Construct User Guide

Stephen Dipple, Michael Kowalchuck, Neal Altman, Kathleen M. Carley
sdipple@andrew.cmu.edu, mkowalch@andrew.cmu.edu, kathleen.carley@cs.cmu.edu

May 2021
CMU-ISR-21-102

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Center for the Computational Analysis of Social and Organization Systems
CASOS technical report

This report/document supersedes the following CMU-ISR Technical Reports:
CMU-ISR-14-105R, "Construct User Guide", December 2014

This work was supported in part by the Knight Foundation and Office of Naval Research under a Minerva Grant (Dynamic Statistical Network Informatics, N000141512797) and Multidisciplinary University Research Initiatives (MURI) Program (N000141712675). Additional support for Construct was provided by the Center for Computational Analysis of Social and Organizational Systems (CASOS) and the Center for Informed Democracy and Social Cybersecurity (IDeaS) at Carnegie Mellon University. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Knight Foundation, the Office of Naval Research, or the U.S. Government.

Keywords: Construct, multi-agent simulation, dynamic network analysis, agent-based modeling, information diffusion, belief diffusion, agent-based simulation, modeling, and simulation.

Abstract

This technical report provides users and researchers information on the configuration and use of the newest version of Construct, the CASOS dynamic network, agent-based, information and belief diffusion simulation of complex socio-technical systems. The report provides a quick start guide to Construct, a detailed discussion of its configuration, and use through a sample problem and virtual experiment configuration exemplar, and a set of appendices with additional useful information. This document is both an introduction to Construct for casual modelers as well as a reference guide for researchers, modelers, and simulationists.

Table of Contents

Table of Figures

Table of Tables

Construct User Guide

Introduction

Construct is a software framework enabling agent-based network-centric simulations. Construct's primary model, the Standard Interaction Model can be used to examine the co-evolution of agents and the socio-cultural environment (Carley, 1990, 1991). Construct enables easy examination of the evolution of networks and the processes by which information moves around a social network (Carley, 1995; Hirshman et al., 2007a, 2007b). Construct's models capture dynamic behaviors in groups, organizations, and populations with different cultural and technological configurations (Schreiber et al., 2004). Groups and organizations are complex systems and the variability of human, technological, and organizational factors among such systems are captured through heterogeneity in information processing capabilities, knowledge, and resources. Multiple non-linearities in the systems generate complex temporal behavior on the part of the agents.

Constructuralism is a mega-theory that states that the socio-cultural environment is continually being constructed and reconstructed through individual cycles of action, adaptation, and motivation. This theory is at the heart of Construct's design. Many social science theories and findings are part of the constructural theoretical approach including structuration theory (Giddens, 1986), social information processing theory (Salancik & Pfeffer, 1978), symbolic interactionism (Manis & Meltzer, 1978; Stryker, 1980), social influence theory (Friedkin, 1998), cognitive dissonance (Festinger, 1957), and social comparison (Festinger, 1954). In addition, several cognitive processes are embedded such as transactive memory (Wegner, 1987). Construct allows for these theories to coexist and operate while minimalizing potential conflicts.

Construct has several advantages as an agent-base model framework. First, the experiment designer has complete control over a wide range of inputs used for interaction over the course of a run and facilitates as much customization as theories allow. Second, Construct contains a suite of agent models, which enable diverse socio-technical conditions to be modeled. Third, general agent characteristics can be easily configured *a priori* using empirical data or they can be based on hypothetical data. To use Construct, the researcher specifies both the relevant agents (Hirshman et al., 2007b) and the relevant networks (Hirshman et al., 2007a). Additional information about the Construct interaction model can be found elsewhere (Carley 1991; Hirshman et al., 2007b).

Agent Based Models

One of the most used and intuitive approaches to Social Networking Services (SNS) is Agent Based Models (ABM). ABMs employ a bottom-up approach in which a set of heterogeneous

agents, their behavioral properties, the "rules" of interaction, the environment, and the interaction topology that the agent populates is explicitly modeled. Complex social behavior emerges from simple individual level processes. In ABMs, many computational entities, with varying levels of cognitive complexity, interact with one another in a manner similar to the real-world entities they represent. These agents are simplified versions of their real-life counterparts (e.g., ants, people, robots, or groups), only retaining elements salient to the phenomena being studied. Agents interact in a virtual world and can be constrained and enabled by the network position they occupy.

In most ABMs, the topology of the virtual world is a simple 2-D grid and agents form "networks" as they occupy the same or neighboring spaces or the agent's network is prescribed as the set other agents within so many spaces of ego. Networks generated from grid-based interactions or defined in terms of grid-nearness tend not to have the same properties as true social networks, i.e., the distribution of ties, the method of tie formation and dissolution, and the relation of ties to physical space are not realistic. Most ABM toolkits support this type of grid-based modeling of the social topology.

There is, however, a growing interest in and a growing number of ABMs where the agents exist and move in a socio-demographic or network topology rather than a grid topology. The Construct models are an example. In these models, the agents occupy a social network position defined in terms of which other agents the ego agent can interact with. In other words, rather than physical adjacency, social adjacency is used. This network topology may be static or dynamic. This latter type of model where agents exist in dynamic social networks rather than on grids is where most research on SNS is focusing. This approach, referred to as agent-based dynamic-network modeling, is the approach we found to be most valuable for modeling the adversary and it is embodied in Construct.

ABMs vary in how the environment is represented. This could be as simple as a single dimension or array where ego interacts with those other agents that are within so many squares left or right of ego. This is the case in Kaufman's NK model. Traditionally, however, the environment was a grid and the agents interacted with other agents in and/or could move to those squares that surrounded them. Most early studies explored the relative impact of von Neuman (squares left, right, up, down of ego) or Moore (eight squares around ego) or extended Moore neighborhoods (squares within some distance of ego). In these traditional approaches, the structure of the social network is directly tied to the physical position of the agents. Examples of such models are the Game of Life (Gardner, 1970), the original Schelling segregation model (Schelling, 1971, 1978) and the more recent SugarScape models developed by Epstein and Axtell (1996). In general, it is difficult to get realistic social networks in this representation of the environment. Further, as early results showed, unless the grid is bent into a torus, the resultant social behavior is largely dictated by "edge effects"; i.e., restrictions on activity caused by being at the edge of the physical grid.

More advanced models place agents in a socio-demographic space and separate the physical and the social space. In such models, very few have explicitly modeled the social network. Increasingly, however, researchers are incorporating more realistic network representations, such as small-world, scale-free, or other types of network generators. The most advanced of these models are the dynamic-network ABMs in which the networks and the agents co-evolve (the first model of this type was Construct). In some cases, the models are instantiated with networks that are derived from real data. These models will often generate or import an appropriate graph before the simulation agents are initialized, and then assign each agent to a graph position when the simulation starts. Other models use a social network gathered from empirical studies. These networks have the advantage of being as realistic as possible but may potentially bias the simulation results due to the structure and nature of the particular social network gathered. Correctly specifying the topology of a social network in an agent-based model has important implications for the conclusions drawn. In modeling an adversary, it is valuable to use the social network of the adversarial group.

The quality of the social network modeling can have important effects on simulation outcomes. For instance, in the Construct's Standard Interaction Model, the social network topology has a non-linear effect on knowledge diffusion rates in the system. Construct uses sophisticated agents that can interact and choose partners with which to exchange knowledge and belief. A stylized meta-network, which specifies the pattern of potential partners with which an agent can interact, can be imposed to limit the form of the evolved networks. We use Construct to model the adversary. Our results indicate that the most effective type of intervention depends on how the adversary is structured, e.g., Al Qaeda and Hamas have different structures and the same intervention, such as isolation of the top leader, in the two cases can lead to performance decrements in one and performance improvements in the other.

Although frequently lumped together, ABMs vary widely in complexity and computational cost – some are extremely inexpensive (e.g., Swarm) and allow hundreds of thousands or even millions of agents to operate in the same simulation, while others are rather expensive and often require the support of an entire processor per agent (e.g., SOAR or ACT-R). This increase in computational expense, however, is matched by construct validity to the actions of cognitively bounded humans: the least computationally expensive (per agent) simulations replicate the behavior of insects (specifically ants) while ACT-R has been able to replicate the brain activation patterns of children solving algebra problems and SOAR has replicated fighter pilot operations in concert with human pilots.

Although economics are an important consideration in picking an agent-based simulation, they should not be the only consideration; the specific phenomena of interest should impose its own set of criteria. For problems of traffic analysis or collision avoidance, swarm agents are particularly appropriate. However, in phenomena with significant cultural freight, such as those involving deception, leadership, participation in group activities, and/or compliance with group norms, these swarm-based technologies offer little useful insight to the policy analyst without

additional (expensive) modification and incurring significant increases in computational cost. At the same time, not all group-based phenomena require the detail and expense imposed by high-fidelity models of individual agents. Construct, which can support hundreds and thousands of agents, supports an appropriate middle ground. It also supports one of the only agent-based models which explicitly unites Herb Simon's dual requirement of bounded rationality, that rationality should be bounded both cognitively, and socially (Simon, 1957). Most of the highest-fidelity models constrain interaction to explicit messages, if at all, and many works entirely in isolation from other agents. Construct, thus, is less expensive and yet more useful for studying group phenomena.

A common query is to which specific theory of group behavior does Construct adhere? Construct does not subscribe to a specific theory of group behavior. Indeed, the question can reflect a fundamental misunderstanding of interesting modeling work – rather, the level at which a simulation is specifically coded/designed is its least interesting level of analysis. Analysis at the level in which a model is coded suggests merely how well the simulation programmers did their work, this is an important verification question, but not of practical application interest to model consumers. It is necessary, but not sufficient, for a model to be correctly coded. Instead, the more interesting question, available to be asked of agent-based simulations, is what are the larger implications with how these agents interact? We call this principle "emergence", what larger phenomena "emerge" from the interactions of these modeled agents. Construct is, as previously said, an agent-based simulation, and thus represents a theory of individuals and how they choose to interact. The Standard Interaction model makes a claim based on research that people tend to interact with other people based on two competing drives. One, that people tend to interact with others because they believe they are similar (the drive for homophily), and two, that people tend to interact with others who they believe have valuable knowledge they do not have (the drive for knowledge expertise). Both of these human drives are cross-cultural.

Emergent properties of the simulation, then, are much more interesting to the agent-based simulation modeler than the direct consequences of their modeling decisions. Based on agents interacting with others due to knowledge expertise and homophily, Construct models have been able to replicate many group-level behaviors found in people: the S-Shaped curve of diffusion, yes, but also that beliefs are more durable than the information used to support a belief. Construct has examined cultural norms in organizations, belief-changes in national decision-makers, and group stability. In practice, Construct is a valuable support for group-level behavioral theories because it provides an explanation rooted in individuals for the origin of these phenomena. These emergent properties, however, may not always be intuitive to the model consumer or model developer. At such points, it is important to recheck questions of verification, that some bug in the model process is not to blame for the errant results. But more interesting is when the model's code is not in error, but the results are still surprising.

Although not directly attributable to programming error, there may be other sources of surprising results that should be described. One, the model simulation is, at its core, not a

sufficiently good model of the atomic primitive it represents; this is often the case when extending swarm agents beyond issues of traffic and navigation. Two, the experimental approach was not well matched to the empirical reality – if, for example, 75% of adults in the population are internet-literate, but the model assumes that only 10% of the agents will receive information from internet sources, the model will significantly underestimate the prevalence of information from internet sources, and there may be further cascading effects of that error. Three, the results may simply not be well communicated. Relating accurately (and conservatively) the implications of models is itself a skill that must be polished.

However, sometimes, the results are non-intuitive and yet none of these errors appears to be present. In such a case, this is the value and joy in modeling counter-factual scenarios – we can place our simulated humans in situations that do not exist and will never exist and be surprised and intrigued by how they behave.

Introduction to the Report

Construct Versions and This Report

Construct is, like all but end-of-life software, undergoing continuing development in both its capabilities and its implementation. Experiment developers and designers should ensure they are using the most current version of Construct available on the CASOS public web site on the Construct Software page. They should also ensure they are referencing the most current set of documentation to reduce the probability of a disconnect between the documentation and the application. Finally, experiment developers and designers should consider subscribing to the CASOS's ORA Google Group for ad-hoc and peer-to-peer assistance as well as assistance from students, staff, and faculty of CASOS.

Conventions Used in This Document

Where feasible, this document quotes a provided example of a Construct experiment configuration file. The sample file is shown in Appendix A, in a 2-up printed format, using the courier new font in a reduced font size. The sample file is also available for download at the Training and Sample Data page on the Construct page of the CASOS website. This report uses the following typeface conventions:

Code snippets will also be written in the `Courier New`, 11 pt text. These snippets are quotes from the demonstration input file. We will also frequently call the input file the input deck, or shorten the name to deck, throughout the document. The origins of this use of the word 'deck' will deliberately remain in the mists of our collective memory lest the authors prove how old they really are.

Construct keywords, will also use the `Courier New`, 11 pt font (the *Code* style in MS Word). Additionally, variables and network names will use the same style.

A blue box and text inside the box indicates information the experiment developer and designer, researcher and simulationist should be particularly aware of when using Construct.

We will reduce the extended list of potential audience members from "experiment developer and designer, researcher and simulationist" in most cases, to "researcher" and/or "simulationist" throughout the document.

Egos and *Alters* are common referents in social science literature that we will use throughout this report. Their use simplifies establishing frames-of-reference and scoping of interaction possibilities. When we refer to a single agent, it will most often have the label of ego. When we refer to the agents or other entities that the ego is connected (in any sense of the word), they will most often have the label of alter or alters. Agents in the simulation not connected to an ego are beyond the scope of awareness of the ego, and do not directly affect the ego.

Organization of This Overall Report

The report has three main components and does not need to be read or referred to in front-to-back sequence. The three parts are below:

PART ONE: Quick-Start Guide - for a relatively quick progression from introduction to execution of Construct

PART TWO: Construct in Detail - for an in-depth explanation of Construct, complex inputs and outputs and complex experiments

Appendices - for additional useful sets of information ranging from additional exemplar input decks, to the use of Construct in High Performance Computing (HPC) environments such as HTCondor, to brief synopses of peer-reviewed projects where Construct played a role.

A Motivating Example

One method of introducing a set of concepts and the application of those concepts to problem solving is by a motivating example. In this report, we adopt this method and present a motivating example for both the questions of interest (QoI) as well as an experimental configuration that can help answer the QoI.

Like all scientists, if we are not attempting to answer a specific QoI, or even a set of QoI, it behooves the reader to take some amount of time to focus the upcoming effort. It is appropriate at this time to remind the experimenter that Construct's roots lie in social network, information diffusion and belief diffusion modeling. This motivating example will stay with this core capability and defer discussions of additional capabilities and experimental purposes to PART TWO: Construct in Detail.

Core Mechanisms

As previously mentioned, Construct is a framework which can be seen in Figure 1. In this framework we have nodes, networks, models, input, and output. Construct's primary function is to properly interface all these components together like the stairs and hallways in a house. Unlike a regular house however, Construct can expand, and contract as needed to facilitate an end user's requests. Construct is able to handle an arbitrary number of nodesets and networks and inputs for those areas. Construct is also able to handle one to an arbitrary number of models assuming there are no conflicts between models as well as many different types of output. The models and output however are limited to what is already built into Construct.

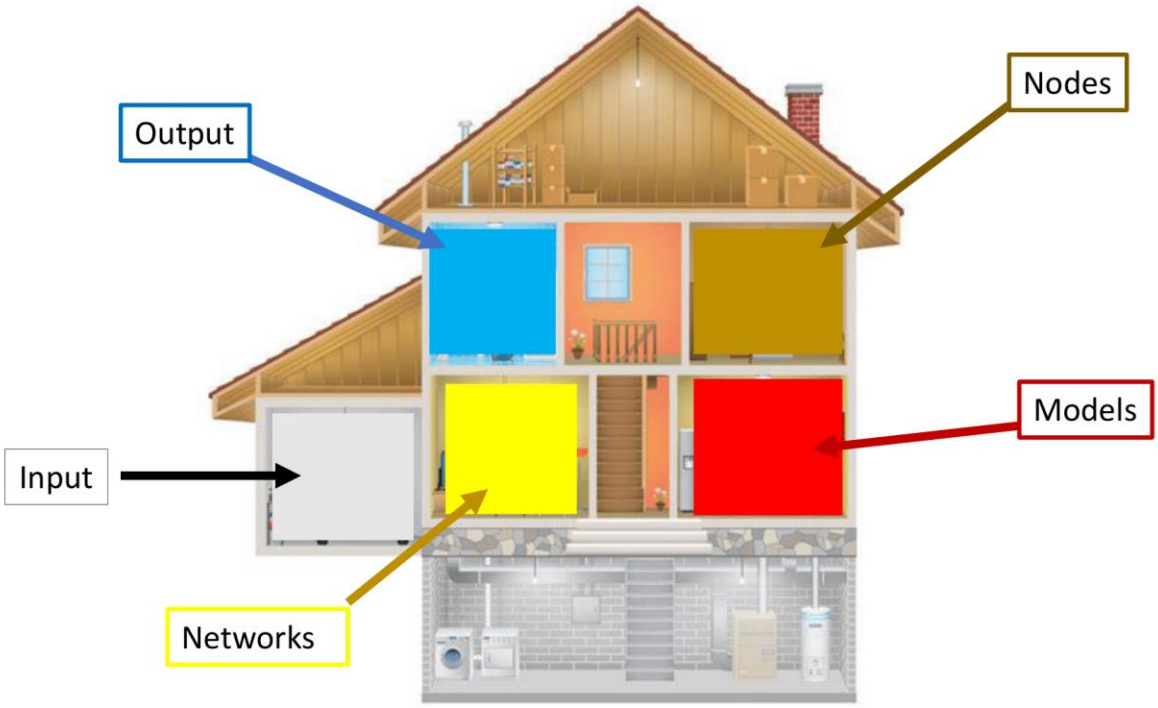

Figure 1: A visualization of the Construct framework as a house.

While Construct is a framework for agent based simulations, the models built into Construct are where the magic happens. The Standard Interaction model is Construct's signature model and the starting point for additional models and modifications. This model combines many different aspects such as decision making, technology restrictions, and information propagation. Agents make decisions about who to interact with, what information to transfer during an interaction, and which communication medium to use for that interaction. The communication mediums present technological limitations as mediums such as books, which are rich in information, but is only one way or face-to-face conversations, which happens instantaneously in both directions.

Finally, when these interactions take place the resulting information spread affects the decision making of each agent such as the search for more exclusive information as an agent's repertoire expands. PART TWO: Construct in Detail goes over the specifics of this and other Construct models.

A Scenario

We, the researchers, are analysts that Acme, Inc. has hired to help Acme design two software development teams in a 'clean room' configuration. Acme wants the two teams to be co-developing a product. Acme also wants structural mechanisms in place to control how much information flows between the two teams as a method to help reduce the probability of unintentional release of Acme's intellectual property. One way of visualizing this scenario is in Figure 2. In this figure, we also call each team a cluster, aligning with the social network analysis literature when groups of entities are meaningfully connected to each other.

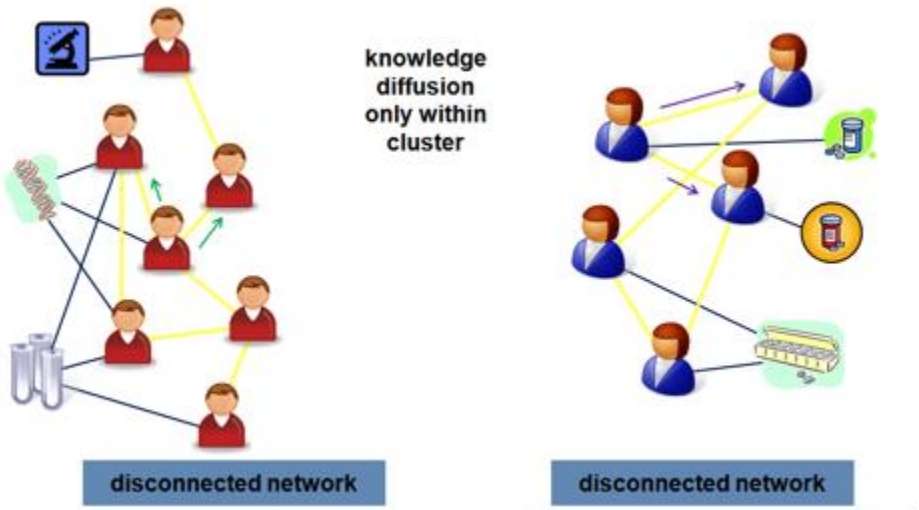

Figure 2. A depiction of two 'clean-room' teams of product developers.

In the figure above, possible questions of interest that are appropriate for the model to help forecast answers could be:

Without direct modeling, is there any leak of knowledge from one team/cluster to the other? If so, how fast does the information flow?

Assuming no friendship networks or other communication networks not modeled, how fast does specific knowledge or specific beliefs within each team spread?

Assuming a requirement to have a controlled mechanism to support the teams passing limited information back-and-forth, to whom would such an intermediary best talk in each team for rapid spread of information or beliefs?

Does either team have any organizational weak point that can be structurally overcome?

After stability is reached within teams for knowledge saturation/diffusion, what kinds and how large are impacts of personnel turnover of various sizes and frequencies have on the group? How long, if at all, does the team take to return to pre-turnover levels for specific measures of interest?

These and other questions can be explored within the Construct framework. In Part 1, we will describe the entities and key relationships between those entities. The treatment in Part 1 is intended to be useful towards further orienting a potential model builder or a model consumer. Part 2 describes mechanisms at a high-level of detail and is suitable to act as a reference even to a regular user of Construct.

PART ONE: Quick-Start Guide

This section is an introduction to core mechanisms of Construct and its Standard Interaction model, introduces three of the most important networks to understand, and suggests a set of experiments that may be of some interest to the model consumer. It is intended to provide an initial suggestion of how Construct may be useful to the model developer. More detail is provided in the second part of this report. We assume that the example deck included in this technical report is available to the reader of this guide.

We begin this guide by providing a summary of key objects within Construct and provide examples of the various semantics between these key entities. We then describe, in more detail, the more precise semantics of three critical networks in the Standard Interaction model. Next, we show a suggestion of some experiments that could be done using only those key networks, referencing the motivating scenario. Finally, we go over how to include additional models into Construct and a high-level discussion of how models interact with each other.

The Objects

Construct organizes classes of objects into what we call node sets. Some examples are 1) agents, 2) knowledge, and 3) time. A singleton example of each of these object classes is referred to (respectively) as 1) an agent, 2) a knowledge bit, and 3) a turn. In addition, node sets are globally available in Construct for usage in constructing networks, operations in models, and various other areas in Construct.

Agents

Agents are the most important class of objects in Construct's model library. Agents have, appropriately, agency, and thus make choices that can potentially affect other agents. Typically, agents represent human-like entities, but researchers can also represent other types of entities such as sources of information (e.g., newspapers, radio programs, or television ads) and information technology (IT) systems (e.g., databases, data-stores). Agents have various critical capacities and capabilities that we will address briefly here and more thoroughly throughout the report.

Individual agents possess different bits of knowledge and can be aware of other agents. An important lesson this guide hopes to teach is how to manipulate both what agents know, and who they know.

Agents may also be members of groups. This can be useful for labeling and categorizing outputs and making per-group analysis easier. Group members, like in our motivating example, tend to have many more connections within the group than outside of it. It can often be easier, but not semantically important, to define groups of agents contiguously. If I were, for example going to group my digits by which hand they are on, it would be easier for me to simply count them off, so that my right hand's digits were 0,1,2,3, and 4, while my left hand's digits were

5,6,7,8, and 9. Then, all I need to remember is that my right hand's digits start at 0 and end at 4, while my left hand's start at 5 and end at 9. Alternatively, I could count them off by functional role (right thumb 0, left thumb 1, right pointer 2, left pointer 3, etc.), but that would quickly become confusing if their membership in my hand groups was their most salient characteristic.

