Construct User Guide

Kathleen M. Carley, David T. Filonuk, Kenny Joseph,
Michael Kowalchuck, Michael J. Lanham, Geoffrey P. Morgan

May 2014
CMU-ISR-14-105

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Supercedes "Construct User Guide" CMU-ISR-12-112, November 2012

This work was supported in part by the IRS project in Computational Modeling, the Air Force Office of Sponsored Research (MURI FA9550-09-1-001 mathematical methods for assisting agent-based computation), and the NSF IGERT in CASOS (DGE 997276). In addition support for Construct was provided in part by Office of Naval Research (N00014-06-1-0104 and MURI N000140-81-1-186 a structural approach to the incorporation of cultural knowledge in adaptive adversary models), and the National Science Foundation (SES-0452487). Additional support was provided by the Air Force Office of Sponsored Research (MURI N00014-08-1-1186 cultural modeling of the adversary). Further support was provided by CASOS - the Center for Computational Analysis of Social and Organizational Systems 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 Internal Revenue Service, the National Science Foundation, the Office of Naval Research, the Air Force Office of Sponsored 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 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 intended both as an introduction to Construct for casual modelers as well as a reference guide for researchers, modelers, and simulationists.

Construct User Guide

INTRODUCTION

Construct is an agent-based network-centric simulation. Although frequently lumped together, agent-based simulations 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 is also 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. Most of the highest-fidelity models constrain interaction to explicit messages, if at all, and many work 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. Construct 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 has 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.

But 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.

Table of Contents

Table of Figures

Table of Tables

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 at www.casos.cs.cmu.edu. 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 CMU-CASOS 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 Report

Where feasible, this technical report quotes a provided example of a Construct experiment configuration file. The file is also included in a 2-up printed format in Appendix A and is also available for download at the Training and Sample Data page on the Construct page of CASOS's website. To help you follow along, this report uses a few conventions in type face: Construct keywords, such as variable or network names, will use a monospace font to clearly differentiate the example from the surrounding text.
Code snippets will also be written in the `courier new`, as these snippets are quotes from the demonstration input file. We'll also frequently call the input file the input deck, or shorten the name to deck, throughout the document. The origins of this use of 'deck' will deliberately remain in the mists of our collective memory lest the authors prove how old they really are.

Boldface and an exclamation mark (**!**) indicate information the experiment developer and designer, researcher and simulationist should be particularly aware of when using Construct. We'll reduce that string of potential audience members, 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 impact 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 shown in the list below

- **Quick Start Guide** - for a relative quick movement from introduction to execution
- **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 Condor, to brief synopsis of peer-reviewed projects within which 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 through the use of 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 Constructs 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 2.

Construct's Core Mechanisms

Figure 1 offers one depiction of the interior workings of a Construct simulation that helps us scope our motivating example to enable a researcher to rapidly move from introduction to experimentation. Starting at the ten o'clock position and moving clockwise, the reader will note agents without which the remainder of this report and use of Construct is pointless. At the eleven o'clock position, each agent is capable of having mental models (often referred to as transactive memory (Wegner, 1987) of what the agent knows, what the agent believes, and perhaps most importantly, what its alters know and believe. This perception is, also importantly, error prone, personal, and both learned and forgotten over the course of a simulation. The one o'clock position depicts agents embedded in social, communication, and other networks with other agents. Some alters may be as cognitively robust as the egos, while others may represent Information Technology (IT) resources, or mass media (e.g., newspapers, TV, radio). Agents are also potentially aware of stylized representations of social and social-demographic information about themselves and their alters, which shape the agent's decisions during the interaction and knowledge cycle. At the three o'clock position, agents have culture as a consequence of their learning knowledge. Technology, at the five o'clock position, is most often modeled as agents capable of receiving, storing, retrieving, and transmitting knowledge to other agents in the simulation. The five and six o'clock positions in Figure 1 represent the ability of

Construct to incorporate such stresses as personnel turnover and time-dependent task-completion modeling, though we'll defer discussion of those capabilities to Part 2.

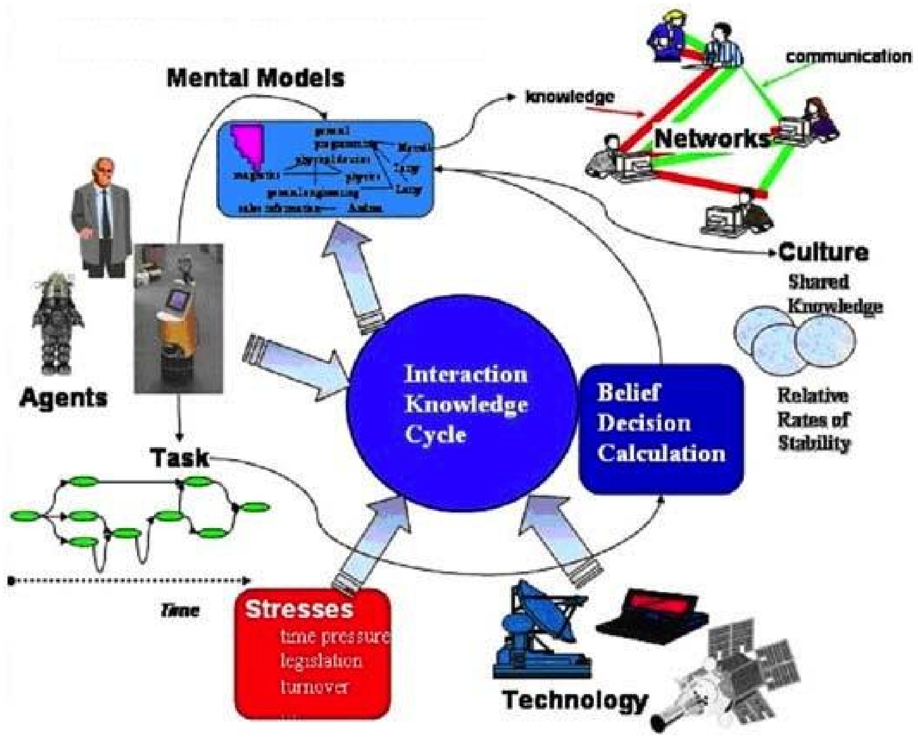


Figure 1. A graphical depiction of the interior workings of a Construct simulation

In the center of the diagram are two blue circles that are, after the calculations to determine which alter, if any, each ego will interact with, the most important components of Construct. The interaction and knowledge cycle represents the process each ego goes through in its decision to interact, or not, with its alters. Each agent's decision takes into account that agent's current knowledge, its current perception of similarity to its egos (knowledge homophily), its current perception of unique knowledge each alter has that the ego does not, as well as the social, physical, and socio-demographic similarity of the ego and alter. On a probabilistic basis, should interaction occur, each agent will exchange messages. The ego and alter both build their message from their own knowledge or beliefs sets or their perception of their own alters' knowledge or belief sets. After message exchange, agents may learn, with and without error, the contents of those messages as well as forget previously learned knowledge that has not been referenced recently.

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 groupsits a deliberate choice 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.

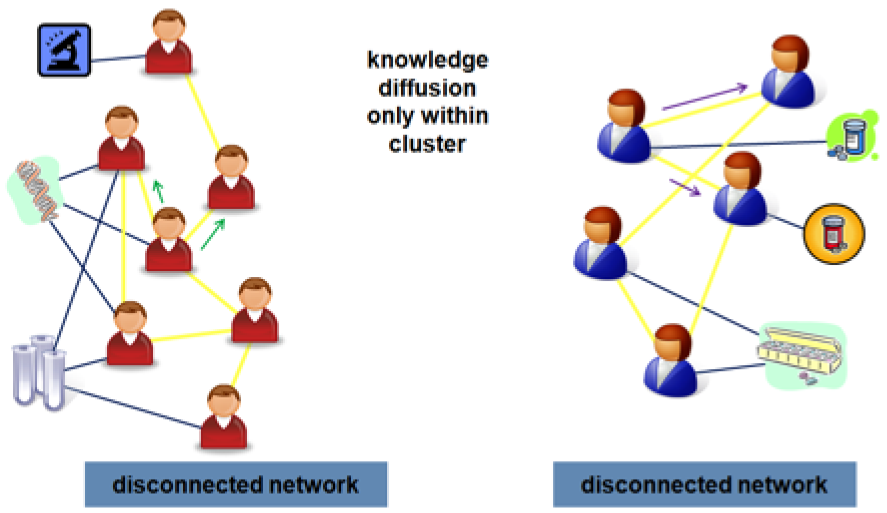


Figure 2. A depiction of two 'clean-room' teams of product developers

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: Construct Quick-Start

This is an introduction to core mechanisms of Construct, 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 Construct. We will then conclude with a suggestion of some experiments that could be done using only those key networks, referencing the motivating scenario.

The Objects

There are five classes of objects in Construct. These are 1) agents, 2) knowledge, 3) tasks, 4) beliefs, and 5) time. A singleton example of each of these object classes is referred to (respectively) as 1) an agent, 2) a knowledge bit, 3) a task, 4) a belief, and 5) a turn.

Agents

Agents are the most important class of objects in Construct. 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'll address briefly here and more thoroughly throughout the report.

Individual agents possess different bits of knowledge and they are aware of other agents. Each person has a unique, error-prone perception of those other agents' knowledge and beliefs that they learn throughout the course of a simulation from some starting condition. This guide discusses how to manipulate both what agents know, who they know, and what they think other people know.

People may be members of groups. Groups are not explicitly defined in Construct but it is useful concept to remember. Group members, like in our motivating example, tend to have many more connections within the group than outside of it. It is usually 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're on, it'd be easier on 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'd quickly confusing if their membership in my hand groups was their most salient characteristic.

Individuals can also have beliefs, and work to do (as described by tasks), and they may not remain unchanged by 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. Construct represents real-world knowledge through a stylized and simplified series of bits (0 or 1). Any particular knowledge bit should represent 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 try and 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.’

Collections of knowledge, which we characterize as expertise, can be assembled by labeling a range of bits as relevant to that larger expertise. The relative size of each range is intended to be representative of the amount of effort required to achieve a given level of expertise. For example, a child’s understanding of the solar system may be represented some 30 facts (the names of the planets, names of interesting moons, relative distances of the planets to the sun, and some representation of relative size), while the requirements of celestial navigation (the role of seasons, star identification, etc) requires a significantly larger set of facts, one that may be estimated usefully if not precisely. We call this form of knowledge specification “stylized knowledge.” Another example of this sizing decision would be if a simulation involves agents with knowledge about recent movies, and also recent literature – a researcher may decide that, because there are fewer movies made than books written in most years, that there is correspondingly less to know and correspondingly fewer bits in the one expertise collection than the other. These sizing decisions may end up being poor modeling decisions, but the researcher must make them and communicate them to the model consumer.

Knowledge can be used to inform the quality of decision-making tasks agents can perform, and also used as evidence either in support of or opposed to a belief, but these connections are outside the scope of this guide.

When a researcher links knowledge to one or more beliefs, the possession of knowledge will impact the strength of the held beliefs as well as the likelihood of changing those beliefs. Beliefs have a more in-depth discussion below as well as in Part 2.

Tasks

Tasks in Construct represent, appropriately, tasks. Specifically, these tasks can best be thought of “decision tasks”, where agents (see previous) need information (see previous!) to perform the task adequately.

Tasks are outside the explicit purview of this quick-start guide, see Part 2.

Beliefs

Beliefs in Construct represent, also appropriately, beliefs. These differ from information because beliefs cannot, it is presumed, be judged for their inherent truth. Also, agents may or may not possess any particular knowledge bit, but they may have believe or disbelieve a belief more or less strongly. Beliefs may or may not be linked to information. Beliefs linked to information are sometimes labeled “Evidence-Based Beliefs”.

Beliefs are outside the scope of this guide, see Part 2.

Time

Turns, in Construct, represent chunks of discrete time. Agents each have some opportunity to interact with other agents during each turn. Agent order is randomized each turn, to avoid agents early in a static order having an unfair primacy advantage. Agents interacting with other agents may not be able to support further interaction. It is usually good practice to attempt to identify, loosely, a length of time with 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).

Time is part of every model, but a detailed discussion of Time is out of scope of this guide.

Their Relations

In Construct, we note how each of these various objects are related to each other through the use of dense matrices. Each matrix, usually referred to as a network, represents a meaningful and distinct tie between objects. Matrix values may be binary (either 0 or 1) or weighted (any real number). 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. 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 so:

	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 of the different matrices present in Construct, their real-world meaning, and their practical impact within Construct. This guide will focus on three key matrices: the interaction sphere, the knowledge network, and transactive memory. 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 are NOT connected within the interaction matrix, they will never be able to directly interact with each other. Agents who are connected in the interaction matrix may still never interact.

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_node class_type="agent" target_node class_type="agent"
id="interaction sphere network" link_type="bool" network_type="dense">

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

This is your first jolt of Construct XML, so it may seem a little daunting at first, but let's attempt to parse this XML line by line.

```
<network src_node class_type="agent" target_node class_type="agent"
```

The network at the beginning indicates we're defining one of the matrices used in Construct. The argument src_node class_type tells us that the matrix we're defining should have rows defined by agents, and the target_node class_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, this name is important to 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">
```

Typically, most networks will have this argument network_type set to dense. This means that every possible cell combination should be defined.

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

The generator is a new object, it's being defined to help us fill in the values of the interaction sphere. There are different generator types - this one, 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 in Appendix D.

```
<rows first="0" last="node class::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 indice 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, after all, counted out five numbers (0,1,2,3,4). 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's handy shorthand so you don't need to keep track of how many agents exist.*

All generators assume (except one) 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 (this is covered in the CSV generator section). Thus, if you want two or more groups of agents, you may want to keep track of the start and end of those groups. This is why it's easier, almost always, 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="node class::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 1 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?

If we set this to a different value, such as “0.20”, what would that mean for the simulation? It would mean that each agent would be able to interact with roughly 20% of the potential alters. How is this calculated? If the implicit uniform random number generator generates a value less than 0.2, it will output a one, otherwise it will ouput a zero – this mean is as accurate as any mean when evaluated in the context of the Law of Large Numbers, not necessarily true for a particular set of generated numbers.

```
<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 don’t alway have access to the boss. If the president wants to see me, he will, but I can’t 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 current generator with a mean parameter set to “1.0”? How about when the mean parameter is set to “0.20”? Multi-mode matrices should not have the symmetric_flag set to true.

</generator>

This indicates that the generator has been fully defined, 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’s 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_node class_type="agent" target_node class_type="agent"
id="interaction sphere network" link_type="bool" network_type="dense">
      <generator type="randombinary">
            <rows first="0" last="node class::agent::count_minus_one"/>
            <cols first="0" last="node class::agent::count_minus_one"/>
            <param name="mean" value="1.0"/>
            <param name="symmetric_flag" value="false"/>
      </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 100% of the cells of this matrix. The generator is functionally but not explicitly symmetric."

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. Construct updates the knowledge network throughout the run of a simulation.

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_node class_type="agent" target_node class_type="knowledge"
id="knowledge network" link_type="float" network_type="dense">
      <generator type="randombinary">
            <rows first="0" last="node class::agent::count_minus_one"/>
            <cols first="0" last="node class::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.

Transactive Memory

Transactive Memory is how Construct implements perceptional differences from reality. In simulation, if agents receive state information directly from the simulation, then they have no "perceptual filter", rose-colored or any other shade. Humans, however, must perceive signals from their senses and grapple with that signal to make sense of it, to turn it into symbols. For example, if my stomach feels empty and I hear it growling, I may eventually realize that I am hungry. Retreating from larger philosophical issues, perception is an important source of human error. Thus, most simulations that attempt to address human-like behavior have some sort of perceptual mechanism. In Construct, that perceptual mechanism is transactive memory. The following figure displays an example of transactive memory.

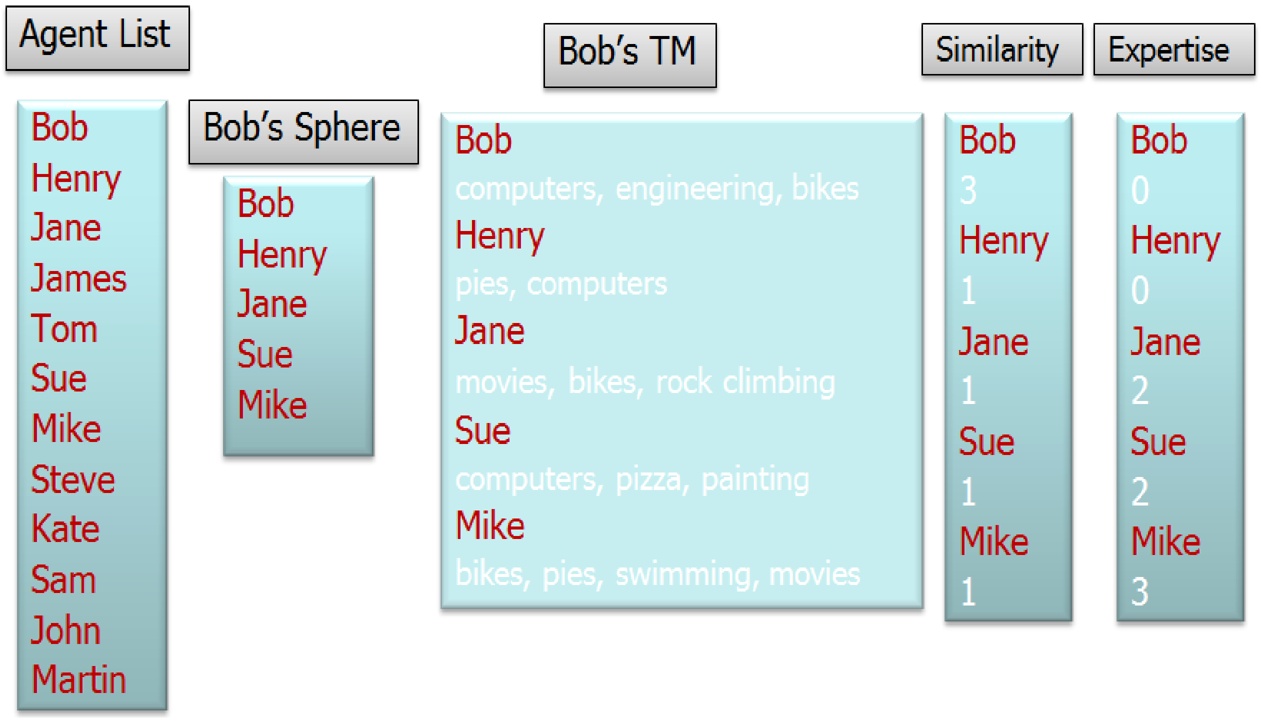


Figure 3. Bob's Transactive Memory

But what is transactive memory? It is a three dimensional matrix, representing what every agent (A) thinks every other agent (A) knows (K) or believes (B). There are, currently, two separate transactive memory matrices, a knowledge transactive memory (A x A x K) and a belief transactive memory (A x A x B). In Construct's implementation of Transactive Memory, ! **each ego maintains transactive memory only of alters it is connected to in the interaction sphere.** ! Agents do not necessarily (and often do not) have a good grasp of what other agents actually know. You have probably met people that thought you knew things you didn't, or, conversely, you may have assumed that somebody else didn't know much about a topic dear to your heart, but they actually knew quite a lot about it. Both of these real-life experiences can be approached via appropriate modification of Transactive Memory.

These perceptual processes are important because these agents use their perceptions, not the ground-truth of the simulation (who "actually" knows what) to inform their twin primary motivations for interaction. Those twin drives are *homophily* and *knowledge expertise.* While defering a more detailed conversation about homophily and knowledge expertise to Part 2, a brief discussion about both these drives is appropriate.

Homophily, in its most general description, is the tendency for people to prefer to interact with people who are like themselves. This perception of ‘like themselves’ is, in Construct, a function of the amount of knowledge an ego and an alter share. In the real world, people may assume that others people are like themselves, even when that is not true. These egos may interact with their alters, because of the perceived similarity. In the actual event the ego is not like the alter, through the exchange of information, both could end up changing their knowledge and end up being more similar to each other than when they started.

The second primary interaction motivation is *knowledge expertise*. This motivation reflects human’s tendency to seek out knowledge they do not have from others--in the real social world, this behavior is most frequently seen when needing knowledge to successfully complete one or more tasks. In Construct, an ego with a perception that an alter has knowledge the ego does not, will have a higher probability of interacting with the alter than it might otherwise have.

This guide focuses on knowledge transactive memory (AAK), and this is the Construct XML required to define the knowledge transactive memory in our example deck:

