depending on the time scale these relations will change among the agents, alter the overall network structure and thus the meaning of position within the network. The modeler should represent these dynamics in order to keep the model valid, or at least consider them.

It becomes challenging in case the social model of an agent deviates from the real network. For instance, due to an argument between Anna and Frederik, their link vanishes. But since Linus and Mara talk merely weekly to Frederik, they get informed with a delay of days during which their social model does not change. Furthermore, the argument might be distressing to Frederik and he won't tell Linus and Mara at all, whereby it takes months to get instructed via Roland and Sarah. In case the real network and the perceived one differ, the agents need their own network representation with all nodes and relations within their range of vision.

2.2. The Ideal Agent-Based Modeling Framework

Software frameworks have several advantages in comparison to developing an agent-based model from scratch. These are listed during the next subsection. Afterwards, features that are meaningful in order to evaluate a framework are discussed.

2.2.1. Advantages of Framework Solutions

Instead of programming the basic components like simulation control, support for experimentation or project organization practitioners are able to spend more resources on theoretical and content modeling work when they use existing frameworks. The modeler does not need to be an experienced programming expert familiar with certain design patterns or architectural questions, since professional developers have already done this job for the framework. Moreover, reliability and efficiency of the software products are likely to be increased (Tobias2004). A comprehensible documentation assumed this should also outweigh the fact that sometimes it is hard to deal with software written by others.

Besides, a well-conceptualized framework helps with design decisions such as types of agents, when they should execute which behavior, or how to observe simulation results by the means it offers (Railsback2006). Therefore, the framework should provide a well-defined concept and should support a range of appropriate elements. It should not be too restrictive in providing options in order to prevent the modeler from making inadequate choices.

Specifying the model in object-oriented software is mainly put on a level with a greater flexibility and nearly unbounded potentialities. Since a software framework is used and tested by many it is in general more reliable and stable (Tobias and Hofmann, 2004). Last but not least, it is an advantage to use widespread, well-known, and sometimes even standardized software because of support and the possibility to exchange models.

2.2.2. Framework Key Features

There are a number of software frameworks that aim to ease the creation of agent-based models. In order to identify the best ones regarding purposes of network integrating models, network related properties as well as general demands on agent-based modeling frameworks need to be considered.

Some of the frameworks are developed for quite certain purposes and others claim to be universal. However, there is likely no ideal framework for all kinds of agent-based models, since requirements for different models may contradict each other. For instance, for models of large populations computing efficiency is especially significant and forces a framework to toss out components that are needless for the special case. This is in contrast to universally featured software. A possible solution would be a modular assembly system that offers all ever required features, and allows the modelers to choose only the ones they need.

Since the aforementioned all-in-one device suitable for every purpose is currently unknown, the remainder of this section is to identify the requirements on the architecture and attributes of an agent-based modeling framework in order to develop social network models. Frameworks that may or may not have these properties are then reviewed in section 3.2. Railsback et al. (2006) identify some factors for agent-based modeling frameworks that make them more productive:

- comprehensive and comprehensible documentation, including an application programming interface (API), tutorials and references
- a strong conceptual framework
- enhanced scheduling of actions
- tools for automating simulation experiments required for sensitivity and uncertainty analyses, possibly without graphical interfaces[9]
- tools for statistical output
- features that simplify tasks like graphical output, random number generators, or GIS
- researching technologies for understanding how simulation results arise

[9]Desirable is also a “scenario" mode that varies a certain parameter value and a “replicate" mode that only reinitializes the random number generator.

A special key feature of software frameworks is its documentation. The most helpful tools and the most sophisticated procedures will be useless if practitioners do not know how to apply them. Especially for scientific software it is furthermore crucial to get an insight into how methods work and thus how their output needs to be interpreted. Documentation for agent-based modeling frameworks should therefore comprise instructions for basic steps like setting up a model, adding agents, the definition of their behavior, scheduling actions, data in- and output. Tutorials that are easy to follow often enable a steep learning curve for beginners and should not only cover the basic tasks in developing agent-based models. Code templates complement convenient examples and accelerate typing. The documentation is finally completed by a comprehensive listing of the API, and descriptions of calculations and results of any routines such as scheduling, statistical distributions, statistical analysis, or network measures.

An active support by developers for instance via mailing lists supplements and sometimes even outweighs documentation. It might mean a faster familiarization with the framework, less effort while finding faults and achieving specific tasks; maybe required extensions are even developed by the support team on demand (Tobias and Hofmann, 2004). A generous framework license like BSD, GPL or LGPL is important for an unrestricted distribution of the model, and availability of source code is important to adapt the framework to specific needs and to search for bugs.

One of the core features of agent-based modeling frameworks is their scheduling and execution of various actions like agent movement, agent interaction, output of statistical data, or rendering of visualizations. The framework should ease the scheduling of these actions and provide different fine-grained adjustments that define the order of execution. Some models require synchronous agent actions, which are mostly achieved by random order, while others require the actions to be performed in a certain sequence according to the agents' power, for instance. Visualization rendering should take place at the end of every model step. Furthermore, it is sometimes helpful to schedule actions dynamically, i.e. adding them during simulation run to begin at a specific time and running for a certain number of steps (Railsback et al., 2006)..

Tools that help the user to organize model instances and parameter settings are desirable for any kind of analysis. It should be possible to store certain parameter settings to a file, in order to load and adapt these later on. This saves a lot of both nerve-racking and fault-prone entering of parameter values. For sensitivity analysis, batch mode simulation facilities that vary parameters over a certain range are highly appreciated. Switching off overhead such as graphics and diagrams often accelerates execution speed dramatically.

It is often required to generate agents according to some empirical data like age, gender, income etc., particularly for social science simulations, and network data might also be represented as matrices. Tools that read in such data and generate the agent population accordingly spare a lot of programming effort.

Simulation output is often useless without further summarization and compression of the mass of data. This requires statistical analysis features starting from simple averages of agent data up to statistical tests. Often it is sufficient to provide comfortable ways to post-process the data by

third party software like R[10], Matlab or Weka[11]. This mainly means to provide adequate output formats for generated data. Of course, linking statistical libraries directly to the agent-based modeling framework could be an even more comfortable way.

Furthermore, the programming language the framework is developed in should not be neglected. Of course, the matter under investigation recommends object-oriented languages since agents are naturally objects encapsulated with properties and functionality. However, meanwhile there are a lot of these. Java seems to be appropriate since its automatic garbage collection keeps the developer from unpleasant memory management, its strong typing is less fault-prone, and it becomes more and more standard as an increasing number of developers are educated in Java. However, when searching for the optimal software it is also crucial to consider who shall use it afterwards and what she is experienced with. Especially among modeling scientists skills in using and programming software differ strongly. For someone who lacks any programming experience, software that requires deep knowledge of an object oriented language like Java might be a nightmare and would not help at all. On the other hand such frameworks often have more potential and are unavoidable for more complex models.

2.3. What Is Required to Analyze and Visualize Social Networks?

This section lays its focus directly on the requirements for software frameworks that shall analyze and visualize social networks during simulation. Key features are identified and possible solutions are sketched. It is important that a solution incorporates all tasks like simulation, visualization and analysis seamlessly. Otherwise, dynamic analysis is prevented and users are confused by time-consuming, fault-prone switching between applications (Perer and Shneiderman, 2006).

Obviously, the demands on visualization aspects are more engaging than those of the analysis of network measures. This is partially due to an abundance of literature on network visualization, since it is prominent in other fields besides social science. Further, it should be considered that network visualization also deals with adequate representation of network measures.

2.3.1. Analysis Measures

Many different network measures arise in the field of SNA. On the one hand these are helpful to characterize a network that emerged during the simulation run in order to interpret it. On the other hand, network measures might be used for instance to formalize an agent's position within the network in order to adapt its behavior or its neighbors' behavior to its personal position. To consider the directed version of the running example, Frederik might choose to ask Linus at first if he knows someone who could lend him a trailer. However, if Mara cuts the connection to Linus and establishes one to Anna, Mara would be the first person Frederik asks, since she has more friends. Often it is required to experiment with measures in order to find the one that best incorporates the desired meaning and to apply sensitivity analysis.

[10]R is an integrated suite of software facilities for data manipulation, calculation and graphical display (Cribari-Neto1999).

[11]Weka is a collection of machine learning algorithms for data mining tasks containing tools for regression, clustering, data pre-processing, classification, association rules, and visualization.

To spare the modeler the effort to implement all possible measures for the network at hand an appropriate framework should provide a wide range to choose from. The framework should make it easy to add certain measures for calculation and output these for comparison. Selecting only the ones that are required in a certain model run is important due to performance, because measure calculation may be computationally demanding. Since there are many measures that often have very similar meanings a thorough documentation of the measures' relevance, their parameters and their algorithms is crucial. Besides, the arrangement of measure in groups and categories makes it easier to interpret and manage them. Since the framework shall ease implementation of, and access to, new network measures, it therefore needs to declare clear interfaces.

When a network measure is used to influence an agent's behavior, it is very important to distinguish between real and perceived network models, as described above in section 2.1.1. The framework should therefore provide means to adjust the network area as a base for measure calculations. For instance, every agent could have a property that indicates his radius of vision, and the calculation then cuts the network behind the furthermost visible node. If the measure is centrality for example, the framework could either assign zero or an average value to all other agents.

2.3.2. Visualization

Even quite small networks are hardly lucid when their nodes are highly interlinked. This is especially true when the relations these links represent have different meanings. To make the structure of a social network easily comprehensible, it is important to layout the nodes of the network in a sound way. This is even more serious in the light that a viewer always tries to interpret the arrangement of nodes and their distances in between (Jansen, 2006).

As Brandes and Wagner (2004) state, network visualizations need to fulfill two demands: The structural properties of the modeled network should be encoded accurately in visualization and the viewer should be able to comprehend this information easily. Since aesthetic network drawings that are easy to capture barely model the structure, exactly these demands are hard to achieve simultaneously. Furthermore, it is important to decide which aspects of network structure shall be represented. Correct node positions that depict centrality adequately and locate the most central node in the middle surrounded by less central ones are potentially different from those representing prestige.

However, with respect to the observation that the viewer tries to interpret the network representation, it is worth considering whether some crucial aspects may be combined even if accuracy suffers a little. For instance: one could imagine a network representation that shows the most central node in the middle and also maintains adequate distances between vertices according to their geodesics. Unfortunately, developing such a layout algorithm seems to be complicated, and at short-term appropriate workarounds need to be investigated. In any case, it is important to document the aspects that influenced the layout process.

Once agent-based models are developed, they should be productive and simulate interactions and behavior among the represented population as it evolves over time. Thus, regarding the network visualization, it is important to consider the dynamics in network structure. One need is that nodes are located more or less statically on the graph, moving just slightly while the network

changes. This is essential to be able to identify the altering parts of the network. Rebuilding the network from its roots and twirling it while updating is quite useless in that sense. The viewer may only follow the network dynamics if one sees what is changing, and not only that something changed. One challenge here is to insert new nodes in a sound way that preserves the current arrangement, but finds a position for the new node that is coherent with the other vertices. Some authors refer to this visualization feature as preserving the mental map between successive layouts and categorize it as the crucial objective in dynamic graph drawing (Baur and Schank, 2008).

Whether this is possible or not depends strongly on the layout algorithm that is chosen to visualize the network. However, for most algorithms one can imagine a solution. For the large family of force-directed algorithms a node could be added at a certain adequate position and only the forces and thus positions of neighboring nodes need to be updated. Baur and Schank provide a more thorough method when they also introduce forces between consecutive visualizations. Minimizing the overall stress therefore also means increasing the stability of the dynamic layout. See subsection 4.7.2 for a discussion on challenges of achieving sound display updates.

Another possibility to ease the comprehension of network dynamics is to mark changing elements by special styles that differ from the ones of static objects. A node that was just removed may be painted in red, while vertices that have emerged just before may be drawn in green. An even more thorough approach is to fade added or removed nodes during a certain amount of time steps (Perer and Shneiderman, 2006). This way the viewer recognizes the differences and may inspect and compare the situations before and after the changes. Of course, similar considerations apply to edges. Section 4.7.3 shows how fading network elements were realized in ReSoNetA.

A crucial duty of network visualization is not only to layout the network in a sound way but also to display the potentially numerous values and measures in a coherent manner. This requires a sophisticated selection mechanism for demanded measures, since otherwise the display gets crowded and confusing. So-called probing enables the user to display enhanced data for a certain agent that is selected. An alternative is tooltips that show additional information when the cursor is located at the particular node. Probing is also a way to adjust specific agent properties during simulation run, for instance to investigate how a change in income for a certain agent alters the simulation results. Both are important for entering numbers, the ease of entering for instance by sliders and the accuracy provided by entering numbers in a text field (Railsback et al., 2006).

In order to identify certain nodes within the network, various shapes and colors according to social variables or other node properties are useful. Freeman suggests different icons to distinguish gender, age classes or ethnic groups (Freeman, 2005). The running example uses different shapes to identify environmentalists (stars), politicians (circles) and local firm owners (crosses), respectively. However, color-coding may furthermore be used to visualize the extent of an agent's property value. For instance, the higher the degree, the darker the agent's color.

Ranked node lists also help to identify agents whose property values are relatively high or low when the agent that is marked in the list is also highlighted in the graph. Dynamic node filters

expect a threshold value for a certain property from the user and display only those nodes which exceed this threshold. This way it is possible to exclude all nodes from a graph whose properties fall below a certain value and are thus uninteresting for specific analysis. For subgroups in a network there are some ways to ease orientation. They might be embedded in a (colored) hull, replaced by a super node that sums up all in- and outgoing edges or by assigning notes to them (Perer and Shneiderman, 2006).

Snapshots from static network visualizations and videos that capture the network dynamics are helpful to analyze and compare different simulation runs. Videos also provide the possibility to investigate the network evolution in reverse, or shift back and forward between steps several times. Linking the produced pictures to their according simulation setup comprising the parameters is important for a careful analysis. This might be accomplished by printing the data in the visualization or identification via distinct file names.

Of course, the way a visualization is instantiated and adjusted is also meaningful. Worthy features are the potential to add and adjust displays during run time via a graphical user interface, to have more than one display at a time to compare different layouts, or different network areas, and reinitialization of network layouts that miscarried.

2.4. Dealing with Existing Models

Once again, the aim of this work is to provide means for practitioners that assist them in modeling social networks. As stated in section 2.2, agent-based modeling frameworks are good bases to start from. A promising possibility to add easily applicable features is a module or library, which interacts with a certain framework and user code. It needs to be anchored on the existing model code and framework with little effort and minimal changes.

The decision left is which agent-based modeling framework to choose, and which version. Apart from the demands discussed, before an answer one should also consider which software was used to implement existing social network models. Many enhanced agent-based models regarding human-environment interaction systems have been developed over years. Their implementation was started in past, using a certain framework, and the effort to convert such complex models to another, more recent agent-based modeling framework is simply too much. That is because the framework developers often either do not take backwards compatibility into consideration, or that the improved concepts of new software may require completely different model implementations. That's a pity, since newer versions of agent based modeling frameworks often provide useful features for visualization or analysis that could facilitate the modelers work. Therefore, the model's further development requires flexible solutions such as modules that are easily dockable even if the used framework is not up-to-date any more, and that make accessible the benefits of recent modeling software. This could be accomplished by a kind of converter library that translates the model structure as the new framework requires, or provides interfaces for specific features to the new framework.

3. Review: Existing Tools in the Field of Network Analysis and Modeling

When new software or new features for an existing application are to be realized, in general there are two options: Implementing the software or features on your own from scratch, or search for products to do the job or at least help in implementing it.

The second option is not always best choice since there is often a confusing abundance of software, and it might take a lot of time to examine it in order to find the one that fulfills the needs. Nevertheless, it is often worth looking for an existing application or several tools to start from. In particular in the fields of network analysis and modeling there are dozens of general purpose programs, specific stand-alone applications or libraries. For finding an appropriate tool for analyzing social networks in multi-agent based models it is a good idea to get a detailed synopsis of existing tools and applications.

A promising way in order to develop a suitable solution is to explore multi-agent modeling frameworks and their capabilities to support network analysis directly. While some of these have a build-in network support that may or may not include visualization features, others require of the user that he implements network arrangements and their analysis on his own. Of course, for evaluation also other features than network support need to be taken into account, since their absence might weight out the network features. Such multi-agent modeling frameworks are discussed during the second section.

There are plenty of certain purpose libraries that are useful to be integrated in either stand-alone applications or frameworks. The functionality the libraries were developed for reaches from general visualization to network analysis. This chapter reviews two of those libraries which play a meaningful role in network analysis and visualization. Since some of the frameworks fall back on those practical software collections they are described before the frameworks.

Besides, many diverse stand-alone applications have emerged in the field of SNA. Nearly all of them provide basic network measure calculations and descriptive statistics. Import and export functionality of network data exists, but not all applications support the common GraphML format. The majority of tools also integrate means to visualize networks with varying extent. Apart from such basic features the applications are mostly intended for specific purposes like aggregating parts of a network to super nodes, blockmodel analysis, enhanced visualizations, or dealing with large networks.

Compared to the demands for social network analysis in modeling as stated in the previous chapter there are some disadvantages for such stand alone tools across the board. One of the main issues is data transfer between the model and analyzing tools which often needs to be done time and resource consuming via files. Furthermore, in general it is impossibility to explore network evolution since the analyzing application is not fed by single network events like an added node or a re-linked edge. Thus, it has difficulties to visualize changes in network structure. Because of these obstacles a comprising review is omitted, even if certain tools could assist social network modelers. Various social network analysis applications are discussed in Huisman and van Duijn (2005).