Individuals can also be connected to any other node set and they may change over time. Information on this is out of scope on this portion of the guide but will be discussed in Part 2.

Just as with people, some agents may have more capacity than others to send or receive information. As with people, they may have more or less retentive memories than others. And, as with people, they may have more or less social reach than others. Specifics on how to implement any of these (and other) characteristics is included in Part 2.

Knowledge

Knowledge represents information. The Standard Interaction model interprets real-world knowledge through a stylized and simplified series of bits (0 or 1). Any particular knowledge bit typically represents a single atomic piece of information, such as "Sol is the name of the star at the center of our solar system", or "Each water molecule is comprised of two hydrogen and one oxygen atom." It is incumbent on a researcher to keep the stylized representation consistent in their experiments – one bit should not represent "How to pilot a 747-jumbo jet" while another bit represents 'flight departed', without proper modification to how those bits connect to the rest of the model.

Time

Turns, in Construct, represent chunks of discrete time and is the only node set that is embedded in Construct's architecture. Each turn various events can happen such as an agent's opportunity to interact with other agents. It is usually good practice to attempt to identify, loosely, a length of time represented by each turn. Turns may be minutes, days, weeks, or months. This mapping of turns to time periods should be chosen relative to the knowledge being transmitted during each turn – it is unrealistic for highly complex knowledge, such as "Civilian Flight Operations", to be conveyed in less than some number of months or years. Thus, either the number of knowledge bits that represents Civilian Flight Operations is very large, or turns are likely to represent weeks or months in this model (or both).

Object Relations

In Construct, we note how each of these various objects are related to each other using dense matrices. These matrices are global and can be operated on by any component of Construct including various models. Each matrix usually referred to as a network or a graph, represents a meaningful and distinct tie between objects. Matrix values may be binary (either 0 or 1), weighted (any real number), or relational (strings or characters). These networks can represent relationships between objects of the same class (single mode), or between objects of different

classes (a multi-mode matrix). The objects listed down the rows are always listed first, then the objects in each column.

'0' is usually a safe default value for matrices, indicating no connection between the entities. Non-zero values usually indicate that the two entities (represented by the row-column pair) are "connected". There are exceptions, discussed in Part 2, for the various 'weight' networks.

For example, a binary (0 or 1s) Agent x Knowledge multi-mode matrix might look like:

	Biology	Physics	Sociology
Aba	1	1	0
Jane	0	1	1
Lu	0	1	1
Raj	1	0	1
Fred	1	0	0

In practice, each of these large areas would be represented by a range of knowledge bits, since none of these sciences are single atomic facts, but as an example we hope it suffices.

Part 2 will discuss all the different matrices present in Construct's models, their real-world meaning, and their practical impact within Construct. This guide will focus on two key networks that prevalent through much of Construct's models: the interaction sphere, the knowledge network. It will also show you the snippet of XML code required to specify each of these key networks.

The Interaction Sphere

The interaction sphere defines "who may know who". It is a single-mode, Agent x Agent, binary matrix.

> *If two agents in the interaction matrix have no connections, they will* never *be able to interact directly with each other. Agents that have connects in the interaction matrix may never interact due to random number generation and probabilities.*

Because agents must be able to interact to pass information, it is easy to see how changes to the interaction sphere can change how the experiment will play out. Generally, agents should not be connected in the interaction sphere if it is unlikely, they would ever have reason to interact. Separate organizations, for example, may not have any connections to each other, save perhaps through explicit liaison personnel.

Here is the code required to specify the interaction sphere:

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="interaction sphere network" link_type="bool" network_type="dense">

   <generator type="randombinary">
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0" last="nodeclass::agent::count_minus_one"/>
      <param name="mean" value="1"/>
      <param name="symmetric_flag" value="false"/>
      <param name="mean" value="1"/>
   </generator>
</network>
```

This is your first exposure to Construct XML, so it may seem a little daunting at first, but let us parse this XML line by line.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
```

The `network` at the beginning indicates we are defining one of the matrices used in Construct. The argument `src_nodeclass_type` tells us that the matrix we are defining should have rows defined by agents, and the `target_nodeclass_type` argument tells us that the columns should also be defined by agents.

```
id="interaction sphere network" link_type="bool"
```

The `id` argument gives us the name of this network. As Construct loads in graphs arbitrarily, this name is used as a reference to find specific graphs in the various components of Construct. The `link_type` argument tells us that this network is Boolean (stored as a binary value), either a link exists (1/True/T), or it does not (0/False/F).

```
network_type="dense">
```

Currently all networks will have this argument `network_type` set to dense. This means that every possible cell combination should be defined. It is the goal for any developer to try to plan for the future, and we intend to add additional network types in future.

```
<generator type="randombinary">
```

The `generator` is a new object; it is being defined to help us fill in the values of the interaction sphere. There are different generator types - this type, a `randombinary` generator, will generate only 1s or 0s. It generates 1s at a given rate. A more complete discussion of the various generator types is under the Generating Random Numbers heading.

```
<rows first="0" last="nodeclass::agent::count_minus_one"/>
```

> *All numbers used to count things in Construct XML use cardinal numbers, also known as "computer science counting", where the first index value is 0, not 1.*

Thus, the last digit on my right hand is digit number 4, not 5, even though I have five digits. I have, using cardinal numbers, counted out five numbers (0, 1, 2, 3, and 4). The rows object tells the generator in what parts of the matrix it should assign numbers. You can (and often will) use multiple generators for one network. In this case, the generator should assign values for all rows of the matrix - `0` is the first agent, and `agent::count_minus_one` is a built-in mechanism for Construct to identify the number of nodes in a node set. It works with all defined node sets in the input file. *It is a handy shorthand, so you do not need to keep track of how many agents exist.*

All generators (except one) assume that they should fill in all values inclusive of and between the first and last of both the row and column arguments. The one exception, not discussed in detail here, is reading in a network from a file. Thus, if you want two or more groups of agents, you may want to keep track of the start and end of those groups. Therefore, it is usually easier to number your agents contiguously by their most important group affiliation, as discussed previously with hands and digits in the Agents section above.

```
<cols first="0" last="nodeclass::agent::count_minus_one"/>
```

This serves the same purpose as the previous line, except it defines what columns the generator will be assigning values to. As you can probably guess, we are assigning values (either 1s or 0s) to all columns as well. This means this generator will provide a value for every cell in the matrix.

```
<param name="mean" value="1"/>
```

This is the parameter that defines how often a "1" is likely to come up. In this case, a 1 should populate every cell in this matrix. What does that mean for our simulation? Think about it for a second. Done? In this case, it means that every agent can talk to every other agent. If you were going to modify this code for use in our motivating example, how might you go about it?

```
<param name="symmetric_flag" value="false"/>
```

The `symmetric_flag` is very important, and important to understand. Not all relationships go both ways. My boss, for example, may have access to me, but I do not always have access to my boss. If the president wants to see me, he will, but I cannot bully my way into the Oval Office. If the symmetric flag is set to the true, then none of the relationships in your group will be asymmetric - they will all go both ways. If it is set to false, then some asymmetries may arise, but not necessarily. Would there be any asymmetrical relationships in this network, given the generator, as you understand it to date? Multi-mode matrices should not have the symmetric_flag set to true.

```
</generator>
```

This indicates that the generator definition is complete and closes the object.

```
</network>
```

This closes the definition of the network, remember, you may have multiple generators in a single network definition. I include the entire XML snippet again for easy review; we hope it is easier to understand the second time. Beneath it, I give my read-aloud version of how I parse this network and relate it verbally.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="interaction sphere network" link_type="bool" network_type="dense">
   <generator type="randombinary">
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0" last="nodeclass::agent::count_minus_one"/>
      <param name="mean" value="1"/>
      <param name="symmetric_flag" value="false"/>
      <param name="mean" value="0.20"/>
   </generator>
</network>
```

"This is the interaction sphere network; it is an agent by agent network with boolean/binary links. It uses a random-binary generator, which will define values for every agent to every agent. This random-binary generator will put 1's in approximately 20% of the cells of this matrix. The generator is not explicitly symmetric."

> *It is important to note that each column agent that has a link to a row agent is in that row agent's sphere of influence. In the case the network is not symmetric, some ego agents may not be in an alter's sphere of influence, but the alter agent in the ego agent's sphere. This is an important aspect when construct communities and connections between communities.*

The Knowledge Network

The knowledge network defines "who knows what". It is a multi-mode, Agent x Knowledge, non-binary matrix. A '1' in this matrix indicates the agent "knows" the fact represented by that bit. The knowledge network can be updated throughout the run of a simulation by a variety of models. Other models may simply observe the knowledge network.

Agents can only communicate knowledge that they "know," or have access to, when they interact with other agents. Thus, changes in the knowledge network will have strong effects on how the simulation proceeds.

This is the Construct XML used to define the knowledge network in our example deck and how I would read it aloud:

```
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
id="knowledge network" link_type="float" network_type="dense">
   <generator type="randombinary">
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0" last="nodeclass::knowledge::count_minus_one"/>
      <param name="mean" value="0.1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

> *This is the knowledge network; it is an agent by knowledge network with non-binary links. It uses a random-binary generator, which will define values for every agent to every knowledge bit. The random-binary generator uses a probability of '.1' to place a 1 in each cell. The generator is not, both explicitly and functionally, symmetric.*

Most of the XML looks very similar to the previous example.

Outputs

Researchers and simulations usually compare outputs of Construct simulations by examining files written over the course of the simulation. It is outside the scope of this quick start guide to offer in-depth suggestions on how to deal with large quantities of simulation data. We will instead go over the basic tools available in this guide.

> *When Construct writes matrices to file(s), as in this example to a comma separated value file, it will separate each row from the others with a line termination symbol appropriate for the host operating system (Carriage Return/Line Feed for Windows-type OS). If a researcher has Construct write multiple time periods to a single file, each time period is separated from others with a single empty line.*

Here is an example that we will start with followed by a break down each component.

```
<outputs>
   <output name="output_graph">
      <parameter name="network_name" value="knowledge network"/>
      <parameter name="output_file" value="knowledge.csv"/>
      <parameter name="timeperiods" value="all"/>
   </output>
   <output name="output_graph">
      <parameter name="network_name" value="interaction network"/>
      <parameter name="output_file" value="sphere.csv"/>
      <parameter name="timeperiods" value="last"/>
   </output>
</outputs>
```

A quick overview shows the output element of the deck with two outputs. The knowledge network is being dumped by the first output in the file “knowledge.csv” for all time periods. In the second output, the interaction network is being dumped into sphere.csv, but only the network as it appears at the last time period.

```
<output name=”output_graph”>
```

This indicates what type of output is requested. Outputting the graphs is the primary method, but other methods are available in Construct.

```
<parameter name="network_name" value="knowledge network"/>
```

This line indicates which network is to be output. Only one network can be output at a time per output. It is important to note that the network name much match the network name in the input deck. Additionally, networks not supplied in the input deck can still be output if a model creates its own network, see “interaction network” in Part 2.

```
<parameter name="output_file" value="knowledge.csv"/>
```

An output file name is required for output. Files are saved in the working directory from which construct is executed.

```
<parameter name="timeperiods" value="all"/>
```

Two options are available for output, `all` or `last`. When `all` is called, all time periods are output as described above. `last` only outputs the last time period. This can be useful when only the end result is required.

Models and Construct Program Flow

Models in Construct operate using a plug and play methodology. Ideally, each model can run simultaneously while operating on the same set of nodes and networks. However, there are always limitations when attempting to create an arbitrary interface for which the models to interact. For example, a previously existing model would not know to access a new network created for a newer model.

One method that allows better compatibility between models is its implementation of a message exchange. By allowing each model to manipulate the messages that other models may want to send, certain behaviors can be obtained without editing a model's source code in a way that would not be possible by editing of networks. As we will show below, models can cause standard messages to be delayed in a mailbox type data structure or create irregularities in a message based on an agent's lack of literacy. In addition, viewing these messages can give models usage statistics for behaviors like a "use it or lose it" style of forgetting knowledge.

To accomplish this goal, all models have a similar structure. First models load only the nodesets and networks that the model requires from Construct. Each model then performs a standard set of instructions which can be seen in the Figure 3. Each model has a set of five functions that are called in the order shown that ensure models can create, manipulate, parse, and digest messages. Each function is completed by all the models before any model advances to the next step (i.e., each model completes the **Think** function before any model performs the **Update** function). Additionally, most models can be separated into interaction models, in which the primary purpose is to determine how interactions form and create messages to be sent between an interaction pair and manipulation models, in which the primary purpose to manipulate flow and content of messages. Below we dive into the five functions that allow the sufficient generality to achieve plug and play functionality of models.

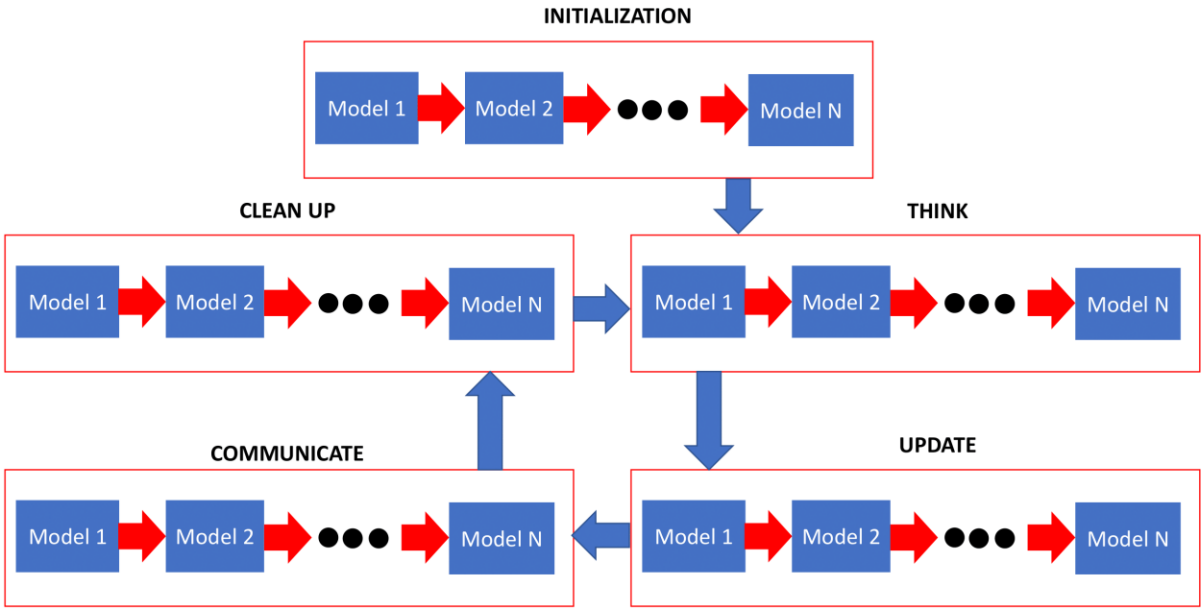

Figure 3. Model execution cycle: after INITIALIZATION executes once, each complete time period begins with all models performing THINK and ends with each completing CLEAN UP.

Initialize Function

The **Initialization** function is called at the beginning of the simulation once. The primary purpose of the initialize function is to check for the existence of other models. As much as this project aims for models to be independent, it is inevitable that some models are mutually exclusive with other models. The initialize function allows each model to check for other mutually exclusive models after all other models have been loaded in. In addition, some models may change behavior based on the presence of other models which can also be checked here. This function is only performed once, prior to the start of the model execution cycle proper.

Think Function

The **Think** function is a critical function for the interaction models as this function's primary purpose is generating messages. Messages creation in this step is generally independent of other models. Some possible secondary dependencies may arise if one model modifies a network another model uses, however this is not done by any of the currently developed models. This function is the first function executed in the model execution cycle during a time period.

Update Function

The **Update** function allows each model a chance to manipulate messages created by other models. This ranges from adding additional information to a message, modifying existing information in the message, removing information in a message, removing a message entirely, to

copying a message to send to another recipient, as well as additional fringe cases. This function is the second executed during a time period.

Communicate Function

The **Communicate** function takes in each individual message and parses its contents. Each model is responsible for parsing the contents of messages it creates as well as any information it tacked onto another model's message. This allows each model the partition itself from each other which having to worry about additional content that may be in a message. This can be particularly useful as the node and message information space increases as previously existing models will not require modification. This function is the third executed during a time period.

Clean Up Function

The **Clean Up** function allows each model to update various strategies and characteristics based on the communicated messages in preparation for the next time period's **Think** function. This function is the last function executed in the model execution cycle during a time period.

Models

Below shows the list of available Construct models. There are two types of models: Interaction Models and Modification Models. Interaction models generate messages that agents exchange, where Modification models primarily modify already existing messages.

- Standard Interaction Model
    - The most fundamental version of Construct's interaction models which relies on proximity, similarity, and expertise to find well suited interaction partners.
- Knowledge Transactive Memory Model
    - An expansion on the Standard Interaction model, this model utilizes an error prone memory of who knows what to provide more realistic interaction seeking.
- Twitter Interaction Model
    - A model that describes how individuals use the Twitter social media platform and how opinions can spread in this environment.
- Mail Model
- Knowledge Learning Difficulty Model
- Literacy Model
- Forgetting Model
- Subscription Model

Here is an example of calling a model. Each model is unique and adding multiple models with the same name will result in warnings that redundant models are being discarded. Models may also have parameters that are specific to that model which is elaborated upon in each model's section in PART TWO: Construct in Detail.

```
<models>
   <model name="Knowledge Transactive Memory Interaction Model"/>
      <param name=”false_positive_rate” value=”0.2”/>
      <param name=”false_negative_rate” value=”0.3”/>
      <param name=”threshold” value=”0.1”/>
   <model name="Forgetting Model"/>
</models>
```

Thoughts on Experimentation

In this guide, we have discussed a set of primitive objects in Construct and three key networks. These networks are:

- the interaction sphere networks, which defines “who could ever know who”,
- the knowledge network, which defines “who knows what”, and
- knowledge transactive memory, which defines “what people think other people know”.

The example deck, as provided, does not quite the serve the needs of the scenario as given. This is intentional. The changes required are relatively minor, and can be confronted with a variety of approaches, but should be explored directly. The motivating scenario suggests:

- Two groups of agents
- Each group has unique knowledge to their group.
- The two groups are, initially, completely isolated from each other.

Whereas, the deck, as shown here says that:

- All agents connected to all other agents.
- All agents have similar knowledge.

Obviously, some effort will need to be made to reconfigure the interaction-sphere and the knowledge network so that groups can be isolated and that groups may have unique knowledge. We leave it up to the reader to consider how such a change may be achieved. Remember that multiple generators can be used to define values for portions of the matrix space.

ORA is a complementary tool to Construct. ORA is a network analysis tool capable of creating and analyzing meta-networks, a collection of nodes and networks, and dynamic meta-networks, a collection of nodes and networks that can vary over time. Naturally, this tool can be used to analyze the time dependent networks that Construct produces. Construct thus supports output in the DyNetML XML file format that ORA uses to import dynamic meta-networks. Usage of this method can be seen in Section DyNetML Output.

In addition, ORA can be used to create and manipulate nodes and networks which can be imported into Construct. This is expanded upon in PART TWO: Construct in Detail in the relevant sections regarding node and network generation. A common problem is misspellings when creating nodesets and networks in ORA for use in Construct. It is critical that any networks or nodesets created in ORA by the user have the exact name that Construct expects (i.e., “knowledge network” or “agents”). In addition, many models expect attributes in a nodeset. In

ORA, this can be done by importing attributes from a text file, or by adding attributes to existing nodesets and editing the attribute values with tools such as Transform Attribute Values to manipulate attribute values.

PART TWO: Construct in Detail

This section of the report, to some degree, repeats information provided in Part 1: Construct Essentials. This is a deliberate choice by the authors.

Part 2 provides in-depth details of the workings of Construct. Topics include the operations of an example deck, the outputs from an example deck, agents, knowledge, binary knowledge, non-binary knowledge, forgetting, transactive memory of knowledge, beliefs, belief formation equations, tasks, binary task selection, energy tasks, biased binary task selection, interactions, and additional special topics included as appendices. Throughout this portion of the report and the appendices, we assume that readers have some familiarity with general programming concepts and terminology – which may lead us to skip details that an introduction to programming text would include but would seem pedantic here.

Variables

Declaring, Defining, and Casting Variables

Variables in Construct are generally user specified constants for a specific simulation. Modifying the values of Construct variables is part of the Scripting Language support, which Appendix D discusses in detail. Researchers frequently use variables to make the input file easier to read and adjust for future simulations – changing a value in a single place makes maintaining consistency easier than relying on 'search and replace.' Examples of variable use include setting the total number of agents, changing agent group sizes as a function of the number of agents, and many other uses. Construct expects variables to be at the top of the input file enclosed in paired `<construct_vars></construct_vars>` ConstructML tags. Modelers declare and assign initial values to variables once, and then reference that variable whenever needed throughout the input deck. Below is a sample of ConstructML showing the four ways a modeler can declare and define variables:

1. declare and define as a constant (`var1`, `var2`, `var3` below).
2. declare and define in terms of another, previously declared, variable (`var4` below).
3. declare and define in terms of a mathematical or logical operation on other prior-declared variables or constants (`var4` and `var5` below).
4. declare them and assign constant values read from a Comma Separated Variable (CSV) file (`time_count` below).

> *Modelers must declare variables prior to using them, or Construct's parser will fail.*

```
<construct_vars>
```

```
<var name="[name]" value="[value]"
with="[delay_interpolation|verbose|details|preserve_white_space|preserve_s
paces_only]"/>
...
<!-- examples of var declarations and definitions -->
   <var name=”var1” value=”1” />
   <var name=”var2” value=”’var2 as string’” />
   <var name=”var3” value=”var3 as another string” />
   <var name=”var4” value=”construct::intvar::var1” />
   <var name=”var5” value=”construct::intvar::var1+1” />
   <var name=”var6”
      value=”construct::intvar::var1+construct::intvar::var1” />
   <var name="param_file_name" value="params.csv"/>
   <var name="param_name_column" value="0"/>
   <var name="param_value_column" value="1"/>
   <var name="time_count"
      value="readFromCSVFile[construct::stringvar::param_file_name,
      construct::intvar::num_turns_param_row,
      construct::intvar::param_value_column]"/>
</construct_vars>
```

The keywords that a modeler can insert into the `with` attribute of the `var` tag cause Construct to perform in specific ways.

- `delay_interpolation` causes Construct to not evaluation variables or expressions while conducting the first pass of parsing. This prevents the XML parser from interpreting the xml attribute as code during the initial parsing phase and instead treats it as text.
- `verbose` causes Construct to be verbose as it evaluates the `name` and `value` attributes. The value of the parameter both before parser initialization, after parser completion, and after evaluation are printed for diagnostic and debugging purposes. Additionally, should the parser encounter an error, this keyword tells the parser to provide a more verbose error message.
- `details` can be used in conjunction with the verbose parameter to determine the values of macro substitution parameters. While the verbose keyword can be used to debug a simple math expression, it may be necessary to see additional information about the state of the parser as it evaluates macro expressions. The details parameter prints out any information about variables in use, in addition to some very specific information about the internal state of the parser as it examines the input string. Should the parser encounter an error, it should also provide more information about the parameter value.
- `preserve_white_space` causes Construct to treat all white spaces as important to the expression. Construct will not remove tabs, returns, spaces, and comments as the expression is evaluated. By default, all white space is removed during the creation of variables. If included as a keyword, however, any white space in the value will be preserved when the parser is run.
- `preserve_spaces_only` causes Construct to treat only spaces as important to the expression. Thus, Construct will preserve spaces but ignore returns, tabs, and comments. Using this parameter will allow for newlines to be placed in scripts which must preserve spaces.