```
<network id="'knowledge transactive memory network'"
     ego_node class_type="agent"
     src_node class_type="agent"
     target_node class_type="knowledge"
     link_type="bool" network_type="TMBool"
     associated_network="knowledge network">

      <generator type="perception_based">
            <ego first="0" last="node class::agent::count_minus_one"/>
            <alter first="0" last="node class::agent::count_minus_one"/>
            <transactive first="0"
                  last="node class::knowledge::count_minus_one"/>

            <param name="false_positive_rate" value="0.0"/>
            <param name="false_negative_rate" value="0.5"/>
            <param name="rounding_threshold" value="0.0"/>
            <param name="verbose" value="true"/>
      </generator>
</network>
```

Whew! Well, it might seem intimidating at first, but much of it is very similar to things we’ve seen before, but carried to the third dimension. **! It is essential that this network id contain the single quotation marks (‘) inside the double quotation marks. !**

The argument ego_node class indicates the agents that have these perceptions, the set of egos in the network. The argument src_node class indicates the agents for which the ego have perceptions, and the argument target_node class shows that, further, the perceptions are about what these other agents know. The associated_network indicates that the ground-truth network these perceptions will be based on is the knowledge network. A specific example with named agents could be Mike thinks Geoff knows about dancing. In more generalized form, this is a matrix that stores what each ego believes their connected alters know.

The associated_network particularly matters for this special type of generator - perception_based. Again, it's very similar to the other generator, taken to a third dimension. The arguments ego, alter and transactive are parallel to the node classes defined in the network: ego_node class, src_nodelcass, and target_node class, respectively. The parameter false_positive_rate indicates how likely egos are to perceive that their alters know things they do not, in this case, not at all likely. The parameter false_negative_rate indicates how likely agents are to assume that agents do not know things they actually do - this happens approximately 50% of the time in this example. The parameter rounding_threshold is useful when knowledge bits are not integer values. Values other than integer zero and integer one represent a knowledge bit that is partially known. This "sorta known" state is, for the purposes of transactive memory, binarized using the parameter as the cut-off point--values below the threshold will become zero and values equal to or greater than the threshold will become one. This parameter is necessary but not useful in this example. Part 2 will discuss its utility in depth. The final parameter verbose, if defined and set to true, will cause Construct to write a set of progress indicators to the console's standard out informing a researcher how far along the initialization process has progressed. If the parameter is undefined, Construct defaults to it being false.

The full snippet of XML, and how it could be read aloud, follows:

```
<network id="'knowledge transactive memory network'"
  ego_node class_type="agent"
     src_node class_type="agent"
     target_node class_type="knowledge"
     link_type="bool" network_type="TMBool"
     associated_network="knowledge network">

      <generator type="perception_based">
            <ego first="0" last="node class::agent::count_minus_one"/>
            <alter first="0" last="node class::agent::count_minus_one"/>
            <transactive first="0"
                  last="node class::knowledge::count_minus_one"/>