3.1. Libraries

Libraries in general are software collections that implement features for specific uses and topics. Libraries support the modular assembly concept, since they provide code that is invoked partly or as a whole by other applications. For network analysis there are libraries that display and layout graphs, calculate measures, constitute general graph frameworks, or provide classes for efficient data representation. This section describes two of these which are widespread in graph visualization and computation and might ease the development of a network visualization and analysis library.

3.1.1. Piccolo

Piccolo is a toolkit for creating custom user interface components and is robust, full-featured and integrates smoothly with the standard application code. The library is developed at the University of Maryland as free and open source software under the BSD license, and addresses primarily educational and research users that are not interested in the low level details. It supports both Java and .NET applications.

Piccolo features a so-called scene-graph model which is a hierarchy of objects and cameras in which each node has a transform method, which allows every object to be arbitrarily translated, scaled, and rotated. Besides, the toolkit supports animation, transparency, panning and picking, which is identifying the object that is currently under the mouse cursor. The camera model supports multiple views and overviews. Since the library clearly identifies the regions that need to be updated, repainting the screen is efficient (University of Maryland, 2008).

Piccolo is designed as a monolithic toolkit, which means that there is a huge base class that provides all functionality. Many subclasses inherit this functionality and adapt it to their needs as a text node, slider or image. In contrast, polylithic toolkits use compositions of many specialized classes like faders, zoomers, or transformers to achieve certain functionality. While polylithic designs are more flexible since they enable easy addition of new functionality classes, monolithic toolkits are easier to use: the developer does not need to keep track over the many different classes but works with a general hierarchy of types (Bederson et al., 2004). Thus, the visual components that are provided by piccolo may be easily adapted by sub-classing and overwriting methods. For instance by defining a new **paint()** method of a text node one could realize semantic zooming, which hides details when the scale factor in a zoomable user interface (ZUI) falls short of a certain value.

3.1.2. Java Universal Network/Graph Framework (JUNG)

JUNG is kind of a general language to model, analyze and visualize any data that may be represented as vertices and edges. It supports networks with undirected, directed, and even parallel relations. Besides, multi-mode networks and hypergraphs may be handled (OMadadhain et al., 2007). The first version was released in August 2003, and version 2.0 in April 2009 with 23 releases in between show an active development. JUNG 2.x is a redesign that overcomes many of the design decisions that in past turned out to impede further improvements. For instance, it allows objects of any type to be nodes of a network, and even be part of several networks, and introduces generics for flexible implementations.

The JUNG API package serves as a basis for the library and provides core interfaces that define graphs and their behavior. This enables the user to create his own custom graphs that implement the according interfaces and thus work with the rest of JUNG (OMadadhain et al., 2008). The graph and especially the network interfaces provide a lot of useful methods to alter the network or access specific elements like a random node of the neighborhood of a given vertex. The possibility to filter the set of nodes for operations or visualization facilitates certain analysis of complex graphs which shall be done for social networks. For instance, one might like to consider only men or only women for a certain analysis.

There are plenty of algorithms for calculation of network measures such as network distances, flows and centrality-, PageRank-, and HITS-based importance implemented, and possibilities in drawing graphs reach far. A number of algorithms from graph theory, data mining, and social network analysis, such as routines for clustering, decomposition and optimization are considered as well. Graph layout algorithms comprise the most popular ones like Fruchterman-Reingold, Kamada-Kawai, clan-based decomposition, or circle layouts. See section 4.7 for a detailed discussion.

Algorithms to generate a network are currently limited to random graphs. These comprise scale-free Barabasi-Albert networks, power-law distributed Eppstein graphs, and the binomial model of Erdos-Renyi. Import/export features already exist but should be enlarged in future. At the moment JUNG supports reading and writing of Pajek files and matrices. For GraphML the library only provides a reader, and the possibility to output graphs in this format is lacking.

3.2. Modeling Frameworks

This section reviews five common agent-based modeling frameworks which are NetLogo, Repast J, Mason, and Repast Simphony. Criteria to evaluate features and shortages and to estimate the benefits of a certain framework were discussed in section 2.2. The following evaluation also focuses on the spared effort by users and developers because of functions the framework already provides and on the necessity and the developer's work for compensation of particular functions that have not been included in the software (Tobias and Hofmann, 2004). Table 3.1 shows a synopsis of the reviewed software.

***Table 3.1**: Compact information on the reviewed agent-based modeling frameworks*

Name	*Language*	*License*	*Developer*
NetLogo	Java	source code only partly available / non-commercial use only	Northwestern University Evanston Ilinois
Repast J	Java	BSD	Repast Organization for Architecture and Design
Repast S	Java	BSD	Repast Organization for Architecture and Design
Mason	Java	Under own license (Open Source)	George Mason University Fairfax Virginia

Name	*Latest Rel.*	*URL*	*References*
NetLogo	12/2007	http://ccl.northwestern.edu/ netlogo/	(Wilensky, 2007)
Repast J	08/2005	http://repast.sourceforge.net/ repast_3	(Collier, 2002)
Repast S	07/2008	http://repast.sourceforge.net	(North et al., 2005)
Mason	06/2008	http://cs.gmu.edu/~eclab/ projects/mason/	(Luke et al., 2005)

3.2.1. NetLogo

NetLogo is a programmable modeling environment for simulating natural and social phenomena, originally specialized in mobile individuals with local interactions in a grid space. It is written in Java and continuously developed at the Center for Connected Learning and Computer-Based Modeling of the Northwestern University in Evanston/Ilinois. Special is its own, easily comprehensible modeling language representing the next generation of the series of multi-agent modeling languages that started with StarLogo (Wilensky, 2007). This language has plenty of built-in primitives and thus is comparatively compact.

NetLogo comprises extensive documentation including a large manual, tutorials, and an enfolding example model. Many kinds of 2D and 3D displays including a flexible plotting system may be added using both, menus and drag-and-drop. They allow for scaling, rotating and movie creation. The application furthermore enables the user to save and restore model states, and runs are exactly reproducible across platforms. Its scheduling capabilities are only basic, but a switch for pseudo-concurrent scheduling is provided.

It must be noted that NetLogo was primary designed for educational purposes down to the elementary level. Thus it provides an easy start in agent-based modeling by its drag-and-drop manner and its own modeling language, but this also limits the areas of application (Railsback et al., 2006). Noteworthy is also that NetLogo's source code is only partly available - in contrast to all other reviewed frameworks.

3.2.2. Repast J

Repast J (Recursive Pouros Agent Simulation Toolkit in Java) is now developed under the BSD license at the Argonne National Laboratory under responsibility of the Repast Organization for Architecture and Design (ROAD). To produce a toolkit based on its well-grounded key abstractions, but implemented in Java, was the original motivation for developing Repast. From the beginning it should comprise facilities for creating, running, and displaying multi-agent-models and collecting data from their simulation. Goals that influenced Repast's development were ease of use, a steep learning curve, extensibility, and robustness (Collier, 2002). While at first with Repast J the focus was laid on experienced Java users, later on Repast Py was released as a version that should be usable without programming skills. Repast .NET is quite similar to its Java counterpart.

Repast J supports a range of spatial relationships the agents can interact within, such as 2D and 3D grids, hexagonal grids, hexagonal tori, vector spaces, GIS environments and networks, but only one at a time. Visual capabilities include histograms and sequence graphs, snapshots, and

quicktime movies. The schedule engine is quite powerful since it provides both a preparatory execution and cleanup stage at which actions might be scheduled. Furthermore, actions are optionally invoked for one-time, repeated, at pause, or at end execution. Actions are represented as **BasicActions** which could be grouped for running in an **ActionGroup**. However, some certain schedules require a rather complicated work around like scheduling display actions at the end of every tick (Railsback et al., 2006).

Since Repast J is explicitly intended for social science simulations and modeling socially acting agents, it offers a comparatively enfolding network modeling support. This comprises a node-generating factory that reads in network definitions from file or initializes them as small world, random density or square lattice networks, a recorder that may output networks as adjacency matrices, and a few simple network statistics for analysis. Furthermore, Repast J enables the creation, storage, and loading of model parameter files, probing of agent properties, a batch mode, and extensive library support for random number generation.

Tobias and Hofmann (2004) evaluated four freely available agent-based modeling frameworks developed in Java[12] with respect to their modeling purpose, and concluded that "the most suitable simulation framework for applications-oriented theory and data based modeling is clearly RePast" (section 5.26). Highlighting the ability to restart models via GUI, the experimental manager, and the geographical as well as network support also Railsback concluded that Repast J is "certainly the most complete Java framework" (Railsback et al., 2006, page 622). Nevertheless, the paper criticizes an insufficient documentation and lack of features for statistical analysis.

3.2.3. Mason

Mason is a conceptual framework for organizing and designing general multi-agent-based models, but focuses on computationally demanding models with many agents executed over many iterations. While it does not support a specific domain like social science simulation, there are libraries that the core model interacts with. The developers provide extensions for certain purposes such as *socialnets* which provides basic network statistics or *jung* which is a bridge to the JUNG library. The open-source and free framework is developed in pure Java at the George Mason University in Fairfax/Virginia (Luke et al., 2005) and intended for experienced coders that want "something general and easily hackable to start from" (George Mason University, 2008). The latest version released in June 2008 and 14 releases since 2003 indicate an active development.

Mason's significant strengths that distinguish it from other multi-agent simulation frameworks are the separation of core model and visualization, its possibility to interrupt model simulations and make checkpoints in order to run them on different platforms, and its compact and fast implementation. All of these issues ease simulation of computational intensive since large agent populations.

Visualization features comprise 2D and 3D displays that enable zooming, export of movies and snapshots, and the creation of a variety of diagrams. The philosophy of separating the core model and its visualization is consistently realized to such an extent that the user needs to install many visualization features additionally, like the Java Media Framework for creating

[12]Besides Repast Tobias et al. investigated Swarm, Quicksilver and VSEit

movies or JFreeChart for making charts. However, the Mason developers provide a zip file that contains all separately required libraries. The guarantee of complete reproduction of results across platforms assures the user freedom in distributing simulation runs over several computers. However, Mason is not meant to support parallel execution of a single simulation across multiple networked processors (Luke et al., 2005).

Further aspects are a central model class which serves as a communication device by holding information needed by other objects and its real-valued schedule. The scheduling concept is designed for execution speed but complicates programming, since there is only one method which might be scheduled per object. Thus, more sophisticated behaviors require long winded work-arounds (Railsback et al., 2006).

3.2.4. Repast Simphony

Repast Simphony is a completely renewed Repast framework first released in December 2007. Currently, version 1.2 is available. It is fully embedded in eclipse and comes as a plug-in for the well-known integrated development environment. The most impressive new feature is click-and-point model building, which also seems to be a major reason for integration into eclipse. The user drops several building blocks like properties, behaviors, triggers or control elements on a drawing area and connects these via arrows. Afterwards she specifies names, types and behavior through several text fields and drop down lists. Once the diagram is finished, Groovy source code is generated, which is then compiled to ordinary java classes[13] (North et al., 2007).

Building models this way is mainly intended for users who are inexperienced with Java programming. However, the click-and-point functionality is still in its infancy. While theoretically there is no limit using task blocks for every single statement that otherwise would be written directly in the Java file, this is cumbersome and inefficient for complex behaviors. Currently just a few patterns which represent particular code blocks are sufficiently documented through the user interface and make modeling accessible to non-programmers, but the number of supported templates grows from release to release. Since help functions and input control regarding the correctness of user inputs are still lacking its usage is also fault-prone and requires good attention[14].

Repast Simphony incorporates many of the recommendation given in Railsback et al. (2006): The conceptual framework was revised with respect to a modern way of organizing agents and environments. Repast S introduces *contexts* that host agents and are organized in a hierarchical manner. Contexts allow the agents to orient themselves with respect to a set of data, depending on which context they belong to (Tatara, 2007). *Perspectives* are responsible for the spatial arrangement of agents and comprise continuous space, grids, GIS, or networks.

---

[13]Groovy is a recently published, dynamic language that compiles straight to Java bytecode but has additional features. It tries to combine the ease of scripting languages with the functionality of Java (Henry, 2006; Codehouse Foundation, 2008).

[14]For instance it is important that agent class names match the input in the model score file. Thus, it would be helpful if the application gave a warning when there is no such class name as given in the model score file.

The framework provides data export facilities and interfaces to common (statistical) analyzing toolboxes like R, VisAD, Matlab, or Pajek for network analysis. But connections to the JUNG library for analysis are still poor. The user may apply statistics on a specific network by clicking a button. She may select from several degree distribution statistics for the network, discrete distribution, or graph statistics on the layout. But only the selection of degree distribution statistics for networks operates and displays a table of values for degree, in-degree and out-degree. Storing the data to a file or even copying is not possible.

Repast Simphony's batch mode is based on Repast J and was further developed. Noteworthy are also extensive 2D and 3D visualizations, freeze-dry of simulation runs, data export, and the generation of portable models that may be easily executed on any system that supports Java. User data that are required to configure displays or define outputs are collected via comfortable wizards. The integration of several libraries for genetic algorithms, neuronal networks, random number generation, and automated Monte Carlo simulations also belongs to Repast Simphony's strengths. Social network representations were improved by integrating the JUNG library and by enabling access to many graph visualizations. So the network projection is actually a wrapper for JUNG networks. By this high level integration of the advanced JUNG software Repast Simphony seems to be the most supporting framework for social network analysis.

A drop of bitterness is the rather sparse documentation which comprises only a Java API, two tutorials that only span certain applications, and an incomplete reference for some common concepts. However, the developers are aware of the need for an enhanced manual as stated on the very active user's mailing list and recently compiled an FAQ page. Features for simplification of more common tasks and providing researching technologies for understanding how simulation results arise could be further enlarged.

3.3. Summary of Reviewed Frameworks and Libraries

After evaluating four agent-based modeling frameworks one needs to make a decision on which is the most appropriate for social network analysis. Since the frameworks differ in their conceptual and architectural character it would mean too much effort to develop a module for more than one framework.

NetLogo has its strengths in educational applications, which is reflected by the enfolding documentation and visualization features. However, its restricted source code availability makes it hard to add additional network features.

Repast J is a comprising and stable framework which is especially intended for social science simulation and includes a comparatively wide range of network features. It is probably the most wide-spread ABM software even if it gets substituted by Repast Simphony, and at least partly designed to be extended.

Mason is appropriate for experienced programmers working on models that are computationally intensive (Railsback et al., 2006). It supports basic network analysis and seeks to integrate the JUNG library. Its focus on experienced programmers that are looking for something easily hackable is a drawback for most social scientists that have no comprising programming education. On the other hand Mason's modular principle which for instance allows switching on and off visualization should be a paragon for other frameworks. A meaningful advantage of both

Repast versions compared to Mason is their spread among social science simulation while only a few models were developed in Mason and even less of the field of social science, as it seems. Furthermore, most social network models that shall be analyzed and visualized are limited in their size and may not require a framework that is specialized to computationally demanding models.

Repast Simphony finally seems to belong to the next generation of agent-based modeling frameworks that is also appealing to non-programming experts. Even if some parts are still immature and require further development, it offers a wide range of functionality and is anticipated to spread out more and more, especially in the community of social science simulation. Its network functionality can be viewed as a well-founded basis that is worth to be extended.

However, many existing and on-going projects were written in Repast J and would benefit from access to modern and extended means for network analysis and visualization. This way, those models could be developed further without the effort to convert them into the completely new software. Therefore, a plausible solution is either to extend Repast Simphony and port older models, or to develop a library for Repast J models that integrates features already present in Simphony in order to extend these with further, requested functionality. In a second step these additional features could then be integrated in Repast Simphony.

Regarding the reviewed libraries it should be noted that JUNG is currently the most comprising, free network library. Piccolo is a potential visualization collection. It is mature and already incorporated by Repast Simphony, why it is plausible to stick to this library for visualization.

4. ReSoNetA: Accessing Promising Features for Social Network Modeling

After demands on a network analysis and visualization software for modeling are identified and a solution is roughly delineated, this chapter first seeks to summarize requirements for the new module that is developed throughout this thesis, and then presents that library. Its name, ReSoNetA on the one hand reflects its purpose that is to say the **A**nalysis of **So**cial **Net**works. One the other hand it is intended for **Re**past J models that were originally written in version 3.1 or before. The software, which is licensed under the BSD license according to the Repast frameworks, may be downloaded at http://www.cesr.de > Downloads > Software > ReSoNetA together with the Java API documentation.

Since Repast Simphony is seen to be promising software that provides meaningful features for network representation it shall be investigated in more depth with a special focus on its architectural concept during the second section. The challenges and limitations of integrating both Repast versions is delineated in the third section. The following segments are then meant to clear the library's software design (section four) and to describe its features in some detail (section five). The remainder then introduces features for dealing with network measures and describes some of these, and discusses visualization techniques in more detail.

4.1. Summary of Demands

To help modelers in dealing with social networks the software module should be easily compoundable with existing model code and make it simple to use additional features for analysis and visualization. The visualization of networks, their analysis through network measures, and simple means for customization constitute the main aspects throughout developing the new library.

Network measures and their time series are significant means in analyzing networks and should be simply accessible without much coding effort. The user needs to get an overview of existing measures and their meaning, should have the possibility to add interesting ones for calculation, and view the results. Output features are necessary to store and post-process gathered measures, and their scheduling should be adjustable to define the amount of data that is stored.

The demand on visualization techniques is to facilitate understanding of network evolution via fading network elements, only slightly changing network layouts in order to preserve the mental map, and information displays directly on network elements. In order to interpret the viewed network correctly information on how the layout was generated is mandatory. Diverse networks and different focuses in analysis require various layouts to choose the appropriate from, and to compare several aspects.