Construct variables are not case sensitive. Construct converts variables to lowercase for internal use.

Like many programming languages, Construct requires variables start with an alphabetical letter. Variables can use ASCII alphanumerics and the underscore; other special characters will cause the parser to fail. Variable names are globally accessible throughout a simulation's input file and must therefore be unique across the simulation's input file; there is no lexical scoping or overloading. Construct supports the multiple variable types though the astute reader will note the above ConstructML has no explicit typing associated with each variable. The supported variable types are floats/decimal values, integer values, strings, Booleans, and even expressions that can be evaluated as scripts. Appendix D discusses scripts, scripting, and evaluation of script segments in detail. To reference a variable, a modeler would type the following as a general syntax:

```
construct::[type]::[variable name]
```

An example of a variable reference would be:

```
construct::boolvar::short_experiment
```

While the `[variable name]` field can refer to any variable defined within the simulation, there are a limited number of `[type]` values that Construct accepts. The use of `[type]` helps Construct cast the `[variable name]` to the appropriate C++ type for processing.

If a modeler omits construct::[type]:: as a preface to [variable name], Construct will attempt to deduce the variable type.

Modelers who rely on Construct's built-in type heuristics for type guessing may get unexpected results and the authors highly encourage the verbose method of referring to variables in input decks! The five supported values for [type] are shown, in alphabetical order, below.

§ `boolvar`, defines the variable as a Boolean (true or false). Construct follows the C convention that zero is false, non-zero is true. The authors highly recommend modelers to use the newer convention of zero is false, and one is true. If the modeler is attempting to cast a variable to a float, the following casting rules are in place.

- If casting from a non-zero integer or float, Construct casts the value as true.
- If casting a zero-valued integer or float, Construct casts the value as false.
- If casting from a string, if the string is "true" (case insensitive) or evaluations to a non-zero integer or float, Construct casts the value as true; otherwise, it casts the value as false.

§ `floatvar`, defines the variable as a float (sometimes referred to as double in this report). Construct supports positive and negative floats. If the modeler is attempting to cast a variable to a float, the following casting rules are used.

- If casting from an integer, Construct simply adds a decimal place and zeros.
- If casting from a bool, Construct treats false as 0.0 and true as 1.0.
- If casting from a numeric string (e.g., '2', '2.15'), Construct will cast to a float and maintain or add decimal place digits as appropriate.
- If a mathematical function uses an integer value as a float variable, the result will be a float value.
- If casting from a non-numeric string or other variable that cannot be cast as a number, Construct silently casts the value as 0.0. There is no mechanism to warn a modeler of this situation in the deck during parsing, nor during execution.

§ `intvar`, defines the variable as an integer. Construct supports positive and negative integers. If the modeler is attempting to cast a variable to an integer, the following casting rules are used.

- If casting from float/double to integer, Construct silently truncates the original value. There is no mechanism to warn a modeler of this situation in the deck during parsing, nor during execution.
- If casting from a bool, Construct treats false as 0 and true as 1.
- If casting from a numeric string (e.g., '2', '2.15'), Construct will cast to an integer, and silently truncate, as it does with floats/doubles.
- If casting from a non-numeric string or other variable that cannot be cast as a number, Construct silently casts the value as 0. There is no mechanism to warn a modeler of this situation in the deck during parsing, nor during execution.

§ `stringvar`, defines the variable as a string.

Construct can cast all variable types to strings.

If the modeler decides to omit the single quotation marks in the variable declaration (e.g., `var3` above), Construct may still treat the variable as a string. It does this if the first white-space separated word in the string is not a Construct-reserved word. This behavior is silent. There is no mechanism to warn a modeler of this situation in the deck during parsing, nor during execution.

§ `expressionvar`, defines a variable as an expression. Construct evaluates the expression and returns it. An expression can evaluate to any of the other four [type] though it may require the modeler to cast the result to the desired final `[type]`.

There are at least two ways of casting a variable, or an expression composed of variables. The first is to cast within the value attribute of a var tag. Some examples are below. The second is to assign the value of one variable to another variable and cast it during the assignment process.

```
<var name="cast_example1" value="(4/2):bool"/>
<var name="cast_example2" value="(4/2):string"/>
<var name="cast_example3" value="(4.0/2.0):int"/>
<var name="cast_example4" value="(4/2):float"/>
<var name=”cast_example5” value=”construct::stringvar::cast_example1” />
<var name=”cast_example6” value=”construct::intvar::cast_example2” />
```

Evaluating Variables

Like many programming languages and applications, Construct reads its input deck from top to bottom, left to right. Variable names are read and stored before variable values.

> *Construct evaluates mathematical and logical expressions, as well as casting between variable types, from right to left, though modelers can make use of parentheses to specify a different evaluation ordering.*

The example mathematical expressions below in Table 1 provides another mechanism to allow this important point to be retained by modelers.

Table 1. Mechanism for evaluating variables in Construct.

Variable Declaration	Actual Value	Expected Value	Warning	Non-Intuitive Explanation
<var name=”var1” value=”3/5.0+1” />	0.5	1.6	X	5.0 + 1 happens first
<var name=”var2” value=”(3/5.0)+1” />	1.6	1.6		
<var name=”var3” value=”3-1-1” />	3	1	X	1-1 happens first
<var name=”var4” value=”3-1+1” />	1	3	X	1+1 happens first
<var name=”var5” value=”(3/5):float” />	0	.6	X	Integer division happens first
<var name=”var6” value=”(3/5.0)” />	0.6	.6		
<var name=”var7” value=”(3.0/5)” />	0.6	“0.6”	X	if either operand in division is a float, the result is a float
<var name=”var8” value=”(3/5.0):string” />	“0.6”	“0.6”		
<var name=”var9” value=”(3/5.0)” with=”delay_interpolation/>	“3/5.0”	“3/5.0”		See section on Variables, Macros, and with Statements

Construct supports the use of a macro language. With macros, users can automate the creation and use of variables to make their simulation input decks more flexible – at the expense of adding a level of complexity.

With dollar sign ($) delimited macro variables, a modeler can create a complex set of variables for use. With the use of dollar sign macros, a modeler must also use a `with` attribute in the `var` tag that declares the variable. Examples of macro use to declare and define variables are below.

```
<construct_vars>
<var name="letters" value="x,y,z" />
<var name="numbers" value="2,3" />
<var name="var_$i$" value="$i$"
   with "$i$=construct::stringvar::numbers" />

<var name="$letters$_$numbers$" value="$letters$$numbers$"
   with "$letters$=construct::stringvar::letters,
   $numbers$=construct::stringvar::numbers"/>

<var name="variable_1" value="$i$" with $i$=1 />
<var name="variable_2" value="construct::intvar::variable_$i$"
   with $i$=1 />

<var name="variable_$I$"
   value="construct::intvar::variable_$I:int$ + 1"
   with "$I$=3" />

<var name="variable_$i$" value="$i$"
   with "$i$=construct::stringvar::letters" />

<var name="variable_4" value="$j$"
   with "$i$=3, $j$=construct::intvar::variable_$i$,verbose"/>

<var name="attacktime_list"
   value="construct::intvar::attack_start_time..construct::intvar::attack_
end_time" />

<var name="attacktime_output_list" value="
   $currTime$ = construct::intvar::attack_start_time - 1; /* start output
at timer period: attack-1 */
   $step$ = 1;
   $result$ = '' + $currTime$;
   foreach $i$ (attacktime_list){
      if ($i$:int == ($currTime$ + $step$) ){
         $result$ = $result$ + ',' + ($currTime$ + $step$);
         $currTime$ = $i$:int;
      } else {
         $currTime$ = $currTime$; /* non-harm else statement, since 'else'
                                  is not optional in Construct if then
                                  else statements */
      }
   }
```

```
   $result$ = $result$ + ',' + ($currTime$ + $step$); /*stop output at
attack + 1*/
   return $result$;"
   with="$result$" />
   <!-- with="$result$,verbose" /> --> <!-- verbose here forces lexer
                              to dump to screen the processed
                              results of the scripting-->
</construct_vars>
```

Table 2. Variables as evaluated.

Variable Name	Value
Letters	“x,y,z”
Numbers	“2”
var_2 var_3	2 3
x_2 y_2 z_2 x_3 y_3 z_3	x2 y2 z2 x3 y3 z3
variable_1	1
variable_2	1
variable_3	4
variable_x variable_y variable_z	X y z
variable_4	4

Like non-macro variables, variables defined using macros must start with an alphabetic character. A macro of i is lexicographically distinct from I. Additionally, no macro should use a reserved word from the scripting language discussed in Appendix D. The declaration of a macro is valid only within the var tag it is in. Attempting to reuse a macro, such as i in a new var tag will create a new macro, not reuse the previous instance of i. Macros are expanded before any further evaluation of the variable occurs. Modelers that attempt to use macros without the `with` statement will receive a Construct error when parsing the input deck.

Construct macro variables are case sensitive.

The `with` attribute within a var tag can accept several pre-defined values as shown below.

- `verbose` - will print to the console standard out the evaluation of the parameter. Values will reflect the value before the Construct initializes the parser, after the parser complete, and after the evaluation is complete. It will also cause Construct to provide additional error information if there is an error during parsing the input file.
- `details` - when the modeler uses this value inside the `with` attribute within a `var` tag in conjunction with the `verbose` value, to allow the modeler to see the values of the macro substitutions. It will also cause Construct to provide additional error information if there is an error during parsing the input file.
- `preserve_all_white_space` - will cause Construct's parser to retain all white space (e.g., tabs, linefeeds, carriage returns, spaces) when evaluating the expression.
- `preserve_spaces_only` - will cause Construct's parser to retain all spaces.
- `delay_interpolation` - will cause Construct to not evaluate the value of the variable during its declaration and definition. Instead, Construct will evaluate the value of the variable each time the simulation deck includes it in a `construct::expressionvar::[variable name]`.

Using Variables

When introducing variables earlier we provide a few examples of uses of variables within a Construct input deck. Below, we will discuss these uses more to provide examples of the ways researchers within CASOS have used variables.

Variables as logical flags. One common use for variables is to create logical flags in the input deck. An example of changing values to the variable time_count variable, which is dependent on `short_experiment`, is below:

```
<var name="short_experiment" value="true"/>
<var name="time_count" value="if(construct::boolvar::short_experiment)
   {
      50
   } else {
      100
   }"/>
```

This is telling Construct to change time_count value to 50 if `short_experiment` is true, otherwise set time_count to 100.

Another example could be to declare a debug variable that allows deck-wide enabling or disabling of verbose output. Putting such a variable near the top of the deck supports making the change quickly and easily.

```
<var name="debug_output" value="true"/>
```

Variables for important or key quantities. Another use is to specify values that control the experiment. Examples of such values could be the number of agents, the number of knowledge facts, the number of beliefs, as well as the size of other node classes. An example of changing such a quantity, as a function of whether debug is enabled, is:

```
<var name="debug" value="true"/>
<var name=”agent_count”
value=”if (construct::boolvar::debug) {15} else {150}” />
<var name="num_groups" value="4"/>
```

Variables for defining bounds. Another example could be setting up the start and end values for agents in adjacent groups, when creating groups of agents is important to the modeler's experimental design.

```
<var name="group_size" value="15"/>
<var name="group0_start" value="0"/>
<var name=”group0_end”
value=”construct::intval::group0_start + group_size - 1” />
<var name="group1_start" value=”construct::intval::group0_end + 1 />
<var name=”group1_end”
value=”construct::intval::group1_start + group_size - 1” />
```

Redefinitions of key values for logical clarity. A modeler may think about the average degree, or the average number of connections, per agent. The various network generators require an average density as a parameter. Both measures are related, so using a variable and a bit of math, allows the modeler to use their concepts while meeting the input expectations of Construct.

Common Gotchas

In no order are lessons from the authors, both as modelers and as developers.

Construct's parser will silently ignore any XML tags within the `<construct_vars>` `</construct_vars>` pair that are not `<var>` tags.

Using an editor that can check for well-formed XML will generally save a modeler significant amounts of time in avoiding Construct parser errors. Use of scripting support throws most such editors for a loop, so we are still looking for viable ways others have used to help reduce badly formed-XML errors.

> *ConstructML requires both the `name` and `value` attributes of a `var` tag to be non-empty strings. Empty strings (e.g., “”) will cause Construct's parser to fail.*

Networks within Construct represent connections between nodes of the various node classes (e.g., agents, tasks, knowledge, time). Most networks have names that include spaces (e.g., "interaction sphere network"). If a modeler needs to store the name of a network in a `stringvar`, the authors strongly recommend using the `with="preserve_spaces_only"` attribute when declaring the variable.

Parameters

Parameters are global values that control how construct operates and are used to modify the experiment. All parameters should be set within the parameters tag of the input deck, and using the following syntax:

```
<construct_parameters>
   <param name="[name]" value="[value]">
</construct_parameters>
```

Parameter names are limited to those predefined by Construct, like the parameters listed below, and the values for parameters must be valid depending on the type of parameter, otherwise Construct will produce errors. The following are common parameters used in Construct simulations.

Seed

Seed is a parameter used to control the random seed for the simulation. For a time dependent seed, set this parameter value to 0, otherwise set it to an integer value to get a fixed sequence of random values if the experiment is to be run multiple times.

```
<param name="seed" value="1"/>
```

> *The seed parameter must be the first parameter to ensure its loading and use!*

Verbose Initialization

Verbose initialization is used to determine values of every construct variable and every value when defining nodes and networks. It is recommended to enable this parameter as true to aid in debugging a simulation.

```
<param name="verbose_initialization" value="true"/>
```

Nodes

Nodes are the entities that Construct simulates. Nodes are grouped into groups of like nodes, called node classes, and are related to each other using networks. This section describes some of the nodes and node classes in Construct: specifically, the nodes and node classes in the demo input deck.

The Construct simulation system uses the idea of "nodes" and "networks", as opposed to the more common formulation of "agents" in the agent-based modeling community. This is because Construct grew out of the social and dynamic network analysis tradition (Carley, 1991; Carley & Reminga, 2004) and PCANS framework (Krackhardt & Carley 1998). Groups of similar nodes are grouped by node classes. Thus, all agent nodes are in the agent node class. Classes of nodes can be associated with other classes of nodes to create networks. Links in these networks are then manipulated when Construct is running. New links in the network can be added or modified: for instance, if the agent learns knowledge, a new link between the specific agent node and the relevant knowledge node can be created. Thus, as a Construct simulation runs, the relationship among different nodes will be modified.

Node classes specify the node's behavior in the simulation. For instance, agent nodes are the nodes that interact and learn. While all agent nodes are alike in the sense that they are in the same node class, each agent node can be associated with (have links to) different knowledge or have different influentialness values. Agents in Construct are just one class of node. Another example node class is the knowledge nodeclass. As with the agent node class, different nodes in the knowledge node class are alike in the sense that they represent knowledge from the simulation's perspective but are different in the way that they represent different knowledge bits. Other node classes include timeperiods, groups, and other entities.

The general XML code segment for declaring a node class in Construct is shown below. Each nodeclass has a `name` element associated with it. This both gives the nodeset its identifier as well as a root for the names of nodes where a name is not explicitly given (e.g., agent_1, agent_2, agent_43). There are two methods to create nodes, individually or with a generator. In either case, an individual node or generator may have required node attributes. Each individual node may have unique values for attributes, however all nodes created using a single generator gain all same attributes from that generator.

Below is an example for creating a nodeset.

```
<nodeclass name=”agent”>
   <generator type=”constant”>
      <count value=”20”/>
      <attribute name= “learning_rate” value= “0.5"/>
      <attribute name= “can_send_knowledge” value= “true”/>
      <attribute name= “can_receive_knowledge” value= “true”/>
   </generator>
   <node name= “Sam>
      <attribute name= “learning_rate” value= “0.75"/>
      <attribute name= “can_send_knowledge” value= “false”/>
      <attribute name= “can_receive_knowledge” value= “true”/>
   </node>
</nodeclass>
```

In this example, twenty-one agents are created with the agent at index 20 being named "Sam". The first twenty agents can send and receive knowledge, but have a low learning rate

compared to Sam. In this way, many nodes can be created without having to individually specify each node's attributes.

Agent Node Class

The agent node class represents the actors in the simulation. Agents interact with each other, exchange messages that contain information, and make decisions based on interaction. This nodeclass is universal among all of Construct's standard models and used in many models.

The second method will read the agents in from a DyNetML file. All the property tags are necessary for proper use and parsing of the input file. In the example shown, the input file is in the same working directory as the construct.exe executable, and there is no path information preceding the file name.

```
<nodeclass type="agent" id="agent">
   <generator type="dynetml">
      <attribute name="file_name" value="path_to_file.xml"/>
   </generator>
</nodeclass>
```

> *If the `generator_doc_path` is absolute and not relative, quote the entire path with single quotes (') to force Construct to treat the value as a long string and prevent the Construct lexer from failing with less than transparent errors. The colon and back slashes in Windows paths will simply cause the lexer to believe it is parsing a Construct script instead of a long string. In Unix environments, the slash will equally cause the lexer to believe it is processing a math operator.*

Agentgroup Node Class

The agentgroup nodeclass represents a collection of agents.

CommunicationMedium Node Class

Just like light or sound, communication requires a medium, and different mediums have different properties which effect the entity that moves through it.

```
<nodeclass name="CommunicationMedium">
   <node name= "face_to_face">
      <attribute name="maxMsgComplexity" value="10"/>
      <attribute name="maximumPercentLearnable" value="1"/>
      <attribute name="time_to_send" value="0"/>
   </node>
   <node name= "email">
      <attribute name="maxMsgComplexity" value="2"/>
      <attribute name="maximumPercentLearnable" value="1"/>
      <attribute name="time_to_send" value="1"/>
   </node>
```

```
</nodeclass>
```

Here we have an example of two mediums. In face-to-face interactions, information transfer typically takes little time and conversations are usually very rich with information beyond written words. The attributes above quantify these characteristics.

1. "maxMsgComplexity" gives an upper limit on the information content of a message. In practice, this means that when the number of items attached to a message is larger than the "maxMsgComplexity", items are removed randomly until the number returns to the upper bound. Additionally, adding an item after a message has been created will cause the message to randomly remove an existing item to make room for the new item.
2. "maximumPercentLearnable" sets an upper bound on the link strength between an agent and knowledge node in the knowledge network. The stronger that link strength, the more difficult it is to be broken in models like the Forget model.
3. "time_to_send" dictates how many time periods a message must wait before being delivered.

This class can also be read in from a DyNetML file, presuming the modeler has the appropriate attributes just discussed. An example of a code snippet that would read the CommunicationMedium in is below.

```
<nodeclass name="CommunicationMedium">
   <generator type="dynetml">
      <attribute name="file_name" value="path_to_file.xml"/>
   </generator>
</nodeclass>
```

Dummy Node Class

The dummy node class is designed to act as a placeholder node class to create column vectors for other node classes. One of the principal ways Construct works is by manipulating two dimensional networks, and with the dummy node class, one can create an agent by dummy node class network that acts as a one-dimensional network. This essentially makes visualization of networks much easier, as well as data manipulation.

```
<nodeclass name="dummy_nodeclass">
   <node name="dummy_node"/>
</nodeclass>
```

Most networks used by Construct avoids this implementation in favor of using node attributes. As an example, one could create a Boolean network connecting agents to the dummy node class to represent whether that agent can send communication. Most models instead attach that information to a node as a node attribute.

Knowledge Node Class

The knowledge node class represents knowledge that can be exchanged between agents. Each knowledge bit is represented by one node. In this example, ten knowledge nodes are created.

```
<nodeclass name="knowledge">
   <generator type=”constant”>
      <count value="10"/>
   </generator>
</nodeclass>
```

Time Period Node Class

Nodes in the time period node class represent one simulated time period in the simulation. The length of the simulation is represented by the number of nodes in this node class. While all of Construct’s standard models require other nodesets such as agent or knowledge, Construct inherently requires this node class as the only hard requirement in the Construct architecture.

```
<nodeclass name="timeperiod">
   <generator type=”constant”>
      <count value="10"/>
   </generator>
</nodeclass>
```

Other Node Classes

While the node classes listed above are standard node classes in construct models, Construct is not limited to these node classes. In the future, new models may be available that require new nodeclasses.

Networks

Networks are the main data structures in Construct. Since construct is a network-based simulation, most of the data that goes into input for simulation are in the form of network. Networks are the relationships between node classes listed in the section above. The algorithms in Construct reference these networks to perform tasks. For example, the agent by knowledge, knowledge network represents which knowledge is known by which agents.

Table 3 shows specific networks with their relationship to node class as well as a brief description.

Table 3. Network relations to node classes.

Network Name	Source and Target Node Classes	Function or Purpose in Demo Input Deck
access	agent x agent	which agents have access to each other (supplement to interaction sphere)

Network Name	Source and Target Node Classes	Function or Purpose in Demo Input Deck
agent active timeperiod	agent x timeperiod	which agents are active during which timeperiods
agent group membership	agent x agentgroup	which agents are associated with which agent groups
agent initiation count	agent x timeperiod	number of times agent can seek a partner, actively initiating communication
agent mail usage by medium	agent x CommunicationMedium	how likely it is that an agent will send a via mail message using a given medium
agent reception count	agent x timeperiod	number of times agent can be sought out, passively receiving communication
communication medium access	agent x CommunicationMedium	which agents can send messages on a given medium
communication medium preferences	agent x CommunicationMedium	how likely an agent is to send a message over a medium
communication medium preferences 3d	agent x agent x CommunicationMedium	how likely the ego is to send a message over a medium to a given alter
interaction knowledge weight	agent x knowledge	weight placed on knowledge-based factors when selecting interaction partners
interaction	agent x agent	record of interactions between agents
interaction probability	agent x agent	record of probabilities for agents to pick each other for interactions
interaction sphere	agent x agent	which agents can potentially interact with another
knowledge	agent x knowledge	the knowledge associated with an agent, i.e., what an agent currently knows
knowledge expertise weight	agent x timeperiod	weight placed on interacting with alter agents with knowledge the ego agent lacks
knowledge forgetting prob	agent x knowledge	the probability that a piece of knowledge not being used will incrementally be forgotten
knowledge forgetting rate	agent x knowledge	the rate at which a piece of knowledge is forgotten
knowledge group membership	knowledge x knowledgegroup	what knowledge bits are associated with which knowledge groups
knowledge learning difficulty	agent x knowledge	probability that an agent is unable to learn a piece of knowledge

Network Name	Source and Target Node Classes	Function or Purpose in Demo Input Deck
knowledge message complexity	agent x timeperiod	amount of info an agent can send when communicating
knowledge priority	agent x knowledge	priorities placed on knowledge bits when sending a message
knowledge opinion	agent x knowledge	an agent's opinion on whether a piece of knowledge is true
knowledge similarity weight	agent x timeperiod	weight placed on shared knowledge when choosing interaction partner
learnable knowledge	agent x knowledge	which knowledge an agent can learn
medium knowledge	CommunicationMedium x knowledge	determines which knowledge can be sent over a CommunicationMedium
physical proximity	agent x agent	how close each pair of agents are physically
physical proximity weight	agent x timeperiod	weight placed on physical proximity when choosing interaction partner
public propensity	agent x timeperiod	how likely an agent is to make a message public
social proximity	agent x agent	how close each pair of agents are socially
social proximity weight	agent x timeperiod	weight placed on social proximity when choosing interaction partner
sociodemographic proximity	agent x agent	how close each pair of agents are socio-demographically
sociodemographic proximity weight	agent x timeperiod	weight placed on sociodemographic proximity when choosing interaction partner
subscription	agent x agent	which ego agents will receive a forwarded message from an alter agent if that message is made public
subscription probability	agent x agent	how likely an ego agent is to subscribe to an alter agent message when the alter makes their message public

It is quite possible for a user to create more networks than there are listed in Table 3. Networks are specified within the <networks> ConstructML tag.

```
<networks>
   <!-- Put all networks here -->
</networks>
```

Each networks tag must have five attributes: network id, source nodeclass, target nodeclass, link type, and network type.

```
<network src_nodeclass_type="[nodeclass]"
   target_nodeclass_type="[nodeclass]"
   id="[name]" link_type="[type]" network_type="dense">
   <!-- Set links and generators here -->
</network>
```

Network ID is the name that refers to a given network. Network ID's are unique, and attempting to include multiple networks will result in warnings that degenerate networks are being dumped. The source node class type and target node class type indicate the node classes that are related by the network. Relationships between networks are weighted and unweighted relations between node classes. Network type specifies the storage mechanism used to represent the network. The link type defines the type of relation stored in the network. There are Boolean link types, integer link types, floating number link types, and string link types. The link type must be the same for all links in the network i.e., it is not possible to have a Boolean relationship in integer networks. Links can be specified either through the `<link>` tag or the `<generator>` tag.