            <param name="false_positive_rate" value="0.0"/>
            <param name="false_negative_rate" value="0.5"/>
            <param name="rounding_threshold" value="0.0"/>
            <param name="verbose" value="true"/>
      </generator>
</network>
```

"This is the knowledge transactive memory network, it is an agent by agent by knowledge network with binary links. It uses the knowledge network as its ground-truth. It uses one perception-based generator and will populate values for all egos and how they perceive all alters and all of their associated knowledge. Egos will be pessimistic, they never assume agents

have knowledge they do not, and often assume agents do not have knowledge they actually do."

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 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 also 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 matrice space.

Outputs

Researchers and simulations usually compare outputs of Construct simulations by examining files written over the course of the simulation. It's outside the scope of this quick start guide to offer in-depth suggestions on how to deal with large quantities of simulation data, but the deck comes prepared with a set of handy outputs. A brief English summary of each output is below.

! **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**. !

- The knowledge network at every time period, with each agent separated by a new line from other agents. Each time period is separated by a blank line, but is otherwise un-numbered.

- Per-Agent diffusion values (# of bits agent has/# of all knowledge bits)
- Who interacted with whom every time period.
- Who was likely to interact with whom every round

Use these outputs (particularly the diffusion values) to examine questions of interest, see how they change (going up or down) as you manipulate the construction of the interaction sphere and the knowledge network. Do so, and you will quickly become comfortable with Construct.

High Level Diagrams of Construct Program Flow

The set of figures below show Construct's program flow. It is helpful for the user to keep track of what is set up for the simulation and these figures aid in describing the overall picture. They are intended for both Consruct users and also as a helpful referent for Construct developers.

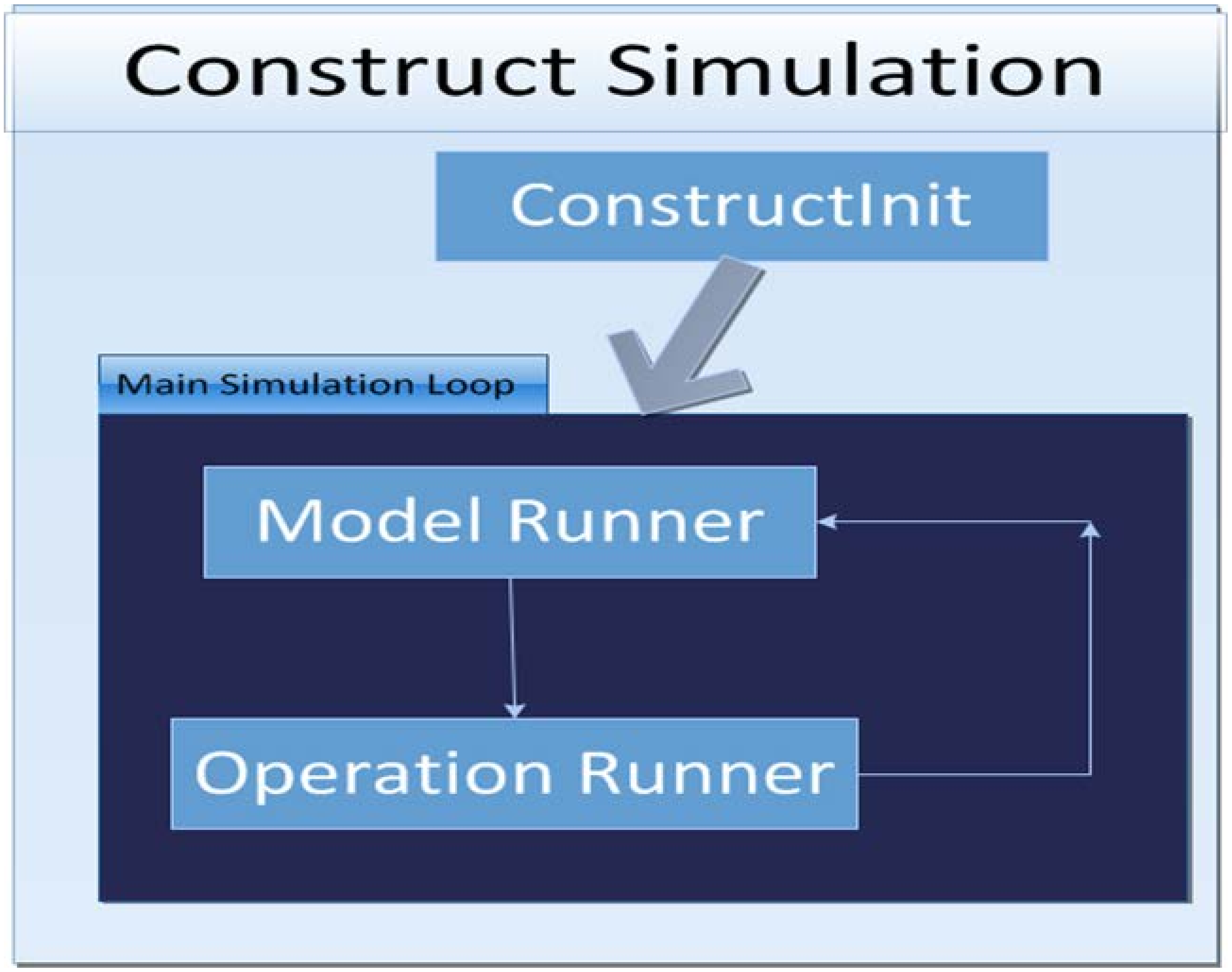


Figure 4. Construct's process has three main components.

This illustrates the three main components to a Construct run. They are an initialization section, then a loop with all models and output operations running repeatedly until the simulation ends. It is important to know that certain processes take place only within the Constuct initialization phase, such as setting up Transactive Memory and determining possible interactions partners for each ego.

Construct developers can create multiple models – this tech report focuses on the standard interaction model, but this model can be extended or amplified with special case models. Many, but not all, special case extensions are then folded into the larger simulation as appropriate.

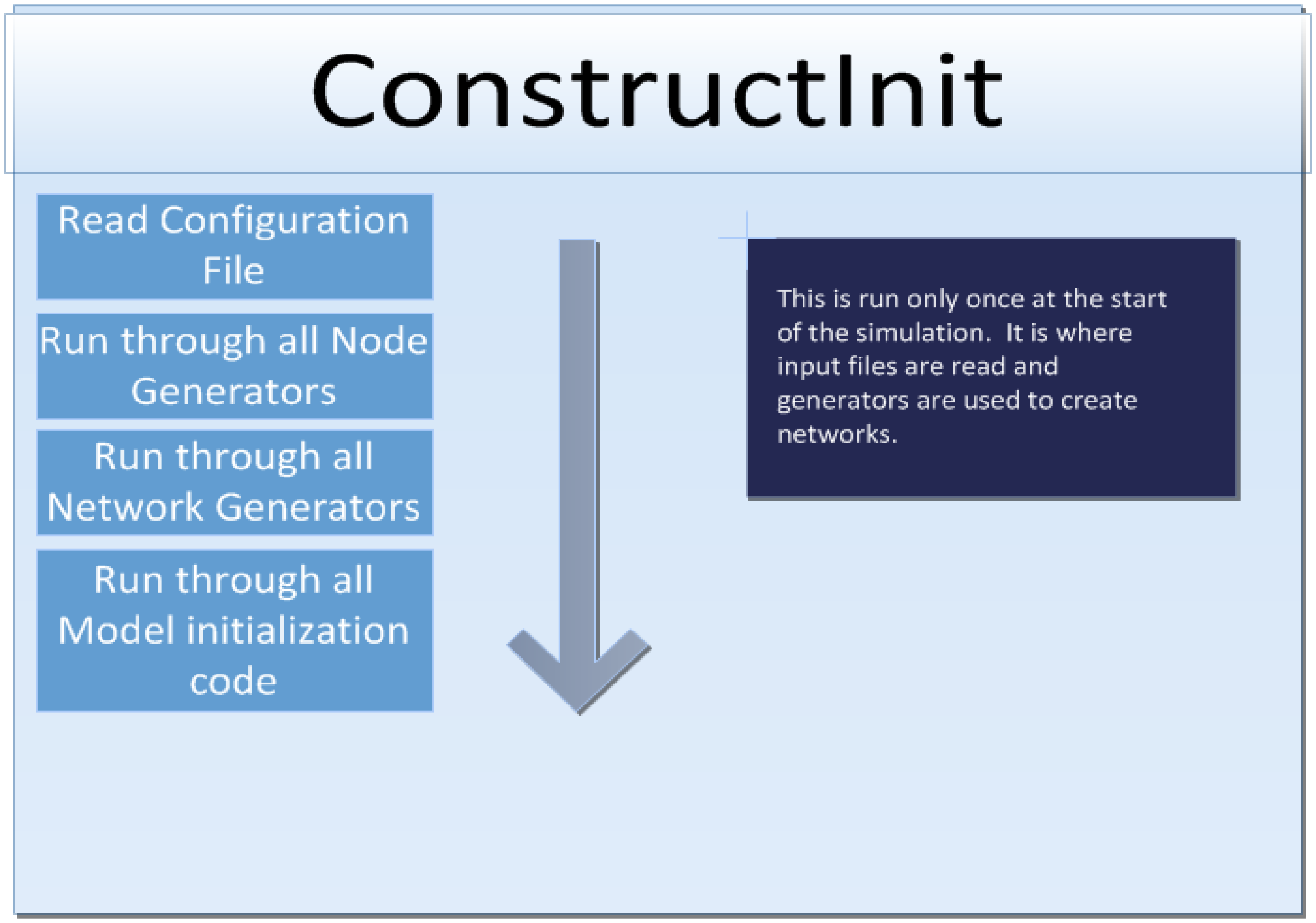

ConstructInit
Read Configuration File
Run through all Node Generators
Run through all Network Generators
Run through all Model initialization code
This is run only once at the start of the simulation. It is where input files are read and generators are used to create networks.

Figure 5. Construct's intialization process starts by reading the deck, then initializes nodes and networks, then goes through model specific setup.

Here we see a list of the components of the initialization section of Construct. They are run from top to bottom. The input deck is read. Nodes are created, and then relationships between nodes ae defined. All models enabled in the Construct deck are then initialized.

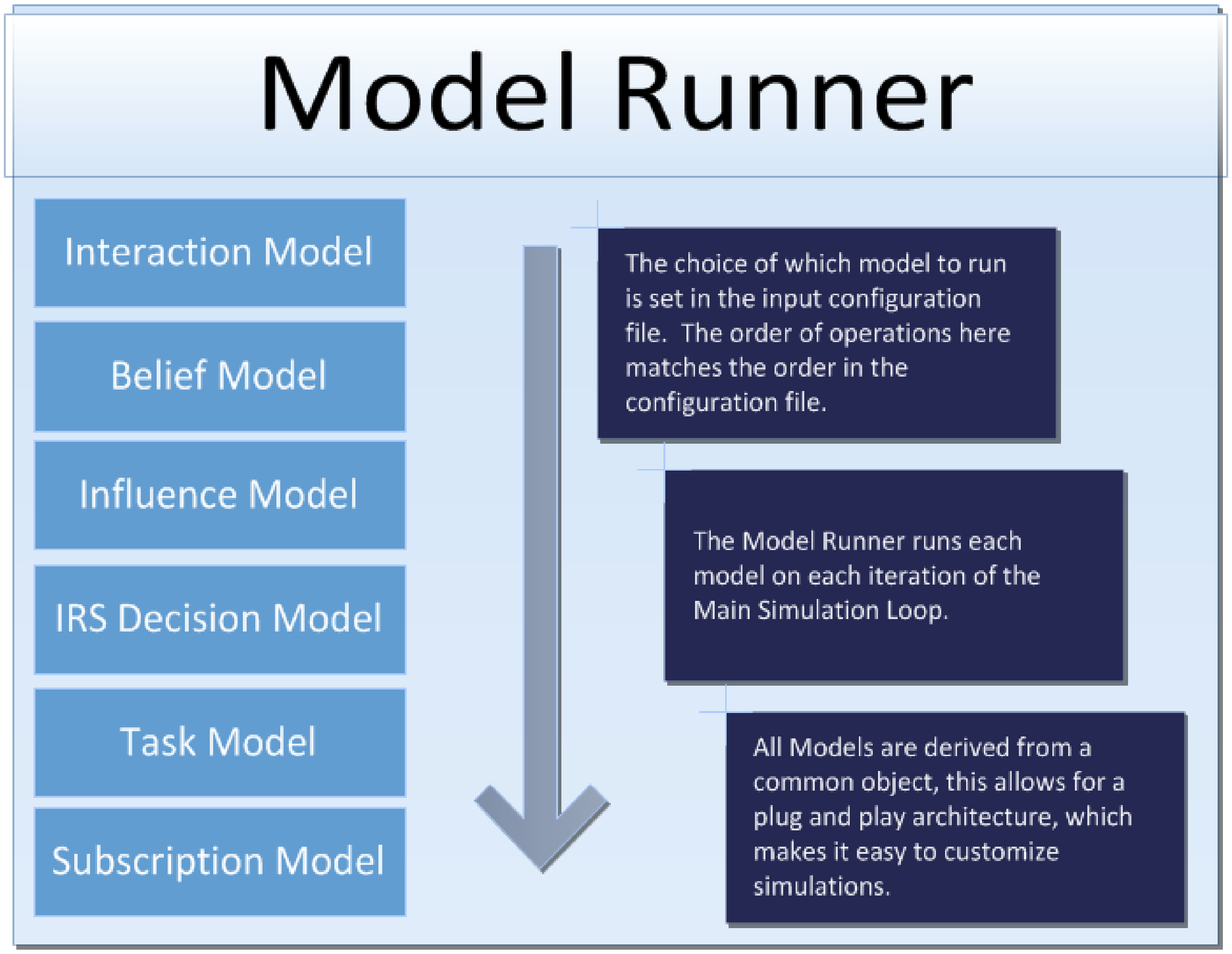

Model Runner
Interaction Model
Belief Model
Influence Model
IRS Decision Model
Task Model
Subscription Model
The choice of which model to run is set in the input configuration file. The order of operations here matches the order in the configuration file.
The Model Runner runs each model on each iteration of the Main Simulation Loop.
All Models are derived from a common object, this allows for a plug and play architecture, which makes it easy to customize simulations.

Figure 6. Stables of models can be run each turn in Construct. They run linearly, in an order defined by the user.

Here we see that models can be arranged in different orders and that they run in sequence from top to bottom. The order of models in the Construct Deck specifices the order in which they run in the simulation. Also, new models can be introduced into the stack at any point by including them in the input-deck.

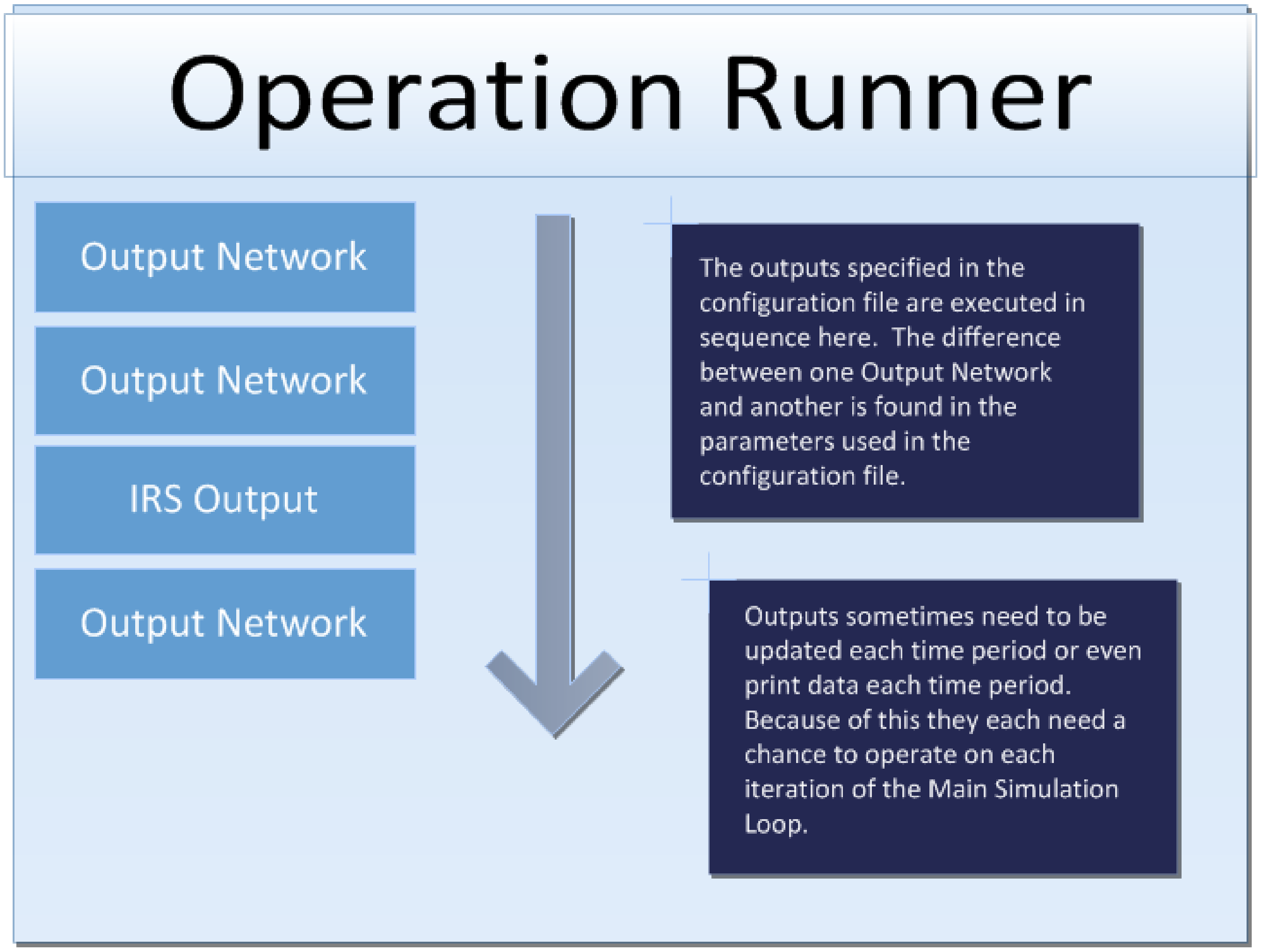

Operation Runner
Output Network
Output Network
IRS Output
Output Network
The outputs specified in the configuration file are executed in sequence here. The difference between one Output Network and another is found in the parameters used in the configuration file.
Outputs sometimes need to be updated each time period or even print data each time period. Because of this they each need a chance to operate on each iteration of the Main Simulation Loop.

Figure 7. Operation Runner allows for various operations to take place. Operations can be ordered by the user.

Just as with the models, the output operations are run in order from top to bottom and can be reordered in the same way. They run in order based on their order of appearance (from top to bottom) in the Construct Deck. This can include model-specific operations, but often involves outputting networks.

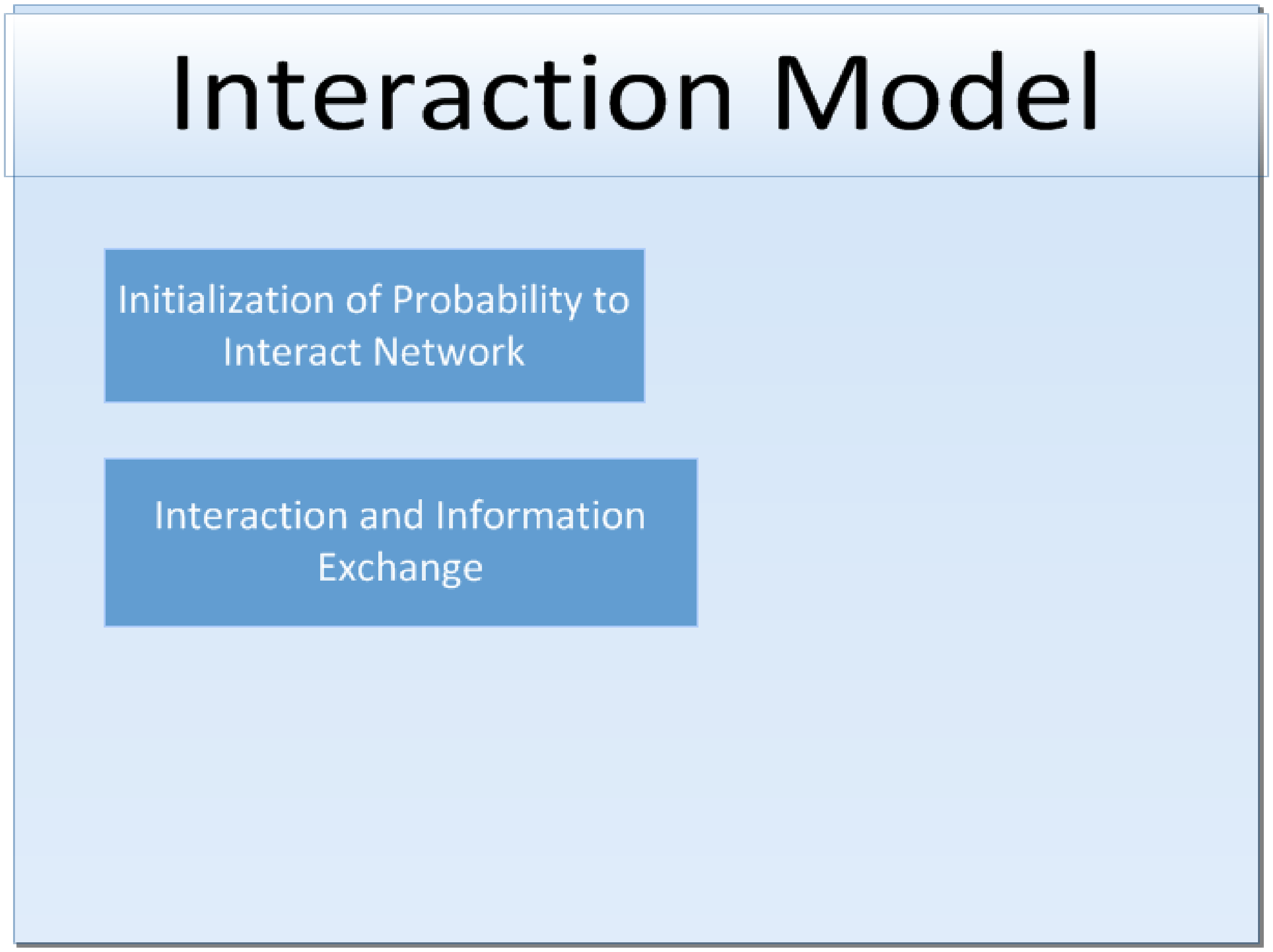

Interaction Model
Initialization of Probability to Interact Network
Interaction and Information Exchange

Figure 8. The Interaction Model is a core part of the Construct.

The Interaction Model is the most widely used Construct model. It has two parts. First, the system is initialized. Certain networks (specifically the Interaction Sphere) are read during this process and not again, changes to those networks after initialization will not produce useful change. Afterwards, this model is responsible for determining the likelihood of interaction with all available partners, and tracks information gained to update those interactions over time.

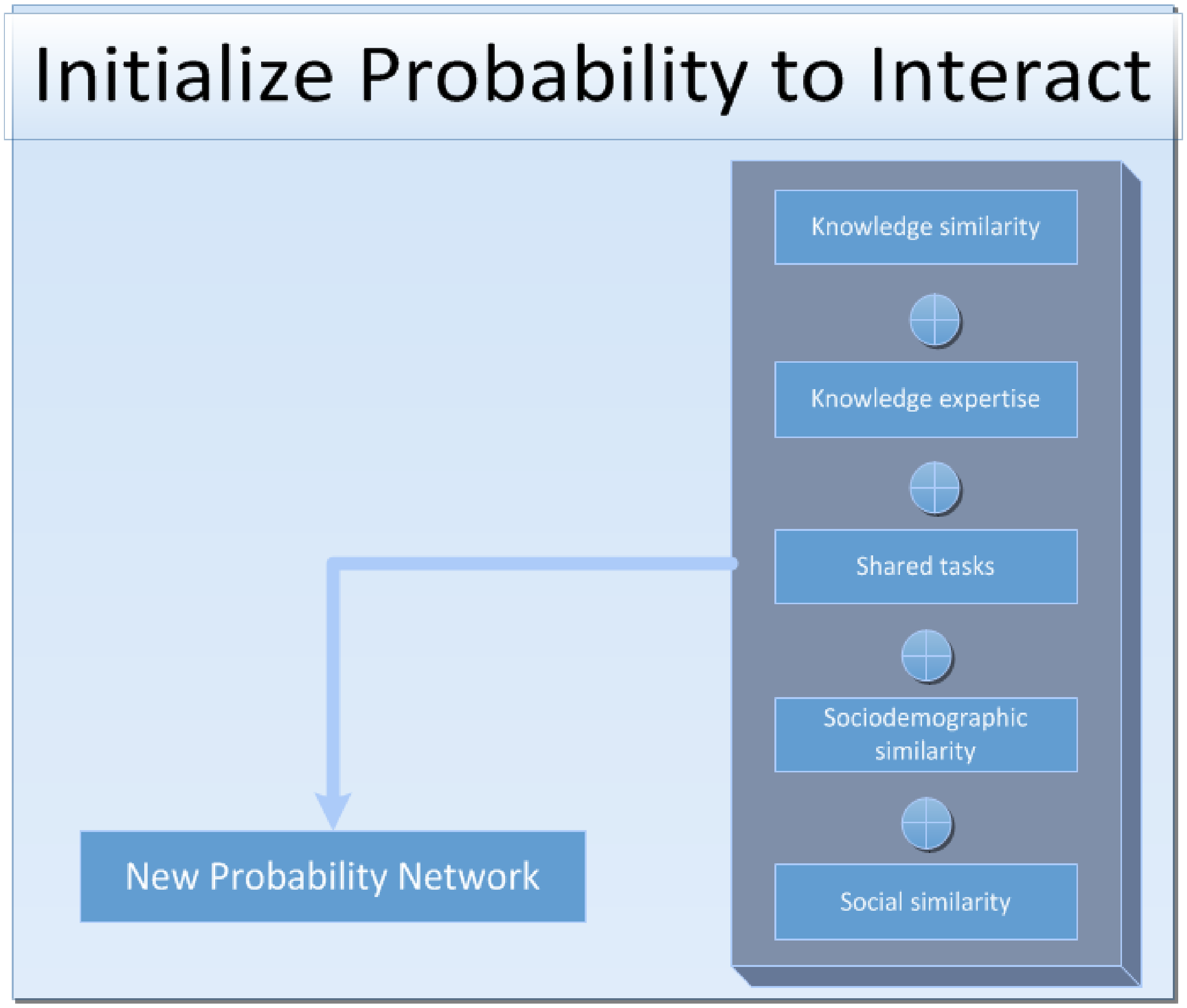


Figure 9. The probability network for "who talks to who" is an output of a variety of factors, some static, and some dynamic.

The probability that two agents will form a communication pair is dictated by similarity and expertise. Some of these values will change as a result of interaction. These changes will also influence future interaction. Together the influence of interaction on the similarity and expertise values as well as the similarity and expertise influence on interaction creates a feedback loop inside the Interaction Model.

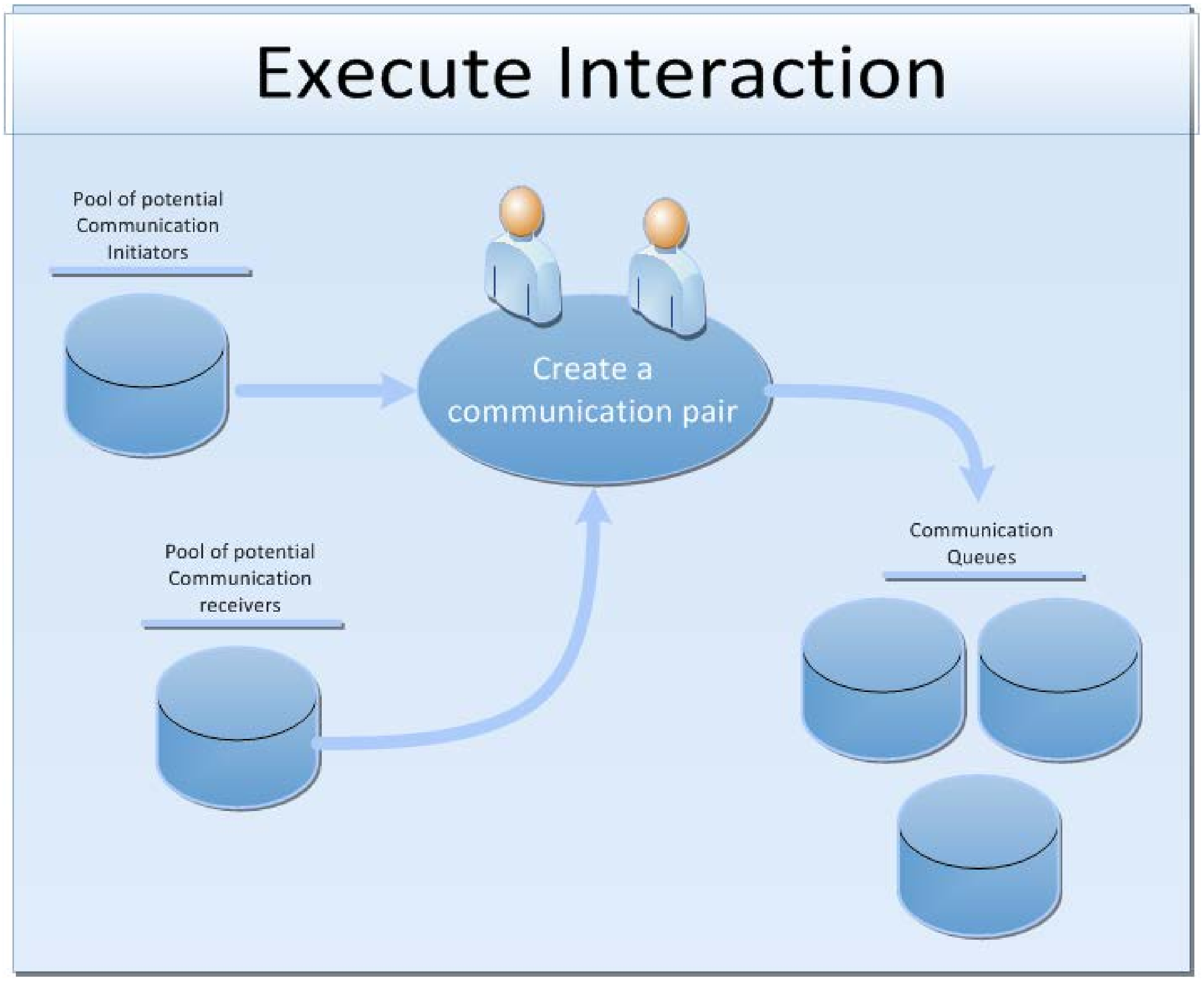

Execute Interaction
Pool of potential Communication Initiators
Create a communication pair
Pool of potential Communication receivers
Communication Queues

Figure 10. Interactions are created through matching up available initiators and receivers.

Agents are placed into pools of communication initiators and receivers. Agent pairs are drawn from these pools and placed in communication queues. The Interaction Probability Network influences which pairs are created.

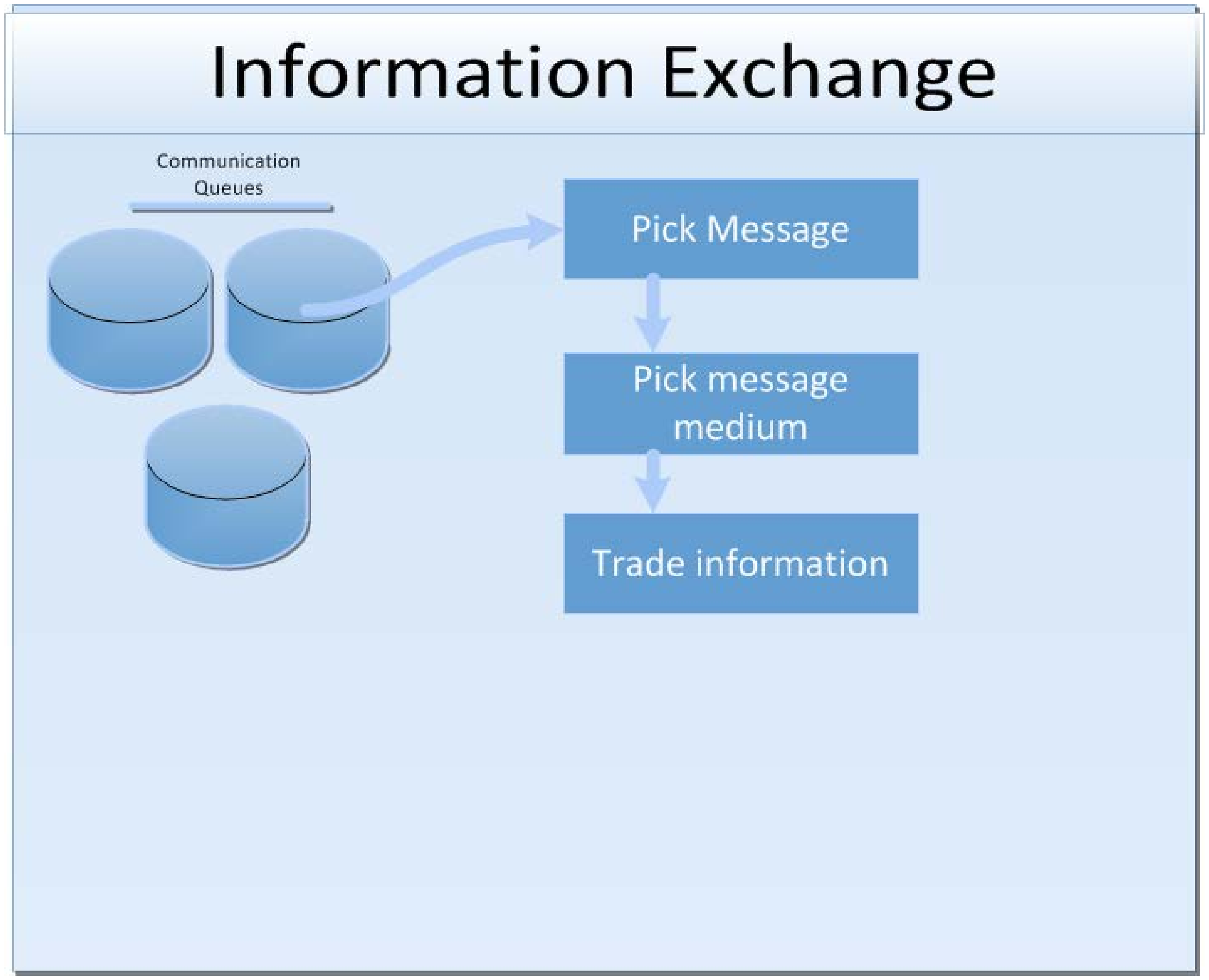


Figure 11. Information Exchange relies on both medium and message.

During Information Exchange, three important things occur. One, the information chosen to exchange is determined. Two, the medium over which the information will be sent is determined. Three, the actual exchange of information takes place.

PART TWO: Construct in Detail

This section of the report is, to some degree repetitive to the information 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 of 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 will be included in the appendix. Throughout this portion of the report and the appendices, we make an assumption 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 E 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 a <construct_vars></construct_vars> ConstructML tags. Modelers declare and define assign variables once, and then reference that variable whenever needed throughout the input deck. **! Modelers must declare variables prior to using them, !**or Construct’s parser will fail. Below is a sample of ConstructML showing the three ways a modeler can declare and define variables. The three ways are: declare and define as a constant (var1, var2, var3 below); declare and define in terms of other, prior-declared, variable (var4 below); and finally to declare and define in terms of a mathematical or logical operation on other prior-declared variables or constants (var4 and var5 below).