In order to simplify customization the user should be able to add both her own components and newly developed features to the framework without trouble. This is particularly important for network layouts and network measures. All features need to be documented thoroughly to make the features available to users. Well commented examples further facilitate access to the library.

4.2. Repast Simphony in Detail

This section at first discusses contexts and projections to get familiar with the way agents are organized in Repast Simphony and the possibilities it offers.

The next subsection then reviews the steps that are required to build a model in order to get to know Repast Simphony's way of defining the infrastructure and configuration. This enables a judgment whether it is efficient to port existing Repast J models to Repast Simphony, or if a more sophisticated solution is required.

The remainder is then dedicated to the (network) projection and visualization architecture, which is crucial to display the networks and to anchor user control elements.

4.2.1. Concepts

Contexts and Projections: Probably the most important concept is that of contexts and projections, as also mentioned in section 3.2.4. However, these are not isolated but part of a wider concept of agent organization. A context is a kind of reservoir of agents. Every agent needs to belong to a context, and contexts may also form a hierarchical order. An agent may also be hosted by more than one context and change them dynamically. Contexts represent the agent's environment and the entities may adapt their behavior to the current context they are within. Therefore, a context may have an internal state, possibly represented by a simple data field, and optional behavior that influences its state. However, the Repast developers have not yet provided a clear example of how such dynamical behavior might be achieved.

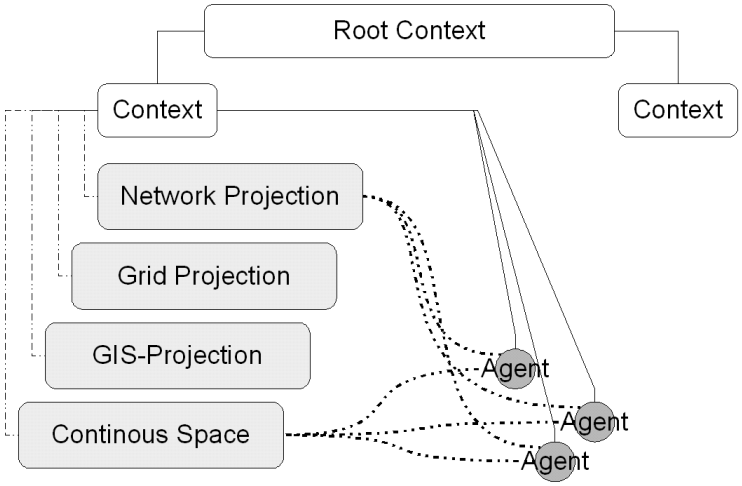


***Figure 4.1**: Contexts and projections in Repast Simphony*

Projections specify the environment the agents are within and impose a structure on the context. This might be continuous space, a grid - both in 2D or 3D -, GIS information, a network or any user implementation of the **Projection** interface which assures listener and predicate support. Predicates are specific queries to a projection, for instance whether two agents are linked in a network.

In general a projection is valid for all entities in a context, and a context may include more than one projection, for instance a grid that assigns spatial positions to the agents and a social network which represents friendship relations among them. That makes the concept quite flexible

since by interchanging projections agents might be tested in different environments. Without any projection agents would not be able to interact or communicate with each other since they could not address the others. Nevertheless, projections and agents do not need to know each other and are independent unless agents have to deal with their projection, which is also the natural way (Howe et al., 2006). Figure 4.1 outlines the relations between contexts, agents and projections. The root context may consist of several sub-contexts which contain the agents. One or more projections may belong to a context, and the agents of that context are associated with each projection.

Furthermore contexts may contain value layers which are mostly associated with a certain projection and provide data, for instance a data field per grid cell which agents may access and take into consideration regarding their next actions.

Data Collection: Repast Simphony provides a sound concept for collecting data in order to store, visualize or post-process them. Through the graphical interface the user defines data sets containing single agent data or aggregations. In a second step these data sets are selected for file output, to be displayed as a chart or passed to post-processing applications like R, visAD or MatLab.

4.2.2. Building the Model

Building a Repast Simphony model in general comprises three steps:

1. Defining the model infrastructure
2. Generation of Java/groovy-code
3. Configuration of displays, diagrams and model outputs

The model infrastructure is defined in the so-called model-score file which contains a tree-like structure. The base is the root context, to which nested contexts may be added. Agents then join the contexts and are linked to their corresponding Java classes. It is also possible to define a Java class as root context that adds all the rest programmatically and also creates agents in a network or on a grid. Such a class needs to implement **ContextBuilder** (Tatara et al., 2006).

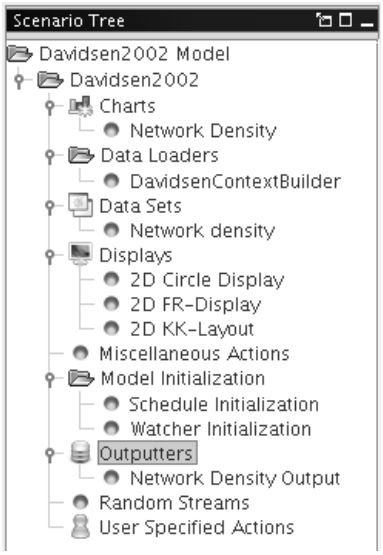


***Figure 4.2**: Screenshot: Repast Simphony's Scenario Tree*

The click-and-point procedure for creating simple agents and manual coding for complex structure also can be combined. During model run Repast Simphony provides a so-called scenario tree that shows the context hierarchy and contains all configurations like defined data loading, data sets, charts that visualize data sets, displays and output files for each context. A screenshot of the visual representation is shown in figure 4.2. The user may edit existing items by double clicking or add new entries by a right click, and in any case a wizard appears that asks the user to give some input.

For displays the visible projections need to be specified as well as the styles for each network and agent classes which facilitate distinction between different agent types. All these settings can be saved to a folder with the model score file and a number of

XML files and of course can be restored.

Once the configuration is done, the user starts the simulation. Charts and displays are shown next to the scenario tree and the user may for instance add new agents to a grid, delete or clone some and edit their properties. Furthermore agent entities may be moved on the grid by mouse, or relations could be added to a network (North et al., 2007). Even simulation runs may be saved (which is called freeze-dry) to a file or a database at any point and restored later on. This is helpful in particular when simulations need a pre-run before parameters are going to change.

4.2.3. Visualization in Repast Simphony

This section reviews the visualization architecture. This is in particular interesting since Simphony's visualization capabilities are modern, comprehensible and also address social network visualization. Docking these capabilities to existing Repast J models seems promising.

Algorithm 1: *The* ***update()*** *method of the interval updater*

```
public void update() {
       counter++;
       if (hasCondition(Condition.ADDED,Condition.MOVED,Condition.REMOVED)){
              layout.update();
              updatePerformed = true;
       if (counter == interval) counter = 0;
       } else {
       if (counter == interval) {
              layout.update();
              updatePerformed = true;
              counter = 0;
       } else updatePerformed = false;
       }
       conditions.clear();
 }
```

It is possible to display several projections one over the other, for instance a network over a grid space[15]. Compared to Repast J the renewed version of Simphony furthermore provides adjustable updating for displays. That means, the user may decide whether a certain display shall be updated on a given interval, every time an element belonging to the projection was added or removed or every time an element moved in case of a grid projection, for instance.

Unfortunately, as algorithm 1 shows, the interval updater does not work consequentially and also updates the display every time a new element moved, was added or removed even if the interval is not met (lines 3-5). This is annoying if the user knows about regular element changes and uses the interval updater on purpose to prevent the display from being redrawn every time step.

In Repast Simphony the agents as entities of a display and also projection elements like network relations are painted according to style definitions. There are some basic style classes like **DefaultEdgeStyle2D** that may be sub-classed and further specified by user implementations and **EditedEdgeStyle2D** which is editable via a wizard that stores information in XML-files. Agent styles may comprise data such as shape, label appearance, label font, and color and size values that may optionally be adjusted by agent properties. That means the user specifies a range for the shape size and the actual size changes according to a certain agent value like its energy.

[15]There are exceptions for GIS projections: Only one GIS projection can be shown at a time in a certain display

Displays are configured by the user via a **DisplayConfigurationWizard** that is invoked from the **DefaultDisplayMenuItem** in the scenario tree. The item also creates a **DisplayComponentControllerAction** for each display and adds this to the registry. The controller action instantiates the **DisplayProducer** which in turn creates the user specified layout and an **IDisplay**-object. This is an instance of **Display2D** for two-dimensional displays or an instance of **Display3D** for spacious visualizations respectively. The display parameters as given by the user are passed on to the producer by the **DisplayDescriptor**. For network projections the display consists of a **NetworkDisplayLayer2D** that holds visual items for edges and **StyledDisplayLayer2D** for every type of agents.

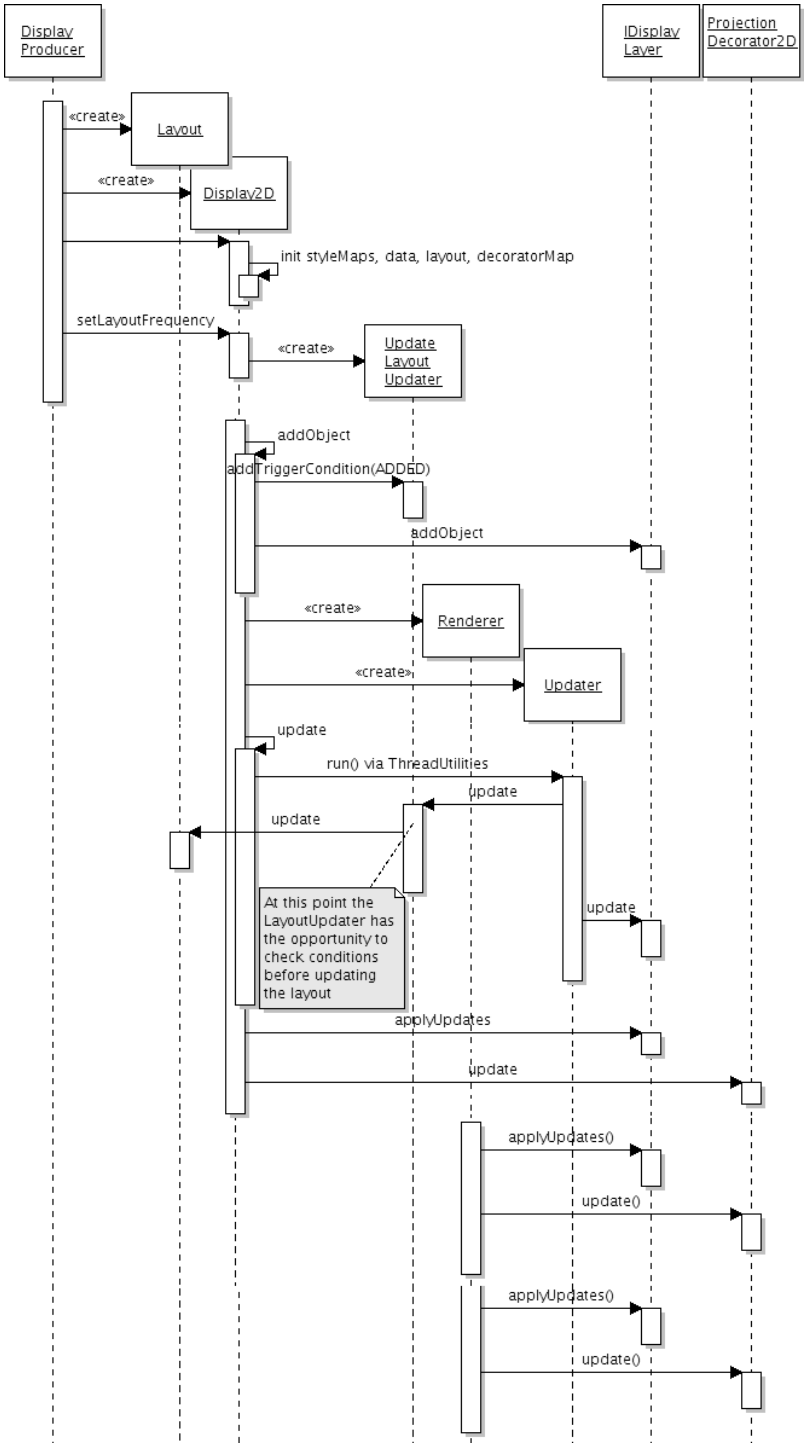


***Figure 4.3**: UML Sequence Diagram: Updating network visualization in Repast Simphony*

The producer causes the display to instantiate a certain **LayoutUpdater** which decides when a layout is updated as discussed before. Updating of displays is done in two steps that are triggered by the display whose **update()** method is scheduled by the controller action. First the display's **Updater** invokes the layout updating and layer updates that create visual components for new objects for instance. During the second stage the updates are applied which means adding and removal of visual items to or from the layer and repainting. The update processes are executed in their own threads. Figure 4.3 shows main aspects of the visualization in Repast Simphony in a sequence diagram.

4.2.4. Data Sets

Data sets need to be defined in order to plot charts or output data to a file. They always refer to a particular context. Simphony offers a wizard that helps the user to collect the necessary instructions for generating data sets. At first an agent class needs to be specified that is then scanned for methods which may provide data to gather. Since the wizard only asks for one agent class it is obviously not possible to combine data of several agents into one data set. In a second step the user adds rows to a table which might represent simple data like a return value of an agent's class method, aggregated data like the average of values or more complicated data generated by a formula script. Afterwards one specifies how often and when during a time step the data shall be gathered.

4.2.5. Porting from Repast J to Repast Simphony

Since Repast Simphony has a different concept and architecture it is not compatible backwards, i.e. models developed for Repast J do not immediately run in Simphony. Nevertheless it is possible to adapt the model code and port these models to the new framework. This adaption process comprises a number of steps:

1. Eliminate visualization code
2. Transfer scheduling instructions and parameters from the **SimModel** class to an instance of **ContextBuilder**
3. Create a model score file
4. Re-configure displays and loggers via the scenario tree

Depending on a concrete model this is a lot to do and certain effort needs to be redone. The complete set of visualizations is no longer usable and every single chart needs to be added to the Repast Simphony scenario tree again. Thus, for some models developed in Repast J the porting process is not reasonable.

4.3. A Library as a Bridge between Repast Versions

As the summary of section 3.3 concludes and the difficulties of porting indicate, the aim should be to develop a software library that makes extended network analysis and visualization features available for Repast J models. This could be achieved partly by making Repast Simphony code accessible. Therefore and because new features could be integrated into Simphony later on such a module could be seen as a bridge between the Repast versions. This section discusses

challenges and limitations which is helpful to further outline the necessary architecture of the new library.

4.3.1. Challenges in Integrating Repast Simphony in Repast J

Since Repast Simphony is based on a complete redesign of Repast J there are a lot of architectural differences which mostly mean obstacles in combining features of both. Some of these are due to the conceptual changes regarding contexts and projections; others have their roots in the extended network support of Repast Simphony which is lacking in Repast J.

The following subsections treat differences in the network representation, the communication of network changes and the process of scheduling. Two more segments address challenges due to the context and projection concept and the question how to deal with necessary adaptations.

Network Representation: By integrating the JUNG library in Repast Simphony much network support comes with this new version. Networks are represented by the **ContextJungNetwork** which serves as a wrapper for the pristine **JungNetwork**s. In Repast J the network is represented in an ordinary **ArrayList**, mostly hold by the user class that implements **SimModel**. Networks are based upon the **Edge** and **Node** interfaces and their implementing classes. Such a simple list is not compliant with any network analyzing library and needs to be converted in order to use their features. Furthermore it is more complicated to find out whether a certain edge exists in the network, how many edges the net contains or to detect neighbors of a given element. The modeler needs to implement such functionalities on her own or use work-arounds like traversing the list of agents.

Algorithm 2: *Edge creation in Repast J*

```
Edge edge = new DefaultEdge(source, target);
source.addOutEdge(edge);
target.addInEdge(edge);
Edge otherEdge = new DefaultEdge(target, source);
target.addOutEdge(otherEdge);
source.addInEdge(otherEdge);
```

Adding edges in Repast J is kind of troublesome. The user needs to take care for adding an edge at both the start and target nodes separately. Furthermore there is no explicit distinction between directed and undirected networks. For representing an undirected link two edges should be added in both directions. While in Repast Simphony a simple **network.addEdge(source, target)** is sufficient to add an edge to an undirected network, in the older version a couple of lines are necessary to achieve the same result as depicted in algorithm 2. This is quite fault-prone in maintaining the network.

While Repast Simphony accepts objects of any type as vertices in a network and only provides **RepastEdge** for tie representations Repast J requires nodes to implement the **Node** interface. Whereas Simphony stores the edges with the network object, in the earlier version the node objects hold connections and therefore need facilities to add and remove edges or return any out-going ties. Clearly, Repast Simphony is more flexible in this regard.

Contexts and Projections: Repast J neither provides universal containers for both agents and the definition of their projections, nor a general interface for all kinds of projections. In order to integrate specific functions like generating data sets ReSoNetA needs to incorporate contexts or

provide appropriate substitutes. Required is a new object that is compatible to Simphony's **Context** interface on the one hand and makes it easy to add any desired objects from Repast J on the other.

Projections are necessary since the visualization system is based on them, and the **ContextJungNetwork** as a central class with respect to ReSoNetA's purposes of network analysis is a projection itself. Consequently, besides the context adaptor there is also need for a projection adaptor. Its task is further described during the next subsection.