The syntax for link generation is listed below:

```
<link src_node_name="[id]" target_node_name="[id]" value="[value]"/>
```

The following sections are more detailed descriptions of the networks listed above in Table 3. The ID Construct uses to identify the network is listed in quotations in each network description as well as in the example for each network.

Access Network

The "`access network`" is an addition to the interaction sphere and can restrict which alters agents the ego agent can communicate with. If the entry for a row agent and column agent is zero then that row agent can not initiate interactions with the column agent, but the column agent can initiate interactions with the row agent if the symmetric entry is nonzero. A use case for this network could include the temporary restriction of communication by an agent by making its row elements all zero or false. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="access network" link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::agent::count-1"/>
      <param name="constant_value" value="true"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Agent Active Timeperiod Network

The "agent active time period network" determines which agents are active during each time period. Active agents during a time period can interact and exchange messages. This is an optional network in the Standard Interaction model and Twitter Interaction model.

```
<network src_nodeclass_type="agent" target_node class_type="timeperiod"
id="agent active timeperiod network" link_type="bool"
network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
      last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value"
         value="construct::boolvar::bridging_agents_active"/>
   </generator>
</network>
```

Agent Group Membership Network

The "agent group membership network" is used to identify related sets of agents.

```
<network src_nodeclass_type="agent" target_node class_type="agentgroup"
id="agent group membership network" link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
         last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="0"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
      <cols first="1" last="1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
      <cols first="2" last="2"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

Agent Initiation Count Network

The "agent initiation count network" specifies the number of times each agent can initiate an interaction with another agent each time period. If an agent cannot find another agent to initiate on, the agent is forced to interact with self. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
id="agent initiation count network" link_type="int" network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
         last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

Agent Mail Usage by Medium Network

The "agent mail usage by medium network" indicates the probability that an agent will use a mailing system for a particular medium. This is an optional network for the Mail model.

```
<network src_nodeclass_type="agent"
target_nodeclass_type="CommunicationMedium" id="agent mail usage by medium
network" link_type="float" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0"
last="nodeclass::CommunicationMedium::count_minus_one "/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Agent Reception Count Network

The "agent reception count network" specifies the number of times each agent can be initiated on by another agent for interactions each time period. The number of times an agent

is initiated on does not need to be equal to its reception count, and can in fact, be lower under the correct conditions. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
id="agent reception count network" link_type="int" network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
         last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
   <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

Communication Medium Access Network

The "`communication medium access network`" determines which agents have access to which communication mediums. Set the value to zero (0) for the agent to prevent access to the specified medium.

```
<network src_nodeclass_type="agent"
target_nodeclass_type="CommunicationMedium" id="communication medium
access network" link_type="float" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0"
         last="nodeclass::CommunicationMedium::count_minus_one"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Communication Medium Preferences Network

The "`communication medium preference network`" specifies an agent's preferences for mediums regardless of who they are interacting with, i.e., if someone prefers, across the board, to use face-to-face rather than email, you would indicate that preference with this network. Link values correspond to probability weights for an agent to use the corresponding medium when constructing a message. A 3D version of this graph exists which is mutually exclusive to this graph. While that graph gives greater control, it also introduces additional complexity.

```
<network src_nodeclass_type="agent"
target_nodeclass_type="CommunicationMedium" id="communication medium
preferences network" link_type="float" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0"
last="nodeclass::CommunicationMedium::count_minus_one"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>

</network>
```

Communication Medium Preferences Network 3d

The "communication medium preferences network 3d" is a network represented by a matrix with three dimensions. In this case, each ego agent has a two dimensional network that represents their preference for sending messages on a specific medium to a specific alter agent.

```
<network src_nodeclass_type="agent" inner_nodeclass_type="agent"
target_nodeclass_type="CommunicationMedium" id="communication medium
preferences network 3d" link_type="float" network_type="dense3d">
   <generator type="constant3d">
      <rows first="1" last="nodeclass::agent::count_minus_one"/>
      <inners first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0"
         last="nodeclass::CommunicationMedium::count_minus_one"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Interaction Knowledge Weight Network

The "`interaction knowledge weight network`" specifies how much weight agents will put on particular knowledge bits when computing knowledge similarity and expertise. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
id="interaction knowledge weight network" link_type="float"
network_type="dense">
   <generator type="randomuniform">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

Interaction Network

The "`interaction network`" is used by the Standard Interaction model; however, this graph is reset at the beginning of each time period. This entity is a network so that it may easily

be output. One does not need to include the interaction network in the input deck in order for it to be used as output.

Interaction Probability Network

Like the above interaction network, the "`interaction probability network`" is used by the Standard Interaction model and is reset each time period. Like the interaction network, input for this network is not used and this network is not required in the input deck. It can however be output for analysis.

Interaction Sphere Network

The "`interaction sphere network`" is the starting point for the modeler to enumerate which agents have an interaction link to other agents.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="interaction sphere network" link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0" last="nodeclass::agent::count_minus_one"/>
      <param name="constant_value" value="1.0"/>
      <param name="symmetric_flag" value="true"/>
   </generator>
</network>
```

Knowledge Expertise Weight Network

The "`knowledge expertise network`" specifies how much weight is placed on knowledge expertise when calculating probability weights. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
id="knowledge expertise weight network" link_type="float"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1.0"/>
      <param name="symmetric_flag" value="true"/>
   </generator>
</network>
```

Knowledge Forgetting Prob Network

The "`knowledge forgetting prob network`" specifies the likelihood that an agent forgets knowledge that they learned. When previous conditions are met, the relevant link corresponds to the probability that an agent will forget some or all of a piece of knowledge. This is an optional network in the Forgetting model.

```
<network src_nodeclass_type="agent" target_node class_type="knowledge"
id="knowledge forgetting prob network" link_type="float"
network_type="dense">
```

```
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="constant_value" value="0.5"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Knowledge Forgetting Rate Network

The "knowledge forgetting rate network" specifies how quickly agents forget knowledge that they learned. When all necessary conditions are met, the relevant link weight is used to decrease the corresponding link in the "knowledge network". This is an optional network in the Forgetting model.

```
<network src_nodeclass_type="agent" target_node class_type="knowledge"
id="knowledge forgetting rate network" link_type="float"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="constant_value" value="0.25"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Knowledge Group Membership Network

The "knowledge group membership network" is used to identify related sets of knowledge bits.

```
<network src_nodeclass_type="knowledge" target_node
class_type="knowledgegroup" id="knowledge group membership network"
link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::knowledgegroup_K1_start"
         last="construct::intvar::knowledgegroup_K1_end"/>
      <cols first="0" last="0"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::knowledgegroup_K2_start"
         last="construct::intvar::knowledgegroup_K2_end"/>
      <cols first="1" last="1"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

Knowledge Learning Difficulty Network

The "knowledge learning difficulty network" gives the probability that a receiving agent cannot learn a piece of knowledge. This is a required network for the Knowledge Learning Difficulty model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
id="knowledge learning difficulty network" link_type="float"
network_type="dense">
   <generator type="randomuniform">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

Knowledge Message Complexity Network

The "knowledge message complexity network" specifies how many items that communicate knowledge an agent will attempt to add to a message each time period. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
id="knowledge message complexity network" link_type="int"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

Knowledge Network

The "knowledge network" specifies which agents have what knowledge. Knowledge is ubiquitous in all of Construct's library of interaction models. This network is a required network for various models.

```
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
id="knowledge network" link_type="float" network_type="dense">
   <generator type="randombinary">
      <rows first="construct::intvar::agentgroup_A_start"
         last="construct::intvar::agentgroup_A_end"/>
      <cols first="construct::intvar::knowledgegroup_K1_start"
         last="construct::intvar::knowledgegroup_K1_end"/>
      <param name="mean" value="0.20"/>
   </generator>
   <generator type="randombinary">
      <rows first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
      <cols first="construct::intvar::knowledgegroup_K2_start"
         last="construct::intvar::knowledgegroup_K2_end"/>
      <param name="mean" value="0.20"/>
```

```
   </generator>
   <generator type="randombinary">
      <rows first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
      <cols first="construct::intvar::knowledgegroup_K1_start"
         last="construct::intvar::knowledgegroup_K1_end"/>
      <param name="mean" value="0.10"/>
   </generator>
   <generator type="randombinary">
      <rows first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
      <cols first="construct::intvar::knowledgegroup_K2_start"
         last="construct::intvar::knowledgegroup_K2_end"/>
      <param name="mean" value="0.10"/>
   </generator>
</network>
```

Knowledge Priority Network

The "knowledge priority network" specifies the priority level of a particular piece of knowledge when building a message. In this network, links correspond to probability weights which when normalized creates a cumulative distribution function from which the knowledge selection process draws. This is optional network for the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
id="knowledge priority network" link_type="int" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Knowledge Opinion Network

The "knowledge opinion network" links indicate how strongly an agent believes that a piece of knowledge is true. This is an optional network for the Twitter Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge" id="
knowledge opinion network" link_type="float" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="constant_value" value="0.5"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Knowledge Similarity Weight Network

The "knowledge similarity weight network" specifies how much weight agents place on shared knowledge when calculating probabilities of interaction. It is measured by

comparing an agent's knowledge against its perception of another agent's knowledge. Knowledge similarity is increased when the ego knows a knowledge bit and perceives that an alter also knows the same knowledge bit. The increase will be equal to the agent's knowledge of the bit.

```
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
id="knowledge similarity weight network" link_type="float"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Learnable Knowledge Network

The "`learnable knowledge network`" links indicate whether an agent can learn a piece of knowledge. An experimenter may want to restrict which groups of agents are capable of learning knowledge bits and can do so using this network. This is an optional network for the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
id="learnable knowledge network" link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Medium Knowledge Network

The "`medium knowledge network`" limits what knowledge can be sent on a given medium. This is an optional network for the Standard Interaction model.

```
<network src_nodeclass_type="CommunicationMedium"
target_nodeclass_type="knowledge" id="medium knowledge network"
link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::CommunicationMedium::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Physical Proximity Network

The "`physical proximity network`" specifies how close two agents are to each other physically. Physical proximity is one of three factors that determines the proximity of two

agents. A higher proximity indicates an increased probability to interact. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="physical proximity network" link_type="float" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::agent::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Physical Proximity Weight Network

The "`physical proximity weight network`" specifies how strongly agents will value physical proximity when determining overall proximity with another agent. Setting a value to zero will cause an agent to determine overall proximity independent of physical proximity. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
id="physical proximity weight network" link_type="float"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Public Propensity Network

The "`public propensity network`" indicates the probability that an agent will make their message public. When a message is made public, other agents have the opportunity to subscribe to the sending agent, and that message is then forwarded to any current subscribers. This is an optional network for the Subscription model.

```
<network src_nodeclass_type="agent"
target_nodeclass_type="CommunicationMedium" id="public propensity network"
link_type="float" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <cols first="0"
last="nodeclass::CommunicationMedium::count_minus_one "/>
      <param name="constant_value" value="0.1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Social Proximity Network

The "social proximity network" specifies how close two agents are to each other socially. Social proximity is one of three factors that determines the proximity of two agents. A higher proximity indicates an increased probability to interact. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="social proximity network" link_type="float" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::agent::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Social Proximity Weight Network

The "social proximity weight network" specifies how strongly agents will value social proximity when determining overall proximity with another agent. Setting a value to zero will cause an agent to determine overall proximity independent of social proximity. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
id="social proximity weight network" link_type="float"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Sociodemographic Proximity Network

The "sociodemographic proximity network" specifies how close two agents are to each other demographically. Sociodemographic proximity is one of three factors that determines the proximity of two agents. A higher proximity indicates an increased probability to interact. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="sociodemographic proximity network" link_type="float"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::agent::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Sociodemographic Proximity Weight Network

The "sociodemographic proximity weight network" specifies how strongly agents will value sociodemographic proximity when determining overall proximity with another agent. Setting a value to zero will cause an agent to determine overall proximity independent of sociodemographic proximity. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
id="sociodemographic proximity weight network" link_type="float"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Subscription Network

The "subscription network" indicates whether each row agent is subscribed to a column agent. When a message is made public, an agent may subscribe to the sender and all agents subscribed to the sender are forwarded a copy of the message. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="subscription network" link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::agent::count-1"/>
      <param name="constant_value" value="false"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Subscription Probability Network

The "subscription probability network" indicates the probability each row agent will subscribe to a column agent when the column agent makes a message they make public. This is an optional network in the Standard Interaction model.

```
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
id="subscription probability network" link_type="float"
network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::agent::count-1"/>
      <param name="constant_value" value="0.1"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

Network Generators

To use a generator to specify links, use the following syntax:

```
<generator type="[type]">
   <!–
   additional attribute for rows, inners, and cols can be values="",
      where values is a comma separated list
   additional attribute for rows, inners, and cols can be groups="",
      where groups is a comma separated list
   additional attribute for rows, inners, and cols
      group_membership_network="", where group_membership_network is
      a string that names the group membership network for this
      row/col/inner
   -->
   <rows first="[firstrow]" last="[lastrow]"/>
   <cols first="[firstcol]" last="[lastcol]"/>
   <param name="[name1]" value="[value1]"/>
   <param name="[name2]" value="[value2]"/>
   ...
</generator>
```

Table 4 summarizes each of the network generators available to an experiment designer. Below Table 4 is a detailed example and description of how to use and invoke each generator.

Table 4. Types of network generators available.

Type	What does it do?
constant	Used to create a 2D matrix with a constant value in all cells of the network.
constant3d	Used to create a 3D matrix with a constant value in all cells of the matrix.
csv	Import from a comma separated value file. If the `load_style` parameter is present, Construct only understands the value `sparse_to_dense_convert` and will treat any other values as if it needed to load a dense network. With this parameter, construct will load a sparse network, where only the cells with values are represented in the csv file.
csv_binarize	Import from a comma separated value file, but dichotomize the values imported based on the specified cut-off. All values less than or equal to the cut off are converted to zero (0) and all values greater than cut off are converted to one (1).
csv3d	Import a 3D network for use initializing Transactive Memory from a comma separated value file.
do_nothing	Useful for debugging an input file. To disable a generator, one must comment it out, delete it or change its type to do_nothing. For large generators with many lines of XML it can be more convenient to change the type to do_nothing than finding the beginning and end of the XML code to comment it out.
gen_from_text	Retrieve generator parameters from text file instead of from within XML of the input file.
group_to_group	Store generator parameters in one network and use them in generating a different network.
erdos_renyi	Generate network with values generated using the Erdos Renyi distribution model.
multi_dimensional_preprocess_based	Use a collection of networks to create a probability distribution function for agents being associated with any value found in the networks. From this, values are chosen for each agent.
periodic	Set cells to a constant value based on given period.
preprocessor_based	Use a given network to find probabilities that an agent will have one of the values found in the network, then use these probabilities to generate cell values.

randombinary	Given a mean, generate a sequence of 0,1 with the approximate number of 1's defined by the mean.
randomuniform	Given a max and min value, generate a sequence of values between the two in a uniform distribution
randomvalue	Given a list of possible values and their weights, one is chosen for each cell.
sociodemographic_similarity	Generate a network where links are created if the sociodemographic similarity between the source and target node is within a minimum and maximum bound.
tied	Set values equal to the corresponding values in a different part of the network.
xml_generator_loader	Load a generator from a separate XML file.

Constant Network Generator

Used to create a 2D matrix with a constant value in all cells of the network.

```
<generator type="constant">
   <!-- additional attribute for rows, inners, and cols can be values="",
where values is a comma separated list
    additional attribute for rows, inners, and cols can be groups="",
where groups is a comma separated list
    additional attribute for rows, inners, and cols
group_membership_network="", where group_membership_network is a string
that names the group membership network for this row/col/inner
   -->
   <rows first="0" last="nodeclass::agent::count_minus_one"/>
   <cols first="0"
last="nodeclass::CommunicationMedium::count_minus_one"/>
   <param name="constant_value" value="1.0"/>
   <param name="symmetric_flag" value="false"/>
</generator>
```

Constant3D Network Generator

Used to create a 3D matrix with a constant value in all cells of the matrix.

```
<generator type="constant3d">
   <!-- additional attribute for rows, inners, and cols can be values="",
where values is a comma separated list
    additional attribute for rows, inners, and cols can be groups="",
where groups is a comma separated list
    additional attribute for rows, inners, and cols
group_membership_network="", where group_membership_network is a string
that names the group membership network for this row/col/inner
   -->
   <rows first="1" last="nodeclass::agent::count_minus_one"/>
   <inners first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0"
last="nodeclass::CommunicationMedium::count_minus_one"/>
   <param name="constant_value" value="1.0"/>
   <param name="symmetric_flag" value="false"/>
</generator>
```

CSV Network Generator

Import from a comma separated value (CSV) file. If the `load_style` parameter is used, Construct only understands the value `sparse_to_dense_convert` and will treat any other values

as if it needed to load a dense network. With this parameter, Construct will load a sparse network, where only the cells with values are represented in the csv file.

```
<generator type="csv">
   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::timeperiod::count_minus_one" />
   <param name="filesystem_path"
      value="agent_initiation_count_network_fname" />
   <param name="csvrow" value="construct::stringvar::agent_list" />
   <param name="csvcol" value="construct::stringvar::timeperiod_list" />

   <!-- optional parameters that default to false -->
   <param name="load_style" value="sparse_to_dense_convert" />
   <param name="skip_first_row" value="true" />
   <param name="subtract_one_from_indices" value="true" /> -->
</generator>
```

CSV_binarize Network Generator

Import from a comma separated value (CSV) file, but dichotomize the values imported based on the specified cut-off. All values less than or equal to the cut off are converted to zero (0) and all values greater than cut off are converted to one (1).

```
<generator type="csv_binarize">
   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::agent::count_minus_one" />
   <param name="filesystem_path"
      value="interaction_sphere_network_fname" />
   <param name="csvrow" value="construct::stringvar::agent_list" />
   <param name="csvcol" value="construct::stringvar::agent_list" />
   <param name="symmetric" value="true" />
   <param name="binarization_threshold" value="0.0" />

   <!-- optional parameters that default to false -->
   <param name="skip_first_row" value="true" />
   <param name="load_style" value="sparse_to_dense_convert" />
   <param name="subtract_one_from_indices" value="true" />
</generator>
```

Csv3d Network Generator

Import a 3D network for use initializing transactive memory from a comma separated value (CSV) file.

```
<generator type="dynetml">
   <param name="filesystem_path" value="my_file.xml" />
   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <inner first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::knowledge::count_minus_one" />
</generator>
```

Gen_from_text Network Generator

Retrieve generator parameters from text file instead of from within XML of the input file.

```
<generator type="gen_from_text">
```

```
   <param name="filename" value="my_file.txt" />

   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::knowledge::count_minus_one" />
</generator>
```

Group_to_group Network Generator

Store generator parameters in one network and use them in generating a different network.

```
<generator type="group_to_group">
   <param name="row_grp_membership_net" value="agent membership network"/>
   <param name="col_grp_membership_net" value="knowledge membership
network"/>

   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::knowledge::count_minus_one" />
</generator>
```

Erdos_renyi Network Generator

Generate network with values generated using the Erdos Renyi distribution model.

```
<generator type="erdos_renyi">
   <param name="density" value="0.3" />
   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::knowledge::count_minus_one" />
</generator>
```

Multi_dimensional_preprocess_based Network Generator

Use a collection of networks to create a probability distribution function for agents being associated with any value found in the networks. From this, values are chosen for each agent.

```
<generator type=”multi_dimensional_preprocess_based">
   <param name="mechanism_names" value=”age,race,gender,occupation" />
   <param name="filesystem_path" value=”my_probability_file.csv" />

   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::knowledge::count_minus_one" />
</generator>
```

Periodic Network Generator

Set cells to a constant value based on given period.

```
<generator type=”periodic">
   <param name="constant_value" value=”10" />
   <param name="period" value=”2" />

   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::knowledge::count_minus_one" />
</generator>
```

Perception_based Network Generator

Generate knowledge transactive memory based on an agent's sphere of interaction, false positive rate, and false negative rate.

```
<generator type="perception_based">
   <param name="false_negative_rate" value="0.1" />
   <param name="false_positive_rate" value="0.1" />
   <param name="rounding_threshold" value="0.1" />
   <param name="perceived_network" value="knowledge network" />

   <ego first="0" last="nodeclass::agent::count_minus_one" />
   <alter first="0" last="nodeclass::agent::count_minus_one" />
   <transactive first="0" last="nodeclass::knowledge::count_minus_one" />
</generator>
```

Preprocessor_based Network Generator

Use a given network to find probabilities that an agent will have one of the values found in the network, then use these probabilities to generate call values.

```
<generator type="preprocessor_based">
   <param name="mechanism_names" value="age" />
   <param name="filesystem_path" value="my_probability_file.csv" />

   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::knowledge::count_minus_one" />
</generator>
```

Randombinary Network Generator

Given a mean, generate a sequence of 0,1 with the approximate number of 1's defined by the mean.

```
<generator type="randombinary">
<rows groups="agent_grp1,agent_grp2" group_membership_network="'agent
group membership network'"/>
<cols groups="agent_grp3" group_membership_network="'agent group
membership network'"/>
   <param name="mean" value="0.7" />
</generator>
```

Randomuniform Network Generator

The randomuniform network generator uses a random uniform distribution with "min" and "max" as the lower and upper bounds of the distribution.

```
<generator type="randomnuniform">
   <rows first="0" last="nodeclass::agent::count_minus_one" />
   <cols first="0" last="nodeclass::timeperiod::count_minus_one" />
   <param name="min" value="0"/>
   <param name="max" value="1"/>
</generator>
```

Randomvalue Network Generator

Given a list of possible values and their weights, one is chosen for each cell.

```
<generator type="randomvalue">
```

```
    <param name="values" value="1,2,3,4,5" />
    <param name="weights" value="0.1,0.2,0.4,0.5 " />

    <rows first="0" last="nodeclass::agent::count_minus_one" />
    <cols first="0" last="nodeclass::timeperiod::count_minus_one" />
</generator>
```

Sociodemographic Similarity Network Generator

Generate a network where links are created if the sociodemographic similarity between the source and target node is within a minimum and maximum bound.