```
<construct_vars>
      <var name="[name]" value="[value]"/>
      ...
      <!-- 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” />
      ...
```

```
</construct_vars>
```

! 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 while 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 lexigraphical 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 E 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]
```

And an example of a specific variable 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 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 that 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 acceptable 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 stick with 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 an non-zero integer or float, Construct casts the value as true.
- If casting an 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 refered 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 in place.

- 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 an 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 moder is attempting to cast a variable to an integer, the following casting rules are in place.

- 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 int, and silently truncate, as it does with floats/doubles.
- If casting from a non-numeric string or other variable that cannot be cast as an 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

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”/>

```
</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. **!** Construct macro variables **are case sensitive**. **!** A macro of i is lexicographically distinct from I. Additionally, no macro should use a reserved word from the scripting language discussed in Appendix E. 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. Macro's 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.

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 an 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'll 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 verbose output or not. Putting such a variable near the top of the deck would supporting 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.

```
<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, assuming groups of agents are 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 thing 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 keep their concepts while meeting the input expectations of Construct.

Common Gotchas

In no particular 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 throughs most such editors for a loop, so we are still looking for viable ways others have used to help reduce non-well-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 a 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 syntaxed as follows:

```
<construct_parameters>
      <param name=“[name]” value=“[value]”>
</construct_parameters>
```

Parameter names must be valid like the parameters listed below and the values for parameters must be valid depending on the type of parameter, otherwise Construct will yield 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 dependant seed, set this parameter value to 0, otherwise set to an integer value to get constant results if the experiment were to be run multiple times.

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

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"/>
```

Dynamic Environment

This parameter determines whether or not to include an “outside world” agent that possesses different knowledge and can exchange information each turn of the simulation, which would introduce new information to agents in the simulation. The default value is false.

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

Default Agent Type

This parameter determines which kind of agent is set to be the default type agent. The default type is set to human.

```
<param name="default_agent_type" value="human"/>
```

Learning and forgetting

These parameters determine how an agent gains or loses information during the simulation. There are various forms of forgetting and learning that agents can take on. Forgetting determines if agents can lose facts that they learned and is a boolean parameter. If true, forgetting can decay at a sest rate under binary forgetting network. Binary Forgetting determines if agents are to lose the fact entirely or not. The agent either loses the entire fact or loses nothing. Binary Learning determines if agents can either learn the entire fact at once or not at all or learn part of the fact. When true, the agent either leanrs the entire fact at once or not at all. When false, the agent can learn a portion of the knowledge fact.

```
<param name="forgetting" value="false"/>
<param name="binary_forgetting" value="true"/>
<param name="binary_learning" value="true"/>
```

Use mail

The Use_mail parameter enables or disables mail communication, which allows agents to send a message at one period that agents can read at a later period and acquire knowledge. For agents to use mail, they must use the communicationMechanism called mail, and must employ various additional networks and parameters. For more detail on the mail system, reference CMU-ISR-08-114.

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

Default communication weights

This parameter determines what kinds of messages are sent whenever agents communicate with each other. Agents can communicate complex messages with multiples components, including knowledge, belief, and transactive memory. The communication weights set the type of content of the message. Belief Weight is set as the probability that an agent chooses to include its belief on any belief in the message. Transactive memory Belief weight is set as the probability that an agent decides to send its perception of any third party’s belief in the message. Fact Weight is set as the probability that an agent chooses to include a fact in the message. Transactive memory belief weight is set as the probability that an agent decides to send its perception of any third party’s knowledge in the message. These weights need to sum to a value of 1, otherwise they will normalize.

```
<param name="communicationWeightForBelief" value="0.2"/>
<param name="communicationWeightForBeliefTM" value="0.1"/>
```

```
<param name="communicationWeightForFact" value="0.5"/>
<param name="communicationWeightForKnowledgeTM" value="0.2"/>
```

Thread count

The thread count parameter sets the number of threads construct can use to parallelize a construct process. It is best to keep this set to 1 and seperate the processes and runs rather than to increase the threads.

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

Active models

This parameter specifies the models that are active in the simulation to govern interaction. There are three main models. The first is the Standard interaction model which uses homophily and expertise to guide interaction among agents. The Standard influence model contains influence and influencibility networks that determine how an agent's beliefs are influenced. The Standard belief model updates beliefs based on an agen'ts knowledge, belief weights, and beliefs of others. Below, Table 3 lists the required networks for all three standard models.

Table 3. List of networks for Construct

Required Networks for Standard Interaction Model	Required Networks for Standard Influence Model	Required Networks for Standard Belief Model
for agent interaction (23)	for agent influence (3)	for agent belief (4)
agent initiation count agent reception count agent message complexity agent selective attention effect agent learning rate agent forgetting rate agent learn by doing rate knowledge physical proximity sociodemographic proximity social proximity physical proximity weight sociodemographic proximity weight social proximity weight binarytask similarity weight binarytask assignment	beInfluenced influenceability agent belief	beInfluenced knowledge agent belief belief knowledge weight

binarytask requirement binarytask truth knowledge similarity weight knowledge expertise weight interaction knowledge weight transmission knowledge weight knowledge priority learnable knowledge		
The Five Core Networks (always required, regardless of model) interaction sphere access agent active timeperiod agent group knowledge group		
note: the word “network” has been omitted from the end of all network names		

Active mechanisms

This parameter specifies network post processes that are performed when setting up a simulation. This parameter includes access constraints such as literacy and information mechanisms.

```
<param name="active_mechanisms" value="none"/>
```

Transactive Memory Model

Modern Construct allows an option for how Transactive Memory should be handled, should Construct:

1) Should use group stereo-types for unknown alters, or
2) Attempt to keep a full transactive memory alter x knowledge network for all known-about alters.

The new stereotype model is much more efficient as the number of agents increases. There are several parameters involved:

```
<param name="tm_model" value="multi_level"/>
```

This parameter tells Construct that rather than using the default (Option 2), it should use group stereotyping. Several additional parameters are used to control aspects of stereo-type process.

All four of these parameters do nothing if “tm_model” is not set to “multi_level”, as shown in the line above.

```
<param name="activation_threshold_agent" value="0.0"/>
<param name="activation_threshold_group" value="-5.0"/>
```

In the new model, interaction between agents (both as themselves and as representatives of their groups) raises the agent’s interest. But the agent can’t keep track of everyone they meet; these two parameters control how quickly the agent dumps their knowledge representation of an agent or group (respectively) by defining the minimum value for an ‘interesting’ alter or group. As currently set in this example, group representations tend to endure much longer than alters.

```
<param name="agent_annealing_halflife" value="6"/>
<param name="group_annealing_halflife" value="25"/>
```

Our conceptions of our friends and of social groups tends to harden over time, these two parameters indicate the amount of time (in turns) that the definition of a group or an alter can be adjusted while they are an active alter. If an alter is lost, but then regained, this timer is effectively reset.

Nodes

Nodes are grouped into classes called node classes. All agent nodes are within agent node classes, and all nodes of the same type are in the same node class. node classes can be associated with each other to create networks through node links. Node links are manipulated within construct and can be added or modified. An example of this would be agents learning knowledge. There is a node link between the agent node and the knowledge node that is now created as the agent learns. Below in Table 4, are some common node classes with some important networks that contain links between nodes, in an input deck.

Table 4. Common node classes in Construct

NODE CLASS	Agent	Belief	Knowledge	Binary Task	Timeperiod	Dummy_ node class
Agent	interaction sphere ntwk	Belief Network	knowledge network	task assign. Network	agent active time ntwk	agent type network
Belief			belief weight ntwk			
Knowledge				task truth ntwk		

Binary Task				▨	▨	▨
Timeperiod					▨	▨
Dummy						▨

Agent node class

The agent node class represents the actors in the simulation. Agents interact with each other, exchange messages that contain beliefs and facts, and make decisions based on interaction. Many networks in the input deck are associated with agents as shown in the table___ above. Some common types include agent by knowledge and agent by belief networks. The agent node class must be present in the simulation and must have at least one node, otherwise the experiment has nothing to model. Agent nodes also have an associated agent type, which determines things that an agent can do, such as give and receive knowledge. Most agents are set as human, however this is not always the case. In some cases a user may want to simulate an intervention as a source of information such as a website that human agents can gain knowledge from.

Knowledge node class

The knowledge node class represents knowledge that can be exchanged between agents. Each knowledge bit is represented by one node. There are two types of knowledge, stylized and specific knowledge. Stylized knowledge would be something that drives interaction between agents while specific knowledge would have specific meaning such as an agent knowing how to pass a certification exam. Agents are associated with knowledge through the knowledge network. Agents can have knowledge of entire bits or partial knowledge by adjusting knowledge link weight in the knowledge network. Agents keep track of other agents' knowledge through knowledge transactive memory network. In the input deck, there are knowledge count knowledge bits within the node class.

```
<node class type="knowledge" id="knowledge">
      <properties>
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value=
         "construct::intvar::knowledge_count"/>
      </properties>
</node class>
```

Belief node class + belief formation equations

Beliefs represent whether or not an agent agrees or disagrees with a principle. These principles are represented by nodes in the belief node class. Agents are associated with these beliefs through the agent belief network, either with positive beliefs (agreements) or negative beliefs (disagreement). The standard belief model defines complete agreement with a value of 1.0, and complete disagreement with a value of -1.0. Neutral belief is set at 0.0. These values are set for a single belief, however in some cases multiple beliefs rather than one single belief will be criteria for a decision that an agent makes. In terms of agents' perception on other agents' beliefs, their perception is not perfect, i.e they don't always know exactly what another agent believes. Their perceptions are stored in the belief transactive memory network and agents will refer to this when determining social influence. So in essence, an agent's belief as well as their perception of what other agents believe, will play a role in their decision making and will show the effects of social influence on decisions. The number of beliefs in the input deck is set by modifying belief_count.

```
<node class type="belief" id="belief">
      <properties>
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value=
         "construct::intvar::belief_count"/>
      </properties>
</node class>
```

Binary task node class

Binary tasks are actions that agents can perform during the simulation. These actions are represented by nodes in the belief class. To perform these tasks, knowledge relevant to the task is required by the agent and can be attempted multiple times throughout an experiment time period. An example of this would be an agent having to pass a test; with insufficient knowledge during the first time period, the agent would not pass, but throught the second time period the experiment could be set up to allow that agent to gain enough knowledge bits to perform the task of passing the test. If an agent is able to perform a binary task, which is set in binary task assignment network, agents will use a subset of their knowledge related to the task. This knowledge is set in the binary task requirement network. That subset is then matched up against the true values that confirm the task to be performed. This is set in binary task truth network.

Agents will perform binary tasks in the following manner:

1. The agent is assumed to be able to complete the task correctly
2. All required knowledge bits are checked relevant to the agent's knowledge in the binary task truth network. If the agent's knowledge matches the required knowledge, the agent's accuracy is unchanged. If it doesn't match, the agent guesses with 50% probability. If correct, the next knowledge bit is checked, and if guessed incorrectly, the agent does not perform the task accurately.
3. There is a chance that an agent can misrepresent information. This is set in the misrepresentation probability network and inverts the decision that would be made based on knowledge and guessing. For example, an agent who should be able to pass a test, won't pass the test with a probability equal to the misrepresentation rate.

4. If misrepresentation is not set, the agent will always guess when their knowledge does not match the knowledge bits in the truth network. Agents that are assigned the same binary task will become more similar to each other. Similarity can be changed via the binary task similarity network. Agents who are assigned the same binary task would typically have similar knowledge bits, however they do not gain transactive memory through through binary tasks.

```
<node class type="binarytask" id="binarytask">
      <properties>
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value=
                  "construct::intvar::binarytask_count"/>
      </properties>
</node class>
```

Energy task node class

Energy tasks, represented as nodes in Construct, are actions that require a specific amount of effort rather than knowledge. Their ability to complete an energy task are setup in the energy task assignment network, and will expend energy on the task until they meet the required amount of energy to complete the task, which is set in the energy task requirement network. Similar to binary tasks, energy tasks can be attempted through different time periods in the simulation, however they are completely independent of binary tasks and knowledge.

The following procedure is used for an energy task:

1. An energy task instance is created and set by the energy task time network for a given agent.
2. The total number of energy task instances is tallied for the agent, which determines the total number of instances which an agent can devote energy.
3. For all incomplete energy tasks, the total amount of energy devoted to completion is equal to the reciprocal of the total number of tasks. So if an agent has three tasks, the amount of energy added to each task is equal to ⅓. Once the agent has spent the required amount of energy on the task, the task is complete and the agent will no longer spend time working on the task.
4. Any energy not spent on the task is lost and isn’t saved for future time periods.
5. There is no measurement of accuracy with energy tasks. Instead, the total number of energy tasks completed is measured following a simulation.