Communicating Network Changes: The recently introduced concept of projections constitutes a basis for Repast Simphony to organize the communication of changes in the network, i.e. adding or removal of nodes or edges, in a much sounder way than before. Since nodes and edges are part of the network which is a **Projection**, whenever user code adds or removes objects the network keeps track of these actions and may inform registered **ProjectionListener**s. For instance, the **Display2D**-object registers itself at the projections it represents and informs the layers to which the changed object belongs whenever it receives a **ProjectionEvent**. This way the layer may handle the representing visual item accordingly.

Repast J does not provide the feature for layouts to react on added or removed nodes or edges. The layouts may only update themselves at every time step or a certain interval. So the layout does not need any information about changes within the network. To have the complete set of network elements in an ordinary list and their relations at every time step renewed is enough. Since the list does not provide any information there is no possibility for a layout to register at the network in order to receive network change events without altering the model severely.

To base the update process on the occurrence of network changes one either has to store the network's state of the previous time step and compare it with the current, or to develop an observer-subject mechanism. That means that some kind of projections serves as listener on changes in the user model. While the first solution is quite demanding on computational resources the latter one seems to require rather severe interventions to the existing Repast J framework.

Scheduling: The scheduling framework is essential for every agent based modeling framework. It is responsible for agent actions, for instance weekly interactions with others or daily water consumption. Besides the model's specific actions also visualization updating, data import and export or network measure calculations need to be scheduled.

The scheduling frameworks of Repast J and Simphony are similar with respect to the possibilities they provide like generation of action groups or concurrent scheduling. Scheduled actions may be one-time or repeated and are specified at both frameworks by their starting time, interval length, duration, and priority that determine whether an action is executed at the end of a time step or not.

However, Repast Simphony does not use the same schedule classes as Repast J does, and thus Repast Simphony actions may not be scheduled at the **ScheduleBase** of Repast J. While this class uses **BasicAction**s as foundation for all actions Simphony provides the class **DefaultAction** that implements the **IAction** interface. Therefore any actions of a Simphony process that shall be used within the Repast J controller need to be converted into

BasicActions. Since the conversion is straight forward this is not difficult but makes is necessary to edit any Simphony code that schedules something.

Reimplementing vs. Extending: Since the library is to be based on large parts of Repast Simphony features, many classes of Repast Simphony need to be included in the library code. The question is whether to re-implement these classes or include existing Repast Simphony libraries in order to extend and adapt them.

Extending existing classes has several advantages: Only those parts have to be reimplemented that need an adaptation, and all the rest can be left untouched. Furthermore, if this is well-done in most cases of changes e.g. bug fixes in the original Repast Simphony code these changes are adopted automatically. It is sufficient to exchange the old library with the new one instead of troublesome incorporation of changes resulting from the ongoing Repast Simphony development.

In case of Repast Simphony extending becomes difficult because of many private members that are not allowed to be overwritten by adapted code. For instance, the **DisplayProducer** references a **DefaultDisplayDescriptor** as private member. If for some reason the descriptor class needs to be interchanged there is no possibility for a subclass to fulfill this adaptation and the whole code that uses the descriptor needs to be reimplemented. Desirable are protected members that offer access from subclasses and classes of the same package but prohibit alteration of members for any other object.

Of course, it actually is a good habit to keep as many entities of an object as possible private in order to encapsulate regions of code that belong together and protect these against unauthorized change. On the other hand, an application that shall be open for extensibility needs to define interfaces whenever possible. Especially in the area of agent based modeling where many institutions are interested in extending the software since their requirements are quite different, interfaces are important to both enable and facilitate valuable extension. Repast Simphony defines some of such interfaces like for adding own visualizations or projections. However, further progress in this field would be appreciated.

There is one more disadvantage of extending that counts pro reimplementation: Some classes which need to be adapted contain a lot of methods ReSoNetA does not use. Furthermore these methods depend on potentially many extra classes that need to be included but are never used. For example many classes provide functionality for both 2D and 3D features but the library is not meant to support 3D-visualization.

4.3.2. Limitations

ReSoNetA can not handle complete model conversions from version 3.1 to Repast Simphony since such a process will either require plenty of changes to the user code or very complicated steps to inspect that code. The library will make rather important features accessible to existing Repast 3.1 models.

Some interesting Simphony features like freeze-dry and 3D visualizations will not be available. One reason is the difficulty to bring together both simulation engines. Each has its user interface that controls the simulation. Since a crucial aspect of ReSoNetA's developments is its application to Repast 3.1 models with as little effort as possible that would mean for instance to substitute the Repast Simphony control panel by the Repast J one. Another problem could be due to

performance since running both systems in parallel would waste a lot of computational resources. Two schedules would be necessary to be maintained and two graphical user interfaces for visualizations would be required in order to use both frameworks in their complete magnitude of features.

4.4. The Library's Concept and Software Design

Before describing the library's core features during the next section this one is dedicated to the clarification of leading design decisions and architectural principles. These pay attention to the challenges listed above and shall ease understanding of feature implementations that otherwise would seem to be strange.

4.4.1. Connecting the Module

ReSoNetA can be combined with existing model code by just a few lines of code as anchors. These instantiate and configure objects which either substitute Repast 3.1 objects with adapted ones or which are completely new objects. There are defined places for these few lines of code and the overall structure of the established model is preserved. Classes like **ReSoNetAInitializer** provide static methods that combine several instructions and need to be invoked once inside a certain method of the user model. For detailed instructions on how to adapt the user model code refer to the user manual at http://www.cesr.de > Downloads > Software > ReSoNetA.

4.4.2. Mapping of Agents

While Repast Simphony claims to allow any type of object to be part of a network, Repast 3.1 uses certain agent classes that may operate as nodes of a network. Since these agent classes are the only characteristics of objects that belong to networks apart from their membership to an ordinary **ArrayList**, ReSoNetA uses this welcome restriction to assure certain network features. As a main feature agents who belong to a network need to report changes regarding their edges to a listener. As stated in subsection 4.3.1 this is the only efficient way to inform for instance network visualizations of network changes. Thus, the new library works with a range of interfaces for agent classes that allow specifying the functionality for agents as network nodes. It provides default classes but the interfaces also give the user the freedom to design implementing classes on his own.

The variety of interfaces not only preserves flexibility but also seems to be confusing at first sight, but knowing the idea behind should make it easy. The classes are depicted in figure 4.4. Starting at the top of the hierarchy, **ChangeFiring** is merely a marker interface for any objects that fire property changes to registered listeners without specifying any requirements on implementing classes. Its only purpose is to point to the fact that any objects the ReSoNetA module keeps track of need to report their changes to it.

The **NetworkChangeFiring** interface clears demands on objects that shall fire network changes. It defines methods to add and remove listeners and one that fires the **NetworkChangeEvent**s to registered observers. The **NetworkChangeEvent** is designed quite similar to a Repast Simphony **ProjectionEvent** including a change type and the changing object.

Since the user model part of the library does not know about Repast Simphony edge objects the **NetworkProjection** itself has to look up the edge object in the network by its source and target which are provided by the Repast 3.1 edge object as subject of the **NetworkChangeEvent**. Since Repast Simphony does not allow for multiple edges at a certain relation this is possible.

The **NetworkChangeFiringNode** adds requirements of an **EdgeHandlingNode** to the change firing support while **NetworkChangeFiringNodeList** is a marker interface for change firing supporting collections that report network related changes of their elements. **AnyChangeFiring** is intended for future extensions of the module regarding changes related to other projections like grid or GIS. Implementing classes need to provide facilities for any such listeners, but for the moment it is equivalent to **NetworkChangeFiring**.

To achieve alignment with the Repast 3.1 node classes **DefaultNode** and **DefaultDrawableNode**, ReSoNetA provides **DefaultFiringNode** and **DefaultFiringDrawableNode** which both extend their Repast 3.1 counterparts and implement the **NetworkChangeFiringNode** interface. Thus they override methods that change their edges, fire events to listeners and then forward to the overwritten methods of their super classes. The same applies to the **NetworkChangeFiringArrayNodeList** which implements the **NetworkChangeFiringNodeList** interface.

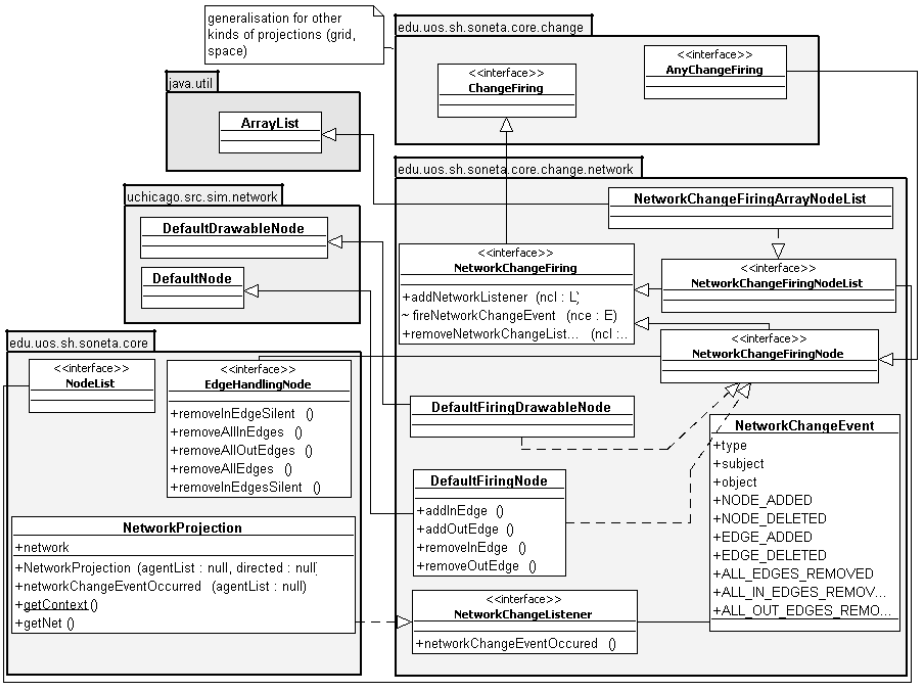


***Figure 4.4**: UML-class diagram of the network change support*

To be compatible with the Repast Simphony context architecture for every agent list a new **DefaultContextAdapted** is created that holds the agents belonging to that list. This class also stores its instances in a static collection accessible via **getContext(NodeList<T>)**.

4.4.3. Configuration

User configurations are mainly required for visualizations and data output. Repast J provides only a few means to edit parameters via a graphical user interface. Instead, adjustments need to be

done programmatically. Since ReSoNetA aims to apply Simphony's visualization and data set framework it is plausible to incorporate the way they collect user data via wizards that guide the user as well. These wizards ask the user for parameters in a sound order and may validate her input immediately to prevent annoying runtime errors later on. ReSoNetA seeks to inform the user by reasonable warning messages every time he gives an invalid value as early as possible.

As already known from the introduction of Repast Simphony in chapter three the collected input is stored in certain descriptors that are passed to actions which initialize a display or command a data recorder to fetch data and store it. ReSoNetA uses these descriptors and completes them by additional fields that are typical for the library. The descriptors may not only be used in combination with graphical data input but may also be configured programmatically if the modeler wants to hard code determined configurations.

4.4.4. Extensibility

The potentiality of extending the library was identified as one of the important demands in section 4.1. Thus, this module aims to provide an architecture that makes it easy to add new components like layouts or network measures. A sound way to enable the addition of new classes that provide network measure calculations for instance is the composite design pattern (Gamma et al., 1995) that organizes supplying objects in a hierarchical manner. The library provides a base class that offers registration means for children which in turn might appear as parents. The root class could then collect network measures its children provide. On request for a specific measure it asks its children recursively to provide a calculating method. Custom network measure offering classes may be registered at the base class somewhere in the user model code. Implementation details of the mechanism are depicted in the next section.

4.5. The Library's Features

The following subsections discuss the main features of the ReSoNetA library in detail. These are accessing network measures, the handling of data output, network visualizations, and further improvements in the last segment.

4.5.1. Accessing Network Measures

In section 2.3.1 two issues were marked as crucial aspects regarding network measures: At first, a framework should provide a wide range of network measures to choose from and experiment with. A second demand on the software is to ease implementation of and access to new network measures; and therefore to declare clear interfaces.

The JUNG library already provides a wide range of network measures. However, accessing these for experimental purposes is rather difficult: the measure rankers need to be instantiated with the graph and certain parameters, its calculation needs to be scheduled, and the results have to be fetched from the rankers. Because there is no sufficient documentation, it is also hard to find the proper class and guess what it exactly produces. ReSoNetA simplifies dealing with network measures by their classification in various measure categories and making them accessible via a graphical interface.

In ReSoNetA a measure consists of a **Measure** object that knows about its calculation, its value type, possible parameters for calculation, and its description. The measure object prepares the calculation, for example by instantiating JUNG network rankers. Parameters are passed to the calculation classes or define scheduling values like the interval the computation is scheduled at and are represented by **String\Object** pairs in a map. When defining measures the map might be prepared by **String** keys and optionally their default values. These values might then be read by user code. Adequate values can be added, and the map is passed to the measure object. The utilities class may also scan the parameter map and asks for input, if the value object is of type **Double**, **Integer** or **Boolean**. Afterwards it passes the parameters to the **Measure** object, where the parameter values are sent to the object that calculates the measure values. Figure 4.5 in the middle shows the measure classes.

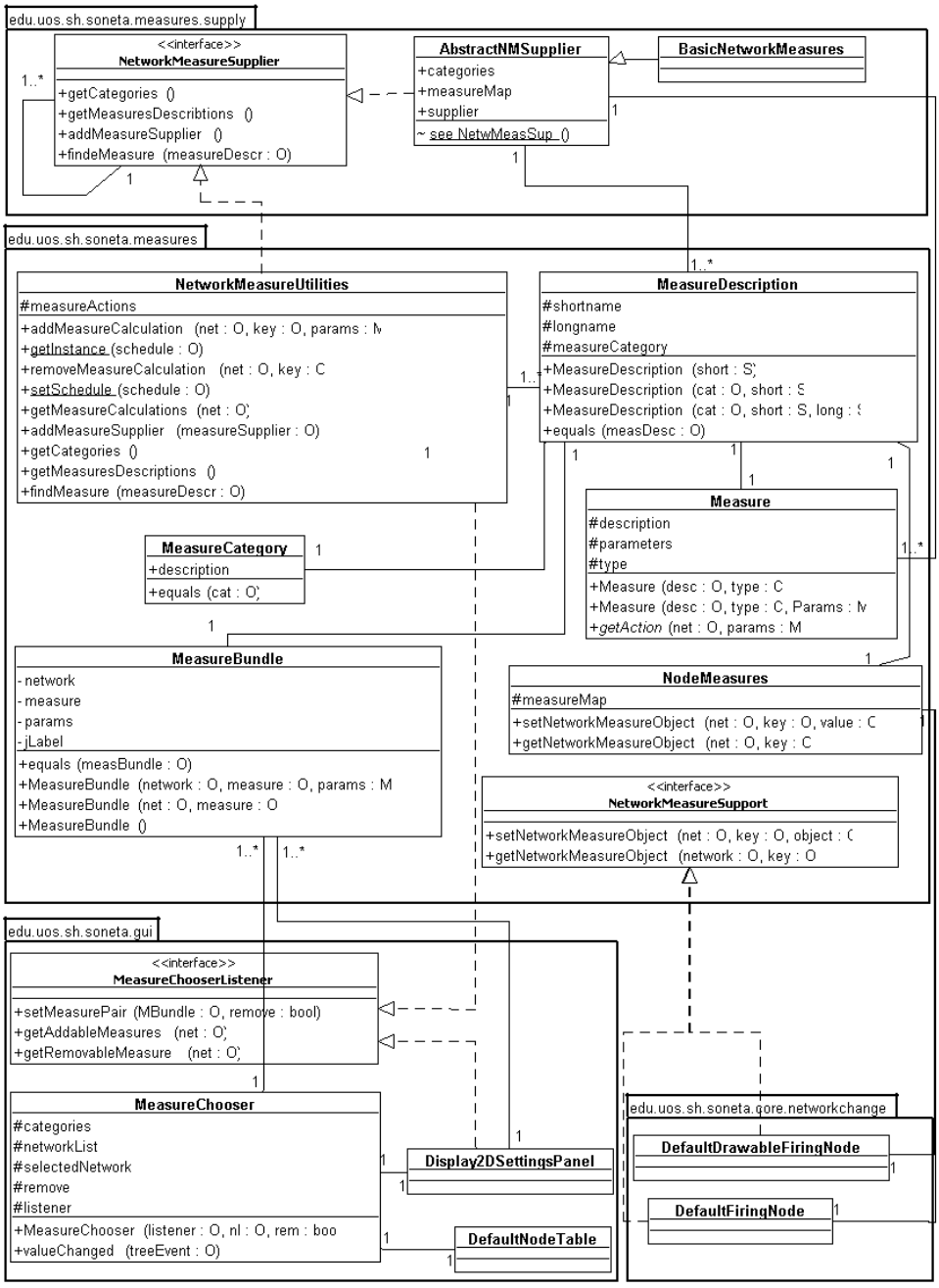


***Figure 4.5**: The UML-Diagram shows all classes that contribute to the network measure support*

The user may finally schedule certain measures for computation and display the calculated ones in the so-called node table, or directly at the nodes in the network visualization. For these settings ReSoNetA provides the clearly arranged measure manager which lists measures as rows and possible applications like displaying in the node table or storing in a data set as columns. This matrix' inner fields are checkboxes by which the user may adjust whether the according column function applies to the measure or not. Buttons allow adding new measures and data set configurations to the matrix. Measures and data sets are also deletable by remove buttons next to the rows and according columns.