```
<generator type="sociodemographic_similarity ">
    <param name="minsimilarity" value="0.1”/>
    <param name="maxsimilarity " value="0.6”/>

    <rows first="0" last="nodeclass::agent::count_minus_one" />
    <cols first="0" last="nodeclass::timeperiod::count_minus_one"/>
</generator>
```

Tied Network Generator

Set values equal to the corresponding values in a different part of the network.

```
<generator type="tied">
    <param name="tiedrow" value="1”/>
    <param name="tiedcol" value="6”/>

    <rows first="0" last="nodeclass::agent::count_minus_one" />
    <cols first="0" last="nodeclass::timeperiod::count_minus_one"/>
</generator>
```

Xml_generator_loader Network Generator

Load a generator from a separate XML file.

```
<generator type="xmlloader">
    <param name="fileystem_path" value="my_xml_file.xml”/>

    <rows first="0" last="nodeclass::agent::count_minus_one" />
    <cols first="0" last="nodeclass::timeperiod::count_minus_one"/>
</generator>
```

Interaction Models

Interaction Models are the backbone of Construct models as other types of models rely on modifying the messages created by interaction models. Many of the interaction models utilize both static and dynamic networks to decide how individuals choose interaction partners and what information to transmit. Currently three interaction models are available for use, the Standard Interaction Model, the Knowledge Transactive Memory Interaction Model, and the Twitter Interaction Model. Only the Standard Interaction Model and Knowledge Transactive Memory Interaction Model are mutually exclusive. The ID for each model is listed in quotations at the beginning of the description followed by an example of its use and each model can only be called once.

Standard Interaction Model

The "`Standard Interaction Model`" is the most basic model in Construct. Below is an example for the input XML deck. Note this model has no model specific parameters.

```
<model name="Standard Interaction Model"/>
```

The model assigns interaction pairs based on perfectly known homophily, expertise, proximity, and interaction access. Agents simultaneously seek other agents that have similar knowledge to themselves while also seeking agents that know knowledge that the interaction seeker does not. Interaction pair formation is further impacted by how proximal agents are to another either by physical distance, social status, or demographic state. Finally, certain agents may not be able to interact with another if the agent is unaware that an agent exists or if there is no medium with which communication can occur. First, we will discuss probability weights that determines who an agent attempts to interact with. Then we will discuss how messages are formed and the result of an empty message. Finally, we will discuss message parsing and what happens when the message is received.

The required networks for the Standard Interaction Model are the knowledge network and the interaction sphere network. The interaction sphere it is a sparse matrix that is treated separately in Construct and describes which alter agents an ego agent is aware of for initiating interactions. As models treat any optional network which is not provided by the user as a homogeneous network, a user-specified knowledge network is required because if it were a homogeneous network the simulation becomes trivial. In one case, no agent possess knowledge, and thus cannot spread any information. In the other case, all agents have perfect knowledge, and no spread of information can happen as all agents already possess all knowledge. All other networks mentioned in this section are optional networks. Further information is provided in the Model Dependencies Section.

Proximity between two agents ($PX_{i,j}$) is comprised of three factors, the physical proximity network ($PP_{i,j}$), social proximity network ($SP_{i,j}$), and sociodemographic proximity network ($DP_{i,j}$). Each factor has an associated weight to determine importance in calculating proximity, physical proximity weight network ($PPW_i(t)$), social proximity weight network ($SPW_i(t)$), and sociodemographic proximity weight network ($DPW_i(t)$). These factors are based on the agent attempting to initiate an interaction and can vary over time. Overal proximity is then,

$$PX_{i,j} = PPW_i(t)PP_{i,j} + SPW_i(t)SP_{i,j} + DPW_i(t)DP_{i,j}.$$

The other two factors for determining probability weights are knowledge similarity ($KS_{i,j}$) and knowledge expertise ($KE_{i,j}$). These two factors are sums over the same quantities just with different indices to sum over. Using K_i^* as the set of knowledge that agent i knows we calculate knowledge similarity and expertise as,

$$KS_{i,j} = \sum_{k \in K_i^* \cap K_j^*} KW_{i,k}\,, \qquad KE_{i,j} = \sum_{k \in K_i^* \cap K_j^*} KW_{i,k},$$

where $KW_{i,k}$ is the knowledge weight network that the initiator agent places on being connected to a knowledge node. These two factors are weighted by the knowledge similarity weight network ($KSW_i(t)$), and the knowledge expertise weight network ($KEW_i(t)$).

Combining these factors get an overall probability weight $(P_{i,j})$ which is stored in the interaction probability network,

$$P_{i,j} = PX_{i,j} + KSW_i(t)KS_{i,j} + KEW_i(t)KE_{i,j}.$$

These are probability weights and not probabilities as additional restrictions are applied. The agent active timeperiod network defines the timeperiods an agent is active and, when not active, the agent cannot interact, and the associated probability weights are set to zero. Additionally, the agent access network applies a hard restriction on interaction. If an (row agent, column agent) element is zero then the row agent cannot initiate interactions with the column agent however, this element does not dictate if the column agent can initiate communication with the row agent. This is true only if the symmetrical element is also non-zero. The interaction sphere in this model operates almost identically to the access network except, the interaction sphere employs a sparse representation which improves performance. Agents also only have so many times they can both initiate and be initiated upon in each timeperiod which set by the agent initiation count network and agent reception count network, respectively. This creates another set of restrictions as interaction pairs are formed that makes a normalization of probability weights futile. Finally using either communication medium access network 3d or communication medium access network, interaction probability is set to zero if a common medium both agents have access to cannot be found.

As mentioned earlier, interaction pair formation is dynamic and as pairs form the interaction probabilities within a set of agents can change. First, an agent with remaining available initiations is chosen randomly with equal probability of selection and then their available initiations is decremented if the pair is formed. Using the probability weights discussed above an agent is chosen with remaining available receptions, which as with initiations, is then decremented if the pair is formed. Agents can self-interact, in which case reception count is not decremented. Except for self-interactions, a given pair can only interact once in a time period regardless of which agent was the initiator. This process continues until no more initiators are available.

Once a potential interaction pair is selected, it is not formally formed until at least one interaction message is created. Interaction messages require a communication medium from the communication medium nodeset. For a potential pair to be formally formed, as discussed previously, both agents must have a common medium. Either communication medium preferences network 3d or communication medium preferences network is used to determine

which medium is used. To avoid potential conflicts, these networks should be non-zero if using non-default values.

From here, pieces of information or items are attempted to be added to a message. A message minimally contains information about the sender, receiver, and communication medium, but additional information, knowledge in this case, is contained in a set of items attached to the message. First, a check is done on the sending agent for the node attribute `"can_send_knowledge"` and on the receiving agent for the node attribute `"can_recieve_knowledge"` in conjunction with the knowledge message complexity network. If the values are `"true"`, `"true"`, and non-zero respectively, knowledge items are added based on the knowledge priority network. Lastly, for each piece of knowledge, the medium knowledge network and learnable knowledge network are checked to make sure that the piece of knowledge is learnable and can be sent over the message medium.

Though knowledge items are added based on their weight in the "knowledge priority network", these items are in randomized order for the creation of the message. When a message is created if the number of items is larger than the medium's `"maxMsgComplexity"` node attribute, items are removed to ensure the maximum message complexity is enforced. While some knowledge may have priority over others, this method is meant to reflect that there may be other types of items in a message and allows the property to be enforced without bias. One may think of this decision in the example of an individual intending to prioritize content in a message, but depending on the medium and receiver, the actual content communicated may be limited in ways unintended by the sender.

Finally, both messages are checked to see if they contain a non-zero number of items. If either message count is non-zero, the interaction pair is formally created. Due to the large amount of complexity that can arise from heterogenous initial conditions, this step is the only step in the model in which progression is not deterministic. If the interaction pair is not created, the process continues with no change to simulation state. Once the interaction pair has been formed its message is added to Construct's central message queue. If the number of times an interaction pair formation fails becomes greater than the number of agents squared, pairing is ended prematurely, and the model's **Think** function completes.

The Standard Interaction Model's **Communicate** function then parses all knowledge related items in all messages in the message queue before the queue is cleared at the end of the time period. The knowledge network is then increased based on the receiver node's attribute "learning_rate"(lr) in the following equation,

$$K_{i,k} = K_{i,k} + lr\left(1 - K_{i,k}\right).$$

This keeps the knowledge value in the range $[0,1]$ for learning rates in the same range. While this particular method is not critical to the overall model, it does allow other modification models to employ a leak of knowledge which will be elaborated upon in the relevant models.

Knowledge Transactive Memory Interaction Model

The "Knowledge Transactive Memory Interaction Model" model expands the Standard Interaction Model by adding a transactive memory (Wegner, 1987) for knowledge for each agent. Knowledge Transactive Memory (KTM) is a data storage for each ego agent on what knowledge alter agents in the ego agent's sphere knows. This storage is incomplete as it does not directly use the knowledge network. Instead, the storage relies on recording interactions to populate what agents know about each other. This model builds upon the Standard Interaction Model and adopts many of the functions used therein. Because of this, the Knowledge Transactive Memory Interaction Model is mutually exclusive with the Standard Interaction Model. All required networks in the Standard Interaction model are also required for the Knowledge Transactive Memory Interaction Model and similarly for optional networks. An example for calling this model can be seen below.

```
<models>
   <model name="Knowledge Transactive Memory Interaction Model">
      <param name="false_positive_rate" value="0.3"/>
      <param name="false_negative_rate" value="0.2"/>
      <param name="threshold" value="0.1"/>
   </model>
</models>
```

Implementation for loading an arbitrary sparse graph such as the knowledge transactive memory is not yet complete. Instead, to seed the transactive memory three model parameters are used, "false_positive_rate", "false_negative_rate", and "threshold". The threshold simply dictates what percentage of entries to seed. The false negative rate is the fraction of knowledge that an alter agent in the ego agent's memory does not know that the alter agent actually knows. Likewise, the false positive rate is the fraction of knowledge the ego agent thinks an alter agent knows, but in actuality does not.

The primary differences are a modification for how similarities and expertise are calculated, an additional type of item that can be added to a message called a KTM item, which requires the transactive knowledge message complexity network which is used in a way similar to the knowledge message complexity network that was used, and additional parsing to handle this additional type of message item.

For this new type of message item, rather than being the sender sharing a piece of knowledge to the receiver, the sender instead shares the information that another agent knows a piece of knowledge, which we will refer to as a KTM item. When an agent receives either this type of item or a knowledge item, that agent adds that information to their transactive memory. Each agent then has a memory about what other agents knows. This memory is not perfect however, as any secondhand information is not guaranteed to still be true. One can imagine a game of telephone (AKA Chinese whispers) where a chain of individuals secretly communicates a message to the next person in the chain in hopes of preserving the message. In a perfect system, this would be achievable, but it is almost a certainty that in a real setting that an ego agent will

eventually send an item about an alter agent, that the ego believes to be true, but is not. The existence of this divide between reality and perception allows Construct agents to better match social theory and real-world behaviors. For example, Ren et al. (2006) used Construct's transactive memory mechanisms to show evidence that people trained on a task in a group setting are better able to solve a problem than those trained individually and then forced into a group setting.

This modifies the calculation of similarity and expertise in that instead of using the true knowledge that two agents know, the ego agents deciding who to initiate interactions with instead uses their own transactive memory which is incomplete and in certain situations in error. In addition, it can be a heavy burden to remember everyone in a system. Individuals typically only retain information about agents in their sphere of influence. When deciding about agents outside their sphere of influence, ego agents instead reference a generalized group or generalized other.

The nodeset "agentgroup" is not a strictly required nodeset but is a requirement for generalized groups. If the nodeset is not present, the simulation defaults to the generalized other in such cases. In such cases as group are used, this model takes as an optional network agent group membership network which simply specifies if an agent is in a group. Naturally, an agent can be in a single group, no group, or many to all groups. If the agent belongs to one or many groups, one of those groups is selected randomly for the purpose of knowledge similarity and expertise. A group knows a piece of knowledge with probability equal to the fraction of agents in that group that know that piece of knowledge. This probability is static and recalculated once every time period. If the agent is not in a group or no groups exist, the generalized other is used instead which has its knowledge probability determined in the same way as generalized groups using the entire population of agents.

In addition to adding knowledge items to a message in the exact same way as the Standard Interaction Model, KTM items are added in a similar way using "can_send_knowledgeTM" and "can_receive_knowledgeTM" from a node's attributes. KTM items however have no knowledge priority. Items from transactive memory are included in a uniformly random manner. This means that if more knowledge is known about an alter agent, it is more likely any piece of information about that alter agent will be added to the ego agent's message. When the items are then added to the message if it is larger than the medium's "maxMsgComplexity" node attribute, items are randomly removed regardless of whether they are knowledge or KTM items.

Additions to message parsing is as minor as the additions to message formation. Knowledge items are parsed in the same way except it is also added to the receiver's transactive memory with the alter agent being the sender of the message. KTM items are added to the receiver's transactive memory with the alter agent coming from the item the sender attached rather than the sender being the alter agent. As an example, Agent A may send a message to Agent B that Agent C knows knowledge K. Agent A will then add that Agent C knows knowledge K into their transactive memory. Some obvious situations are avoided when sending a KTM item. An agent

cannot send a KTM item about themselves or the intended receiver as both would have perfect memory about what knowledge they know.

Twitter Interaction Model

The "Twitter Interaction Model" aims to simulate the Twitter environment. Twitter-style communications has frequently been modeled in the world of network analysis and agent based simulations. To simulate Twitter, we need two primary concepts, how do agents decide when to create a tweet, reply, quote, or retweet and how do agents decide which tweets, replies, quotes, or retweets to read. To facilitate simulating this, event entities will be constructed rather than creating various proxy functions to emulate the environment without saving the information in a tweet like format. Additionally, methods are constructed to maintain long-term stability and provide a soft cap on the memory usage of the simulation, as well as implementing twitter followers. Unlike the Knowledge Transactive Memory Interaction Model, this model can be run side by side with the Standard Interaction Model and other interaction models.

The Twitter Interaction Model requires two networks, the "knowledge network" which serves a similar function in this model as other models, and the "twitter follower network" which provides a seed network to begin simulations. A homogeneous network for twitter followers is allowed by simulation standards and would produce non-trivial results, however almost any situation would call for an initial seeding of the network. Additional requirements include three model parameters "relax_rate", "interval_time_duration", and "maximum_post_inactivity". Relax rate determines how quickly opinions change, interval time duration is the amount of time between two time periods, and maximum post inactivity is how long an event like a tweet needs to be inactive before it is removed. Each of these will be further discussed in this section. Below is an example XML input.

```
<models>
   <model name="Twitter Interaction Model">
      <param name="relax_rate" value="0.3"/>
      <param name="interval_time_duration" value="1.0"/>
      <param name="maximum_post_inactivity" value="10.0"/>
   </model>
</models>
```

It is difficult to talk about both primary concepts at the same time and also to introduce one at a time. First, we will go over how users create tweets, both spontaneously and in response to reading tweets, replies, etc. before going over how users choose which events to read. A general twitter event can be any type of content a user posts to Twitter. Users only spontaneously create original tweets. The number of tweets a user will generate in a time period is equal to an agent's "Twitter_post_density" node attribute times the "interval_time_duration".

Most events follow a similar structure for creating an event. The event contains an id, user, timestamp, and a single piece of knowledge. When a tweet is generated, a random piece of knowledge is selected. In addition, an opinion is attached to a tweet which is a statement by the

posting user whether they believe that piece of knowledge is true or false. Users state that a piece of knowledge is true with probability received from the knowledge opinion network ($O_{i,k}$). Any event may also mention other users. This will only become significant when examining how users decide which events to read.

The other types of events: replies, retweets, and quotes, are created in response to reading an event. When an event is read, the reading user has probability based on the agent's node attribute `"Twitter_reply_probability"` to reply to an event, `"Twitter_repost_probability"` to retweet a tweet, and `"Twitter_quote_probability"` to quote an event. When an event is responded to, the response will contain/be about the same piece of knowledge. This is meant to signify that tweets are typically short and primarily about one topic/piece of knowledge. Quotes also attach the same opinion used in the parent event, but only if the user's opinion aligns with the parent's opinion. This uses the same probability sampling from the knowledge opinion network. Replies and retweets use independent opinion generation in the same way an original tweet is generated.

When reading an event, the reading user has the option to follow the sending user. The decision to follow is made by comparing the reader's perceptions about the sending user's opinions with their own. Perceptions ($R_{i,j,k}$) operate similarly to transactive memory except for opinions rather than knowledge. In this case, a differentiating factor is required for not only true and false, but also if the ego agents have no information on an alter's opinion. For this reason, perceptions take on +1 for true, -1 for false, and 0 for no information. The probability that agent i follows agent j after reading agent j's post is then,

$$PF_{i,j} = C_j\left(1 - \frac{1}{|K_i^*|}\sum_k \left|\frac{R_{i,j,k}+1}{2} - O_{i,k}\right|\right),$$

where C_j is an agent's node attribute `"Twitter_charisma"` and $|K_i^*|$ is the number of pieces of knowledge agent i knows. If an agent's node attribute of `"Twitter_autofollow"` is "true", an ego agent will follow an alter agent when the alter agent follows the ego agent if the alter agent is not already following the ego agent.

In addition to reading events, agents passively add followers and remove followers each time period. The follow recommendation on Twitter is determined by proprietary software that is not disclosed to the public. As such, we attempt to reproduce a recommendation system based on the graph of mentions. Currently, users will mention a random agent when generating any type of post except for a quote, with plans to expand this mechanism with more complex mechanics in future. More methods for removing followers as can be found in Kwak et al. (2012). We use a graph of responses, unidirectional relationships, and Jaccard similarity of follows to indicate how likely an agent is to unfollow another, as many of the other factors cannot be utilized due to various idealizations this model has made to the Twitter environment. Two scale factors `"Twitter_add_followers_scale_factor"` and `"Twitter_remove_followers_scale_factor"` which are agent node attributes determine the

strength of this effect with the former increasing likelihood of following, the later decreasing the chance of unfollowing, and are both scaled by the `"interval_time_duration"`.

Another feature that is not currently public knowledge is how Twitter orders its Twitter feed for each user. For this, we will build on several core concepts. When an agent is mentioned in a post, that post should appear with high priority in an agent's feed. In addition, replies to an agent post should also have a high priority of appearing in an agent's feed. Direct responses like these are critical to forming an engaging dialogue between individuals which is obviously a goal for a platform that wants its base to constantly use that platform. In addition, agents would naturally want to see posts from other agents they follow. Finally, if an agent has no mentions, replies, or follower posts to read, the agent will receive posts based on popularity. In an ideal system, these posts would be related to other content the agent likes to absorb. Given the obscure nature of how Twitter accomplishes this goal and a desire to keep this base model simple we ignore this possible mechanism.

Posts are separated into these three priority levels of replies and mentions, follower posts, and all other posts. Within each level, the posts are ordered by their popularity, which we define as how large a post's cascade is divided by how old the post is. In this definition, we only consider the cascade below a post. Even if a reply is a part of a large cascade, if that reply does not itself have a large resulting cascade, we do not consider that reply as being "popular". Lastly, random posts are sprinkled through the feed after this ordering to promote the agent to branch out and explore new topics/agents. This is achieved by swapping two posts position in a feed. Ten percent of a feed is swapped in this fashion.

Agents read posts during the **Think** function after generating original post. The number of posts an agent reads is determined by the agent's node attribute `"Twitter_reading_density"`. Multiplying this density by the `"interval_time_duration"`, gives the number of posts an agent will read in a time period. Like creating original posts, agents will only read posts if that agent is active via the agent active timeperiod network. Agents read posts in order in their feed and only read posts once. Therefore, when a post is read is removed from an agent's feed and can never reenter an agent's feed. With this restriction, posts are still capable of affecting an agent over multiple time periods.

When an agent reads a post, they gain the knowledge that post is communicating as well as adding the posting agent's opinion to their perceptions about that agent. In addition, the reading agent will also absorb information from the local sphere of posts around the read post. Because every post in a cascade is about the same piece of knowledge, the reading agent will only add opinions to their perceptions. When reading a root or original post, all direct replies to that post are read and their opinions added to the ego's perceptions. These replies do not have the opportunity for their posting agent to be followed in this case. For retweets, the retweeted post is parsed for opinions as well as any of the retweet's direct replies. The same is true for quotes. Finally, replies to a reply are parsed for opinions as well as the post the reply was replying to. In the case where the reply is replying to something other than the root post, the root post is also

parsed for opinions. In this scenario, it is possible for a post's opinions to be parsed multiple times.

Agent's opinions change over time based on information they gather from other agents. If multiple independent sources claim a something is true or false, one might expect that claim is factual. In this way, other agent's opinions are used inform the ego's opinion in the following equation,

$$O_{i,k}(t) = (1 - rr)O_{i,k}(t-1) + \frac{rr}{\sum|R_{i,j,k}|}\sum\left|R_{i,j,k}\right|\left(\frac{R_{i,j,k}+1}{2}\right),$$

where rr is the "relax_rate". This keeps opinions in the range [0,1] and does not bias opinion dynamics if a user does not have a perception of all agents in the simulation.

Finally, to maintain simulation longevity, posts are removed from the history if they remain inactive for an amount of time equal to "maximum_post_inactivity". A post is inactive if no posts are added to its cascade. In this way, entire cascades are removed at the same time both for efficiency and also to prevent potential conflicts of looking up a post that has been deleted. Removed posts are also removed all agent's feeds.

As a final note, communication mediums typically take the form of books, billboards, email, conversations, etc. Broadly, Twitter would be considered an internet medium if implemented in the Standard Interaction Model, however, to ensure proper cooperation with other interaction and modification models, the Twitter Interaction model creates its own communication medium which is obscured from other models. Many features in the **Update** and **Communicate** functions operate regardless of the communication medium used. To avoid potential unintended conflicts functions that do depend on communication medium will simply ignore this extra communication medium. An obvious example would be potential conflicts with the Mail model which can delay a message being sent.

Modification Models

These models do not inherently generate new messages. Rather a modification model aims at manipulating, removing, or in some cases generating new messages in response to existing messages. An example is the Subscription Model, which can copy messages and forward them different agents. In addition, these models may also modify networks.

Mail Model

The "Mail Model" constructs mailboxes for each user. Messages will be placed into mailboxes with a probability from the agent mail usage by medium network. If this happens, that message is removed from Construct's central message queue. Agents may then check their inbox each time period using the probability from mail check probability network. If this happens, all messages in that agent's inbox return to Construct's central message queue. When implemented in this way, messages can enter a mailbox and subsequently leaving when the mailbox is checked in the same time period.

Knowledge Learning Difficulty Model

The `"Knowledge Learning Difficulty Model"` scans Construct's central message queue during the **Update** function for any message items that are about knowledge. For each knowledge item, the receiver has a probability from knowledge learning difficulty network to not receive that knowledge item. In the case that a message has no more items, it is removed from the message queue. In this model, "knowledge learning difficulty network" is a required network to be included in the input deck.

Literacy Model

The `"Literacy Model"` scans the message queue during the **Update** function. The receiver and sender's literacy are gathered from the literacy strength network. For each item, if that item is a knowledge item, that item is removed if either agent is found to be illiterate, which is found to be true with a probability equal to their literacy.

Forgetting Model

When messages are parsed during the **Communicate** function, the `"Forgetting Model"` records what knowledge was sent to which receiver. Then during the **Clean Up** function, if an agent was not received a piece of knowledge the value of that entry in the knowledge network is decremented with probability from the knowledge forgetting prob network. The amount entry is decreased comes from the knowledge forgetting rate network. Agent's knowledge value cannot fall below zero and an agent cannot forget a piece of knowledge if that agent is the only agent that knows that piece of knowledge.

Subscription Model

When a message is parsed by the `"Subscription Model"` during the **Communicate** function, that message is added to an internal "public" queue with probability from the public propensity network where the relevant agent is the sender. This public queue is not accessible expect by the Subscription model. In the **Clean Up** function agents if they are not subscribed in the subscription network, each agent subscribes with probability from subscription probability network where the column agent is the sender. Finally, during the **Think** function, public messages from the previous time period are copied and forwarded to all subscribing agents.

Model Dependencies

Each of the models have been discussed in detail already. This section aims to summarize what each model requires in the input deck, what are optional and their defaults, and bounds for network weights.

Standard Interaction Model

Table 5. Required nodesets for the Standard Interaction model.

Nodeset name	Expected Attributes	Attribute Ranges

agent	learning_rate	[0,1]
	can_send_knowledge	true or false
	can_receive_knowledge	true or false
knowledge		
timeperiod		

Table 6. Required networks for the Standard Interaction model.

Network Names
knowledge network
interaction sphere network

Table 7. Optional networks for the Standard Interaction model.

Network Names	Default Values
access network	true
agent active timeperiod network	true
agent initiation count network	1
agent reception count network	1
communication medium access network	1
communication medium preferences network	1.0
communication medium preferences network 3d	Uses the 2d network if this network is not present in the input deck.
interaction knowledge weight network	1.0
knowledge expertise weight network	1.0
knowledge message complexity network	1
knowledge priority network	1.0
knowledge similarity weight network	1.0
learnable knowledge network	true
physical proximity network	1.0
physical proximity weight network	1.0
social proximity network	1.0
social proximity weight network	1.0
sociodemographic proximity network	1.0
sociodemographic proximity weight network	1.0

Knowledge Transactive Memory

The Knowledge Transactive Memory Interaction model uses three model parameters to give the transactive memory an initial state. It is also mutually exclusive with the Standard Interaction model.

- false_positive_rate
- false_negative_rate
- threshold

Table 8. Required nodesets for the Knowledge Transactive Memory Interaction model.

Nodeset name	Expected Attributes	Attribute Ranges
agent	learning_rate	[0,1]
	can_send_knowledge	true or false

	can_receive_knowledge	true or false
	can_send_knowledgeTM	true or false
	can_receive_knowledgeTM	true or false
knowledge	N/A	
timeperiod	N/A	

The Knowledge Transactive Memory Interaction model uses all the required networks from the Standard Interaction Model as well as all the optional networks with the same default values. Below are the model specific networks. Not included is the "agent group membership network" which is only required if the "agentgroup" nodeset is in use.

Table 9. Optional networks for the Standard Interaction model.

Network Names	Default Values
transactive knowledge message complexity network	1

Twitter Interaction Model

The Twitter Interaction model uses three model parameters and is compatible with all other interaction models.