```
<node class type="energytask" id="energytask">
      <properties>
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value=
         "construct::intvar::energytask_count"/>
      </properties>
</node class>
```

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. The experimenter can decided whether agents are active during a time period by changing values in the *agent active time period network*. Some models treat the first period as a baseline and may alter some algorithms due to a lack of a previous time period to calculate a change from. For this reason it is better to run simulations for larger time periods.

```
<node class type="timeperiod" id="timeperiod">
      <properties>
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value=
         "construct::intvar::time_count"/>
      </properties>
</node class>
```

Agent group node class

The agent group node class keeps track of collections of similar agents. Construct can use these node classes to calculate network metrics and for simulation analysis. For example, Construct can determine how many agents have learned one specific fact within a group of agents.

```
<node class type="agentgroup" id="agentgroup">
      <properties>
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value=
         "construct::intvar::agentgroup_count"/>
      </properties>
</node class>
```

Knowledge group node class

The knowledge group node class keeps track of collections of similar knowledge bits. Construct can also use these to calculate a number of metrics. For example, Construct can determine how many knowledge bits have been learned by agents in a particular agent group.

```
<node class type="knowledgegroup" id="knowledgegroup">
      <properties>
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value=
         "construct::intvar::knowledgegroup_count"/>
      </properties>
</node class>
```

Dummy node class

The dummy node class is designed to act as a placeholder node class in order to create column vectors for other node classes. Construct works by manipulating two dimensional input and internal 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.

```
<node class type="dummy_node class" id="dummy_node class">
      <node id="constant" title="constant"/>
</node class>
```

Agent type

This node type sets the type of agent nodes within the simulation. The agent node type has the folllowing attributes:

- communicationMechanism - this attribute determines how an agent communicates. The agent can communicate with others either directly or via mail. Direct communication is where agents exchange messages face to face during a current time period, while mail communication delays the information exchange.
- canSendCommunication - this determines whether nodes are able to send information to other nodes of this type. Most agents should be able to send information regardless if they are human or not. Agents who can't send information cannot influence knowledge or beliefs of other agents.
- canReceiveCommunication - this determines which nodes can receive information. Agents who cannot receive information can't learn new knowledge or change beliefs and transactive memory.
- canSendKnowledge - this determines whether nodes can send knowledge when communicating. This mostly has to do with content of messages that agents send.
- canReceiveKnowledge- this determines whether nodes can receive knowledge when communicating. The content of the message received depends on the knowledge of the sender if this is enabled. Agent's who can't receive knowledge will ignore knowledge bits within a message.
- canSendBeliefs - this determines whether nodes can send beliefs.
- canReceiveBeliefs - this determines whether nodes can receive beliefs.
- canSendBeliefsTM - this determines whether or not nodes can send transactive memory about third party beliefs.
- canRecieveBeliefsTM - this determines whether nodes can receive transactive memory about third party beliefs.
- canSendKnowledgeTM - this determines whether nodes can send transactive memory about third party knowledge.
- canReceiveKnowledgeTM- this determines whether nodes can receive transactive memory about third party knowledge.
- canSendReferral - this determines if agents can send referrals to other agents in addition to knowledge and beliefs. An example of a referral would be if an agent is

seeking a specific piece of information and the sender has transactive memory about someone else who has the information desired, the sender can then reffer the agent seeking information to the agent with information. In other words, the sender agent can recommend an expert to the receiver agent.

- canReceiveReferral - this attribute determiens if agents can receive referals from agents in addition to knowledge and beliefs. If the receive wants information and can receive referrals, an agent with transactive memory about an expert agent can send the referall to the receiving agent, directing them to the source of information that they seek.

```
<node class>
      <node id="human" title="human">
            <properties>
                  <property name="canSendCommunication" value="true"/>
                  <property name="canReceiveCommunication" value="true"/>
                  <property name="canSendKnowledge" value="true"/>
                  <property name="canReceiveKnowledge" value="true"/>
                  <property name="canSendBeliefs" value="true"/>
                  <property name="canReceiveBeliefs" value="true"/>
                  <property name="canSendBeliefsTM" value="true"/>
                  <property name="canReceiveBeliefsTM" value="true"/>
                  <property name="canSendKnowledgeTM" value="true"/>
                  <property name="canReceiveKnowledgeTM" value="true"/>
                  <property name="canSendReferral" value="true"/>
                  <property name="canReceiveReferral" value="true"/>
            </properties>
       </node>
</node class>
```

Other node classes

While the ten node classes listed above are standard node classes in construct models, Construct is not limited to these ten node classes. Users can define their own node classes if desired, as long as they follow the same syntax as the node classes above and are unique names.

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 in order to perform tasks. For example the agent by agent knowledge network represents which knowledge is known by which agents.

Table 5 shows specific networks with their relationship to node class as well as a brief description.

Table 5. Network relations to node classes

NetworkName	Source & Target node classes	Function or Purpose in Demo Input Deck
agent type name	agent x dummy	specifies the agent type for each agent, thereby identifying key behavior
agent initiation count	agent x timeperiod	number of times agent can seek a partner, actively initiating communication
agent reception count	agent x timeperiod	number of times agent can be sought out, passively receiving communication
agent message complexity	agent x timeperiod	amount of info an agent can send when communicating
beInfluenced	agent x dummy	how resistant an agent is to changing its belief
influentialness	agent x dummy	how strongly an agent can influence the beliefs of others
agent selective attention effect	agent x dummy	percentage of agent knowledge that an agent will examine when communicating
agent learning rate	agent x knowledge	how quickly an agent will learn new knowledge when communicating
agent forgetting rate	agent x knowledge	how quickly an agent will forget old knowledge
agent learn by doing rate	agent x dummy	how quickly an agent will learn new knowledge when performing tasks
knowledge	agent x knowledge	the knowledge associated with an agent, i.e. what an agent currently knows
agent belief	agent x belief	the beliefs associated with an agent, i.e. what an agent currently believes
belief knowledge weight	knowledge x belief	the impact that each

		knowledge bit has on belief
interaction sphere	agent x agent	which agents are able to potentially interact with, and keep TM, of which
access	agent x agent	which agents have access to which (supplement to interaction sphere)
agent active timeperiod	agent x timeperiod	which agents are active during which timeperiods
physical proximity	agent x agent	how close each pair of agents are physically
sociodemographic proximity	agent x agent	how close each pair of agents are socio-demographically
social proximity	agent x agent	how close each pair of agents are socially

Network Name	**Source & Target node classes**	**Function or Purpose in Demo Input Deck**
physical proximity weight	agent x timeperiod	weight placed on physical proximity when choosing interaction partner
sociodemographic proximity weight	agent x timeperiod	weight placed on s-d proximity when choosing interaction partner
social proximity weight	agent x timeperiod	weight placed on social proximity when choosing interaction partner
binarytask similarity weight	agent x timeperiod	weight placed on shared binary tasks when choosing interaction partner
binarytask assignment	agent x binarytask	which agents are assigned to which binary tasks
binarytask requirement	knowledge x binarytask	which knowledge bits are required to complete which binary tasks
binarytask truth	knowledge x binarytask	what values knowledge bits must have to complete which binary tasks
knowledge similarity weight	agent x timeperiod	weight placed on shared knowledge when choosing interaction partner

knowledge expertise weight	agent x timeperiod	weight placed on different knowledge when choosing interaction partner
interaction knowledge weight	agent x knowledge	weight placed on knowledge bits when choosing interaction partner
transmission knowledge weight	agent x knowledge	weight placed on knowledge bits when sending a message
knowledge priority	agent x knowledge	priorities placed on knowledge bits when sending a message
learnable knowledge	agent x knowledge	what knowledge bits can or cannot be ever be learned
agent group membership	agent x agentgroup	what agents are associated with which agent groups
knowledge group membership	knowledge x knowledgegroup	what knowledge bits are associated with which knowledge groups

It is quite possible for a user to create more networks than there are listed in Figure 6. Networks are specified within the <networks> ConstructML tag.

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

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

```
<network src_node class_type="[node class]"
      target_node class_type="[node class]"
      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. 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 relationships 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]"/>
```

For using generator to specify links, use the following syntax:

```
<generator type="[type]">
      <rows first="[firstrow]" last="[lastrow]"/>
      <cols first="[firstcol]" last="[lastcol]"/>
      <param name="[name1]" value="[value1]"/>
      <param name="[name2]" value="[value2]"/>
      ...
</generator>
```

The following sections are more detailed descriptions of the networks listed above in Table 5.

Access Network

The access network is an addition to the interaction sphere and can restrict which alters agents can communicated with. This network prevents agents from potentially interacting rather than absolute prevention from interacting.

```
<network src_node class_type="agent" target_node class_type="agent"
      id="access network" link_type="float" network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::agent::count-1"/>
            <param name="min" value="1.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Agent Active Time Period

The agent active time period network determines which agents are active during time periods. Active agents during a time period can interact and exchange messages as well as beliefs.

```
<network src_node class_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="node class::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="node class::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="node class::timeperiod::count-1"/>
            <param name="constant_value"
      value="construct::boolvar::bridging_agents_active"/>

      </generator>
</network>
```

Agent Belief Network

The agent belief network specifies how strongly an agent holds a particular belief. During the simulation, agent beliefs can change based on what they learn and what agents around them believe.

```
<network src_node class_type="agent" target_node class_type="belief"
      id="agent belief network" link_type="float" network_type="dense">

      <generator type="constant">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::belief::count-1"/>
            <param name="constant_value" value="0"/>
      </generator>
</network>
```

Agent Forgetting Rate

The agent forgetting rate network specifies how quickly agents forget knowledge that they learned. Agents can partially learn knowledge bits depending on the forgetting method, through forgetting and binary forgetting. If binary forgetting is enabled, knowledge may be forgotten more unevenly, while if it is disabled it may be difficult for agents to completely forget a knowledge bit. These settings depend on the experiment.

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

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::knowledge::count-1"/>
            <param name="min" value="0.0"/>
            <param name="max" value="0.0"/>
      </generator>
</network>
```

Agent Group Membership

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

```
<network src_node class_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

The agent initiation count network specifies the number of times each agent can select other agents to interact with. Agents can either initiate communication with others by calculating a probability of interaction and choosing a partner based on this probability, or can wait until an agent choose to initiate interaction with them. Initiation count specifies the number of times agents initiate communication.

The process for agent initiation is as follows:

1. The probability of interaction is first computed between all pairs of agents that are within their respected interaction sphere.
2. Initiation count network is examined and each agent that can initiate interaction is added to a vector.
3. While there are agents remaining in the vector, a random agent is chosen and named the ego agent. Using the ego agent’s interaction sphere, all possible partners are examined and potential partners are kept. The pre computed probabilities of interaction are then normalized for these potential partners by the total absolute probability. An

interaction partner is then selected from this set of probabilities with a probability equal to that of its relative probability of interaction.

4. If the agent can't find a partner it will interact with itself.

Below is the syntax for initiation count:

```
<network src_node class_type="agent" target_node class_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="node class::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="node class::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="node class::timeperiod::count-1"/>
            <param name="constant_value" value="1"/>
      </generator>
</network>
```

Agent Learn by Doing Rate

The agent learn by doing network specifies how quickly agents learn particular bits of knowledge when performing binary tasks. This allows agents to learn more about knowledge bits that are partially known. This can allow agents to hone knowledge they already have in order to perform a task more accurately, without interacting with other agents to gain the knowledge.

```
<network src_node class_type="agent" target_node
      class_type="dummy_node class" id="agent learn by doing rate network"
      link_type="float" network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="0"/>
            <param name="min" value="0.0"/>
            <param name="max" value="0.0"/>
      </generator>
```

```
</network>
```

Agent Learning Rate

The agent learning rate network specifies how quickly agents learn knowledge. This network also allows agents to partially learn facts.

```
<network src_node class_type="agent" target_node class_type="knowledge"
      id="agent learning rate network" link_type="float"
      network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::knowledge::count-1"/>
            <param name="min" value="1.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Agent Message Complexity

Agent message complexity specifies the number of items an agent can include in its message when communicating with others. The process for message creation is as follows:

1. As long as the agent's message is less than the message complexity, the type of message item to include according to communication weights is randomly chosen. If the agent can't send that message item, another is chosen.
2. An item is selected of the appropriate type and verified whether it has not already been added to the message.

Below is the syntax for message complexity:

```
<network src_node class_type="agent" target_node class_type="timeperiod"
      id="agent message complexity 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="node class::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="node class::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="node class::timeperiod::count-1"/>
            <param name="constant_value" value="1"/>
      </generator>
</network>
```

Agent Reception Count

The agent reception count specifies the number of times each agent can be chosen as an interaction partner. The reception count is the maximum number of times each agent can be selected as an interaction partner each time period.

Below is the syntax for agent reception count.

```
<network src_node class_type="agent" target_node class_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="node class::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="node class::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="node class::timeperiod::count-1"/>
            <param name="constant_value" value="1"/>
      </generator>
</network>
```

Agent Type

The agent type network determines which agents are of which types. While the agent nodes are definedi n the node class, their role in the simulation is not yet defined. To determine which properties an agent has, it must be associated with an agent type. Agent type will determine agent behaviors such as communication mode and what can be communicated. For simplicity, a string network was chosen to represent the agent type.

The following is syntax for setting up an agent type network via <generator> that sets agents to be human and allows direct communication:

```
<network src_node class_type="agent" target_node
```

```
      class_type="dummy_node class" id="agent type name network"
      link_type="string" network_type="dense">

      <generator type="constant">
            <rows first="construct::intvar::agentgroup_A_start"
                  last="construct::intvar::agentgroup_A_end"/>
            <cols first="0" last="node class::dummy_node class::count-1"/>
            <param name="constant_value" value="human"/>
      </generator>
      <generator type="constant">
            <rows first="construct::intvar::agentgroup_B_start"
                  last="construct::intvar::agentgroup_B_end"/>
            <cols first="0" last="node class::dummy_node class::count-1"/>
            <param name="constant_value" value="human"/>
      </generator>
      <generator type="constant">
            <rows first="construct::intvar::agentgroup_C_start"
                  last="construct::intvar::agentgroup_C_end"/>
            <cols first="0" last="node class::dummy_node class::count-1"/>
            <param name="constant_value" value="human"/>
      </generator>
</network>
```

Belief Knowledge Weights

The belief knowledge weight network specifies how much impact a fact has on an agent's belief. This weight allows the user to associate beliefs with particular knowledge bits.

```
<network src_node class_type="belief" target_node class_type="knowledge"
      id="belief knowledge weight network" link_type="float"
      network_type="dense">

      <generator type="constant">
            <rows first="0" last="node class::belief::count-1"/>
            <cols first="0" last="node class::knowledge::count-1"/>
            <param name="constant_value" value="0"/>
      </generator>
</network>
```

Binary Task Assignment

The binary task assignment network specifies which agents are assigned to which binary tasks. Agents can learn knowledge by performing the binary task and also increase their similarity with others who are performing the same tasks.

```
<network src_node class_type="agent" target_node class_type="binarytask"
      id="binarytask assignment network" link_type="bool"
network_type="dense">

      <generator type="randombinary">
```

```
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::binarytask::count-1"/>
            <param name="mean" value="0"/>
      </generator>
</network>
```

Binary Task Requirements

The binary task requirement network specifies which knowledge bits are examined when an agent attempts to complete a binary task. For each knowledge bit required for the task, if the agents knowledge value doesn’t equal the value specified in the binary task truth network, the agent will guess and possibly complete the task incorrectly.

```
<network src_node class_type="knowledge" target_node class_type="binarytask"
      id="binarytask requirement network" link_type="bool"
      network_type="dense">

      <generator type="randombinary">
            <rows first="0" last="node class::knowledge::count-1"/>
            <cols first="0" last="node class::binarytask::count-1"/>
            <param name="mean" value="0"/>
      </generator>
</network>
```

Binary Task Truth

The binary task truth network specifies what the values of the required bits must be for an agent to complete a task without guessing.

```
<network src_node class_type="knowledge" target_node class_type="binarytask"
      id="binarytask truth network" link_type="bool" network_type="dense">

      <generator type="randombinary">
            <rows first="0" last="node class::knowledge::count-1"/>
            <cols first="0" last="node class::binarytask::count-1"/>
            <param name="mean" value="0"/>
      </generator>
</network>
```

Binary Task Similarity Weight

The binary task similarity weight network specifies how much weight agents place on shared tasks. Agents are more likely to interact if they have more similar tasks that they need to perform.

```
<network src_node class_type="agent" target_node class_type="timeperiod"
      id="binarytask similarity weight network" link_type="float"
      network_type="dense">

      <generator type="randomuniform">
```

```
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::timeperiod::count-1"/>
            <param name="min" value="0.0"/>
            <param name="max" value="0.0"/>
      </generator>
</network>
```

Interaction Network

Construct maintains this network as an Agent x Agent matrix of interactions in a particular turn. It is not available for manipulation by the modeler and is reset by Construct at the end of each turn in preparation for the upcoming turn. This network is available for reading and providing output via Construct operations.

Interaction Knowledge Weight

The interaction knowledge weight network specifies how much weight agents will put on particular knowledge bits when computing probabilities of interaction.

```
<network src_node class_type="agent" target_node class_type="knowledge"
      id="interaction knowledge weight network" link_type="float"
      network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::knowledge::count-1"/>
            <param name="min" value="1.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Knowledge – Binary and non-Binary

The knowledge network specifies which agents have what knowledge. Knowledge is used to select interaction partners, perform tasks, and form beliefs. Agents learn and forget knowledge as the simulation runs.

```
<network src_node class_type="agent" target_node class_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 Expertise Weight

The knowledge expertise network specifies how much weight ego agents place on knowledge known only by the alter when calculating probabilities of interaction.