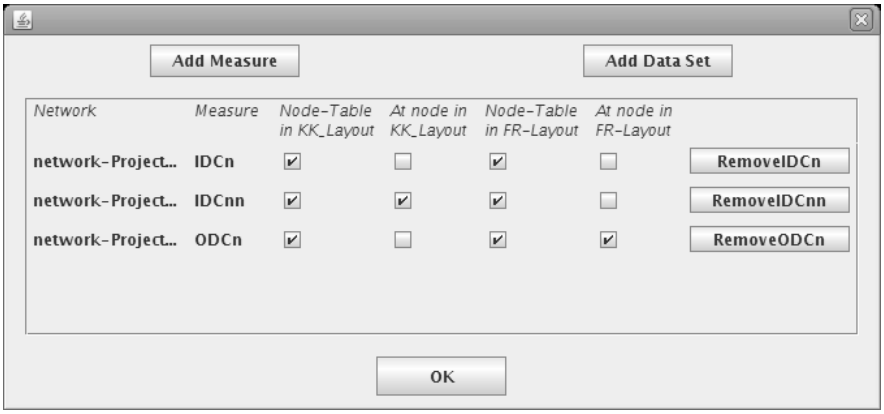


***Figure 4.6**: Screenshot: Measure Manager*

For the measure selection process the **MeasureChooser** is used, which implements a dialog and interacts with a **MeasureChooserListener** (see screenshot 4.8 next page). The listener provides all measures the user might choose from and receives the selected item. In case of adding measures for computation the listener is implemented by **NetworkMeasureUtilities**.

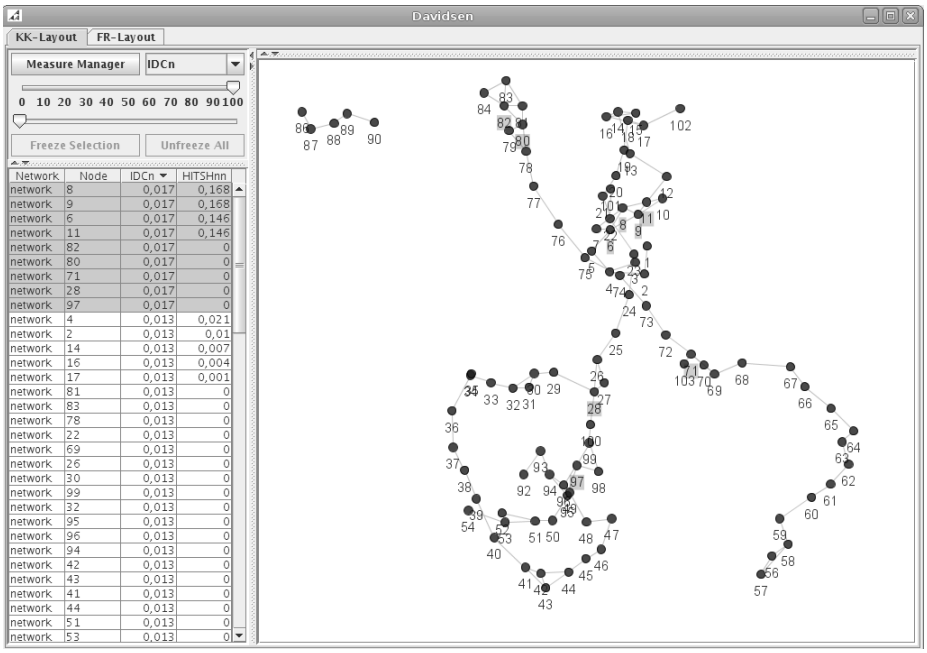
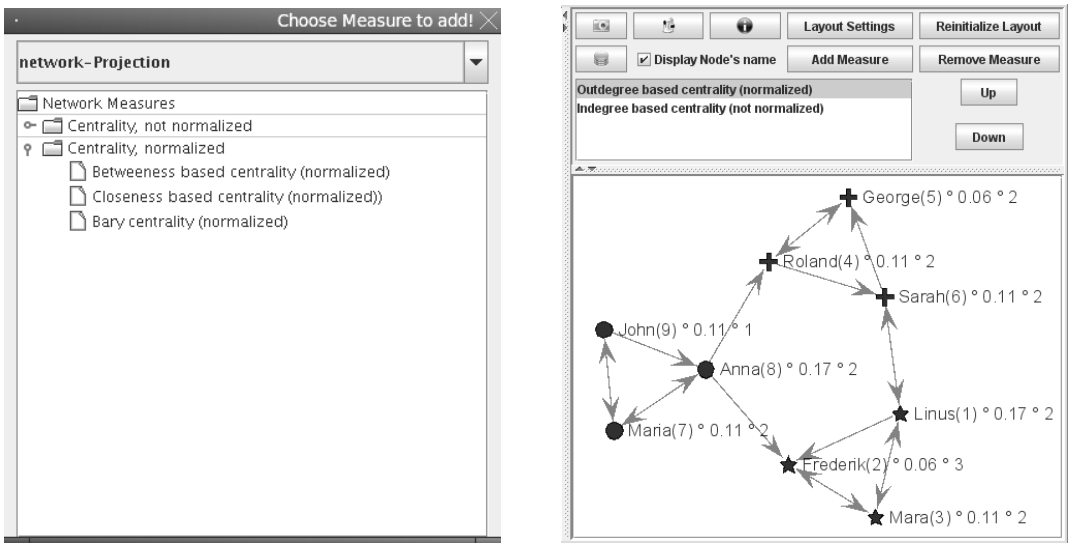


***Figure 4.7**: Screenshot of the node table applied to a network of 100 nodes.*

The node table facilitates the ranking of nodes according to calculated measures. As the screenshot of figure 4.7 depicts, the nodes can be listed in the table according to any computed network measure. The sliders above the table are used to cut the node list from below and from

above. The measure that is used as criterion is selected from the drop down list. Optionally, nodes that are filtered out by the sliders may disappear from the drawn network or marked by a frame around their label. Furthermore, the table allows selecting certain nodes, which are then highlighted by a transparent filling of their label. This is also preserved across updates of the visualization. In the example, using the sliders it may be easily noticed that the most authoritative nodes are located in the middle of the network. It is also possible to adapt the node table in future to incorporate any other agent properties to be used for analysis besides network measures.

Each display contains a panel that is shown in the other screenshot of figure 4.8. Besides the measure manager the user adds measures that were scheduled for calculation before via the button “Add Measure”, which are then added to the nodes the next time the display is updated (for instance via “Reinitialize layout"). All shown measures are also immediately listed below in the presented order which may be adapted by “up” and “down” buttons. A remove button excludes selected items from the nodes.



(a) Screenshot: Measure Chooser Dialog (b) Screenshot: Adding Measures to Display

***Figure 4.8**: Screenshots of adjusting network measures*

The **NetworkMeasureUtilities** class is the centre of the network measure support. In adding measures for calculation it looks up the demanded **Measure** object, calls for its **BasicAction**, schedules the action and stores it in order to possibly remove the measure from computation in future. To let the measure objects themselves instantiate the action object offers a wide range of potentialities. For instance, measure objects could initialize further helping classes and even schedule methods without dealing with the schedule framework. As a disadvantage this procedure is quite specific for the Repast J framework. However, it is possible to convert **BasicAction**s into their Simphony representatives straight forward.

To allow a flexible implementation of and access to new measures the process of looking up measure objects follows the architectural recommendations of subsection 4.4.4. It is a bit complicated but powerful. Implementing a combination of the delegate and composite design patterns (Gamma et al., 1995) it is easy to add new classes that provide additional network

measure objects. All these classes of which **NetworkMeasureUtilities** is the head implement the **NetworkMeasureSupplier** interface. Therefore starting from the measure utilities class new measure suppliers are added to it which in turn might host further suppliers. When asked for all measures that might be added for computation **NetworkMeasureUtilities** asks its children to provide the measure descriptions for which they store **Measure** objects. These in turn ask their children and merge all received measure descriptions before they transfer it to their parental class. Searching for a measure that maps a given description works analogous. The hierarchical order makes it easy to organize different categories of measures in different classes. The process of adding network measures for computation and the look-up of measure descriptions and measures in the network measure supplier framework is depicted in the UML sequence diagram of figure 4.9.

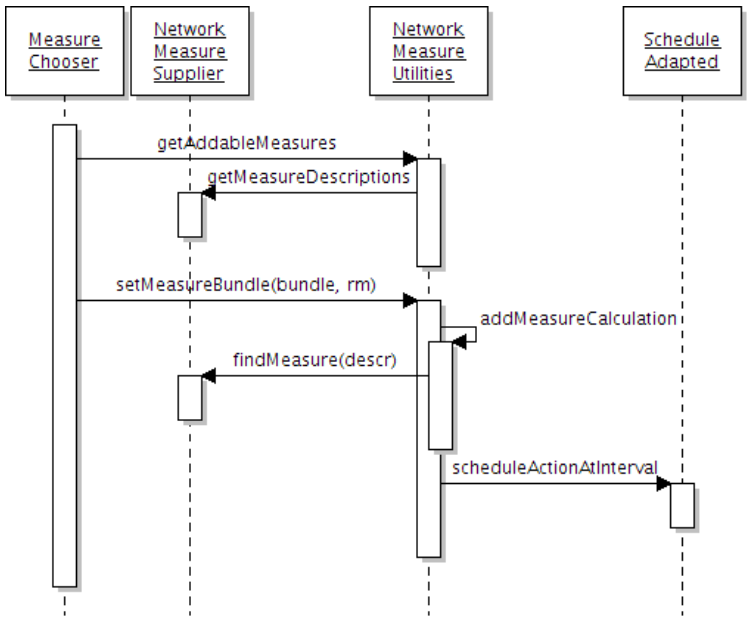


***Figure 4.9**: UML Sequence Diagram of Adding Network Measures for Computation*

The process reduces redundancies compared to a greedy variant that reads and stores all measures at a single object. To save time and space the supplier classes could be even adapted in a way that they instantiate measure objects dynamically as they are requested. The procedure allows users to create whole trees of network measure supplier classes which are added to the library by a single line of code:

```
NetworkMeasureUtilities.getInstance().addMeasureSupplier(userRootSupplier);
```

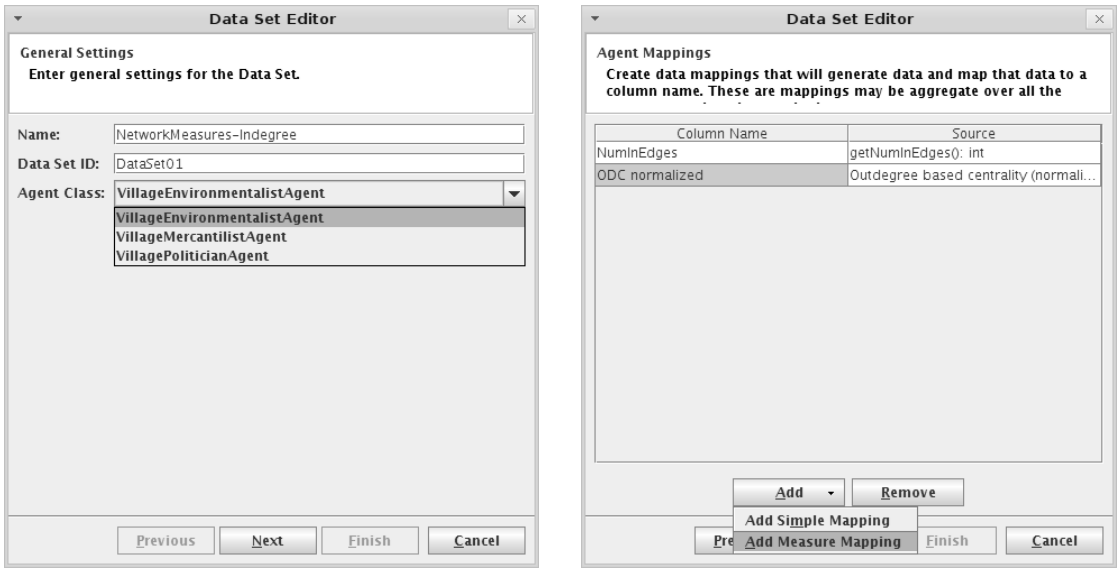
Finally, calculated measures need to be stored somewhere, and the most plausible place is the nodes themselves. Nodes or agents that are capable of network measure calculations need to implement the **NetworkMeasureSupport** interface. It declares methods for setting and getting measure values, given the network the measures are calculated for and their description. ReSoNetA's default agent classes, **DefaultFiringNode** and **DefaultDrawableFiringNode** implement network measure support and thus make it easy to experiment with network measures.

4.5.2. Data Output

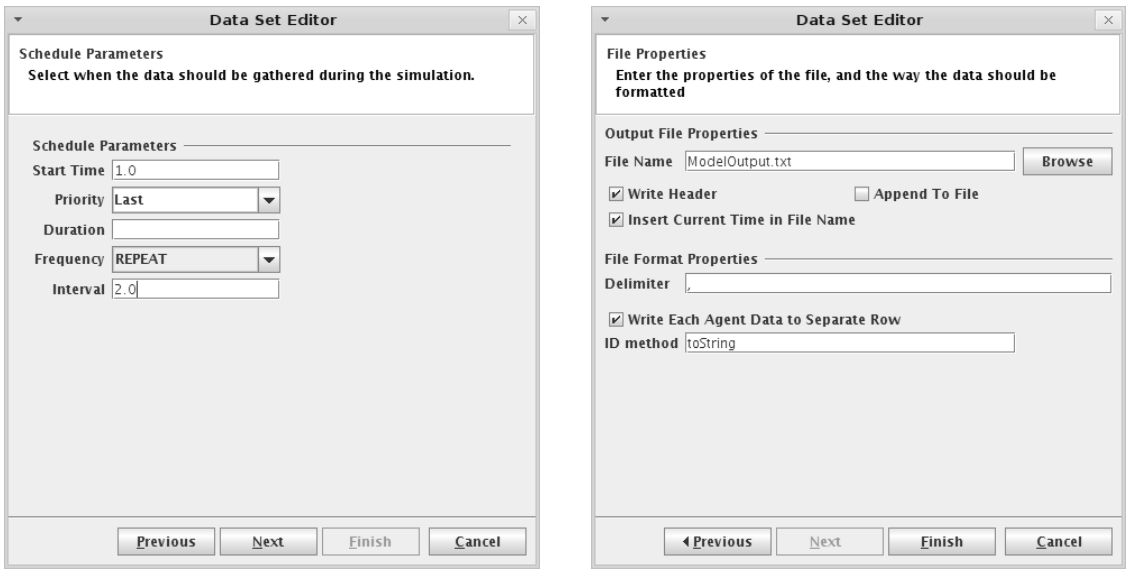
In order to analyze model results and to compare several parameter settings it is crucial to store data at the file system or within a database. Generated files may then be imported into third

party analyzing applications, or used to create diagrams of sufficiently high resolutions for documentation. While Repast Simphony integrates a rich framework as introduced in section 4.2.4, Repast J provides only sparse support for scheduling data output. The **DataRecorder** and its relatives are the only utility it offers, and the complete configuration is done programmatically.

ReSoNetA adds comfortable data storing features that aid the users to collect data sets and schedules their output to a given file. The Repast Simphony code could not be used as is because of different scheduling frameworks (see section 4.3.1). Furthermore, the writing of data sets is quite complicated as demonstrated in chapter 3. Anyway, the Repast Simphony data set and file outputter wizards are appropriate means to collect user data for storing and could be reused. As mentioned before data sets are manageable by the measure manager which lists configured data sets, and offers possibilities to add and remove them. At the beginning, the data set wizard explores all contexts that currently exist. This is necessary since a certain data set is associated with a certain context and stores information about all agents of a given agent class in that context. At the first wizard step it asks for a name and ID to identify the data set and to choose the agent class that shall be inspected. The next step provides opportunities to add data mappings to the set. A simple mapping represents parameter-less agent methods that return any object. To specify a network measure mapping the **MeasureChooser** is used to select one from the measures that are currently computed. Once the data set wizard is completed the measure manager may be applied to remove and add measures from and to the data set. This is only valid when the selected agent class implements **NetworkMeasureSupport**.



(a) Step 1 (b) Step 2



(c) Step 3 (d) Step 4

***Figure 4.10**: Screenshots of the four steps of the Data Set Wizard*

The scheduling parameters are queried during the third step, and the last one asks for filename and column delimiter character. It is also possible to append the data to an existing file in case the given file exists, to decide whether the header of column names shall be printed, and to add describing data lines which might help to identify the simulation run later on. Additionally, the user may specify the way multiple agents are written to the data set. When the box "Write Each Agent Data to Separate Row" is checked, the agents are not appended in a single row for each tick, but each time step comprises as much rows as agents. Only when the box is checked it is possible to enter the name of an agent's class method which is used to fetch the value for the agent ID column. In case the text field stays empty an increasing number is used as ID. Figure 4.10 shows screenshots of the dialog steps.

It is also possible to add any desired string to the top of the output file. For example, the string may contain parameter values that were used to generate the stored data, maybe organized in an XML structure. Thereto, the recorder invokes a specified method at a given object and writes the returning string to the file. The providing object and the method name need to be known to the output descriptor. Therefore, the users may call **setDescriptionProvider(Object)** and **setDescriptionMethod(String).** The recorder recognizes whether a providing object with a suitable method is available or not. Up to now this feature requires the programmatic configuration of data sets.