```
relax_rate
interval_time_duration
maximum_post_inactivity
```

Table 10. Required nodesets for the Twitter Interaction model.

Nodeset name	Expected Attributes	Attribute Ranges
agent	learning_rate	[0,1]
	can_send_knowledge	true or false
	can_receive_knowledge	true or false
	agent_type	{"gen"}
	Twitter_post_density	[0,∞)
	Twitter_reply_probability	[0,1]
	Twitter_repost_probability	[0,1]
	Twitter_reading_density	[0,∞)
	Twitter_add_followers_scale_factor	[0,∞)
	Twitter_remove_followers_scale_factor	[0,∞)
	Twitter_auto_follow	true or false
	Twitter_charisma	[0,1]
knowledge		
timeperiod		

Table 11. Required networks for the Twitter Interaction model.

Network Names
knowledge network
twitter follower network

Table 12. Optional networks for the Twitter Interaction model.

Network Names	Default Values
agent active timeperiod network	true
knowledge opinion network	0.5

Mail Model

Table 13. Optional networks for the Mail model.

Network Names	Default Values
agent mail usage by medium network	1.0
mail check probability network	0.5

Knowledge Learning Difficulty Model

Table 14. Required networks for the Knowledge Learning Difficulty model.

Network Names
knowledge learning difficulty network

Literacy Model

Table 15. Optional networks for the Literacy model.

Network Names	Default Values
literacy strength network	0.9

Forgetting Model

Table 16. Optional networks for the Forgetting model.

Network Names	Default Values
knowledge forgetting rate network	0.25
knowledge forgetting prob network	0.5

Subscription Model

Table 17. Optional networks for the Subscription model.

Network Names	Default Values
public propensity network	0.1
subscription network	false
subscription probability network	0.1

Output

There are three output methods currently provided by Construct, each of which uses a different file format. CSV output produces a comma separated values representation of a dense matrix over time. Dynetml output produces a DyNetML format file for the specified networks. The messages output produces a JSON format file detailing each message and all items contained in the message.

CSV Output

Below is an example usage of the CSV output method (note that type specifier for CSV output is "output_graph").

```
<output type="output_graph">
   <parameter name="network_name" value="knowledge network"/>
   <parameter name="output_file" value="knowledge.csv"/>
   <parameter name="timeperiods" value="all"/>
</output>
```

Here, the network "knowledge network" is being written to the file "knowledge.csv" for all timeperiods. Any network created in Construct can be given as a value for "network_name", even networks that are not explicitly defined in the input XML deck. Any optional network generated by a model can be used for output. This method does require the file name use a .csv file extension.

The structure of the CSV file contains the source nodeset on the rows and target nodeset for the columns. Node names are not included to reduce file size; rather the row number corresponds to the source node index and the column number corresponds to the target node index. Two values can be used for the parameter "timeperiods": "all" or "last". When "last" is used, the network is only output during the last timeperiod of the simulation. For "all", every timeperiod is output. In the CSV file, an empty line indicates the transition from one timeperiod to the next. The row number is then reset at the empty line for determining which row corresponds to which node index. This method is primarily used for large simulations where output files can become excessively large.

DyNetML Output

Below is an example of the dynetml output method.

```
   <output type="output_dynetml">
      <parameter name="output_all_networks" value="false"/>
      <parameter name="network_names" value="knowledge network,interaction
network"/>
      <parameter name="output_file" value="construct_networks.xml"/>
      <parameter name="timeperiods" value="all"/>
   </output>
```

In this example, the "knowledge network" and "interaction network" are being sent to "construct_networks.xml" for all timeperiods. The parameter "output_all_networks" allows for all networks stored in Construct to be output if its value is "true". In this case the parameter "network_names" is not required. For specific network output, the parameter "network_names" takes a comma separated list of network names and only outputs those specified. The output file should always be in XML format (.xml file extension) and the content of the file is consistent with the DyNetML format (DyNetML is an

XML derivative language for exchanging rich social network data). As with CSV output, either the "last" or "all" timeperiods can be specified for the "timeperiods" parameter.

Messages Output

Below is an example of the messages output method.

```
<output type="output_Messages">
   <parameter name="output_file" value="messages.json"/>
</output>
```

Here only the parameter "output_file" is used for specifying the file name to use for output. Files are required to be in JSON format (.json file extension). The JSON representation in the ouput file contains all messages sent each timeperiod. Messages contain the sending agent index, receiving agent index, the name of the communication medium used, and a list of message items. Each message item contains a set of attributes, indexes, and values. Below is an example of a message in JSON format.

```
{
   "sender" : 93,
   "receiver" : 0,
   "medium index" : "CommunicationMedium_0",
   "Items" : [
      {
         "attributes" : ["SIM knowledge","knowledge"],
         "indexes" : {
            "knowledge" : 3
         },
         "values" : {
         }
      }
   ]
}
```

In the above example, agent 93 sent a message to agent 0 using "CommunicationMedium_0". The message contains only one item. That item has attributes "knowledge" and "SIM knowledge". "knowledge" indicates that item contains knowledge that is to be sent to the receiver. "SIM knowledge" is an internal usage attribute that aids in ensuring only one entity parses a knowledge item, in this case the Standard Interaction Model. The indexes indicate that the knowledge node being communicated about has an index of 3. This gives a complete list of all messages sent using Construct's central messaging system.

References

Apache Software Foundation. (2019, September 10). *MapReduce Tutorial*. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Carley, K. M. (1986). An approach for relating social structure to cognitive structure. *Journal of Mathematical Sociology*, 12(2), 137-189. https://doi.org/10.1080/0022250X.1986.9990010

Carley, K. M. (1990). Group stability: A socio-cognitive approach. *Advances in Group Processes: Theory and Research*, (Vol. 7, pp. 1-44). JAI Press. http://www.casos.cs.cmu.edu/events/summer_institute/2014/reading_list/pubs/carley_1990_groupstability.pdf

Carley, K. M. (1991). A theory of group stability. *American Sociology Review, 56*(3), 331–354. https://doi.org/10.2307/2096108

Carley, K. M. 1995. Communication technologies and their effect on cultural homogeneity, consensus, and the diffusion of new ideas. *Sociological Perspectives, 38*(4), 547–571. https://doi.org/10.2307/1389272

Carley, K. M. (2002). Computational organization science: A new frontier. *Proceedings of the National Academy of Sciences of the United States of America*, 99(Suppl 3), 7257-7262. https://doi.org/10.1073/pnas.082080599

Carley, K. M. (2006). A dynamic network approach to the assessment of terrorist groups and the impact of alternative courses of action. In *Visualising Network Information* (pp. KN1-1 – KN1-10). Meeting Proceedings RTO-MP-IST-063, Keynote 1. Neuilly-sur-Seine, France: RTO. https://www.sto.nato.int/publications/STO%20Meeting%20Proceedings/RTO-MP-IST-063/$MP-IST-063-KN1.pdf

Carley, K. M., Martin, M. K., & Hirshman, B. R. (2009). The etiology of social change, *Topics in Cognitive Science*, *1*(4), 621-650. https://doi.org/10.1111/j.1756-8765.2009.01037.x

Carley, K. M., & Maxwell, D. T. (2006). Understanding taxpayer behavior and assessing potential IRS interventions using multiagent dynamic-network simulations. In Dalton & Bliss (Eds.), *Recent Research on Tax Administration and Compliance: Proceedings of the 2006 IRS Research Conference* (pp. 93-106). Washington, D.C. https://www.irs.gov/pub/irs-soi/06carley.pdf

Carley, K. M., & Newell, A. (1994). The nature of the social agent. *Journal of Mathematical Sociology*, *19*(4), 221-262. https://doi.org/10.1080/0022250X.1994.9990145

Carley, K. M., & Reminga, J. (2004). *ORA: Organization Risk Analyzer*. (Technical Report CMU-ISRI-04-106). Pittsburgh, PA, USA: Carnegie Mellon University, School of Computer Science, Institute for Software Research. http://reports-archive.adm.cs.cmu.edu/anon/isri2004/CMU-ISRI-04-106.pdf

Carley, K. M., Robertson, D. C., Martin, M. K., Lee, J. S., St Charles, J. L., & Hirshman, B. R. (2010). Predicting intentional and inadvertent non-compliance. In M. E. Gangi & A. Plumley (Eds.). *Recent Research on Tax: Selected Papers Given at the 2010 IRS Research*

Conference Administration and Compliance (pp. 53-82). Washington, DC. https://iwp-koeln.org/wp-content/uploads/Proceedings_IRS_Conference_2010.pdf#page=162

Epstein, J. M. & Axtell, R. (1996). *Growing artificial societies: Social science from the bottom up*. Brookings Institution Press.

Festinger, L. (1954). A theory of social comparison processes, *Human Relations, 7*(2), 117-140. https://doi.org/10.1177/001872675400700202

Festinger, L. (1957). *A theory of cognitive dissonance*. Row, Peterson.

Friedkin, N. (1998). *A structural theory of social influence*, Cambridge University Press.

Gardner, M. (1970, October). Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Scientific American*, 223, 120–123. https://doi.org/10.1038/scientificamerican1070-120

Giddens, A. (1986). *The constitution of society: Outline of the theory of structuration*. University of California Press.

Hirshman, B. R., Birukou, A., Martin, M. K., Bigrigg, M. W., & Carley, K. M. (2008). *The impact of educational interventions on real & stylized cities*. (Technical Report CMU-ISR-08-114). Pittsburgh, PA, USA: Carnegie Mellon University, School of Computer Science, Institute for Software Research. http://reports-archive.adm.cs.cmu.edu/anon/isr2008/CMU-ISR-08-114.pdf

Hirshman, B. R., Carley, K. M., & Kowalchuck, M.J. (2007a). *Loading networks in Construct*. (Technical Report CMU-ISRI-07-116). Pittsburgh, PA, USA: Carnegie Mellon University, School of Computer Science, Institute for Software Research. http://reports-archive.adm.cs.cmu.edu/anon/isri2007/CMU-ISRI-07-116.pdf

Hirshman, B. R., Carley, K. M., & Kowalchuck, M.J. (2007b). *Specifying agents in Construct*. (Technical Report CMU-ISRI-07-107). Pittsburgh, PA, USA: Carnegie Mellon University, School of Computer Science, Institute for Software Research. http://reports-archive.adm.cs.cmu.edu/anon/isri2007/CMU-ISRI-07-107.pdf

Joseph, K., Carley, K. M., Filonuk, D., Morgan, G. P., & Pfeffer, J. (2014). Arab Spring: From newspaper. *Social Networks and Mining, 4*, 177. https://doi.org/10.1007/s13278-014-0177-5

Joseph, K., Morgan, G. P., Martin, M. K., & Carley, K. M. (2014). On the coevolution of stereotype, culture, and social relationships: An agent-based model. *Social Science Computer Review*. *32*(3), 295–311. https://doi.org/10.1177/0894439313511388

Kim, B. (2001). Social constructivism. In M. Orey (Ed.), *Emerging Perspectives on Learning, Teaching, and Technology*. http://epltt.coe.uga.edu/

Knoeller, J. (2013). *Analyzing job/machine matches using condor_q-analyze* [PowerPoint slides]. Paradyn/HTCondor Week 2013: https://research.cs.wisc.edu/htcondor/HTCondorWeek2013/presentations/KnoellerJ_QAnalyze.pdf

Krackhardt, D., & Carley, K. M. (1998). A PCANS model of structure in organizations. In *Proceedings of the 1998 International Symposium on Command and Control Research and Technology* (pp. 113-119). Vienna, VA: Evidence Based Research.

Kwak, H., Moon, S., & Lee, W. (2012). More of a Receiver Than a Giver: Why Do People Unfollow in Twitter?. *Proceedings of the International AAAI Conference on Web and Social Media*, *6*(1). https://ojs.aaai.org/index.php/ICWSM/article/view/14296

Manis, J. G., & Meltzer, B. N. (1978). *Symbolic interaction: A reader in social psychology*. Allyn & Bacon.

MapReduce. (2020, 3 December). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=MapReduce&oldid=992047007

Moon, I.-C., & Carley, K. M. (2007). Modeling and simulation of terrorist networks in social and geospatial dimensions, *IEEE Intelligent Systems, 22*(5), 40-49. https://doi.org/10.1109/MIS.2007.91

Ren, Y., Carley, K., & Argote, L. (2006). The contingent effects of transactive memory: When is it more beneficial to know what others know? *Management Science, 52*(5), 671-682. https://doi.org/10.1287/mnsc.1050.0496

Salancik, G. R., & Pfeffer, J. (1978). A social information processing approach to job attitudes and task design. *Administrative Science Quarterly, 23*(2), 224-253. https://doi.org/10.2307/2392563

Schelling, T. C. (1978). *Micromotives and Macrobehavior*, Norton.

Schelling, T. C. (1971). Dynamic models of segregation. *Journal of mathematical sociology 1.2*, 143-186. https://doi.org/10.1080/0022250X.1971.9989794

Schmerl B., Garlan D., Dwivedi V, Bigrigg M. W., and Carley K. M. 2011. SORASCS: a case study in soa-based platform design for socio-cultural analysis. *In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)* 643–652. https://doi.org/10.1145/1985793.1985883

Schreiber, C., & Carley, K. M. (2003). The impact of databases on knowledge transfer: Simulation providing theory. In *2003 NAACSOS Conference Proceedings*, Pittsburgh, PA. https://www.researchgate.net/publication/228430691_The_impact_of_databases_on_knowledge_transfer_Simulation_providing_theory_2003_NAACSOS_conference_proceedings

Schreiber, C., & Carley, K. M. (2007). Agent interactions in Construct: An empirical validation using calibrated grounding. In *2007 BRIMS Conference Proceedings*, Norfolk, VA. https://www.researchgate.net/publication/228725767_Agent_interactions_in_Construct_An_empirical_validation_using_calibrated_grounding

Schreiber, C., Singh, S., & Carley, K. M. (2004). *Construct-A multi-agent network model for the co-evolution of agents and socio-cultural environments.* (Technical Report CMU-ISRI-04-109). Pittsburgh, PA, USA: Carnegie Mellon University, School of Computer Science, Institute for Software Research. http://reports-archive.adm.cs.cmu.edu/anon/isri2004/abstracts/04-117.html.

Simon, H. A. (1957). *Administrative behavior: A study of decision-making processes in administrative organization* (2nd ed.). Macmillan.

Stryker, S. (1980). *Symbolic Interactionism: A social structural version*. Benjamin Cummings.

Tsvetovat, M., & Carley, K. M. (2004). Modeling complex socio-technical systems using multi-agent simulation methods. *Künstliche Intelligenz*, 18(2), 23-28. http://www.casos.cs.cmu.edu/publications/working_papers/tsvetovat_2004_modeling.pdf

Wegner, D. M. (1987). Transactive memory: A contemporary analysis of the group mind. In B. Mullen, G. R. Goethals (Eds.) *Theories of Group Behavior* (pp. 185-205). Springer. https://doi.org/10.1007/978-1-4612-4634-3_9

Appendices

Appendix A The Sample Input File (AKA Input Deck)

```
<construct>
  <construct_vars>
    <var name="time_count" value="100"/>
    <var name="agent_count" value="200"/>
    <var name="knowledge_count" value="100"/>
    <var name="knowledgegroup_count" value="3"/>
    <var name="agentgroup_count" value="1"/>
  </construct_vars>


  <construct_parameters>
    <param name="seed" value="1"/>
    <param name="verbose_initialization" value="false"/>
  </construct_parameters>

  <model name=”Standard Interaction Model”/>

  <nodes>
    <nodeclass name="CommunicationMedium">
      <node name="facetoface">
        <attribute name=“maxMsgComplexity" value="1"/>
        <attribute name=“maximumPercentLearnable"
value="1.0"/>
        <attrubite name=“time_to_send" value="1"/>
      </node>
    </nodeclass>

    <nodeclass name="agent">
      <generator type=”constant”>
        <count value="agent_count"/>
        <attribute name="learning_rate" value="1"/>
       <attribute name="can_send_knowledge" value="true"/>
        <attribute name="can_receive_knowledge" value="true"/>
      </generator>
    </nodeclass>

    <nodeclass name="knowledge">
      <generator type=”constant”>
        <count value=”knowledge_count”>
      </generator>
    </nodeclass>

    <nodeclass name="agentgroup">
      <generator type=”constant”>
        <count value=”agentgroup_count”/>
      </generator>
    </nodeclass>

    <nodeclass type="knowledgegroup" id="knowledgegroup">
      <node name="KG1"/>
      <node name="KG2"/>
      <node name="KG3"/>
    </nodeclass>

    <nodeclass type="timeperiod" id="timeperiod">
      <generator type=”constant”>
         <count value=”time_count”/>
      </generator>
    </nodeclass>

    <nodeclass type="dummy_nodeclass" id="dummy_nodeclass">
      <node id="dummy1" title="dummy1"/>
    </nodeclass>
  </nodes>


  <networks>
    <network src_nodeclass_type="agent"
target_nodeclass_type="timeperiod" id="knowledge message
complexity network" link_type="unsigned int"
network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::timeperiod::count_minus_one"/>
        <param name="min" value="1"/>
        <param name="max" value="1"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="timeperiod" id="agent initiation count
network" link_type="int" network_type="dense">
      <generator type="randomuniform">
```

```
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::timeperiod::count_minus_one"/>
        <param name="min" value="10"/>
        <param name="max" value="10"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="knowledge" id="knowledge network"
link_type="float" network_type="dense">
      <generator type="randombinary">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::knowledge::count_minus_one"/>
        <param name="mean" value="0.1"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="agent" id="access network"
link_type="float" network_type="dense">
      <generator type="constant">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::agent::count_minus_one"/>
        <param name="constant_value" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="timeperiod" id="knowledge similarity
weight network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::timeperiod::count_minus_one"/>
        <param name="min" value="1.0"/>
        <param name="max" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="timeperiod" id="knowledge expertise
weight network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::timeperiod::count_minus_one"/>
        <param name="min" value="1.0"/>
        <param name="max" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="knowledge" id="interaction knowledge
weight network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::knowledge::count_minus_one"/>
        <param name="min" value="1.0"/>
        <param name="max" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="timeperiod" id="physical proximity
weight network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::timeperiod::count_minus_one"/>
        <param name="min" value="1.0"/>
        <param name="max" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="timeperiod" id="social proximity
weight network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::timeperiod::count_minus_one"/>
```

```
        <param name="min" value="1.0"/>
        <param name="max" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="agent" id="physical proximity network"
link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::agent::count_minus_one"/>
        <param name="min" value="1.0"/>
        <param name="max" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="agent" id="social proximity network"
link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::agent::count_minus_one"/>
        <param name="min" value="1.0"/>
        <param name="max" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="agent" id="sociodemographic proximity
network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::agent::count_minus_one"/>
        <param name="min" value="1.0"/>
        <param name="max" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="timeperiod" id="agent active
timeperiod network" link_type="bool" network_type="dense">
      <generator type="randombinary">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::timeperiod::count_minus_one"/>
        <param name="mean" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="agent" id="interaction sphere network"
link_type="bool" network_type="dense">
      <generator type="randombinary">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::agent::count_minus_one"/>
        <param name="mean" value="1"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="agentgroup" id="agent group membership
network" link_type="bool" network_type="dense">
      <generator type="randombinary">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::agentgroup::count_minus_one"/>
        <param name="mean" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="timeperiod" id="agent reception count
network" link_type="int" network_type="dense">
      <generator type="randombinary">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::timeperiod::count_minus_one"/>
        <param name="mean" value="10.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
```

```
<!--
  This is the state of the art in my filter work. Have to get
this working properly!
      <generator type="randombinary">
        <rows groups="agent_grp1,agent_grp2"
group_membership_network="'agent group membership network'"/>
        <cols groups="agent_grp3"
group_membership_network="'agent group membership network'"/>
        <param name="mean" value="1.0" />
      </generator>
-->
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="knowledge" id="knowledge priority
network" link_type="unsigned int" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::knowledge::count_minus_one"/>
        <param name="min" value="1"/>
        <param name="max" value="1"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="knowledge" id="learnable knowledge
network" link_type="bool" network_type="dense">
      <generator type="randombinary">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::knowledge::count_minus_one"/>
        <param name="mean" value="1.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="agent"
target_nodeclass_type="dummy_nodeclass" id="agent forgetting
rate network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0" last="0"/>
        <param name="min" value="0.0"/>
        <param name="max" value="0.0"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <!--
    This network determines which agent has access to which
mediums.
    Set access to zero if you do not want the agent to have
access to that medium.
    -->
    <network src_nodeclass_type="agent"
target_nodeclass_type="CommunicationMedium" id="communication
medium access network" link_type="float"
network_type="dense">
      <generator type="constant">
        <rows first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::CommunicationMedium::count_minus_one"/>
        <param name="constant_value" value="1"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <!--
      This network shows what medium is preferred when
communicating with a
      given agent.
      This network is actually a 3d network. It is agent x
agent x medium. One
      way to view it is a collection of agent x medium
networks. There is one of
      these agent x medium networks for every agent, so each
agent has a custom
      agent x medium network that shows what mediums he
prefers to use when
      communicating with any given agent.
    -->
    <network src_nodeclass_type="agent"
inner_nodeclass_type="agent"
target_nodeclass_type="CommunicationMedium" id="communication
medium preferences network 3d" link_type="float"
network_type="dense3d">
      <generator type="constant3d">
        <rows first="1"
last="nodeclass::agent::count_minus_one"/>
        <inners first="0"
last="nodeclass::agent::count_minus_one"/>
        <cols first="0"
last="nodeclass::CommunicationMedium::count_minus_one"/>
        <param name="constant_value" value="1"/>
```