```
<network src_node class_type="agent" target_node class_type="timeperiod"
      id="knowledge expertise weight network" link_type="float"
network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::timeperiod::count-1"/>
            <param name="min" value="0.2"/>
            <param name="max" value="0.2"/>
      </generator>
</network>
```

Knowledge Group Membership

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

```
<network src_node class_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 Priority

The knowledge priority network specifies the priority level of a particular fact when building a message.

```
<network src_node class_type="agent" target_node class_type="knowledge"
      id="knowledge priority network" link_type="int" network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::knowledge::count-1"/>
            <param name="min" value="1.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Knowledge Similarity Weight

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_node class_type="agent" target_node class_type="timeperiod"
      id="knowledge similarity weight network" link_type="float"
      network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::timeperiod::count-1"/>
            <param name="min" value="0.8"/>
            <param name="max" value="0.8"/>
      </generator>
</network>
```

Learnable Knowledge

The learnable knowledge network specifies which agents are able to learn what knowledge bits. An experimenter may want to restrict which groups of agents are capable of learning knowledge bits and can do so using this network.

```
<network src_node class_type="agent" target_node class_type="knowledge"
      id="learnable knowledge network" link_type="bool" network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::knowledge::count-1"/>
            <param name="min" value="1.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Physical Proximity

The physical proximity network specifies how close two agents are to each other physically. Physical distance is a factor in how likely two agents are to interact. With weights put on sociodemographic proximity as well as other forms of proximity, an experimenter can limit interaction between groups of agents.

```
<network src_node class_type="agent" target_node class_type="agent"
      id="physical proximity network" link_type="float" network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::agent::count-1"/>
            <param name="min" value="0.0"/>
            <param name="max" value="0.0"/>
      </generator>
</network>
```

Physical Proximity Weight

Physical proximity weight network specifies how strongly agents will value physical proximity when deciding who to interact with.

```
<network src_node class_type="agent" target_node class_type="timeperiod"
      id="physical proximity weight network" link_type="float"
      network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::timeperiod::count-1"/>
            <param name="min" value="0.0"/>
            <param name="max" value="0.0"/>
      </generator>
```

```
</network>
```

Selective Attention Effect

The agent selective attention effect network will determine how much of an agent's knowledge it will examine when deciding which knowledge bit to to use in a message. The size of the selective attention effect determines how much of an agent's knowledge it will examine when choosing knowledge to communicate.

```
<network src_node class_type="agent"target_node class_type="dummy_node class"
      id="agent selective attention effect network" link_type="float"
      network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="0"/>
            <param name="min" value="1.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Social Proximity

Social proximity network determines how close two agents are socially. This serves as a way of making agents more or less likely to interact based on social factors, such as career type or personality type.

```
<network src_node class_type="agent" target_node class_type="agent"
      id="social proximity network" link_type="float" network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::agent::count-1"/>
            <param name="min" value="1.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Socio-Demographic Proximity

The socio-demographic proximity network specifies how close agents are physically based on sociodemographic distance, and thus determines probability of interaction based on this metric.

```
<network src_node class_type="agent" target_node class_type="agent"
      id="sociodemographic proximity network" link_type="float"
      network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::agent::count-1"/>
            <param name="min" value="0.0"/>
            <param name="max" value="0.0"/>
      </generator>
</network>
```

Susceptibility (beInfluenced)

The beInfluenced network specifies how susceptible an agent is to influence. This network affects how strongly other alter agents can affect an ego's beliefs. Egos with a high susceptibility to influence will be more likely to change their beliefs.

```
<network src_node class_type="agent" target_node
      class_type="dummy_node class" id="beInfluenced network"
      link_type="float" network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="0"/>
            <param name="min" value="0.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Transmission Weight

The transmission knowledge weight network specifies how much weight agents will put on particular knowledge bits when sending a message to a chosen interaction partner.

```
<network src_node class_type="agent" target_node class_type="knowledge"
      id="transmission knowledge weight network" link_type="float"
      network_type="dense">

      <generator type="randomuniform">
            <rows first="0" last="node class::agent::count-1"/>
            <cols first="0" last="node class::knowledge::count-1"/>
            <param name="min" value="1.0"/>
            <param name="max" value="1.0"/>
      </generator>
</network>
```

Transactive Memory

Transactive memory, a term often associated with the work of Wegner (1987) in group process literature, is the term used to refer to an agent’s perception of the knowledge and beliefs of those around them. Transactive memory networks are the way in which Construct allows agents within the simulation to be “boundedly rational”, meaning that they do not have a perfect perception of the knowledge and beliefs of the people around them. Because perception is often imperfect, it will often be the case that an agent’s perception of the knowledge or beliefs of those around him differ from the truth. This notion of bounded rationality (Carley & Newell 1994; Simon 1957) is what allows Construct to be different than most other social simulation tools, which simply allow agents to be homophilous based on the current state of the system, rather than a perception of the environment.

Transactive memory networks store not just networks but each ego agent’s perception of a network. Thus, while most networks in Construct are two dimensional, transactive memory data structures have three dimensions: each agent has a perception of the knowledge or skills of each other agent they are connected to. It is important to note here that this does not mean each agent has a perception of all others, which would be highly memory intensive, but rather only of those agents that it is connected to in the interaction sphere network.

An agent’s transactive memory of others, or their imperfect knowledge of the world around them, is used when computing their probability of interacting with others and when computing the effect of social influence on their beliefs. 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 (Ren et al, 2001) use 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. Below, Table 6 shows the two transactive memory networks used in the demo input deck. Currently, Construct only supports perception (or transactive memory) of knowledge and beliefs- though there are other cases where perception may make sense (such as the perception of tasks one agent has of another agent), such mechanisms do not currently exist within Construct.

Table 6. Key transactive memory networks in the demo input deck

Network Name	Source & Target node classes	Function or Purpose in Demo Input Deck
knowledge transactive memory network	agent x timeperiod	store agent perceptions of other agents’ knowledge
belief transactive memory network	agent x timeperiod	store agent perceptions of other agents’ beliefs

Transactive memory networks in Construct are specified within a `<transactivememory>` ConstructML tag in the input deck. Within this tag, we use the `<network>` tag to define the different types of networks that we will use for transactive memory - in this implementation of Construct, the two networks will be the knowledge transactive memory network and the belief transactive memory network.

So, the outline of our transactive memory network looks like the following

```
<transactivememory>
      <network id=”knowledge transactive memory network”>
            ...
      </network>
      <network id=”belief transactive memory network”>
            ...
      </network>
</transactivememory>
```

Within the `<network>` tag, we need to specify seven different attributes: the network id (name), ego node class, source node class, target node class, link type, network type, and associated network, as seen below.

```
<network id="[name]"
  ego_node class_type="agent" src_node class_type="[node class]"
  target_node class_type="[node class]" link_type="[ltype]"
  network_type="[ntype]" associated_network="[network]">
 ...(Generator code, defined below)...
</network>
```

The network ID [name] is the name that refers to the transactive memory network - as we have seen, each network within Construct has a specific name that must be used to refer to it - if the name is, for example, misspelled, Construct will behave as though it does not exist and will likely exist with an error saying so.

In the present implementation of Construct, the eg_node class_type is assumed to be the agent node class - any other value is expected to behave in a non-obvious fashion, and will likely exit with an error. As we have stated, only agents in Construct are designed to have transactive memory, and they are only designed to have it for the knowledge and beliefs of others. It is also important to note that while certain agents may not use transactive memory in advanced models, it is still necessary to initialize their transactive memory. This is done to simplify certain internal mechanisms in Construct and actually happens to decrease run-time of the model.

While only agents are allowed to have transactive memory, Construct allows for them to have it of varying types of other node classes in the model. Thus, the src_node class_type does not strictly have to be defined as an agent, though models otherwise may be hard to justify. The src_node class_type and target_node class_type are related in the fact that each ego has a perception of how these two node classes relate. One can think of a transactive memory network as a three-dimensional array, where the first index refers to the ego, the second to some source, and the third to what the ego percieves the source to have of the target node class. So, for example, in the knowledge transactive memory network, an ego has, for each other agent they are connected to, a perception of whether or not that alter has the given knowledge.

The associated_network network is the underlying two-dimensional network that the ego perceives the source as having. For example, the knowledge transactive memory network is associated with the underlying knowledge network (The Knowledge Network). Specifying this network helps to initialize the network, particularly with certain types of generators we will discuss below.

The network_type specifies the storage mechanism used within Construct for each transactive memory network. The current implementation provides two options for this storage mechanism - the TMBool mechanism will store boolean values (either true or false), while the TMFloat mechanism will store the values -1, 0 and +1. While these methods have been optimized for large numbers of agents, the storage of these values is a chief reason why Construct runs can consume large amounts of RAM. Future iterations of the model hope to alleviate these issues.

The link_type parameter ltype defines the way in which Construct stores the data - as of the present implementation, the link type parameter can be either bool or float. Note, however, that this need not coincide with the value set for network_type, though it is often of interest to the user to specify the link_type as bool if the network_type is of kind TMBool. This is because bool links are boolean values, which allow certain specially designed storage mechanism for binary values to be utilized within Construct. Float links are stored as two-bit values representing either -1, 0, or +1. Note that the typical definition of floating point integers is really extended to significantly larger numbers of bits. However, an implementation decision was made to store only -1,0, or +1 to conserve large amounts of RAM during runs.

Given this description of the network tag within transactive memory tags, we now move to defining how these networks can be initialized. It is important to note that transactive memory networks, unlike other networks in Construct must be set using <generator> tags (other networks can be set using the explicit <link> command).

The general syntax for a transactive memory network generator, similar to the network generators described in Appendix D, is shown below.

```
<generator type="[type]">
      <ego first="[efirst]" last="[elast]"/>
      <alter first="[afirst]" last="[alast]"/>
      <transactive first="[tfirst]" last="[tlast]"/>
      <param name="[name1]" value="[value1]"/>
      <param name="[name2]" value=="[value1]"/>
      ...
      <param name="verbose" value="[verbose]"/>
</generator>
```

The [type] attribute specifies the type of network that is to be generated - as in previous sections, many different types of generators exist. The <ego> tag specifies the range of egos (the first column of our three-dimensional matrix), the <alter> tag the second, and the <transactive> tag the third. Thus, for the knowledge transactive memory network, to specify all agents and all knowledge bits, we would specify the first and last of <ego> and <alter> to be the ranges of the agent node class, and the transactive as the ranges of the knowledge node class. Note that the same generator-specific <param>s must be supplied in order for the generator to function correctly, as in other network generators. Finally, if desired, a <verbose> parameter can be given to have Construct print out a message indicating every hundredth agent that has been initialized.

Below, we give an example of how to use a constant generator for a transactve memory network:

```
<generator type="constant">
      <ego first="[efirst]" last="[elast]"/>
      <alter first="[afirst]" last="[alast]"/>
      <transactive first="[tfirst]" last="[tlast]"/>
      <param name="constant_value" value="[value]"/>
</generator>
```

Certain generators exist that are specific to the three-dimensional networks that are implied by the structure of transactive memory. In particular, the perception_based generator is specific to transactive memory, and the one used in the sample deck is provided below:

```
<generator type="perception_based">
      <ego first="[efirst]" last="[elast]"/>
      <alter first="[afirst]" last="[alast]"/>
      <transactive first="[tfirst]" last="[tlast]"/>
      <param name="false_negative_rate" value="[fnrate]"/>
      <param name="false_positive_rate" value="[fprate]"/>
      <param name="rounding_threshold" value="[threshold]"/>
      <param name="verbose" value="[verbose]"/>
</generator>
```

This generator will create a “perception” for each agent of the associated_network specified as an attribute in the network tag described above. The accuracy of the values of these perceptions can be modified by the false_negative_rate, false_positive_rate,and the rounding_threshold. These are three different ways to specify perception errors- note, however, that they have certain meanings for the different implementations of network_type, and will only be described as they exist in the demo input deck.

There are three additional points which are important to note about transactive memory generators. First, like other network tags, generators will be run in sequence, and thus any generator code placed after a previous generator will overwrite what has been done previously. Second, if the value for a given piece of the three dimensional transactive memory network is not specified, the value will default to false in TMBool networks or 0 in TMFloat networks.

Finally, after all generator code in the model, Construct will internally make sure that agents have a perfect perception of their own knowledge - thus, for example any perception_based generator used to initialize an agent's perception of themselves that initializes the agent to having imperfect knowledge of themself will be overwritten before the simulation begins.

Knowledge transactive memory

The knowledge transactive memory network is the way in which a user specifies what each agent perceives the knowledge of each of his alters to be. Thus, it is an agent x agent x knowledge network, or as we have referred to it, a three dimensional array. This array, once initialized, cannot be modified by the scripting mechanisms in Construct, introduced in Appendix E. As noted, agents will use this perception to calculate probabilities of interaction - thus, in combination with the other networks of the standard interaction model (of which the knowledge transactive memory is a piece), perceptions will evolve and change probabilities of interaction as the simulation runs.

As we have discussed, because transactive memory network store perceptions, they can be either correct or incorrect. Incorrect perceptions can occur during intialization, via miscommunication, or because of other Construct mechanisms that affect the spread of information. In addition, because the ego can tell an alter about the knowledge of a different alter, incorrect perceptions can actually be learned in Construct through traditional means of knowledge diffusion.

The knowledge transactive memory network is updated when an agent learns a new fact. In the current implementation of Construct, the previous perception an agent has is always overwritten by the newest information they obtain. The only exception to this rule is that an agent will always perceive their own knowledge correctly- thus, they cannot be told something that they know. Agents are therefore always trusting other completely and assuming that information received from others is more up-to-date than the agent's own knowledge.

The knowledge transactive memory network is of network_type TMBool, meaning values can be either 0 (false) or 1 (true). This allows the knowledge transactive memory network to utilize the perception_based generator - given the fact that value can be either 0 or 1, the parameters of this generator are relatively straightforward. The only exception to this is the rounding_threshold parameter, which is used to round values coming from the knowledge network - values that are below the rounding threshold are rounded down to 0, while values above the rounding threshold are rounded up to 1. This is one more point at which misperception can occur, if the knowledge network used is implemented with floating point values. The false_negative_rate is the rate at which the ego perceives an alter having a knowledge bit as not having the knowledge bit, and the false_positive_rate the percentage of time that the ego is initialized to believe an alter has a knowledge bit when they actually do not.

The demo input deck uses a perception_based generator to generate the knowledge network with a rounding threshold of 0.0 (only 0s will be interpreted as having no knowledge), a false negative rate of 50% (egos incorrectly "assume" alters do not have knowledge bit when they actually do 50% of the time) and a false positive rate of 0%. Note that each draw for false negative rates and false positive rates are independent and identically distributed ~Uniform(0,1), and therefore errors can be assumed to be uncorrelated.

The code to initialize is given below.

```
<network id="knowledge transactive memory network"
      ego_node class_type="agent" src_node class_type="agent"
      target_node class_type="knowledge" link_type="bool"
      network_type="TMBool" associated_network="knowledge network">

      <generator type="perception_based">
            <ego first="0" last="node class::agent::count-1"/>
            <alter first="0" last="node class::agent::count-1"/>
            <transactive first="0"
                  last="node class::knowledge::count-1"/>
            <param name="false_negative_rate" value="0.50"/>
            <param name="false_positive_rate" value="0.0"/>
            <param name="rounding_threshold" value="0.0"/>
            <param name="verbose" value="false"/>
      </generator>
</network>
```

Belief transactive memory

The belief transactive memory network is analogous to the knowledge transactive memory network with only a few, but important, differences. First, agents will use the belief transactive memory network when computing social influence, and it is thus a part of the standard influence model (and not the standard interaction model). Second, instead of an agent x agent x knowledge network, the belief transactive memory network is, of course, an agent x agent x belief network. Similar to the knowledge transactive memory network, scripting mechanisms cannot be used to change these values once initialized.

Similar to the knowledge transactive memory network, the belief transactive network can be correct or incorrect, and can be incorrect due to initialization, miscommunication, other active Construct mechanisms or via diffusion from other agents. Furthermore, like the knowledge transactive memory network, agents will update their transactive memory of others beliefs by overwriting any of their perceptions with information coming from other agents. The single exception is that the agent cannot have a misperception of its own beliefs. Note, however, that any and all updates described here will only occur when the standard belief model is active and in mask_mode.

The chief difference between the two transactive memory networks described in this report (besides their reference to different associated_networks) is the difference in how they are stored and, consequently, how values are generated for them. The values in the belief transactive memory network are stored as TMFloats, which can take on three values: -1, 0 and +1. A value of -1 indicates the ego perceives that the alter does not support the belief, a +1 that the alter does, and a 0 the perception that the alter is neutral on the belief. Because of this difference in storage mechanism, the perception_based generator for belief transactive memory works slightly differently than that for knowledge transactive memory. In particular, the rounding_threshold value is used as an absolute value- thus, for example, a value of .3 for the rounding threshold would mean that -.4 is rounded to -1, .4 is rounded to 1, and .2 or -.2 are rounded to 0. The false_positive_rate becomes the rate at which negative values actually are interpreted as positive values (e.g. -.4 misinterpreted as .4) and vice versa for false_positve_rate.

The specification for the belief transactive memory in the input deck is given below:

```
<network id="belief transactive memory network" ego_node class_type="agent"
      src_node class_type="agent" target_node class_type="belief"
      link_type="float" network_type="TMFloat"
      associated_network="belief network">

      <generator type="perception_based">
            <ego first="0" last="node class::agent::count-1"/>
            <alter first="0" last="node class::agent::count-1"/>
            <transactive first="0" last="node class::belief::count-1"/>
            <param name="false_negative_rate" value="0.25"/>
            <param name="false_positive_rate" value="0.25"/>
            <param name="rounding_threshold" value="0.0"/>
            <param name="verbose" value="false"/>
      </generator>
</network>
```

References

Carley, K. M. (1990). Group stability: A socio-cognitive approach. Advances in group processes, 7, 1-44.