4.5.3. Further Improvements

While developing ReSoNetA it became clear that even Repast Simphony has some minor short comings when viewed from the perspective of a social network modeler. ReSoNetA takes the opportunity to overcome some of these short comings by adapting Repast Simphony's code. These improvements include adjustable layout properties, providing layout information, layout reinitialization, and more efficient edge creation.

Adjustable Layout Settings: In Repast Simphony it is not possible to adjust any settings that influence the network layouts. Two crucial examples of hard coded properties are the visualization size and the accuracy of the layout process. The drawing dimension of the `*Layout2D` classes is frozen to 800 * 400 pixels. That might be annoying if one requires, for example, a squared visualization for publication. One might consider clinching the snapshot, but this also distorts the node representations, which then could look quite unappealing. The accuracy is worth being adjustable since it might help with finding the balance between efficiently fast and convenient, clearly arranged visualizations. For some layout it specifies the number of iterations, for others an epsilon value is required that is achieved by calculating the reciprocal of the given integer.

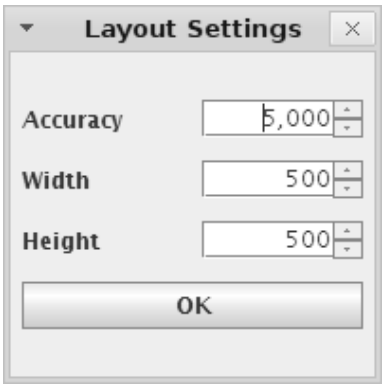


Figure 4.11: Screenshot of the layout settings dialog

ReSoNetA introduces the **LayoutSettingsSupport** interface for layout classes in order to indicate that these implement methods to adjust the layout settings. On the one hand the **DisplayProducerAdapted** fetches attributes from the display descriptor and passes these to the layout, and on the other one the user may invoke the layout settings dialog via the display window. It is shown in figure 4.11 and enables the input of values that are applied during the next update of the display.

Layout Information: Section 2.3.2 identifies the documentation of aspects that influenced the layout process as important since the viewer usually tries to interpret the given visualization for instance by associating central nodes in the display with central nodes in the network. The **Display2DSettingsPanel** offers a button to display a description of the layout that is responsible for the current visualization. However this is only possible when the layout class provides a description which is indicated by the **DescribingLayout** interface.

Layout Reinitialization: Section 2.3.2 also mentioned the reinitialization of miscarried network layouts as a desired visualization feature. Since many layouts are influenced by randomized operations a complete renewal of the layout may help to improve readability. The new library has a button at the display's settings panel to reinitialize the layout, update and render the visualization in order to display the changes.

Efficient Edge Creation: As stated in section 1.4.3 creation of network edges is quite complicated and fault-prone. ReSoNetA provides some methods in **NetworkUtilities** that comprise the many steps for the generation of directed and undirected edges and even for connecting complete groups given an array of agents.

4.6. Ways to Analyze a Social Network

First this section discusses the possibilities users have to analyze a social network. The role of different ways to incorporate measures is demonstrated. Afterwards the most favored network measures are presented in detail, catalogued into the main categories centrality, prestige and authority.

4.6.1. Exploring Social Networks

Investigating social networks often means to identify special or analogical positions among the actors, finding cohesive subgroups, mapping similar parts of a network or to characterize the whole network. As discussed in chapter 2 this is also true for social network analysis in modeling. This chapter is dedicated to providing the reader with concrete means to explore the modeled networks in order to comprehend simulation results on the one hand and to map back network properties to the model on the other. The focus lays on identification of special and analogical nodes within the network, which forms the basis of social network analysis.

As mentioned before finding important positions within the network is a crucial task in order to decide which actor is significant. Therefore one needs measures that indicate a node's importance or prominence, which are synonymously used expressions in SNA. Importance has at least two dimensions which are centrality and prestige. Centrality means the extent to which an actor is embedded at the heart of a network using edges of any kind in case of undirected relations and the extent to which an actor has access to all other nodes via out-going edges in case of directed networks respectively. Calculating prestige requires directed relations and focuses on the in-going connections which indicate how intensely other actors choose the one in question.

It is important to note again that any calculations of measures always depend on the content or meaning of relations that are modeled. A valid meaning in case of centrality would be friendship. Linus might be central in an undirected network in the sense that a relation between Linus and Mara shows that Mara is a friend of Linus and vice verse. In a directed network Linus is central when he has an out-going relation to Mara that shows that Linus may rely on Mara. In case of prestige asking for advice would be a plausible denotation for the edges. Sarah is prestigious when she has an in-going edge from George that indicates that George would ask her for help when he is in trouble.

Given a raw value of any network measure like 7 for the degree makes it hard to interpret and evaluate it. Often it does not become meaningful unless it is compared to results of others. For instance, given that the average degree of a network is 2, a value of 7 is quite high.

There is a widespread method of comparing values of a certain node to others which is normalization. It simply means to divide the values by the largest one. As a result all value range from 0 to 1 and a value close to 1 indicates that it is close to the maximum.

For network measures it is also important to evaluate a given measure value as it is calculated for a specific network independent from concrete other networks. This is possible by standardization of values, usually via dividing the data by the maximally or minimally possible value for the given network structure. For instance, the maximum degree is the number of vertices - 1.

ReSoNetA usually provides three kinds of each measure: The "natural" value, normalized, and standardized data. When interpreting the data it is important to consider which variant of data one investigates.

The remainder of this section will review a couple of network measures to facilitate finding the appropriate one. All metrics that are listed through this section are also supported by the ReSoNetA library.

This section subdivides the measures according to their meaning which is centrality, prestige or authority. Table 4.1 shows available measures and a short description of their meaning.

***Table 4.1**: Synopsis of all reviewed measures*

Key	*Description*	*Return Type*
BwCn	Betweeness based centrality (normalized)	class java.lang.Double
BwCstd	Betweeness based centrality (standardized)	class java.lang.Double
BwCnn	Betweeness based centrality (not normalized)	class java.lang.Double
BaC	Bary centrality (normalized)	class java.lang.Double
BaCnn	Bary centrality (not normalized)	class java.lang.Double
BaCstd	Bary centrality (standardized)	class java.lang.Double
CLC	Closeness based centrality (normalized))	class java.lang.Double
CLCnn	Closeness based centrality (not normalized)	class java.lang.Double
CLCstd	Closeness based centrality (standardized)	class java.lang.Double
EccCn	Eccentricity centrality (normalized)	class java.lang.Double
EccCnn	Eccentricity centrality (not normalized)	class java.lang.Double
EccCstd	Eccentricity centrality (standardized)	class java.lang.Double
HITSAnn	HITS Authority (not normalized)	class java.lang.Double
HITSHnn	HITS Hubs (not normalized)	class java.lang.Double
IDCn	Indegree based centrality (normalized)	class java.lang.Double
IDCnn	Indegree based centrality (not normalized)	class java.lang.Double
IDCstd	Indegree based centrality (standardized)	class java.lang.Double
ODCn	Outdegree based centrality (normalized)	class java.lang.Double
ODCnn	Outdegree based centrality (not normalized)	class java.lang.Double
ODCstd	Outdegree based centrality (standardized)	class java.lang.Double
PInn	Outdegree-Prestige, not normalized	class java.lang.Double
PPnn	Proximity-Prestige, not normalized	class java.lang.Double
PRnn	Page Rank (not normalized)	class java.lang.Double

4.6.2. Centrality

Centrality represents one of two meanings: Either how close a node is to all the others or the extent that node is a transmitter, i.e. how often messages or anything else needs to pass this node on a shortest way between two other nodes. In case the relations mean exchange of information, goods or anything else centrality might be substituted by activity. Inhomogeneity of a centrality index is used to define the centralization of a graph with respect to that index (Freeman, 1979).

Degree Centrality

The degree centrality is the simplest centrality measure since it just counts the number of ties with which an actor is connected to others (d_i). For directed networks the outdegree is used to calculate degree centrality since centrality focuses on the choices the actor makes by him.

Bary Center

A node's Bary center measures the sum of the shortest paths to every other node in a connected network. Therefore, this measure is undefined for disconnected networks. A low value indicates that the node does not depend on many middlemen on the way to the other nodes. Thus, adulterations and delays are less probable. If there are more than one geodesic, only one is considered. Thus this measure does not account for stability in case of several shortest paths between two nodes. Attention should be paid since a high Bary center does not mean comparatively high centrality for a given node, but the opposite.

The JUNG-implementation uses the Dijkstra-algorithm to compute the shortest paths, which runs in $O(m\log(n))$[16]. It is efficient by using a **MapBinaryHeap** and caching of previously calculated distances. The Bary center implementation ignores nodes that are not connected to the vertex under consideration. This gives relatively seen the same results as using the maximal distance or any other large constant for unconnected pairs of vertices.

Closeness Centrality: Closeness Centrality is defined exactly as the inverse of Bary Center. Thus a higher value means shorter ways to the others which indicates a higher centrality.

Betweenness Centrality: Betweenness centrality measures a node's potential in being a middleman on a path between two other nodes. For that purpose for each pair among the remaining nodes all shortest paths the particular node lies on are counted. Each number is then related to the overall number of shortest paths between the particular pair including the ones that do not pass the node the measure is calculated for. These shares are then added up for all pairs. For standardization the result is divided by the highest possible number of shortest paths a node may lie on which is $(n-1)(n-2)/2 = n^2-3n+2/2$.

The JUNG library uses a fast algorithm proposed by Brandes (2001). Instead of first calculating the length and number of shortest paths between all nodes and adding up all pair-dependencies[17] it takes advantage of combining these steps. During computing the shortest paths it keeps track whenever a path from a source node s to another node t includes a node v as predecessor of a node w. It then increases the number of shortest paths from s to t that go through w by the number of shortest paths from s to t through v since v is a predecessor of w and all shortest paths through v count for w, too. Afterwards, a node's betweenness centrality, i.e. dependency from others, is accumulated by the relative shortest paths of those nodes, of which it is a predecessor. The accumulation starts recursively with the longest shortest path, repeatedly for every node. The overall complexity of the algorithm is $O(nm)$ for unweighted graphs using breath first search and $O(nm+n^2\log(n))$ using Dijkstra's algorithm (Dijkstra, 1959).

Eccentricity Centrality: The eccentricity of a node is the maximum among geodesics to all other vertices in the network ($\max j(d(i,j))$) (Wasserman and Faust, 1994). To get a centrality measure at what a higher value reflects higher centrality the reciprocals are taken.

The results of this network measure also depend on the handling of unconnected nodes.

[16] As usual m denotes the number of edges and n the number of vertices.

[17] A pair-dependency of a pair of nodes s and t on v is the fraction of shortest paths through v in all shortest paths between s and t.

EccentricityCentrality uses two parameters, **useDiameter** and **unconnectedRepresentative** to determine the value in case of an unconnected pair of nodes. When **useDiameter** is set true, it uses the graph's diameter and otherwise the value assigned to **unconnectedRepresentative**.

4.6.3. Prestige

Prestige measures focus on the actors as recipients (Wasserman and Faust, 1994). Synonyms are deference or popularity. In order to decide whether a node is a sender or recipient, directed ties are required for all prestige measures. Note that, if the meaning of relations is negative, for instance "dislikes", one should not use the notion prestige.

Indegree Prestige: Clearly the simplest prestige measure is indegree prestige. It counts the number of in-coming relations (in-degree) which are considered as choices of connected actors for the node under investigation. Thus, given that the actors may choose among several others to ask, request for help or tell news, a certain node is prestigious when it is selected.

Proximity Prestige: Compared to indegree prestige this measure takes into account every (unilaterally) connected actor and not only direct neighbors. Such nodes that are connected to (not necessarily from) a specific vertex n_j via some path belong to its *influence domain* I_j[18]. Proximity prestige takes the average distance from nodes of the influence domain to node n_j and standardizes it to the graph size by the proportion of the influence domain's size and the graph size (i.e. devision by $|I_j|/(g-1)$). Then it takes the reciprocal of this expression to achieve high values for nodes that are connected from many others by short ways, and low values for vertices that are connected from a few by relatively long ways.

4.6.4. Authority

Authority measures are a bit more complex than centrality and prestige measures. Their meaning is discussed for the particular measures during the following subsections.

Page Rank: The Page Rank ranking (Page et al., 1998) was developed to enhance results of search engines of the World Wide Web and became a major part of the famous Google search engine (Brin and Page, 1998). Keeping in mind the fact that apart from navigation and advertisement links, hyperlinks between pages encode a considerable amount of latent human judgment it simulates a random surfer along the links to rank the importance of a web page. The higher the probability the surfer reaches a certain webpage the higher its rank should be. This probability is determined by the amount of pages that link to that page (so called backlinks) and their importance in turn. Consequently, a page has a high rank if the sum of the ranks of its backlinks is high. In using the web's link structure a search engine is not solely dependant on the content of pages that may be manipulated in order to receive higher search rankings.

The Page Rank further considers the opportunity a surfer has to restart a session and jump to a

[18]Note that Wasserman and Faust (1994) use I_j for the number of nodes that belong to the influence domain which occurs to be confusing since capitals are usually used to denote sets of elements and not their extent.

completely random page[19]. Leaf nodes, i.e. nodes that have no outgoing links need a special treatment since they receive rank score but do not provide any, and thus mean a sink. The original design excluded all leaf nodes for the calculation[20].

The JUNG implementation of Page Rank provides the possibility to specify edge weights, which influence the probability to surf along that edge. To support this feature the modeler needs to implement a custom version of the **Measure** object. Leaf nodes are handled in the way that their rank score is added to all other nodes, proportionally to their previous page rank.

For social networks the Page Rank is quite interesting as it might simulate spreading of rumors or other messages: One carries a message to people that are known to be important information exchange points in order to get back another hot story. Of course, people also visit others apart from that consideration, which relates to the incident restarts at some page.

Page Rank with Priors: The Page Rank with Priors ranking algorithm (White and Smyth, 2003) is a slight generalization of the ordinary Page Rank. It enables the user to define a set of so called root nodes that are meant to be authoritative. The algorithm assigns a starting page rank of 1.0/|*rootset*| only to those nodes which belong to the root set. All other nodes receive 0 as starting page rank. The Page Rank with Priors measure is useful if one can identify actors that have authority independent from their links, just because they hold a reputable office, for instance.

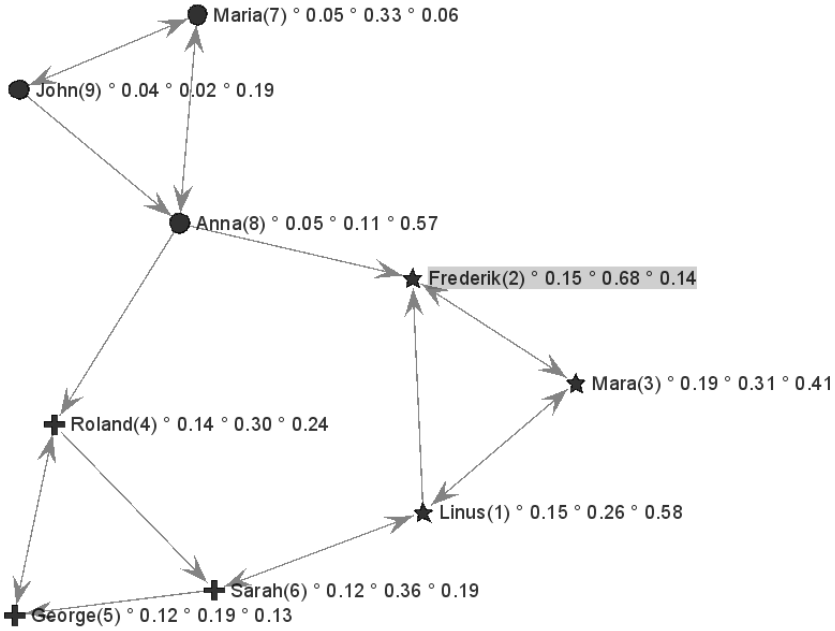


***Figure 4.12**: The screenshot shows Page Rank, authority values of HITS, and hub values of HITS for the directed running example. Watching Frederik one clearly spots that his high authority value is due to high hub values and low authority values of Ana, Linus and Mara. The Page Rank is especially low for John, who is located in a corner to which not many edges lead.*

[19]This is realized in that a page receives *probability that is independent from its incoming links. The fraction may be adjusted by the bias parameter.*

[20]Page et al. (1998) eliminated the dangling links that connect the lead nodes in a few stages: After removal of a dangling link the page the link originated at might become a leaf node itself.

HITS: The motivation for developing the HITS ranking algorithm (Kleinberg, 1999) was to identify interesting pages out of million ones of a search result on the World Wide Web. In that, it is quite similar to the Page Rank measure and even based on it. Regarding the hypertext links in most cases the welcomed strong authoritative pages need to be pointed at from hubs - pages that have links to multiple relevant authoritative pages. For example, at least for competitive reasons electronic online shops do not set links to each other. In contrast, there are some hub pages that compare price and service of a certain product and link to these shops that provide the product.

The HITS algorithm finally tries to identify the most important hubs and authority pages. For that, it maintains and updates numerical hub- and authority-weights for each page, following a simple principle: If a page is pointed to by many pages with large hub-values, then it should receive a large authority-value. Vice verse, if a page is pointed to by many pages with large authority-values it should receive a large hub-value. Algorithm 3 shows the algorithm as implemented by JUNG.

```
iterations := 0
for all vertices do
       authScore := 1.0
       hubScore : = 1.0
end for
while iterations < maxIerations AND precision < desiredPrecision do
       for all vertices v do
              prevAuthScore := authScore
              prevHubScore : = authScore
              for all successors s of v do
                     successorAuthSum + = v:authScore
              end for
              hubScore := successorAuthSum
       end for
       normalize hubScores
       for all vertices v do
              for all predecessors p of v do
                     predecessorHubSum += p:hubScore
              end for
              authScore := predecessorHubSum
       end for
       normalize authScores
       precision := root mean square of (hubScore - prevHubScore) of all v
       precision +=
end while
```

Algorithm 3: HITS ranking algorithm

Since the original HITS procedure is meant to deal with search results and contains some pre-processing for generating the set of pages to rank the algorithm described above may be identified as the core of that procedure.