```
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>

    <network src_nodeclass_type="Communication Medium"
target_nodeclass_type="knowledge" id="medium knowledge
network" link_type="bool" network_type="dense">
      <generator type="constant">
        <rows first="0" last=
"nodeclass::CommunicationMedium::count_minus_one"/>
        <cols first="0" last=
"nodeclass::knowledge::count_minus_one"/>
        <param name="constant_value" value="true"/>
        <param name="symmetric_flag" value="false"/>
      </generator>
    </network>



  </networks>

  <outputs>
     <output name="output_graph">
       <parameter name="network_name" value="knowledge
network"/>
       <parameter name="output_file" value="knowledge.csv"/>
       <parameter name="timeperiods" value="all"/>
    </output>

    <output name="output_graph">
       <parameter name="network_name" value="interaction
network"/>
       <parameter name="output_file"
value="interactions.csv"/>
       <parameter name="timeperiods" value="all"/>
    </output>
  </outputs>
</construct>
```

Appendix B A History of Construct

Construct is the embodiment of constructuralism, a mega-theory which states that the socio-cultural environment is continually being constructed and reconstructed through individual cycles of action, adaptation, and motivation. Many social science theories and findings are part of the constructural theoretical approach including structuration theory (Giddens, 1986), social information processing theory (Salancik & Pfeffer, 1978), symbolic interactionism (Manis and Meltzer, 1978; Stryker, 1980), social influence theory (Friedkin, 1998), cognitive dissonance (Festinger, 1957), social constructivism (Kim, 2001), and social comparison (Festinger, 1954). In addition, several cognitive processes are embedded such as transactive memory (Wegner, 1987).

In 1990, research done by Kathleen M. Carley on group stability initiated early model designs for Construct. In her paper, *Group Stability: A socio-cognitive approach*, she created a socio-cognitive model based on nonstructural theory to predict changes in interaction patterns among workers in a tailor shop in Zambia (Carley, 1990). The model tested behaviors that occurred on individuals, such as social change or stability changes that were derived from interaction, as well as the exchange of information between the workers. The resulting observation and analysis of these behaviors provided an explanation for why the workers were able to go on strike successfully after an aborted first strike. The first basic principle of the model is that in every social group, there are facts within the group that have the potential to be learned by members in the group. Information can be broken down into individual facts, which can then be measured quantitatively for a social group. The second basic principle of the model states that there is a probability that certain individuals will interact with one another and exchange facts, which then leads to shared knowledge. The third basic principle states that similar individuals who share common knowledge are more likely to interact. This implies that individuals consider how much in common they have with others before they choose to interact and communicate information. The combination of these three principles leads to the interaction/knowledge cycle, which is what Construct is designed to simulate. This model initially takes a description of a particular society in terms of culture and structure and predicts the ways in which the society can evolve. With these concepts in place, the Construct model continued to evolve.

With advances in computing throughout the 1990's, the Construct model gained more opportunities and capabilities for real world application. The ability to process large amounts of data to predict outcomes on large, scaled populations was critical in construct's development. One of the key developments for the Construct model computationally was research done on knowledge transfer, and its effect on an organization or social group. In 2003, Schreiber and Carley explored data base technology and its support of knowledge transfer. Virtual experiments using the construct model were run using two group conditions, task complexity and experience, to examine how task and referential data types differ when simulating knowledge transfer (Schreiber & Carley, 2003). Transactive memory is also represented by the model to incorporate perception of other's knowledge in the social group. Each agent in the model is assigned task and

transactive knowledge, which are then represented by task databases and referential databases (Schreiber & Carley, 2003). The virtual experiment showed that these databases influence task complexity as well as experience, and that knowledge transfer can be represented in different forms to effectively simulate transfer within an organization. Task data was shown to be most useful for knowledge transfer of simple to moderate level tasks, while referential data was shown to be more useful for complex tasks.

In 2004 Schreiber, Singh, and Carley, described a more complex version of the original Construct-TM model. In addition to having the ability to interact with other human agents, in this model agents could interact with objects that contain information, such as a book or an advertisement. Agents were given several types of capabilities and limitations; examples included control over the ability to communicate and receive information (Schreiber et al., 2004). The number of agent groups was limited to 3 and the number of agents limited to 101 (Schreiber 2004). The interaction mechanism allowed agents to interact based on proximity, perception of others, referrals, access to information, and the ability of forgetting. Knowledge was represented as binary strings, which determined an agent's decision as well as perception of other agents' knowledge. Knowledge was limited to 500 facts and up to 25 tasks were assigned for each particular knowledge bit.

New mechanisms for belief were abandoned, several different approaches for adding in different communication logics were added, and new telecommunication technologies. The ability to specify event histories in external scripts and new communication regimes supported the ability to model taxpayer behavior (Carley & Maxwell, 2006). Geo-proximity modeling was added to support assessment of terror groups (Moon & Carley, 2007). Collectively these changes and others made the entire system more robust and more powerful at modeling the human condition. At this point, the entire system was refactored, thereby increasing maintainability and speed. The modern system is more expansive and can support many more agents and types of communication technologies such as email, books, news, phone, call-centers, lectures, billboards, web pages and so on. This system was then used to assess social change (Carley et al., 2009) and non-compliance (Carley et al, 2010).

The next major innovation was the incorporation of social intelligence. The agents now perceived their social network, constrained behavior based on socio-cognitive constraints on network formation, thus focusing on their local sphere of influence (Joseph, Morgan et al., 2014). This made it possible to increase the size of the populations that could be modeled, increased the speed of processing, and increased the realism of the results. Memory usage was now approximately linear with the number of agents.

Meanwhile Construct was more tightly integrated with ORA. The toolchain, linking AutoMap (later NetMapper), ORA and Construct, meant that the user could go from text mining to the extraction of networks, to simulation. This process supports model reuse and reduces time to model large populations. It also means that models could be more easily instantiated with real world data. This was used to assess revolutionary activity during the Arab Spring, and so to

predict revolutionary behavior given changes in what was covered in the news (Joseph, Carley, et al., 2014).

Appendix C Additional Construct Generators

Group to Group Generators

Generators are created for a network based on a mapping between groups of nodes instead of mappings between nodes.

Nodes in node groups do not have to be contiguous. The box generator only works on contiguous nodes; if the nodes in a node group are not contiguous, you must use multiple box generators. This can require thousands of box generators.

Thinking in terms of groups is more intuitive. Group to group generators can use a reference to a second network for the mappings as well as references to the two group networks.

```
<network src_nodeclass_type=”agent”
      target_nodeclass_type=”knowledge”
      id=”knowledge network”
      link_type=”float”
      network_type=”dense”>

   <generator type=”group_to_group”>
      <rows first=”0” last=”0”/>
      <cols first=”0” last=”0”/>
      <param name=”src_net_name” value=”ag_to_kg_gen_net”/>
      <param name=”row_grp_membership_net”
         value=”’agent group membership network’”/>
      <param name=”col_grp_membership_net”
         value=”’knowledge group membership network’”/>
   </generator>
</network>
```

The above XML shows the group to group generator in use. Note that this generator is the only generator for the knowledge network. The common rows/cols values are set to zero but are not actually used. Instead, the following parameters are used:

`src_net_name`

- This parameter tells the generator where to find the group to group mapping. It refers to another network that should already be loaded. In the example, the network is called “`ag_to_kg_gen_net`”.
- Note that you can choose whatever network name you wish, but the `src_net_name value` attribute must contain the chosen name.

`row_grp_membership_net`

- This tells the generator where the membership network for the row groups is located. In this case, the network’s name is “`agent group membership network`”.
- A membership network is a mapping of nodes to groups. In this case, it maps agents to agent groups.

- Construct uses the "`agent group membership network`" for its own purposes. You can use it for this parameter as well, but you can also substitute your own node to nodegroup network. Just make sure you give the correct name in the `row_grp_membership_net value` field.

`col_grp_membership_net`

- This tells the generator where the membership network for the column groups is located. In this case, the network's name is "`knowledge group membership network`".
- This is a knowledge to knowledgegroup network.
- It is just like the row_grp_membership_net, but it applies to the column node set type, which in this case is knowledge instead of agent.

```
<network src_nodeclass_type="agentgroup"
      target_nodeclass_type="knowledgegroup"
      id="ag_to_kg_gen_net"
      link_type="string"
      network_type="dense">

   <link src_node_id="ag0" target_node_id="fg0"
      value="'gen_typeXXXrandombinary,meanXXX0.0'"/>
   <link src_node_id="ag0" target_node_id="fg1"
      value="'gen_typeXXXrandombinary,meanXXX1.0'"/>
   <link src_node_id="ag0" target_node_id="fg2"
      value="'gen_typeXXXrandombinary,meanXXX0.0'"/>
   <link src_node_id="ag1" target_node_id="fg0"
      value="'gen_typeXXXrandombinary,meanXXX1.0'"/>
   <link src_node_id="ag1" target_node_id="fg1"
      value="'gen_typeXXXrandombinary,meanXXX1.0'"/>
   <link src_node_id="ag1" target_node_id="fg2"
      value="'gen_typeXXXrandombinary,meanXXX1.0'"/>
   <link src_node_id="ag2" target_node_id="fg0"
      value="'gen_typeXXXrandombinary,meanXXX1.0'"/>
   <link src_node_id="ag2" target_node_id="fg1"
      value="'gen_typeXXXrandombinary,meanXXX0.0'"/>
   <link src_node_id="ag2" target_node_id="fg2"
      value="'gen_typeXXXrandombinary,meanXXX1.0'"/>
</network>
```

The above XML shows the mapping between the agent group and knowledge group nodesets.

- Note the value string: `gen_typeXXXrandombinary,meanXXX0.0`.
- This is parsed to find the parameters for the generator for the mapping between agentgroup: `ag0` and knowledgegroup: `fg0`.

- Once parsed, the gen_type will be randombinary and its sole parameter "`mean`" will be 0.0.
- If there should be more parameters, just follow the pattern and use XXX between parameter name and its value and use a comma between individual parameters.
- Unfortunately, limitations in our XML parser preclude the use of more sensible delimiters. Once fixed, we will change from XXX to something like "||" or ":".

```
<network src_nodeclass_type="knowledge"
      target_nodeclass_type="knowledgegroup"
      id="knowledge group membership network"
      link_type="bool"
      network_type="dense">

   <generator type="randombinary">
      <row first="0" last="2"/>
      <cols first="0" last="0"/>
      <param name="src_net_name" value="ag_to_kg_gen_net"/>
      <param name="mean" value="1.0"/>
      <param name="symmetric_flag" value="false"/>

   <generator type="randombinary">
      <row first="3" last="5"/>
      <cols first="1" last="1"/>
      <param name="src_net_name" value="ag_to_kg_gen_net"/>
      <param name="mean" value="1.0"/>
      <param name="symmetric_flag" value="false"/>

   <!— This group contains facts after the first 5 -->
   <generator type="randombinary">
      <row first="6"
         last="nodeclass::knowledge::count_minus_one"/>
      <cols first="2" last="2"/>
      <param name="src_net_name" value="ag_to_kg_gen_net"/>
      <param name="mean" value="1.0"/>
      <param name="symmetric_flag" value="false"/>
   </generator>
</network>
```

The above XML shows the creation of a network containing the mapping between the knowledge and knowledge group.

- Maps nodes to groups.
- Groups must exist in their own node set.
- Typically, the group nodeset's name has the name of the nodes it will be associated with followed by the word "group".
- Example: agentgroup is a node set of agent groups.

- Example groups: finance_dept, advertising_dept, friendlist.
- Example: knowledgegroup is a node set of knowledge groups.
- Normal generators can be used so CSV, DyNetML, and constant generators are typical for membership networks.
- Usually, the bounds of each group are critical.
- Example: finance has 10 agents in it; advertising has 3 agents, etc.
- If groups are not contiguous then generators can specify those agents too.

Appendix D Scripting

Reserved Words in the Construct Scripting Language and Input File

Variables and 'Decisions' are subsets of the more general concept of scripting within Construct. In this section, we will discuss in more detail the parts of the scripting system. We first begin on a rather important note by stating that when using the scripting functionality of Construct, it is important that the modeler does not use any of the following words, which are reserved for certain uses and, if used incorrectly, may provide unexpected results.

Table 18 Construct Scripting Language Reserved Words①

`agent`	`Random*②`
`bool`	`randomBinary②`
`Construct②`	`randomNormal②`
`delay_interpolation`	`randomUniform②`
`details`	`return`
`else`	`set*③`
`error`	`spaces_only`
`get*③`	`static_if`
`if`	`timeperiod`
`preserve_all_white_space`	`verbose`
`preserve_spaces_only`	`..`

* An asterisk indicates a keyword beginning with the fixed text shown can be followed by additional text. See notes ② and ③ below.

① All words beginning with an alphabetic character will be considered variables, though in many cases it will only be a valid variable with the use of `with`.

② All words beginning with "construct" or "random" should not be used – though the list above specifies all current reserved words, anything beginning with either of these two words may become a reserved word at some point during future development.

③ Note that any string beginning with the word "get" or "set" is reserved for use with referencing networks, and thus the remainder of the network reference must be one of the networks that Construct is aware of, specified in CamelCase. Strings that begin with the word "get" or "set" will be treated as a network reference.

Having stated what a researcher should not use when doing scripting in Construct, we now define the lexemes that are possible within Construct's scripting system. Note in the sections below that an expression surrounded by angled brackets, such as `<text>`, indicates an

expression can be used in place of it. One other note is that in all cases below, square brackets ([]) and curly braces ({}) are deliberate and must be included.

Testing Construct Scripts

Construct has a built-in mechanism to help lexicographically check the validity of a script. To do this, the researcher should include the "lexxy_test" parameter in the input file. The researcher then inserts the script into the value attribute. Construct will parse and process the script and the modeler can increase their confidence it is executing as they intended.

```
<param name="lexxy_test" value="[insert the script here]" />
```

General Syntax

§ Comment: `/* <This is an example comment> */`

Comments within scripts occur within `/* */`, as shown above. It is important to note that the user not put commas or quotes within comments. Beyond this, however, users should feel free to use comments as desired, including the use of newlines, e.g.:

```
/* <This is another
 example comment> */
```

§ Quoted literal: `'<Text>'`

To specify a string of text to be used, the user can enclose it within two single quotes. Again, as with comments, the user should refrain from putting other single quote characters or commas into quoted literals. To avoid errors where quoted literals are mistakenly not closed being too confusing, there is a limit of 100 characters per literal – to create literals with more than 100 characters, concatenate multiple literals together.

§ Numbers: `<Number>` or `<Number>.<Number>`

It is important to note two things when specifying numbers. First, Construct will attempt to represent the number using the smallest type possible – that is, if the user does not explicitly (via :float) or implicitly (via adding a .0) cast a number to a float, it will be stored as an integer. Construct supports up to sixteen-bit, twos complement signed integers in addition to C-style floats.

§ White space: `<space>` or `<tab>` or `<newline>`

All whitespace is ignored by the Construct parser – please be aware that under certain conditions, this may produce unexpected results. If the user wishes to specify a variable addressing networks in Construct, which often have spaces, the user must refer to them within a quoted literal.

Mathematical Expressions

§ Sub-expression: `(<Expr>)`

Parentheses are used to create sub-expressions for two reasons. The first, as discussed in the variables section, is to specify order of operations. The second is to specify subexpressions – for example, when writing an if statement, it is necessary to use parentheses.

§ **Addition:** `<Expr>+<Expr>`

§ **Subtraction:** `<Expr>-<Expr`

§ **Multiplication:** `<Expr>*<Expr>`

§ **Division:** `<Expr>/<Expr>`

§ **Exponentiation:** `<Expr>**<Expr>`

The five mathematical operators are presented here together, for clarity. There are some differences between these – chiefly, the addition operation performs a concatenation if either of the operands used are strings; for instance, adding "test" and "1" will create the string "test1". If both operands are numeric or Boolean, then a standard addition will be performed – note that in addition, as in all cases, the result will be a float if one at least one operand is a float and will be an integer otherwise. Subtraction, multiplication, division, and exponentiation (which evaluates to the C function pow()) can be used as expected and cannot be used with strings.

§ **Concatenation**: `<Expr>,<Expr>`

Not to be confused with string concatenation, the concatenation operator is used to separate two values in a sequence. This can be used to 1) create a list of entries, or 2) internally by the parser to separate parameters from other scripting commands.

§ **Subsequence:** `<StringExpr>/<StringExpr>` or `<StringExpr>|<StringExpr>`

The subsequence (/or |) operator is specifies a group of related items within a single sequence, and can be used to specify a list within a list. It is important to note that the subsequence operator is the same as the division and or operators, and will be used if one or more of the expressions involved are strings. For example, `1/2/3` will be evaluated as a numerical division operation, while `1/2/3:string` will utilize the subsequence operator.

§ **Enumeration:** `<Min>..<Max>`

(AKA list operator or .. operator)

The enumeration operator is used to create a sequence of integers between the values of `<Min>` and `<Max>`. The primary use for the enumeration operator is to create comma-separated lists of integers quickly and concisely without having to utilize a loop.

```
<var name="timeperiod_list" value="1..5" />
```

In the example above the enumeration operator is used in the sequence 1..5, which will generate the sequence 1,2,3,4,5. Further examples, re-illustrating the principle of left-to-right evaluation, are provided below:

```
3+1..5 -> 31,2,3,4,5

(3+1)..5, -> 4,5.
```

Logical Expressions

- **Logical and:** `<Expr>&&<Expr>`
- **Logical or:** `<Expr>||<Expr>`
- **Exclusive or:** `<Expr>^<Expr>`
- **Negation:** `!<Expr>`

The logical operators are defined together here for convenience. In the case that one of the operands is a string, all the given operations will fail. Otherwise, all values will first be converted to Booleans and then the expression will be applied. In all cases, if the expression evaluates to true, the Boolean value 1.0 is returned, otherwise 0.0 is returned.

It is important to note that although Construct will be able to interpret it, the `&&` operator is not a standard XML token, and thus certain text and XML editors may warn that your syntax is incorrect.

- **Equality:** `<Expr>==<Expr>`
- **Inequality:** `<Expr>!=<Expr>`
- **Less than:** `<Expr> < <Expr>`
- **Greater than:** `<Expr> > <Expr>`
- **Less than or equal to:** `<Expr> <= <Expr>`
- **Greater than or equal to:** `<Expr> >= <Expr>`

The comparison operators are listed together here for convenience. If either value the equality or inequality operations is a string, both sides of the equation are first converted to strings, and then the comparison occurs. The less than, greater than, less than or equal to and greater than or equal to operations will all fail if one or more of the operators are a string. If one or more of the values are a float for any of these operations, then all values are converted to floating point values before the comparison is completed. Finally, if both operands are integers, then an integer comparison will be applied.

In all cases, it is important to keep in mind two things. First, if the expression evaluates to true, a Boolean value of 1.0 is returned, otherwise, 0.0 is returned. Second, recall that Construct does not implement operator precedence, and continues with right-to-left evaluations. Thus,

evaluating the expression `2+1==3` will result in a value of 2, because the script will compare 3 and 1, generating a value of 0, and then add this amount to 2. Though we only show this for the == operator, the same holds for all others here.

As with the expression `&&` above, standard XML editors do not allow for the use of < or > (less than or greater than) in places outside of tags, and thus including these expressions in your ConstructML may cause your editor to warn you that you have an error.

Generating Random Numbers

§ generate random number from a uniform distribution:

```
randomUniform(<MinExpr>,<MaxExpr>)
```

§ generate random number from a normal distribution:

```
randomNormal(<MeanExpr>,<VarianceExpr>,<MinExpr>,<MaxExpr>)
```

§ generate random binary values of 0 or 1:

```
randomBinary
```

These three functions allow for the creation of random numbers. In all cases, a new value is generated each time a call to this expression is made, and thus, for example, would generate a unique value for each turn if placed within an expression evaluated on each turn. The random number generator utilized is the same one used by Construct, and hence utilizes the same seed. The random number generator generates a new random number each time it is invoked, meaning that the expression is evaluated as Construct is executed and not when the statement is parsed.

The `randomUniform` expression generates a randomly drawn floating point value from the uniform distribution defined by the parameters `<MinExpr>` and `<MaxExpr>`, which can be any values that evaluate to either integer or float values. If they are not supplied, (e.g., if the expression is written as `randomUniform()`, the default values assumed are 0 and 1. Thus, a call will generate a value inclusive of the minimum and maximum values given. If an integer is desired, for example, between two and five, the user can utilize a call of the form `randomUniform(2,6):int`.

The `randomNormal` number generator generates a float value from a normal distribution with mean `MeanExpr` and variance `VarianceExpr`, and the `MeanExpr`, `VarianceExpr`, `MinExpr`, and `MaxExpr` expressions can be anything evaluating to either a float or an int. If no minimum or maximum values are specific, the range of possible values can theoretically go from negative infinity to infinity. Note that these need not be symmetric, but that because there is usually little need to evaluate infinity, it is often desirable to bound the distribution by something. To adhere to the bounds, Construct uses post-processing – that is, it repeatedly draws random numbers from a normal with the specified mean and variance until it finds values within

the desired range set by the minimum and maximum values, inclusive. Note that in certain cases, this may be a very slow process.

The `randombinary` generator produces a value of 0.0 or 1.0.

Conditional Statements - IF

§ If expression:

```
if(<BoolExpr>) { < Expr> } else { <Expr> }
```

or

```
if(<BoolExpr>) { < Expr> } else if(<BoolExpr>) { < Expr> }
else { <Expr> }
```

The `if` (and subsequent `else if`s and `else` statements) allow the scripting language to evaluate a series of Boolean expressions. These expressions are evaluated sequentially, starting with the first expression (which must be an `if`) through zero or more else if conditions to a final (and required) `else` command. If any of the <BoolExpr> within one of these is true, then the statement within the curly brackets is executed, and the rest of the conditions are ignored. Thus, if expressions will execute only a single expression (or set of expressions) enclosed within curly brackets.

> *Note that there are two significant departures from C-like syntax in Construct's version of an if statement.*
>
> *First, the researcher must use curly brackets when expressing a statement to be executed after an if or an else (note in C-like languages, this is not necessary for single-line expressions).*
>
> *Second, there cannot be an if statement without an else statement – all curly brackets in an if expression that are not part of the final else must be followed by an else (which may be part of an else if).*

Thus, in the case that the user wants to test a single condition, the syntax would look something like the following:

```
if(<BoolExpr>) { < Expr> } else { < Expr> }
```

When testing a conditional inside the parentheses, it is necessary to have as output an explicitly Boolean value. Thus, implicit conversion will not occur for floats or integers, to reduce the possibility of user error. If a user wishes to use an integer, float, or string for the conditional, they must explicitly cast with `:bool`. Finally, the returned types of all expressions executed should match.

§ Static if expression:

```
static_if(<BoolExpr>) { <Expr> } else { <Expr> }
```

or

```
static_if(<BoolExpr>) { <Expr> }
else if(<BoolExpr>) { <Expr> } else { <Expr> }
```

The `static_if` expression differs from the standard if expression in that it is evaluated statically – while if conditional is considered when the statement is executed, the `static_if` is executed at the time at which it is parsed. This can be used in cases where the experimenter is sure that the conditional statement will never change, and in these cases will dramatically speed up execution time, as a `static_if` will be evaluated only once. Such a situation might occur if the user were to test for some constant variable that may be changed once, by the user, in the file, but will stay constant throughout the simulation.

The second use if the `static_if` is that if utilized, only the expressions within the brackets {} of the conditional evaluating to true will be evaluated. Thus, one can introduce whole sections of code conditional on whether a variable is initialized to a certain value, where if it is not, that code will never be utilized by Construct. Simply put, however, the difference between if statement and a static_if is portrayed best in the following example: Consider the two,

```
if(timeperiod > 0)…
static_if(timeperiod > 0)…
```

In the case of the if statement, the condition would be evaluated each turn of the simulation – thus at every time period after the first, the code within the if statement would be run. In contrast, the static_if would be checked one time, when it was parsed. Because it evaluates to false in that case, the code will never be run.

§ Assignment: `$variable$ = <Expr>;`

The assignment operator, or the equals sign, allows the assignment of the value on the right hand side to the variable on the left, which must be surrounded by dollar signs. The right hand side of the equation can be any expression, though note that it must end with a semicolon (;). This expression can include any variables declared previously that have already had values assigned to them.

Upon assignment, the variable with name `variable` will be given the type of the type for whatever the right hand side evaluates to. Assignments take on a global scope within the ConstructML attribute they are defined in – thus, unlike, for example, C, a variable defined as such inside a loop can be utilized outside of it. However, once outside of an attribute, the variable loses that definition – even within the same element, a variable declared in its "name" attribute will be different than one defined in its "value" attribute.

When a variable is used, it is given a variable type. If the right-hand side expression is a Boolean the first time the variable is initialized, the variable will be type as a Boolean. Otherwise, if it is an integer, float, or string, the variable will be typed as an integer, float, or

string (respectively). The most specific type that can be used for a variable will be used to type the variable. If a specific variable type is to be used, the right-hand side can be cast to the desired type using the cast (:) operation.

An additional point to note is that it is necessary to declare any variable first declared on the left-hand side of an assignment using a `with` variable, as is done in the example in Table 19 below with the variable `result`. As can also be seen in Table 19 below, it is necessary to include a return statement in the script to specify which result will be returned. If the end of a script does not contain a return statement, the parser will error.

Table 19. Examples of foreach loops.

Variable	Value

<var name="loop1" value="
$result$ = ''; /* init return var */
foreach i ('a', 'b', 'c', 'd') {
$result$ = $result$ + i;
}
return $result$;"
with="$result$"/>
	"abcd"

<var name="loop2" value="
$result$ = ''; /* init return var */
foreach i ('e', 'f', 'g', 'h') {
$result$ = $result$ + i;
}
return $result$" /* as last command in script, semi-colon optional */
/> <!-- with optional in this case -->
	"efgh"

<decision name="loop2" value="foreach col (0..10) { setKnowledgeNetwork[row:int,col:int,0] }" with="row=2"/>	(sets network row [2,0-10] cells to zero)

§ Error: `error(<StringExpr>)`

The error expression will force Construct to output the string given and then exit immediately after being evaluated. Typically, one will want to use this to debug code to make sure that "impossible" conditions within the code are never hit. If a string expression is not given, a default message of "<no error message provided>") will be returned. Note that the string expression can be a quoted literal but can also be a dynamically evaluated string variable.

Looping - Foreach

§ Foreach expression: `foreach $iterator$ (<IterableExpr>)`

`{ <Expr> }`

The foreach loop allows the user to give a list that can be iterated over to produce an aggregated result. To use a foreach loop, one must specify the `foreach` keyword, then the name of the parameter while will be used to iterate over the list enclosed by dollar signs, followed by the parentheses enclosing what is to be iterated over, and then finally the statements to be run for each element in the list within the brackets {}. For example, the expression "`foreach $val$ (1,2,3)`" will generate a parameter, val, which will be given a value of 1, 2 and then 3 on the first, second and third iterations of the loop, respectively.

Within the brackets, a sequence of any number of statements can be written – in most cases, these statements will include reference to the iterator parameter, and in many other cases will use variables, such as an aggregate variable, outside the loop as well. However, loops can also use the set* operations and therefore will not always need such an aggregate value.

In the case that the user does not, under certain conditions, want to iterate through the entire loop, a return statement, which will break the loop and return a value from the script immediately, can be used within an if statement. In addition, it is important to note that foreach loops can be embedded within other results, and that results coming from a foreach can be used outside of the loop.

Return

§ Return: `return <Expr>;`

The return statement allows for a script to return a value at any point during its execution- it is mostly intended for use with complicated scripts. All return statements must begin with the word `return` but must only have a trailing semicolon *if they are not placed at the end of a script*. Perhaps most importantly, the user must note that in ignoring whitespaces, Construct will interpret something of the form "return_val" as meaning the need to return the variable _val, and thus should not be used. However, expressions after the return statement, such as return $count$+1; will evaluate correctly (i.e., in this case will return the value stored in count plus one).

Another important point is that statements which contain assignments if statements or foreach must contain a return statement so that the script in its entirety returns a value. In the case where this does not happen, Construct should error. Finally, values evaluated as part of an expressionvar are considered to be part of the script- thus, any return statements in the expression variable will serve as returns for the entire script.

Macros

§ Macro variable expression: `$<Name>$`

Macros may be the most important tool in developing an extendible and maintainable deck, but also may be the most confusing to the reader. A macro is defined in two parts. First, within the script, an identifier (name) for the macro is placed between two dollar signs. Like in all cases

in Construct, this identifier should be limited to alphanumeric characters. Second, the variable value for the macro should be specified – in most cases, this will occur in the `with` tag of the enclosing piece of ConstructML. The `with` tag must address the same identifier, and the second dollar sign must be immediately followed with an assignment operator.

The results of macro expansion directly affect the text of the expression. So, for example, in Table 20, the expansion of the variable `$i$` is converted to an integer and incremented in variable `x1`. In contrast, the substituted variable can also create a new lexeme – in variable `x2` the expansion will substitute in the value 1 for `$i$`, creating a new variable, `construct::intvar::x1`.In turn, when this is expanded and evaluated, `x2` will then take on the value of `x1`.

Table 20. Examples of macros.

Variable	Variable of Interest	Value
`<var name="x1" value="$i$:int+1" with="$i$=1"/>`	x1	"2"

`<var name="x1" value="$i$:int+1" with="$i$=1"/>`
`<var name="x2" value="construct::intvar::x$i$"`
`with="$i$=1"/>`
	x2	"2"

`<var name="x1" value="$i$:int+1" with="$i$=1"/>`
`<var name="x2" value="construct::intvar::x$i$"`
`with="$i$=1"/>`
`<var name="x3" value="construct::intvar::x`
`$2*i:int$+1" with="$i$=1"/>`
	x3	"3"

`<var name="x$i$" value="$i$" with="$i$=(4,5)"/>`	x4, x5	"4", "5"

In most cases, variables should be defined inside of the with tag, as is the case with `x1`, `x2`, and `x3`. Note that the value of the macro variable must be written as a string but can easily be cast to any desired type. Also, note that it is possible to have several macro variables each having separate variable lists –thus, if there are three values for macro `$i$` and four for macro `$j$`, then twelve different expansions will be performed.

It is important to remind the reader that not all values surrounded by dollar signs are macros – for example, variables used in assignment operators may be modified dynamically as the script is evaluated, and such variables are thus usually specified as with parameters to ensure that they are recognized by the parser. The difference here is that macros, being defined by the parser, are static and cannot change during execution but may be embedded in more complex variable names. Finally, note that variables created via macros can be accessed in the standard way- for example, we see that the last example gives us x4 and x5, not a variable with the name xi.

Get/Set Network Values

§ **Get network value:** get<NetworkName>[<RowExpr>,<ColExpr>]

§ **Set network value:** set<NetworkName>[<RowExpr>,<ColExpr>,<ValueExpr>]

The word get, when followed immediately by the name of a network in CamelCase (e.g., getKnowledgeNetwork) retrieves the given value from a specific location in a network. The location is indexed by two integers in brackets, where network value operation retrieves the value as a specific location in a network. For this expression to function, the network must exist, the row and column values given must be integers, be enclosed by bracket characters ([]) and be separated by a comma (,). The returned value will have the type of the network the call is made to. Also, it is important to note that these calls are somewhat time intensive, and thus the user should take care when making such calls repeatedly, such as making them inside loops or at every turn of the simulation. Finally, note that variables cannot yet be initialized in this way, as variables are currently initialized before networks.

Similarly, the word set, followed immediately by the name of a network in CamelCase, can be used to change the value at a given row and column in a network. In this case, it is important to verify that the value given in <ValueExpr>, which will be the value that this row and column are set to, is of the same type as the network it is being set in. Otherwise, Construct will exit with an error. Note that this is the case even where an implicit cast would make sense – i.e., from integer to Boolean. It is also important to note that the set command returns a value, which is equal to <ValueExpr>.

§ **Get aggregate network values:** get<NetworkName>[<RowExpr>,<ColExprString>]

This version of the get expression will simply call get on a single row in a table and a series of columns, given by the indices listed in the list ColExprString, a string containing a series of comma-separated integers. The value returned by this function is the sum or concatenation of these values. A common usage of the aggregate get call is to get the network values for a specific group of agents or facts – for example, we can get the number of facts in the knowledge group G that agent 0 knows with the call:

```
getKnowledgeNetwork[0, construct::knowledgegroupvar::G]
```

§ **Set aggregate network values:**
set<NetworkName>[<RowExpr>,<ColExprString>,<ValueExpr>]

The set aggregate network values operation analogous to the set expression, except with a list of columns. In this case, the value returned is the summation or concatenation of all values set. Additionally, one should note that if the expression ValueExpr references a dynamic value, such as a call to a uniform random number generator, then the value will be recomputed for each element in <ColExprString>.

ReadFromCSVFile

§ Get value from csv file:

```
readFromCSVFile[<FileExpr>,<RowExpr>,<ColExpr>]
<var name=”param_val_col” value=”1” />
<var name="attack_prob" value="readFromCSVFile[params.csv,0,
    construct::intvar::param_val_col]:float" />
```

This command reads a value from a CSV file named params.csv in the example above. The file location is relative to the location of the Construct execution directory. The file must also have a value at location `<RowExpr>,<ColExpr>` (row 0,column 1 in the example above). If all these conditions are true, then Construct will return the value at the given row and column of the CSV, otherwise, it will exit with a failure. Note that file input can be extremely time-intensive, and thus this command should be used with care.

Appendix E Construct in High Performance Computing (HPC) Environments

In many ways, the resource we are concerned when we do simulation shifts from the person-hours necessary to complete surveys and in-depth interviews to computational complexity in both time and space. In particular, the goal is to be able to complete a large-scale simulation project with the idea of "single-click" from starting the simulation through result generation, and with an implementation that allows us to quickly tweak simulation parameters and rerun all simulations.

To understand the difficulties associated with simulation in a large-scale project, we now present the scenario we faced in a previous experiment, described in more detail in Carley and Maxwell (2006). In this project, we were faced with approximately 2,000 runs, each of a population of 4,000 agents, along with their attributes, their initial knowledge, and the associated social network. This model, perhaps one of the most complex social simulation models run in Construct, took nearly five hours per run. Thus, the sequential cost of running these simulations for a single researcher on a single processor is just about enough time for a research grant to expire. Luckily, a series of innovations in computing over the past fifty or so years, with which most of us are familiar have saved us from such a fate. In this section, we detail such innovations for the interested user, and then give examples of how to utilize the tools for HPC environments employed at CASOS.

The first innovation, of course, is the ability for computers to talk to each other. This allows us to use a single terminal to run simulations on other computers at our disposal and have them return the results. The second innovation was the development of multi-core processors and computers with multiple processors. Because Construct, by default, runs on a single core of a processor, we can not only run our simulations on other computers, but to run multiple simulations on each of them at the same time, independently of each other. The computing power of our center is likely better than most settings, but by no means ideal. Upon the running of simulations for Carley and Maxwell (2006), our center possessed 234 processor cores upon which simulation runs could be done, though many of these cores were being intermittently used by other members of our research center.

The final innovation of computer science, the MapReduce framework (Apache Software Foundation, 2019; MapReduce, 2020;), answers the question of how we can "black-box" both the distribution of simulations and the coalition of their output to various machines that can be potentially interrupted at any time. In its most basic definition, the MapReduce framework "maps" out simulations to different machines, ensuring in some way that we will receive output from each machine, and "reduces" all our output to a single format which we can specify.

Several open-source packages exist to implement the MapReduce framework on computers that researchers have available to them. Importantly, such a framework allows the researcher to be ignorant of the number of processors he or she has available – the MapReduce concept works

in the same way (though with obvious time increases) on a single core as it does on the millions of cores used by companies such as Google. We use the HTCondor (formerly Condor) High Throughput Computing (HTC) software (https://research.cs.wisc.edu/htcondor/) to connect machines in our center, and their DAGMan (Directed Acyclic Graph Manager) (https://research.cs.wisc.edu/htcondor/dagman/dagman.html) application, along with some straightforward scripting, to implement the MapReduce framework.

The MapReduce framework, along with some well-known interventions, allow our workflow to have two vital properties. First, the given workflow maximizes the resources available to the researcher. A problem which could have naively taken, even under ideal computing circumstances on a single machine, months to complete, has been reduced to a few days at most. Indeed, a researcher need not even obtain more machines, as with the advent of cloud computing, they can access technologies that hide all implementation details of the MapReduce framework and give cheap access to an unlimited supply of machines, such as Amazon's EC2 cluster. Indeed, workflow technologies like SORASCS (Schmerl 2011) are rapidly evolving to allow full workflow to be completed without a research having access to anything other than a single computer and the Internet. If the researcher does have a large supply of machines available, such speedup has been achieved with free, open-source, easy-to-install technologies.

Having explained, at a high level, the concepts incorporated in running Construct in parallel on multiple machines, it is now useful to describe in more detail how such tools can be utilized. The first objective, of course, is to obtain some way of submitting Construct runs to multiple machines. Here, we will discuss the HTCondor cluster framework implemented at CASOS. The first step, of course, is to install HTCondor onto machines in your cluster- this step is not covered here but is described in detail in the HTCondor setup manual, located at https://htcondor.readthedocs.io/en/latest/index.html.

Once installed properly, a machine with HTCondor installed on it and a user with submission privileges from that machine can submit jobs from that machine onto the cluster in a series of simple steps. First, the user should set up a CSV file with the parameters indicating the conditions of the experiment they would like to have changed. From here on out, we will refer to this file as the *conditions file*, to represent the fact that it holds *all the conditions necessary for the entire experiment*. We will differentiate this later with a *parameter file*, which holds the *conditions necessary to run a single cell of the experiment*. In a trivial experiment, where the goal is to test an effect on different population sizes, the conditions file would look something like this:

```
AgentSize,10,100,100
```

The first column of the file simply labels the condition being changed - though this is not necessary (we will never tell Construct to look at this value), it is naturally useful in keeping track of which lines of your parameters file refer to which condition. Once this parameter file has

been specified, we need some way to submit (in this case) three different runs to multiple machines via HTCondor. To do so, we need to complete three further steps.

The first step is to create three different parameter files - one for each of the different conditions. This can be done using your favorite scripting language. Below, we give a simple example, in Python, which reads a conditions file and generates a parameter file (recall that a parameter file is simply a set of conditions necessary to run a single experiment) in a directory whose name specifies the conditions for that directory. (Note that if you are not comfortable doing such programming, for small experiments, it is quite easy to do this step manually).

```
import csv, itertools, os
with open("conditions_file.csv", "r") as condFile:
   reader = csv.reader(condFile)
   values = []
   conditionTitles = []
   for line in reader:
      conditionTitles.append(line[0])
      values.append([val for val in line[1:] if val != ""])
   experimentalSet = list(itertools.product(*values))
   numVals = len(conditionTitles)
   for experiment in experimentalSet:
      condsString = '_'.join(str(i) for i in experiment)
      os.mkdir(condsString);
      with open(os.path.join(condsString,"params.csv"), "w") as paramFile:
         for i in range(numVals):
   paramFile.write(conditionTitles[i]+ "," + experiment[i] + '\n')
```

To run this script, place it in the same directory as your conditions file, name the conditions file “conditions_file.csv”, and use Python (this example was written for Python version 2.7) to run the script. For information on how to download Python and run a script, consult the Python documentation at https://www.python.org/.

Assuming you use the same methodology suggested in the script above, you will now have the following in the directory in which you placed your conditions file and ran the script: your conditions file (conditions_file.csv), the python script (your_naming_of_python_script_above.py) and three Folders 10, 100, and 100, each with one file called params.csv. The second step to submit to HTCondor is to develop your model (i.e., the XML file described above) and to allow the model to read in as a parameter from a CSV file the conditions you are interested in. In this case, we would change the “`agent_count`” variable to be instantiated as follows:

```
<var name="agent_count" value="readFromCSVFile[“params.csv”,0,1]/>
```

As we know from the above sections, this tells Construct to read the agent_count variable from the first (zeroth) row and the second (zero-indexed) column of the csv file “params.csv”. Once we have done this, we can add our XML file to the directory we are working in (i.e., at the same level as conditions_file.csv). Note that this implementation will only require us to have a

single model file, which is desirable with respect to person-hours required to change the model and the amount of space needed to store results.

The final step is to create a *submission file* that HTCondor will use. Though we do not detail in depth the details of HTCondor submission, below is a file that, placed at the same level of the directory as your XML model file, will allow you to run the simple experiment described here. Note that you should replace `YOUR_MODEL_FILE_NAME.xml` with the name of your XML file and include a construct executable with the name "Construct.exe" in your directory as well.

```
universe       = vanilla
requirements   = ((ARCH == "INTEL" || ARCH=="X86_64") &&
((OPSYS == "WINNT51") ||(OPSYS == "WINNT52") || (OPSYS == "WINNT61") ||
(OPSYS == "WINDOWS"))
should_transfer_files   = YES
when_to_transfer_output = ON_EXIT
executable        = Construct.exe
transfer_executable  = true
notification      = Never
arguments         = YOUR_MODEL_FILE_NAME.xml
output         = out_setup_to_construct.txt
error          = err_setup_to_construct.txt
log   = condor.log
transfer_input_files    = params.csv
initialdir        = 10
queue
initialdir        = 100
queue
initialdir        = 1000
queue
```

The file, generally, tells HTCondor where to find your executable and model file, and then to run three times in each of your experimental directories, using the parameter file within that directory. This file also contains requirements for what operating system to run on, and specifies that all files written out by Construct (e.g., in ReadGraph operations) should be transferred back to your machine after they are run. Putting the text above into a file called "condor_submission.sub" and assuming the PATH variable on your machine includes the HTCondor executables, opening a command prompt, changing to the directory we have discussed here, and typing in the following will run the given experiment.

```
THIS_DIRECTORY> condor_submit condor_submission.sub
```

You can use other HTCondor programs, such as `condor_q` to check the status of your runs - for full details, see the Condor manual (https://htcondor.readthedocs.io/en/latest/man-pages/condor_q.html) and (Knoeller, 2013).

Appendix F Construct in Research Literature

Below are some brief descriptions of projects that used Construct. Links to the full publications and project sites are available in the References section.

Predicting Intentional and Inadvertent Non-compliance

By: Kathleen M. Carley, Dawn C. Robertson, Michael K. Martin, Ju-Sung Lee, Jesse L. St. Charles, Brian R. Hirshman (Carley et al., 2010)

Models for predicting intentional and inadvertent errors on tax returns were developed using two approaches: the first was metamodeling using literature on errors, and the second was using statistical machine learning to derive a model from tax audits. The reliability of the models is dependent on the amount of data, the quality of the data, and whether the learning techniques are supervised or unsupervised. IRS audit data does have some reliability issues; the taxpayer's motives are unknown at the time of filing, and the standard is high for proving intentional misreporting. The models take these biases into account through an ensemble modeling approach. The methods shown in this study are useful in creating a predictive model of taxpayer behavior.

Agent Interactions in Construct: An Empirical Validation using Calibrated Grounding

By: Craig Schreiber, Kathleen M. Carley (Schreiber & Carley, 2007)

Schreiber and Carley conducted a validation study for Construct. The focus of the study was on the ability of Construct to produce an initial state of agent interactions which resemble how a real world network communicates. The calibrated grounding technique was used to validate the model. Construct was shown to produce a valid initial state of interactions.

Computational organization science: A new frontier

By: Kathleen M. Carley (Carley, 2002)

According to synthetic adaptation, any entity that is composed of intelligent, adaptive, and computational agents is also an intelligent, adaptive, and computational agent. Organizations are inherently computational because of synthetic adaptation. The behavior of groups and organizations can be explained by using multi agent computational models that are composed of intelligent adaptive agents. Construct is an example of such a model; by combining a network with a multi-agent approach, the model becomes more realistic. A series of virtual experiments use this model to show the power of this approach for analysis of societies and organizations.

A Dynamic Network Approach to the Assessment of Terrorist Groups and the Impact of Alternative Courses of Action

By: Kathleen M. Carley (Carley, 2006)

Dynamic network analysis is based on the collection, analysis, understanding, and prediction of dynamic relations amongst various entities such as actors, events, and resources, and their

impact on individual and group behavior. Using dynamic network analysis, terrorist groups were examined as complex dynamic networked systems that evolve over time. The use of dynamic network analysis tools to analyze a terrorist group is demonstrated. Techniques that are demonstrated include identifying sphere of influence amongst actors, determining emergent leaders in the network, and how using network metrics can assess the impacts of various actions within the group.

Modeling Complex Socio-technical Systems using Multi-Agent Simulation Methods

By: Maksim Tsvetovat, Kathleen M. Carley (Tsvetovat & Carley, 2004)

To study complex social and technological systems, underlying psychological and sociological principles, as well as communication patterns and technologies within these systems must be measured and understood. The creation of high fidelity models of these systems requires a combination of analytical models with empirically grounded simulation, to form multi agent systems. These multi agent systems incorporate learning algorithms as well as other social network phenomena. The power of these methods are demonstrated by creating a multi-agent network model of networks such as terrorist organizations. This ultimately creates a generalizable and valuable process for analyzing complex social systems, by using AI algorithms combined with an analytic approach.

On the Coevolution of Stereotype, Culture, and Social Relationships: An Agent-Based Model

By: Kenneth Joseph, Geoffrey P. Morgan, Michael K. Martin, Kathleen M. Carley (Joseph, Morgan, et al., 2014)

The theory of constructuralism describes how shared knowledge, representative of cultural forms, develops between individuals through social interaction. Constructuralism argues that through interaction and individual learning, the social network (who interacts with whom) and the knowledge network (who knows what) coevolve. In the present work, we extend the theory of constructuralism and implement this extension in an agent-based model (ABM). Our work focuses on the theory's inability to describe how people form and utilize stereotypes of higher order social structures, in particular observable social groups and society as a whole. In our ABM, we formalize this theoretical extension by creating agents that construct, adapt, and utilize social stereotypes of individuals, social groups, and society. We then use this model to carry out a virtual experiment that explores how ethnocentric stereotypes and the underlying distribution of culture in an artificial society interact to produce varying levels of social relationships across social groups. In general, we find that neither stereotypes nor the form of underlying cultural structures alone are sufficient to explain the extent of social relationships across social groups. Rather, we provide evidence that shared culture, social relations, and group stereotypes all intermingle to produce macrosocial structure.