Carley, K., & Newell, A. (1994). The nature of the social agent*. Journal of mathematical sociology, 19(4), 221-262.

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.

Carley, K. M. (2006). A dynamic network approach to the assessment of terrorist groups and the impact of alternative courses of action. CARNEGIE-MELLON UNIV PITTSBURGH PA INST OF SOFTWARE RESEARCH INTERNAT.

Carley, K. M., Robertson, D., Martin, M., Lee, J. S., St Charles, J., & Hirshman, B. (2010). Predicting Intentional and Inadvertent Non-compliance. Recent Research on Tax Administration and Compliance.

Giddens, A. The constitution of society: Outline of the theory of structuration. Berkeley, CA: University of California Press. 1984

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

Ren, Y., Carley, K. M., & Argote, L. (2001). Simulating the role of transactive memory in group training and performance. Pittsburgh, PA: CASOS, Dept. of Social and Decision Sciences, Carnegie Mellon University.

Salancik, G. R. and J. Pfeffer. A social information processing approach to job attitudes and task design. Administrative Science Quarterly, v.23, p.224-253. 1978.

Schreiber, C., & Carley, K. (2003). The impact of databases on knowledge transfer: simulation providing theory. In NAACSOS conference proceedings, Pittsburgh, PA (p. 2).

Schreiber, C., Singh, S., & Carley, K. M. (2004). Construct-a multi-agent network model for the co-evolution of agents and socio-cultural environments (No. CMU-ISRI-04-109). CARNEGIE-MELLON UNIV PITTSBURGH PA INST OF SOFTWARE RESEARCH INTERNAT.

Schreiber, C., & Carley, K. M. (2007). Agent interactions in construct: An empirical validation using calibrated grounding. In 2007 BRIMS Conference Proceedings, Norfolk, VA.

Simon, H. A. (1957). Administrative Behavior: A study of decision-making processes in administrative organization.

Tsvetovat, M., & Carley, K. M. (2004). Modeling complex socio-technical systems using multi-agent simulation methods. KI, 18(2), 23-28.

Wegner, D. M. (1987). Transactive memory: A contemporary analysis of the group mind. Theories of group behavior, 185, 208.

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="0"/>
      <var name="agentgroup_count" value="1"/>

   </construct_vars>
   <construct_parameters>
      <param name="seed" value="1"/>
<!--
      <param name="operation_output_working_directory" value=""/>
-->
      <param name="verbose_initialization" value="false"/>
      <param name="default_agent_type" value="human"/>
      <param name="out_of_sphere_comm_allowed" value="false"/>
      <param name="forgetting" value="false" />
      <param name="use_mail" value="false" />
      <param name="belief_model" value="mask_mode" />
      <param name="interaction_requirements" value="disable" />
      <param name="communicationWeightForBelief" value="0.2" />
      <param name="communicationWeightForBeliefTM" value="0.1" />
      <param name="communicationWeightForFact" value="0.5" />
      <param name="communicationWeightForKnowledgeTM" value="0.2" />
      <param name="thread_count" value="1" />
      <param name="transactive_memory" value="enable"/>
      <param name="active_models"
value="standard interaction model,standard task model"
with="delay_interpolation"/>
<param name="active_mechanisms" value="none"/>
   </construct_parameters>

   <nodes>
      <node class type="agent_type" id="agent_type">
         <node id="human" title="human">
            <properties>
               <property name="canSendCommunication" value="true"/>
               <property name="canReceiveCommunication" value="true"/>
               <property name="canSendKnowledge" value="true"/>
               <property name="canReceiveKnowledge" value="true"/>
               <property name="canSendBeliefs" value="true"/>
               <property name="canReceiveBeliefs" value="true"/>
               <property name="canSendBeliefsTM" value="true"/>
               <property name="canReceiveBeliefsTM" value="true"/>
               <property name="canSendKnowledgeTM" value="true"/>
               <property name="canReceiveKnowledgeTM" value="true"/>
               <property name="canSendReferral" value="true"/>
```

```
        <property name="canReceiveReferral" value="true"/>
        <property name="communicationMechanism" value="direct"/>
      </properties>
    </node>
</node class>

<node class type="CommunicationMedium" id="CommunicationMedium">
    <node id="facetoface" title="facetoface">
      <properties>
        <property name="maxMsgComplexity" value="1"/>
        <property name="msgCost" value="1.0"/>
        <property name="maximumPercentLearnable" value="1.0"/>
        <property name="time_to_live" value="1"/>
        <property name="time_to_send" value="1"/>
        <property name="passive" value="0"/>
      </properties>
    </node>
</node class>

<node class type="agent" id="agent">
    <generator type="count">
    </generator>
    <properties>
      <property name="generate_node class" value="true"/>
      <property name="generator_type" value="count"/>
      <property name="generator_count" value="agent_count"/>
    </properties>
</node class>

<node class type="knowledge" id="knowledge">
    <generator type="count">
    </generator>
    <properties>
      <property name="generate_node class" value="true"/>
      <property name="generator_type" value="count"/>
      <property name="generator_count" value="knowledge_count"/>
    </properties>
</node class>


<node class type="binarytask" id="binarytask">
    <node id="t1" title="ttt1" />
    <node id="t2" title="ttt2" />
    <node id="t3" title="ttt3" />
    <node id="t4" title="ttt4" />
</node class>

<node class type="belief" id="belief">
    <node id="b1" title="b1"/>
    <node id="b2" title="b2"/>
    <node id="b3" title="b3"/>
</node class>

<node class type="agentgroup" id="agentgroup">
    <properties>
```

```
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value="agentgroup_count"/>
         </properties>
      </node class>

      <node class type="knowledgegroup" id="knowledgegroup">
         <node id="FG1" title="FG1"/>
         <node id="FG2" title="FG2"/>
         <node id="FG3" title="FG3"/>
      </node class>



      <node class type="timeperiod" id="timeperiod">
         <properties>
            <property name="generate_node class" value="true"/>
            <property name="generator_type" value="count"/>
            <property name="generator_count" value="time_count"/>
         </properties>
      </node class>

      <node class type="dummy_node class" id="dummy_node class">
         <node id="dummy1" title="dummy1"/>
      </node class>

   </nodes>

   <networks>


      <network src_node class_type="agent" target_node class_type="timeperiod"
id="agent message complexity network" link_type="unsigned int" network_type="dense">
         <generator type="randomuniform">
            <rows first="0" last="node class::agent::count_minus_one"/>
            <cols first="0" last="node class::timeperiod::count_minus_one"/>
            <param name="min" value="1"/>
            <param name="max" value="1"/>
            <param name="symmetric_flag" value="false"/>
         </generator>
      </network>


      <network src_node class_type="agent" target_node class_type="timeperiod"
id="agent initiation count network" link_type="int" network_type="dense">
         <generator type="randomuniform">
            <rows first="0" last="node class::agent::count_minus_one"/>
            <cols first="0" last="node class::timeperiod::count_minus_one"/>
            <param name="min" value="10"/>
            <param name="max" value="10"/>
            <param name="symmetric_flag" value="false"/>
         </generator>
      </network>
      <network src_node class_type="agent" target_node class_type="timeperiod"
id="beinf network" link_type="float" network_type="dense">

         <generator type="randomuniform">
            <rows first="0" last="0"/>
            <cols first="0" last="node class::timeperiod::count_minus_one"/>
            <param name="min" value="0.5"/>
            <param name="max" value="0.8"/>
            <param name="symmetric_flag" value="false"/>
         </generator>

         <generator type="randomuniform">
            <rows first="1" last="1"/>
            <cols first="0" last="node class::timeperiod::count_minus_one"/>
            <param name="min" value="0.5"/>
            <param name="max" value="0.8"/>
            <param name="symmetric_flag" value="false"/>
         </generator>

      </network>

      <network src_node class_type="agent" target_node class_type="knowledge"
id="knowledge network" link_type="float" network_type="dense">
         <generator type="randombinary">
            <rows first="0" last="node class::agent::count_minus_one"/>
            <cols first="0" last="node class::knowledge::count_minus_one"/>
            <param name="mean" value="0.1"/>
            <param name="symmetric_flag" value="false"/>
         </generator>
      </network>

      <network src_node class_type="agent" target_node class_type="agent" id="access
network" link_type="float" network_type="dense">
         <generator type="constant">
            <rows first="0" last="node class::agent::count_minus_one"/>
            <cols first="0" last="node class::agent::count_minus_one"/>
            <param name="constant_value" value="1.0"/>
            <param name="symmetric_flag" value="false"/>
         </generator>
      </network>

      <network src_node class_type="knowledge" target_node class_type="binarytask"
id="binarytask requirement network" link_type="bool" network_type="dense">
         <generator type="randombinary">
            <rows first="0" last="node class::knowledge::count_minus_one"/>
            <cols first="0" last="node class::binarytask::count_minus_one"/>
            <param name="mean" value="0.5"/>
            <param name="symmetric_flag" value="false"/>
         </generator>
      </network>

      <network src_node class_type="knowledge" target_node class_type="binarytask"
id="binarytask truth network" link_type="bool" network_type="dense">
         <generator type="randombinary">
            <rows first="0" last="node class::knowledge::count_minus_one"/>
```

```
          <cols first="0" last="node class::binarytask::count_minus_one"/>
          <param name="mean" value="0.0"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
      </network>

      <network src_node class_type="agent" target_node class_type="binarytask"
id="binarytask assignment network" link_type="bool" network_type="dense">
        <generator type="randombinary">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::binarytask::count_minus_one"/>
          <param name="mean" value="1.0"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
      </network>

      <network src_node class_type="agent" target_node class_type="timeperiod"
id="knowledge similarity weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="timeperiod"
id="knowledge expertise weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="timeperiod"
id="binarytask similarity weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="knowledge"
id="interaction knowledge weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="knowledge"
id="transmission knowledge weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="timeperiod"
id="physical proximity weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="timeperiod"
id="social proximity weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="timeperiod"
id="sociodemographic proximity weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="belief" target_node class_type="knowledge"
id="belief knowledge weight network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::belief::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="belief" id="agent
belief network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::belief::count_minus_one"/>
          <param name="min" value="0.3"/>
          <param name="max" value="0.3"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
      </network>

      <network src_node class_type="agent" target_node class_type="agent" id="physical
proximity network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="agent" id="social
proximity network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="agent"
id="sociodemographic proximity network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="timeperiod"
id="agent active timeperiod network" link_type="bool" network_type="dense">
        <generator type="randombinary">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::timeperiod::count_minus_one"/>
          <param name="mean" value="1.0"/>
```

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

      <network src_node class_type="agent" target_node class_type="agent"
id="interaction sphere network" link_type="bool" network_type="dense">
         <generator type="randombinary">
            <rows first="0" last="node class::agent::count_minus_one"/>
            <cols first="0" last="node class::agent::count_minus_one"/>
            <param name="mean" value="1"/>
            <param name="symmetric_flag" value="false"/>
         </generator>

      </network>

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


      <network src_node class_type="agent" target_node class_type="dummy_node class"
id="public message propensity network" link_type="float" network_type="dense">
         <generator type="randomuniform">
            <rows first="0" last="node class::agent::count_minus_one"/>
            <cols first="0" last="0"/>
            <param name="min" value="0.0"/>
            <param name="max" value="1.0"/>
            <param name="symmetric_flag" value="false"/>
         </generator>
      </network>

      <network src_node class_type="knowledge" target_node class_type="knowledgegroup"
id="fact group membership network" link_type="bool" network_type="dense">
<!-- medium -->
         <!-- This group has ALL facts -->
         <generator type="randombinary">
            <rows first="0" last="node class::knowledge::count_minus_one"/>
            <cols first="0" last="0"/>
            <param name="mean" value="1.0"/>
            <param name="symmetric_flag" value="false"/>
         </generator>

         <!-- This group contains first 5 facts -->
         <generator type="randombinary">
            <rows first="0" last="5"/>
            <cols first="0" last="0"/>
            <param name="mean" value="1.0"/>
            <param name="symmetric_flag" value="false"/>
         </generator>
```

```
        <!-- This group contains facts after the first 5 -->
        <generator type="randombinary">
          <rows first="5" last="node class::knowledge::count_minus_one"/>
          <cols first="0" last="0"/>
          <param name="mean" value="1.0"/>
          <param name="symmetric_flag" value="false"/>
        </generator>

<!--
  DEFAULTS HERE
        <generator type="randombinary">
          <rows first="0" last="node class::knowledge::count_minus_one"/>
          <cols first="0" last="node class::knowledgegroup::count_minus_one"/>
          <param name="mean" value="1.0"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
-->

      </network>


      <network src_node class_type="agent" target_node class_type="timeperiod"
id="agent reception count network" link_type="int" network_type="dense">
        <generator type="randombinary">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::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_node class_type="agent" target_node class_type="dummy_node class"
id="agent selective attention effect network" link_type="float" network_type="dense">
        <generator type="randomuniform">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="0"/>
          <param name="min" value="1.0"/>
          <param name="max" value="1.0"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
      </network>
```

```
     <network src_node class_type="agent" target_node class_type="knowledge"
id="knowledge priority network" link_type="unsigned int" network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::agent::count_minus_one"/>
           <cols first="0" last="node class::knowledge::count_minus_one"/>
           <param name="min" value="1"/>
           <param name="max" value="1"/>
           <param name="symmetric_flag" value="false"/>
        </generator>
     </network>

     <network src_node class_type="agent" target_node class_type="timeperiod"
id="dynamic environment reset timeperiods network" link_type="bool"
network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::agent::count_minus_one"/>
           <cols first="0" last="node class::timeperiod::count_minus_one"/>
           <param name="min" value="0"/>
           <param name="max" value="0"/>
           <param name="symmetric_flag" value="false"/>
        </generator>
     </network>

     <network src_node class_type="agent" target_node class_type="dummy_node class"
id="beInfluenced network" link_type="float" network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::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>

     <network src_node class_type="agent" target_node class_type="dummy_node class"
id="influentialness network" link_type="float" network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::agent::count_minus_one"/>
           <cols first="0" last="0"/>
           <param name="min" value="1.0"/>
           <param name="max" value="1.0"/>
           <param name="symmetric_flag" value="false"/>
        </generator>
     </network>

     <network src_node class_type="agent" target_node class_type="dummy_node class"
id="agent learning rate network" link_type="float" network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::agent::count_minus_one"/>
           <cols first="0" last="0"/>
           <param name="min" value="1.0"/>
           <param name="max" value="1.0"/>
           <param name="symmetric_flag" value="false"/>
        </generator>
     </network>
     <network src_node class_type="agent" target_node class_type="knowledge"
id="learnable knowledge network" link_type="bool" network_type="dense">
        <generator type="randombinary">
           <rows first="0" last="node class::agent::count_minus_one"/>
           <cols first="0" last="node class::knowledge::count_minus_one"/>
           <param name="mean" value="1.0"/>
           <param name="symmetric_flag" value="false"/>
        </generator>
     </network>

     <network src_node class_type="agent" target_node class_type="dummy_node class"
id="agent forgetting rate network" link_type="float" network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::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>

     <network src_node class_type="agent" target_node class_type="dummy_node class"
id="agent learn by doing rate network" link_type="float" network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::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>

     <network src_node class_type="agent" target_node class_type="dummy_node class"
id="agent forgetting mean network" link_type="float" network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::agent::count_minus_one"/>
           <cols first="0" last="0"/>
           <param name="min" value="1.0"/>
           <param name="max" value="1.0"/>
           <param name="symmetric_flag" value="false"/>
        </generator>
     </network>

     <network src_node class_type="agent" target_node class_type="dummy_node class"
id="agent forgetting variance network" link_type="float" network_type="dense">
        <generator type="randomuniform">
           <rows first="0" last="node class::agent::count_minus_one"/>
           <cols first="0" last="0"/>
           <param name="min" value="1.0"/>
           <param name="max" value="1.0"/>
           <param name="symmetric_flag" value="false"/>
        </generator>
     </network>
```

```
      <network src_node class_type="agent" target_node class_type="agent"
id="interaction network" link_type="bool" network_type="dense">
        <generator type="constant">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::agent::count_minus_one"/>
          <param name="constant_value" value="0"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
      </network>

      <network src_node class_type="agent" target_node class_type="agent"
id="interaction probability network" link_type="float" network_type="dense">
        <generator type="constant">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::agent::count_minus_one"/>
          <param name="constant_value" value="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_node class_type="agent" target_node class_type="CommunicationMedium"
id="communication medium access network" link_type="float" network_type="dense">
        <generator type="constant">
          <rows first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::CommunicationMedium::count_minus_one"/>
          <param name="constant_value" value="1"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
      </network>

      <!--
        This network shows what medium is prefered 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_node class_type="agent" inner_node class_type="agent" target_node
class_type="CommunicationMedium" id="communication medium preferences network 3d"
link_type="float" network_type="dense3d">
        <generator type="constant3d">
          <rows first="1" last="node class::agent::count_minus_one"/>
          <inners first="0" last="node class::agent::count_minus_one"/>
          <cols first="0" last="node class::CommunicationMedium::count_minus_one"/>
          <param name="constant_value" value="1"/>
          <param name="symmetric_flag" value="false"/>
        </generator>

      </network>

      <!--
        This network limits what knowledge a given medium may send by limiting
        what knowledgegroups the medium may use.
      -->
      <network src_node class_type="CommunicationMedium" target_node
class_type="knowledgegroup" id="medium knowledgegroup network" link_type="bool"
network_type="dense">
        <generator type="randombinary">
          <rows first="0" last="node class::CommunicationMedium::count_minus_one"/>
          <cols first="0" last="node class::knowledgegroup::count_minus_one"/>
          <param name="mean" value="1.0"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
<!--
    DEFAULT all mediums can use all knowledge groups
        <generator type="randombinary">
          <rows first="0" last="node class::knowledge::count_minus_one"/>
          <cols first="0" last="node class::knowledgegroup::count_minus_one"/>
          <param name="mean" value="1.0"/>
          <param name="symmetric_flag" value="false"/>
        </generator>
-->
      </network>
   </networks>



   <transactivememory>
      <network id="'knowledge transactive memory network'" ego_node class_type="agent"
src_node class_type="agent" target_node class_type="knowledge" link_type="bool"
network_type="TMBool" associated_network="knowledge network">
        <generator type="perception_based">
          <ego first="0" last="node class::agent::count_minus_one"/>
          <alter first="0" last="node class::agent::count_minus_one"/>
          <transactive first="0" last="node class::knowledge::count_minus_one"/>
          <param name="false_positive_rate" value="0.0"/>
          <param name="false_negative_rate" value="0.5"/>
          <param name="rounding_threshold" value="0.0"/>
          <param name="verbose" value="true"/>

        </generator>
      </network>
   </transactivememory>


<operations>
      <operation name="ReadGraphByName">
        <parameters>
          <param name="output_filename" value="km.csv"/>
          <param name="output_format" value="csv"/>
          <param name="time" value="'all'"/>
```

```
            <param name="graph_name" value="'knowledge network'"/>

         </parameters>
      </operation>

      <operation name="ReadKnowledgeDiffusion">
         <parameters>
            <param name="output_filename" value="diff.csv"/>
            <param name="output_format" value="csv"/>
            <param name="time" value="all"/>
            <param name="no_empty_lines" value="true"/>
         </parameters>
      </operation>

      <operation name="ReadGraphByName">
         <parameters>
            <param name="output_filename" value="prob.csv"/>
            <param name="output_format" value="csv"/>
            <param name="time" value="all"/>

            <param name="graph_name" value="'interaction probability network'"/>

         </parameters>
      </operation>


      <operation name="ReadGraphByName">
         <parameters>
            <param name="output_filename" value="im.csv"/>
            <param name="output_format" value="csv"/>
            <param name="time" value="all"/>

            <param name="graph_name" value="'interaction network'"/>

         </parameters>
      </operation>

      <operation name="DeltaFeed">
         <parameters>
            <param name="output_filename" value="deltafeed.csv"/>
            <param name="output_format" value="csv"/>
            <param name="time" value="all"/>
         </parameters>
      </operation>
</operations>
</construct>
```

APPENDIX B A History of Construct

The foundational research from which construct was built upon lies within the fields of sociology and cognitive sciences, particularly in research done on human interaction and information exchange. Construct is based on Constructural theory, which states that social groups create concepts and actions based on reality, learning, and knowledge(a b Beaumie Kim; et al "Social Constructivism" Association for Educational Communications and Technology). Construct was designed to apply this theory computationally.