4.7. Methods for Visualization of Dynamic Networks

Correct and meaningful visualizations are crucial to thoroughly understand and interpret simulated network dynamics. ReSoNetA provides a number of different network layouts that have both advantages and shortcomings for certain networks and purposes. The subsection “Visualization of Networks” presents the layouts, their computations, and preferential applications. Additionally, the library incorporates some further means regarding visualization:

fading of network elements and highlighting of nodes are introduced in subsection “Visualization features”.

4.7.1. Preservation of the Mental Map

Nodes that do not change their location basically but move only slightly were identified as an important requirement for dynamic network visualization in order to allow the viewer to percept changes that are not only due to the updating of the layout. For the force-directed layouts possible solutions were delineated: updating only the neighborhood of added or removes vertices or introducing additional forces between the same nodes of consecutive layouts. Whereas the JUNG library offers the potential to lock certain nodes from being moved by the layout process, Repast Simphony completely neglects this requirement. Because the layout is reinitialized at every update, the list of currently locked vertices gets lost.

Therefore, the challenge is to redesign the Repast Simphony layout wrapper classes, which are also used by ReSoNetA, to avoid reinitializations. Added nodes need to be incorporated into existing collections of nodes that store their location, forces, and distances between each other, while disappeared ones have to be identified and removed from the collections. Since this is quite troublesome, in ReSoNetA it is merely realized for the Kamada-Kawai layout so far.

To activate a rather static layout the nodes that shall not move need to be selected, and the “Freeze Selection” button should be pressed. This procedure makes it possible to explore a particular set of nodes, for example those that have an indegree above a certain threshold, while other vertices are neglected.

4.7.2. Visualization of Networks

Plenty of graph drawing algorithms have emerged over the past decades. Some of these were developed for quite special applications while others may be regarded as rather universal. This section reviews the layout procedures which are available in ReSoNetA and describes their characteristics in order to decide which one is the most appropriate for a certain intention. To provide an insight it also depicts pseudo code of the implementations that mostly fall back on the JUNG library.

In order to decide which layout is the best fitting, one needs criteria. Kamada and Kawai (1989) list often cited criteria:

- uniform distribution of edges and vertices on the display
- minimal number of edge crossings
- preservation of symmetric structures when these occur

For dynamic graph drawing as applied in modeling it is especially important to additionally consider the algorithm’s performance. Table 4.2 includes a synopsis of all available network visualization layouts comprising their advantages, shortcomings and customization.

***Table 4.2**: Layouts for network visualization*

Name	Advantages	Shortcomings	Input Data
Random Layout	- size/format definable	- really simple	-
Circle Layout	- fast - clear arrangement	- changing positions on update - lost space within the circle	radius
Ordered Circle Layout	- easy finding of certain nodes according to sort criteria - identification of connection patterns according to order - static positions	- lost space within the circle	radius, order of vertices
I-Self-Organizing Map Layout	- based on Meyer's self-organizing graph methods - easily adaptable to arbitrary types of visualization spaces (not implemented) - efficient	- nodes are likely to stick together	radius, initialAdaption, maxEpoch, coolingFactor
Fruchterman-Reingold Layout	-simple -fast	-blocking effects -isolated vertices drift away	attrac_multipl, repul_multipl, dimension, maxIter
Fruchterman-Reingold Layout grid-variant	- faster - holds isolated vertice/structures	- unnecessary edge crossings - symmetry of a large graph can be marred	attrac_multipl, repul_multipl, dimension, maxIter
Kamada-Kawai Layout	- seeks to achieve the geodesics as distances between pairs of nodes	- computational demanding	maxIter, length_factor, discon_multipl
Clan-based Graph Decomposition	- identifies subgraphs - supports contraction/expansion of subgraphs - cycles may be handled - dealing with unpleasant edges	- only for (tree-like) directed graphs	-

Circle Layout: Using the circle layout all nodes are located on a circle of a given radius, the angle that separates two nodes from each other defined by 360/*n*. Since this view is symmetric, nodes may be compared to each other easily, for instance regarding their degree. If there are any distinct patterns of relations this will be recognized without difficulty. Anyway, mostly it depends on the order of nodes along the ring whether a circle layout is helpful or not. If there is a predefined ordering of nodes, e.g. when nodes represent farmers along a channel adopting this sequence in the circle layout would ease pulling together network positions and locations on the channel. The JUNG implementation incorporates an **orderVertices(V[])** method, but the default version shuffles the vertices.

Ordered Circle Layout: This renewed version of the circle layout provides means to order the vertices along the circle. Therefore, ReSoNetA introduces the **SortableVertex** interface, which postulates a **compareWithVertex(SortableVertex)** for node objects. The ordered circle layout uses this method to sort the vertices on the circle starting with the first at the top. This

layout class may be also used to arrange vertices that do not implement the **SortableVertex** interface. In this case the layout uses the nodes' natural order and achieves a static layout that does not shuffle the vertices every time it is updated.

Fruchterman-Reingold Layout: This layout was designed to provide a speedy and simple algorithm to arrange nodes of an undirected graph with straight edges to be used in interactive graph drawing. The user should not need to tackle lots of options, change formulas or adjust the number of iterations. Vertices should be distributed evenly in the frame with uniform edge lengths. The guiding principle is that of force-directed placement. It interprets the edges as springs that force neighboring vertices to be attracted, and all other nodes to be repelled (Fruchterman and Reingold, 1991).

The algorithm calculates attractive and repulsive forces, sums them up and assures that every vertex is located within the frame. The extent of this synchronous replacement among all vertices is controlled by the decreasing "temperature" of the current time step. The optimal distance between nodes k is calculated as square root of the area every vertex might occupy ($k=\sqrt{(width*height/|V|)}$). The formulas for repelling and attraction are as follows, which are effective and efficient to compute (d is the current distance between two nodes):

$$f_a(d) := d^2/k$$

$$f_r(d) := -k^2/d$$

As shown in Figure 4.13 these functions assure a distance k. A disadvantage of the algorithm is the blocking effect that sometimes occurs: Repulsive forces between disconnected nodes prevent a node or a structure to move to the other side of a vertex, which would mean less crossing edges.

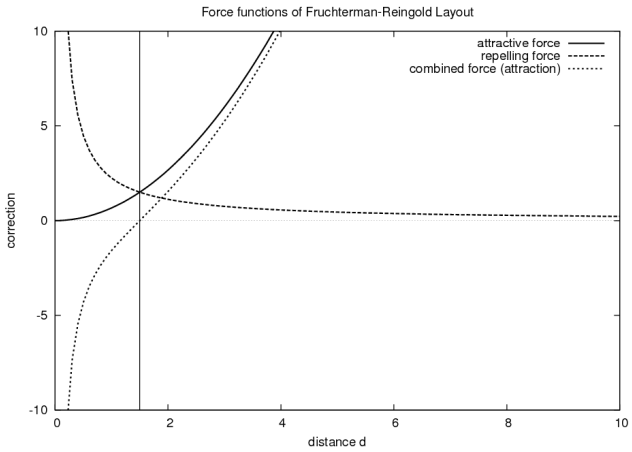


Figure 4.13: *Force functions of the Fruchterman-Reingold-Layout algorithm. The vertical line represents the aspired distance k. For lower distances, the repelling force is strong, for higher distances the attractive force is the strong one.*

The JUNG version of the Fruchterman-Reingold Layout as implemented in **FRLayout** does not support the grid-version of the algorithm: For calculating the repulsive forces, this variant considers only vertices within a radius of $2k$ around the vertex the force is calculated for. It is, therefore, faster and appears fleecier, but exhibits edge crossings more often.

Kamada Karwai Layout: The Kamada Karwai layout is also realized by a force-directed algorithm. In comparison to Fruchterman-Reingold, it does not distinguish between repulse and attraction and computes forces between all pairs of vertices based on the shortest path between them. It therefore aspires to map the geometric distance between vertices in the display to the theoretical distance of the graph represented by the geodesics (Kamada and Kawai, 1989). Furthermore every iteration this algorithm picks the most displaced vertex and replaces it instead of updating all vertices synchronously.

Algorithm 4: *Kamada Karwai layout: pseudo-code*

```
initialize xydata
compute preferred length of edge L
for all pairs of vertices (I,j) do
       Calculate distance d_i,j
end for
while iterations < maxIterations ∧ max Δ_pi > ε do
       for all pairs of vertices (i,j) do
              Calculate preferred distance l_ij = d_ij(min(height;width)/dm)lfactor
              Compute weight factor k_ij = 1=d^2_ij
              Compute energy E = ∑_{i=1}^{n-1} ∑_{j=i+1}^{n} k/
                                 2*(dx^2+dy^2+l_ij^2-2l_ij √((dx^2+dy^2)))
        end for
       Calculate max δ_pi and identify according vertex v_m
       while count < 100 ∨ δ_m < ε do
              Calculate dx;dy for v_m
              Update v_m's location
       end while
       Shift vertices so that the center of gravity is located at the center
end while
if exchangeVertices = true then
       for all pairs (v_i,v_j) do
              if energy > energyexchanged then
                     exchange v_i with v_j
              end if
       end for
end if
```

The original algorithm by Kamada and Karwai runs in $O(nTnU)$ where U is the number of inner loops required to achieve a local minimum of δ_m for the chosen vertex m and T the number of outer loops to achieve minimum values for all δ_i. The JUNG implementation is not that efficient, since it does not store and update δ_i values. Besides it contains a lot of redundant code[21]. Furthermore there is no criterion that stops the outer loop if δ_m falls short of a certain ε. Thus, the algorithm runs the hard coded number of steps every time.

For directed graphs there is another shortcoming: The diameters of such graphs may be less than those of the same graphs where all edges are bidirectional as in an undirected graph. For instance, the directed version of the running example has a diameter of 5 whereas the diameter for the undirected network is 7. Since the diameter influences the preferred distance between two nodes (line 8 in algorithm 4) the visualization's size gets unnecessarily smaller due to an unnecessarily higher diameter. In **KKLayout_Fixed** ReSoNetA therefore creates an undirected

[21]For instance the computation of the energy level ($O(n^2)$) is calculated twice each step.

graph from the given directed, calculates its diameter and adapts the *lfactor*[22] to compensate the larger diameter used by the JUNG implementation.

Clan-based Graph Decomposition Layout: By parsing the graph in subgraphs that are organized in a tree and using that tree to create the layout the authors seek to provide an aesthetically pleasing visual layout for arbitrary directed graphs (McCreary et al., 1998). In considering clans the algorithm is ready to reflect the two-dimensional affinity of these structures in the layout and reduces edge crossings. Specific layout attributes (node size, spacing) are assigned to the tree entries and used to calculate the node's location. Varying views of the graph can be achieved in this way. The Clan-based Graph Decomposition algorithm is roughly depicted in Algorithm 5.

Algorithm 5: *Clan-based Graph Decomposition layout: pseudo code*

```
Parse the graph into a tree of parallel, serial and primitive clans
for all nodes of the parse tree from leaves to root do
       Calculate the bounding box depending on descendants
end for
Add dummy nodes to route long edges and place them in the parse tree
Recompute the bounding box dimensions for the augmented tree
for all node of the parse tree do
       Assign x- and y-coordinates to the bounding box
end for
Place the node labels in a circle
Connect the circles with straight lines when possible and splines at curves
```

Compared to the concept of hierarchically laid out graphs the Clan-based Graph Distribution offers some advantages as listed below. Most of these are enabled by parsing the graph into subgraphs:

- Nodes of a clan are placed close to each other in the drawing.
- By identifying a certain node the smallest non-trivial clan that contains that node may be contracted to a single node and expanded again.
- Unnecessarily long edges are shortened by placing the start node to a layer below when possible.
- Dummy nodes may be used to reroute edges that otherwise disturb the layout by covering nodes or other edges.

Cyclic graphs might be handled by reversing the orientation of the edge that identifies a cycle before parsing. After the layout is ready this edge is reversed again to achieve the original orientation.

However, this layout is only suitable for directed graphs. Nevertheless, in Repast S it is possible to create a CGD-Display based on **CGDLayout** for any projection. If the projection is an undirected network you will be confronted with the error message **Error while creating displays java.lang.IllegalArgumentException: Network is not a tree type network** while that display is created. ReSoNetA adapts the layout class and displays a more reasonable hint when a user tries to use the CGD layout for an undirected network.

[22]Since *dm* is not accessible in **KKLayout** *lfactor* is the only possibility to influence the diameter.

4.7.3. Fading Network Elements

As identified in sub-section 2.3.2, marking changing elements by special styles and even fade the transition during several steps is a useful feature to understand network evolution. ReSoNetA supports fading of elements, as figure 4.14 shows. Duration and fading styles are user-defined separately for in and out transition i.e. appearance and removal as well as different for nodes and edges. The process is accomplished by substituting the node style (**StyledDisplayLayer2D**) and network style layers (**NetworkDisplayLAyer2D**)[23]. Because these are instantiated in **Display2DAdapted** this class is the anchor for fading.

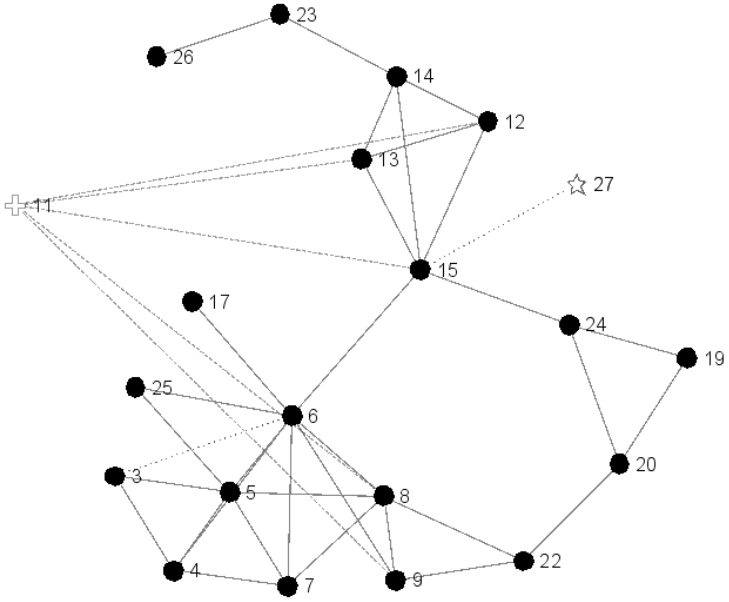


***Figure 4.14**: Screenshot of fading network elements; the snapshot shows an emerging, still transparent node at the top right and a leaving one on the left. The default fading styles assign a star to added nodes and crosses to disappearing network elements. The fading duration is preset to five ticks.*

The display also fetches information about the fading styles from **DisplayDescriptorAdapted** and sets it at the display layers. The user may specify the fading look by setting properties to **DefaultFadingStyle2D** or **DefaultFadingEdgeStyle2D** which inherit from the default style classes and provide additional getters and setters for in and out fading intervals as depicted in figure 4.15. Of course, the user may subclass the fading defaults or create his own styles that implement the respective interfaces for node and edge styles.

The display layers contain methods, which the display invokes every time it receives events for added or removed objects[24]. The layers then need to store particular objects in a map containing the object and an integer that represents the number of steps that element is already fading. Added objects start with 1 and increase, removed objects start with -1 and decrease. The objects

[23]Since the classes require enfolding adaption, sub-classing is not appropriate.

[24]To be precise, this is only true for the node display layer. The network display layer as an object layer is not invoked for added or removed edges. It is registered directly at the network for edge events.

in the map are faded during the render process: If they were removed their transparency increases, and the opposite is true for added items. If the fading duration reaches the user-specified fading interval value, removed objects are also removed completely and added item's style becomes the normal one. Both are finally removed from the map of fading elements. Fading can be suppressed by setting the fading interval to 1 for edges or nodes respectively.

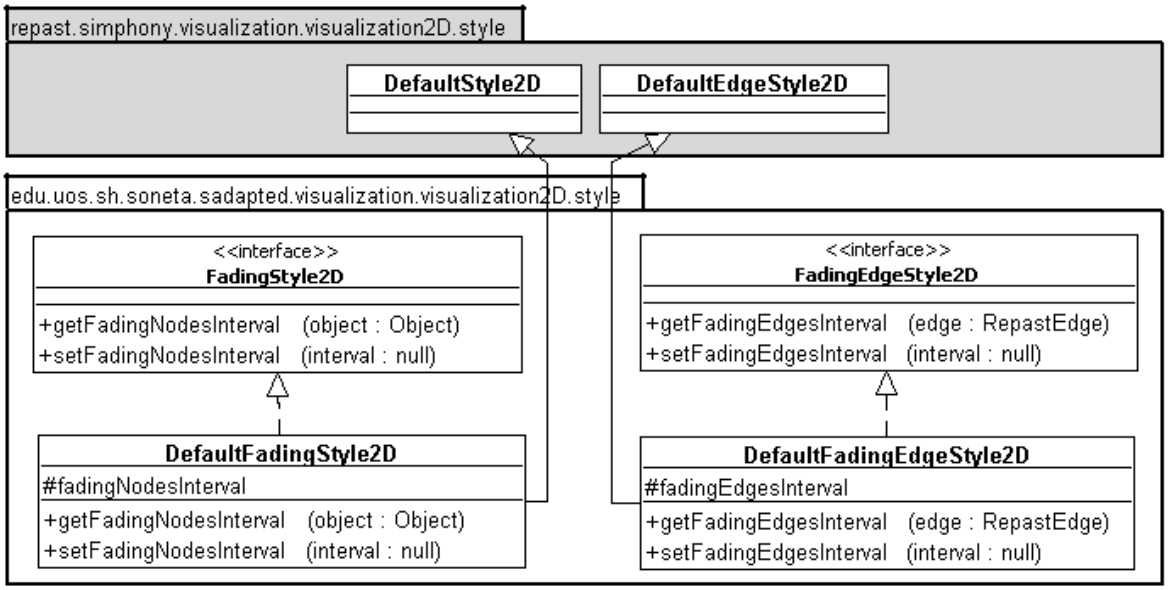


***Figure 4.15**: UML-class diagram of fading style classes*

However, there are some aspects one needs to be aware of. Removed vertices need to stay within the network since otherwise they are unconsidered during the layout process and the network may look unappealing. Thus they have to be added again until the fading process is over. Since the **JungNetwork**s would release an add event which would trigger the fading *in* behavior the removed node is added to the underlying **Graph** directly.

When a node is added the display layer needs to check whether the same node was previously removed. If this is the case, the particular object should be removed from the removal process. Similarly, the same applies to removed nodes which were previously added. If an edge is removed that was added before during the same time tick, the edge's visual item may not have been created. But, in order to fade the edge the visual item is required and needs to be produced. This in turn demands the visual counterparts of the edge's end points, so this too needs to be checked.

4.7.4. Highlighting of Nodes

Highlighting of nodes works in combination with the node table. Rows that represent vertices might be selected and the corresponding nodes are indicated in the network visualization by a colored transparent background. This way it is possible to find nodes with certain values regarding specific network measures in the network. Their interrelations with others may then be investigated.

5. Conclusion

This last chapter seeks to summarize the attainments of and experiences with ReSoNetA. The first part recapitulates insights from the literature studies on modeling social networks and their visualization. The resulting software library is furthermore reviewed, and consequences for modelers are discussed. The second section gives an outlook. A lot remains to be done, both in extending the library and regarding conceptual questions in modeling social networks.

5.1. Attainments

The ReSoNetA software is developed as a library that makes it more comfortable to deal with social networks in multi-agent-based modeling. It is intended for models that were originally written within the Repast J framework, but which seek to incorporate sophisticated network analysis and visualization.

In order to design and implement the library, demands on agent-based models that deal with social networks were identified. Literature of Social Network Analysis and graph visualizations was reviewed. Network measures as means to analyze and evaluate networks, and meaningful visualizations for dynamic graphs are found to be crucial for such software. For sound analysis, these means need to be accessible from the modeling framework directly. Repast was identified as being the most appropriate software for social science simulations, with Repast J being often used and Repast Simphony being a modern framework that comprises a lot of new functionality. ReSoNetA finally tries to make Repast Simphony features accessible and adds further means to handle computation and visualization of network measures, and to ease comprehension of dynamic networks. Docking the library to existing user models by a few determined lines of code is straightforward. Configuration of displays, data sets and network measure calculation is done by setting parameters at so called descriptors (displays and data sets), or calling methods at dedicated classes like the **NetworkMeasureUtilities**. These steps are all well documented in the user's manual[25].

Wizards and dialogs were used to configure data sets and network measure handling for two reasons. At first it is usually more efficient to enter parameters via graphical user interfaces instead of editing the Java code. While the first solution may be done during runtime the second one requires the code to be compiled again and the simulation must be restarted. This is time-consuming, especially for experimenting with lots of network measures. In case of the layout settings dialog, it enables the user to adjust the visualization quality, for instance to improve the network drawing for a particular time step. Furthermore, graphical user interaction like the measure manager also enables non-programmers to use the software and adjust certain settings. Runtime configuration also makes it possible to offer items to choose from, and to check inputs for validity.

With the node table ReSoNetA integrates an important feature for analysis of social networks during simulation. Several measures may be listed in the table for each agent, including the

[25]The user's manual comes with the software and is available at http://www.cesr.de > Downloads > Software > ReSoNetA

possibility to rank the nodes according to a certain measure. The sliders enable the user to focus on a particular set of nodes, and with their linkage to the displayed network allow layouts that are less crowded. Selecting nodes in the table that are consequently marked in the networks provide the necessary combination of both GUI elements.

During development, emphasis was also placed on integrating new components into Repast Simphony later on. Examples are the network measure supplier framework including the measure manager, the node table, the fading feature for network elements, and some of the improvements on layouts. Since these extensions are already mainly based on the recent Simphony framework, it should be easy to make them accessible in that new version. Exchange with the Repast developers is ongoing.

Adaptations to existing software might cause extra work, even if the software is object oriented which actually simplifies extension. However, it is important that the developers take extensions of their software into consideration while designing the software. For instance, private fields make it often hard to subclass certain classes which is necessary if required interfaces are missing. Where it is possible, ReSoNetA uses existing code of Repast Simphony and adapts it to recent needs instead of re-implementing classes. First results show that the library's performance is suitable for networks up to at least 100 nodes and numerous edges in between. Compared to Repast Simphony, which provides comparable visual features but also a lot of additional support, the starting time is quite short. However, the most influencing process is laying out of networks, which is easily adaptable by the layout settings dialog.

5.2. Outlook

Suggestions for future extensions are listed below. Besides, there are some conceptual questions that were not discussed in an appropriate depth. Citations for further considerations are given.

ReSoNetA introduces a new kind of display window that provides unfamiliar possibilities of interaction. Furthermore, it requires additional configuration procedures for data sets and measures, and alteration of user code. Therefore, it would be useful to conduct some user evaluations that uncover shortcomings and lead to improved and more efficient user interfaces. Even if the configuration of displays via Java code is straightforward, this might represent an obstacle for non-programmers. Adapting the Repast Simphony wizards for display adjustments, as already done with respect to the data set configuration, therefore seems to be a worthy effort in future. Since the configuration process is already based on Simphony descriptors, realizing the wizards that interact with it should not be too complicated. Results from user studies regarding the parameter input via wizards should influence the design.

Up to now it is questionable whether the framework's performance is suitable to run large agent populations with several hundreds of nodes and dynamic edge creation and removal. Detailed benchmarks that also compare ReSoNetA to Repast Simphony runs would be quite interesting.

The library could be extended in several directions regarding the area of network analysis. At first, many measures that are currently solely calculated at the actor level could be computed on the network level. This is particularly interesting when comparing different simulation runs or to do sensitivity analysis. Another wide field is clustering and identification of subgroups in a network.

Section 2.1.1 indicates that in some cases it might be important to let each agent have its own social model of its relationships to others and ties among these. Incorporating the individual social models into the library is time and space consuming and a challenge to design efficiently. Since such a model is currently not known, it seems to be plausible to first implement a prototype to identify promising architectures.

Furthermore there are many extensions worth thinking about in regards to the visualization capabilities. Modeling visualizations are demanded that especially care for clear and compact time series analysis. Dwyer, Hong, Koschutzki, Schreiber and Xu (2006) provide a software that implements several visualizations like 3D parallel coordinates, hierarchy based and orbit-based comparison indented to collate different centrality measures for a given network (Ahmed et al., 2006). The requirement of only slightly changing network visualizations has been touched by ReSoNetA, but needs further consideration because of rather complicated adaptations of the layout process. An interface from the network measures to the charting capabilities would enable the user to view the progression of measures during simulation runs. An implementation could be similar to that of displaying measures at the nodes, which incorporated the **MeasureChooser**.

Smaller features that would help comprehension of and orientation in network displays are aggregated nodes and extended vertex or edge filters. Aggregated nodes require the identification of subgroups and substitute these by a single vertex that collects all in-coming and out-going relations to and from nodes within the groups to nodes outside the group. At first, criteria for sub-grouping need to be selected in order to aggregate or expand the elements by mouse gestures.

The current agent based modeling frameworks are intended to support a rather wide range of modeling, either totally general like the reviewed multi-agent modeling framework Mason or focused to the whole field of social science modeling like Repast Simphony. While this supports the spread of such software and enlarges the community of users and developers it also means to develop software which is suitable for many purposes and may not specialize to a certain application like modeling of social networks.

The approach of a modular assembly system as Mason already seeks to incorporate it, seems to be promising. Experts of certain domains may extend the core framework with modules for specific tasks. The users may then choose the modules that are most appropriate for their needs and ignore others. Therefore, the framework needs to provide clear interfaces that such modules may be expanded upon, like interfaces for projections, for visualizations, for data import and export, and for user input. For example, if a modeler does not want to deal with GIS information the framework should never mention any GIS functionality at any place not to confuse users that are unfamiliar with GIS. Of course, it is hard for an agent based modeling framework to separate all these components. For example, when a social network is displayed among agents that are located on a GIS map dependences exist which are difficult to bypass. However, some effort is required to explore the pros and cons.

While social networks and their analysis play a strong role in social sciences they yet seem to be under-represented in modeling, especially in agent based modeling. While social scientists realized the meaning of relations among actors, not many modelers have yet taken on the effort of incorporating social networks into their models. However, studies seem to be missing that

seek to estimate the meaning of representing social networks in modeling. Particularly the variant of the agents' social model on which they base interactions with others has not often been considered.

ReSoNetA seeks to facilitate dealing with social networks in modeling. It supports developers and users in analyzing and visualizing networks and thus makes it easier to incorporate the agent's social surroundings, in particular for Repast J models that needed much effort to achieve this before. Additionally, ReSoNetA offers means and has potentialities to be extended and customized further on.

6. Bibliography

Ahmed, A., Dwyer, T., Forster, M., Fu, X., Ho, J., Hong, S.H., Koschätzki, D., Murray, C., Nikolov, N.S., Taib, R., Tarassov, A. and Xu, K. (2006). Geomi: Geometry for maximum insight. In: P. Healy and N. Nikolov (Eds.), Graph Drawing 2005, LNCS 3843, 468–479. Springer Berlin / Heidelberg.

Baur, M. and Schank, T. (2008). Dynamic graph drawing in visone. Tech. rep., University of Konstanz: Department of Computer and Information Science.

Bederson, B.B., Grosjean, J. and Meyer, J. (2004). Toolkit design for interactive structured graphics. IEEE Transactions on Softare Engineering, 30(8), 1–12.

Brandes, U. (2001). A faster algorithm for betweenness centrality. Journal Of Mathematical Sociology, 25(2), 163–177.

Brandes, U. and Wagner, D. (2004). Analysis and visualization of social networks. Tech. rep., University of Passau, Department of Mathematics & Computer Science.

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the seventh international conference on World Wide Web 7, 107–117. Brisbane, Australia: Elsevier Science Publishers B. V.

Burt, R.S. (1982). Towards a Structural Theory of Action. New York: Academic Press.

Butts, C.T. (2008). Social network analysis: A methodological introduction. Asian Journal Of Social Psychology, 11(1), 13–41.

Codehouse Foundation (2008). Groovy - An agile dynamic language for the Java platform. http://groovy.codehaus.org. Last visited on 08/19/2008.

Collier, N. (2002). Repast: An extensible framework for agent simulation. Tech. rep., University of Chicago: Social Science Research Computing.

Cribari-Neto, F. and Zarkos, S.G. (1999). R: Yet another econometric programming environment. Journal of Applied Econometrics, 14, 319–329.

Dijkstra, E.W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269–271.

Dwyer, T., Hong, S., Koschutzki, D., Schreiber, F. and Xu, K. (2006). Visual analysis of network centralities. In: K. Misue, K. Sugiyama and J. Tanaka (Eds.), Conferences in Research and Practice in Information Technology, vol. 60, 189–197. Australian Computer Society.

Edmonds, B. (1998). Modeling socially intelligent agents. Applied Artificial Intelligence, 12(7-8), 677–699.

Freeman, L.C. (2005). Graphic techniques for exploring social network data. In: P.J. Carrington, J. Scott and S. Wasserman (Eds.), Models and Methods in Social Network Analysis, 248–269. Cambridge University Press.

Fruchterman, T.M.J. and Reingold, E.M. (1991). Graph drawing by force-directed placement. Software-Practice & Experience, 21(11), 1129–1164.

Galaskiewics, J. and Wasserman, S. (1993). Social network analysis: Concepts, methodology, and directions for the 1990s. Sociological Methods Research, 22(1), 3–22.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley.

George Mason University (2008). MASON. http://cs.gmu.edu/ eclab/projects/mason/. Last visited on 08/19/2008.

Gilbert, N. and Troitzsch, K.G. (1999). Simulation for the Social Scientist. Open University Press.

Granovetter, M. (1985). Economic-action and social-structure - the problem of embeddedness. American Journal of Sociology, 91(3), 481–510.

Henry, K. (2006). A crash overview of groovy. Crossroads, 12(3), 5–5.

Howe, T., Collier, N., North, M., Parker, M. and Vos, J. (2006). Containing agents: Contexts, projections, and agents. In: Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects. Argonne, Illinois, USA: Argonne National Laboratory.

Huisman, M. and van Duijn, M.A. (2005). Software for social network analysis. In: P.J. Carrington, J. Scott and S. Wasserman (Eds.), Models and Methods in Social Network Analysis, 270–316. Cambridge University Press.

Jansen, D. (2006). Einführung in die Netzwerkanalyse. Grundlagen, Methoden, Forschungsbeispiele. Wiesbaden: Vs Verlag.

Kamada, T. and Kawai, S. (1989). An algorithm for drawing general undirected graphs. Information Processing Letters, 31(1), 7–15.

Kleinberg, J.M. (1999). Authoritative sources in a hyperlinked environment. Journal Of The ACM, 46(5), 604–632.

Krebs, F., Elbers, M. and Ernst, A. (2007). Modelling social and economic influences on the decision making of farmers in the odra region. In: Proceedings of the 4th European Social Simulation Association Conference. Toulouse, France: University of Social Sciences.

Lem, M. (2006). Epidemiologic considerations in network modeling of theoretical disease events. In: Visualising Network Information, 11.1 – 11.6. Neuilly-sur-Seine, France: RTO.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K. and Balan, G. (2005). Mason: A multiagent simulation environment. Simulation, 81(7), 517–527.

Matthews, R.B., Gilbert, N.G., Roach, A., Polhill, J.G. and Gotts, N.M. (2007). Agent-based land-use models: A review of applications. Landscape Ecology, 22(10), 1447–1459.

McCreary, C.L., Chapman, R.O. and Shieh, F.S. (1998). Using graph parsing for automatic graph drawing. IEEE Transactions On Systems Man And Cybernetics Part A-Systems And Humans, 28(5), 545–561.

Moss, S. and Edmonds, B. (2005). Sociology and simulation: Statistical and qualitative cross-validation. American Journal of Sociology, 110(4), 1095–1131.

North, M.J., Collier, N.T., Tatara, E. and Ozik, J. (2007). Visual agent-based model development with repast simphony. Tech. rep., Argonne National Laboratory.

North, M., Howe, T., Collier, N. and Vos, J. (2005). The Repast Simphony runtime system. In: Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms. Argonne, Illinois, USA: Argonne National Laboratory.

OMadadhain, J.O., Fisher, D., Smyth, P., White, S. and Boey, Y.B. (2007). Analysis and Visualization of Network Data using JUNG. Manuscript submitted for publication.

OMadadhain, J.O., Fisher, D., Smyth, P., White, S. and Boey, Y.B. (2008). JUNG - Java universal network/graph. http://jung.sourceforge.net/. Last visited on 09/12/08.

Page, L., Brin, S., Motwani, R. and Winograd, T. (1998). The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, Stanford University.

Pahl-Wostl, C. (2005). Actor based analysis and modeling approaches. The Integrated Assessment Journal, 5, 97–118.

Perer, A. and Shneiderman, B. (2006). Balancing systematic and flexible exploration of social networks. IEEE Transactions On Visualization And Computer Graphics, 12(5), 693–700.

Railsback, S.F., Lytinen, S.L. and Jackson, S.K. (2006). Agent-based simulation platforms: Review and development recommendations. Simulation-Transactions of The Society for Modeling and Simulation International, 82(9), 609–623.

Tatara, E. (2007). Repast Simphony reference. Tech. rep., Argonne National Laboratory.

Tatara, E., NORTH, M., HOWE, T., COLLIER, N. and Vos, J. (2006). An indroduction to repast simphony modeling using a simple predator-prey example. In: Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects.

Tobias, R. and Hofmann, C. (2004). Evaluation of free java-libraries for social-scientific agent based simulation. Journal of Artificial Societies and Social Simulation, 7(1), 1–23.

University of Maryland (2008). Piccolo. http://www.cs.umd.edu/hcil/piccolo. Last visited on 08/24/2008.

Wasserman, S. and Faust, K. (1994). Social Network Analysis. Methods and Applications (Structural Analysis in the Social Sciences). Cambridge University Press.

White, S. and Smyth, P. (2003). Algorithms for estimating relative importance in networks. In: L. Getoor, T.E. Senator, P. Domingos and C. Faloutsos (Eds.), KDD, 266–275. Washington D.C., USA: ACM.

Wilensky, U. (2007). NetLogo 4.0.2 User Manual. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, Illinois, USA.

University of Kassel · Center for Environmental Systems Research
Kurt-Wolters-Straße 3 · 34125 Kassel · Germany
Phone +49.561.804.3266 · Fax +49.561.804.3176
cesr@usf.uni-kassel.de · http://www.usf.uni-kassel.de