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 (Carley, 1990). 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 (Carley, 1990). 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 (Carley, 1990). The third basic principle states that similar individuals who share common knowledge are more likely to interact (Carley, 1990). 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 (Carley, 1990). With these concepts in place, the Construct model continued to evolve.

With advancements 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 in order 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, Carley and Schreiber 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, in order to examine how task and referential data types differ when simulating knowledge transfer (Schreiber and Carley, 2003). Transactive memory is also represented by the model to incorporate perception of other's knowledge in the social group (Schreiber and Carley, 2003). Each agent in the model is assigned task and transactive knowledge which are then represented by task databases and referential databases (Schreiber and Carley, 2003). The virtual experiment showed that these databases have an effect on 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 (Schreiber and Carley, 2003).

In 2004 Schreiber, Carley, and Singh, 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 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 (Schreiber 2004). Knowledge was represented as binary strings, which determined an agent's decision as well as perception of other agents' knowledge. (Schreiber 2004). Knowledge was limited to 500 facts and up to 25 tasks were assigned for each particular knowledge bit (Schreiber 2004).

APPENDIX C Construct ‘Operations’

At the end of each simulation turn, Construct executes each of the `<operation></operation>` tags within the `<operations></operations>` tag of the simulation input file. There are numerous operations, only some of which are shown in the sample input file of Appendix A. This appendix discusses each of the `<operation></operation>` tags supported by Construct. Construct will ignore any `<operation></operation>` tag that is outside the `<operations></operations>` tags with no indicator of error to the modeler.
Its important for a modeler to keep in mind the processing sequence of Construct. Construct does not process any operation until all agents have finished their interactions, finished learning and task execution efforts, and are otherwise poised for the next turn.

Turn 0

! Turn zero (0) is a special turn in that Construct uses turn zero as the initialization turn. !
There are no interactions and no simulation driven changes to the inputs at turn 0.

One way a modeler can increase their confidence that their nodes and node relationships (networks) are correct is to use the `ReadGraphByName` operation at time zero.

Operations

All the operations discussed in this section provide mechanisms for having Construct provide output to the modeler and simulationist. Modelers can use operations to debug the simulation, both at time point 0 and at other time points, as well as provide ways of communicating simulation behavior to consumers of the model’s outputs.There are three general ways of having Construct output data.
Entire Networks - At specified timepoints, Construct writes the contents of the specified network(s) to file. One written, post-processing can occur which presumably turns the dense matrix outputs into a meaningful measure or set of measures. ! **Entire network outputs can take up significant amounts of disk space, and inflict network congestion when Construct is operating in a high performance computing environment.** !
Process Outputs / Measures - Construct has several in-built matrix analytics that the modeler can use. These include various information diffusion metrics and task accuracy metrics. A primary caveat is that Construct may be applying these measures to agents or knowledge not of interest to a modeler or simulation.
Scripted Outputs - Using the capabilities of Construct’s scripting language, modelers can build customized output operations.

General Operation Syntax

The general syntax for a Construct `<operation></operation>` is shown below. Each of the specific `[name] values` that Construct supports are addressed in sub-sections below.

```
<operation name="[name]">
  <parameters>
<param name="’output_filename’" value="[filename].[extension]"/>
<param name="output_format" value="[csv|dynetml]"/>
<param name="run" value="all"/>
<param name="time" value="[first|last|all| csv timeperiod list]"/>
<!-- the following params are optional with default=all -->
<param name=”first_row” value=”” />
<param name=”last_row” value=”” />
<param name=”first_col” value=”” />
<param name=”last_col” value=”” />
<!-- the following params are optional with default=false -->
        <param name="print_row_names" value="true" />
        <param name="print_col_names" value="true" />
  </parameters>
</operation>
```

Construct will place the output file in the same directory as the input file by default. Modelers can prepend path information to the file name and Construct will write the output to the directory specified by the path name. Users should ensure that if they have opened any output files (e.g., in Excel to view the files), they should either close the file or use an application that does not place a file-level lock on the file (e.g., Notepadd++).

! Construct silently overwrites pre-existing files with no warning. !

The `output_format` supports two values, `csv` and `dynetml`. The `dynetml` format is an XML based format that CASOS uses in its Organizational Risk Analyzer (ORA) network analytic software package. The `csv` format is the only format that allows the modeler to output multiple time periods to a single file.

The `run` parameter remains necessary to support legacy input decks. The `time` parameter shown above is directing Construct to provide output at all time periods. Additional valid values for the time parameter are: `first, last,` and a comma separated list of positive integers that are less than the length of the simulation (`first` is equivalent to 1). The `verbose` parameter, when `true`, will print additional information about the decision during parsing of the input deck. The boolean `header_row` tells Construct to print a header row in the output file if the value is true.

The `<ListExpr>` is a comma separated list of node ids (e.g., 1,2,55,99). A modeler can also use the agent group reference syntax. (i.e. construct::agentgroup::<name> to provide the comma delimited list of agents. For each value in the `<ListExpr>` for the decision_names parameter, the modeler must add a parameter using that value as a decision name and define the decision using scripting syntax. If the `value` attribute does not define the decision (as it does for `d1` and `d2` below), the modeler must include a `type` attribute with a `decision_name_list` value to tell Construct that the definitions of the decisions appear later in the input deck (see also decision `d3` below, which is composed of 2 1-bit decisions, `d4` and `d5`). A more specific example of this syntax is shown below.

ReadGraphByName

This is a general purpose operation that allows a modeler to read an output any specified network. An important syntax issue to remember is to always include the name of the graph within single quotes (‘). If the modeler mis-spells the name, does not include a proper name in single quotes (‘), or Construct cannot otherwise find the named graph, it will fail during it’s first attempt to execute the operation with an error message.

```
<operation name="ReadGraphByName">
        <parameters>
 <param name="graph_name" value="'interaction probability network'"/>
         <param name="output_filename" value="prob.csv"/>
         <param name="output_format" value="csv"/>
         <param name="run" value="all"/>
         <param name="print_row_names" value="true" />
         <param name="print_col_names" value="true" />
         <param name="time" value="0,1,2,3,4,
(construct::intvar::time_count *10)/100,
(construct::intvar::time_count *20)/100,
(construct::intvar::time_count *30)/100,
(construct::intvar::time_count *40)/100,
(construct::intvar::time_count *50)/100,
(construct::intvar::time_count *60)/100,
(construct::intvar::time_count *70)/100,
(construct::intvar::time_count *80)/100,
(construct::intvar::time_count *90)/100,
construct::intvar::time_count-1"/>
        </parameters>
</operation>
```

In the example above, Construct will print the probability of interaction matrix for all agents with row and column headers. Construct will print at time points 0,1,2,3, and 4. It will also print at time points that are in 10% increments of the total simulation run time as defined using the `construct::intvar::time_count` variable.

The remainder of the Construct supported operations will be in alphabetical order

ReadAgentsWhoDoNotInteractWithAnyone

This will print a report to standard out of agents who did not interact with anyone at the end of each time period. It does not support writing the outputs to file. Changing the `output_to_stdout` to `false` will turn this operation off, but still consume processing time.

```
<operation name="ReadAgentsWhoDoNotInteractWithAnyone">
        <parameters>
                <param name="output_to_stdout" value="true"/>
                <param name="run" value="all"/>
                <param name="time" value="all"/>
                <param name="print_row_names" value="true" />
                <param name="print_col_names" value="true" />

        </parameters>
</operation>
```

ReadBinaryTaskAccuracy

This operation prints out an Agent vector with each entry specifying the agent's accuracy across all assigned binary tasks. In order to use this operation, a “binarytask truth network” and a “binarytask requirement network” must be specified.For each task the agent is assigned to, Construct assigns a 1 if accurate and a 0 if inaccurate.

```
<operation name="ReadBinaryTaskAccuracy">
  <parameters>
        <param name="output_filename" value="binaryTaskAccuracy.csv"/>
        <param name="output_format" value="csv"/>
        <param name="run" value="all"/>
        <param name="time" value="all" />
        <param name="print_row_names" value="true" />
        <param name="print_col_names" value="true" />
 </parameters>
</operation>
```

ReadInteractionMatrix

This specific operation has been deprecated by CASOS. Instead, below is the appropriate way of accessing and printing the values of the interaction matrix. Recall the interaction matrix is an Agent x Agent matrix whose cells store integer counts of the number of times the row agent and column agent have interact during the specified turn. The matrix is reset to all zero’s at the end of each turn.

```
<operation name="ReadGraphByName">
        <parameters>
 <param name="graph_name" value='"interaction network'"/>
         <param name="output_filename" value="interaction.csv"/>
```

```
             <param name="output_format" value="csv"/>
             <param name="run" value="all"/>
             <param name="print_row_names" value="true" />
             <param name="print_col_names" value="true" />
             <param name="time" value="first,
(construct::intvar::time_count *10)/100,
(construct::intvar::time_count *20)/100,
(construct::intvar::time_count *30)/100,
(construct::intvar::time_count *40)/100,
(construct::intvar::time_count *50)/100,
(construct::intvar::time_count *60)/100,
(construct::intvar::time_count *70)/100,
(construct::intvar::time_count *80)/100,
(construct::intvar::time_count *90)/100,
last"/>
            </parameters>
</operation>
```

ReadKnowledgeDiffusion

This will print an Agent x PercentKnowledgeDiffused vector at time period 0, time period 1, time periods that correspond to 20%, 40%, 60%, and 80%, and the last time period. Construct will separate each time period from the others by a blank line. In order to determine the number of knowledge facts that an agent has, Construct simply sums the number of bits set to 1 for that agent's index into the knowledge network.

```
<operation name="ReadKnowledgeDiffusion">
  <parameters>
            <param name="graph_name" value="'knowledge network'"/>
            <param name="output_filename" value="diffusion.csv"/>
            <param name="output_format" value="csv"/>
            <param name="run" value="all"/>
            <param name="time" value="0,first,
(construct::intvar::time_count *20)/100,
(construct::intvar::time_count *40)/100,
(construct::intvar::time_count *60)/100,
construct::intvar::time_count *80)/100,
construct::intvar::time_count-1" />
            <param name="print_row_names" value="true" />
            <param name="print_col_names" value="true" />
 </parameters>
</operation>
```

ReadKnowledgeDiffusionByAgentGroup

This will print a Knowledge x AgentGroup matrix at all time periods to a single file. Construct will print row and column headers. Construct will separate each time period from the others by a blank line. For this operation to operate, the modeler must have defined an Agent x AgentGroup matrix. The ReadKnowledgeDiffusionByAgentGroup operation will output a matrix where each row is an agent group, and each column is a fact. The value at a particular row-column cell in this matrix is the percentage of agents who know that fact.

```
<operation name="ReadKnowledgeDiffusionByAgentGroup">
 <parameters>
  <param name="output_filename" value="KnowledgeDiffusionByAgentGroup.csv"/>
  <param name="output_format" value="csv"/>
  <param name="run" value="all"/>
  <param name="time" value="all"/>
  <param name="print_row_names" value="true" />
  <param name="print_col_names" value="true" />
 </parameters>
</operation>
```

ReadKnowledgeDiffusionByFactGroup

This will print an Agent x Percentage of Facts per FactGroup matrix at all time periods to a single file. Construct will print row and column headers. Construct will separate each time period from the others by a blank line. Each cell of the matrix at a given time period will output the percentage of facts in the given FactGroup that the agent at that row of the matrix knows.

```
<operation name="ReadKnowledgeDiffusionByFactGroup">
  <parameters>
          <param name="output_filename" value="KnowledgeDiffusionByFactGroup.csv"/>
<param name="output_format" value="csv"/>
          <param name="run" value="all"/>
          <param name="time" value="all"/>
          <param name="print_row_names" value="true" />
          <param name="print_col_names" value="true" />
  </parameters>
</operation>
```

ReadKnowledgePriorityMatrix

This specific operation has been deprecated by CASOS. Instead, below is the appropriate way of accessing and printing the values of the knowledge priority matrix. This network does not normally change during the execution of the run unless the modeler uses scripting to do so. As such, printing out time periods other than the first is probably not useful.

```
<operation name="ReadGraphByName">
        <parameters>
 <param name="graph_name" value="'knowledge priority network'"/>
```

```
             <param name="output_filename" value="interaction.csv"/>
             <param name="output_format" value="csv"/>
             <param name="run" value="all"/>
             <param name="print_row_names" value="true" />
             <param name="print_col_names" value="true" />
             <param name="time" value="first"/>
            </parameters>
</operation>
```

ReadSphereMatrix

This specific operation has been deprecated by CASOS. Instead, below is the appropriate way of accessing and printing the values of the interaction sphere matrix. Recall the interaction sphere matrix is an Agent x Agent matrix whose cells store boolean values that indicate the row agent can interact with the column agent. This network does not normally change during the execution of the run unless the modeler uses scripting to do so. As such, printing out time periods other than the first is probably not useful.

```
<operation name="ReadGraphByName">
  <parameters>
<param name="graph_name" value="'interaction sphere network'"/>
            <param name="output_filename" value="interaction_sphere.csv"/>
            <param name="output_format" value="csv"/>
            <param name="run" value="all"/>
            <param name="print_row_names" value="true" />
            <param name="print_col_names" value="true" />
<param name="time" value="first"/>
 </parameters>
</operation>
```

‘Decisions’

Construct is capable of generating arbitrary computable output through the use of expressions and scripting on a per-agent, per-turn basis. When an agent executes a ‘Decision,’ the agent is not only able to generate additional non-in-built output, but can modify its internal state in ways the original developers had not necessarily contemplated. In the `<operations></operations>` portion of a construct input file, a modeler can create an operation that supports this functionality. A modeler can also make a ‘decision’ that can affect the entire simulation, and not just single agents or nodes.

ReadDecisionOutput

The modeler will use the `<ReadDecisionOutput></ReadDecisionOutput>` tags when creating an arbitrary decision. The complete syntax is shown in the example below and the name of the tag is case sensitive. Construct will place the output file in the same directory as the input file; there is no capability to write to a different directory or path. The `output_format`, similarly to other output options, supports two values, `csv` and `dynetml`.

```
<operation name="ReadDecisionOutput">
<parameters>
<param name="output_filename" value="[filename]"/>
<param name="output_format" value="csv"/>
<param name="run" value="all"/>
<param name="time" value="all"/>
<param name="verbose" value="<BoolExpr>"/>
<param name="header_row" value="<BoolExpr>"/>
<param name="applicable_agents" value="<ListExpr>"/>
<param name="decision_names" value="<ListExpr>"/>
<param name="<decision name value 1>" value="<ListExpr>"/>
<param name="<decision name value 2>" value="<ListExpr>"/>
<param name="<decision name value ...>" value="<ListExpr>"/>
<param name="<decision name value n>" value="<ListExpr>"/>
<parameters>
</operation>
```

Unlike other operations, the `ReadDecisionOutput` gets executed at the end of every turn. The `time` parameter shown above is directing Construct to provide output at all time periods, though how frequently a modeler wants that output is situation dependent. Note the additional valid values for the time parameter: `first, last, all,` and a comma separated list of positive integers that are less than the length of the simulation (`first` is equivalent to 1). The `run` parameter remains necessary to support legacy input decks. The `verbose` parameter, when `true`, will print additional information about the decision during parsing of the input deck. The boolean `header_row` tells Construct to print a header row in the output file if the value is true.

The `<ListExpr>` is a comma separated list of node ids (e.g., 1,2,55,99). A modeler can also use the agent group reference syntax (i.e. construct::agentgroup::<name> to provide the comma delimited list of agents). For each value in the `<ListExpr>` for the decision_names parameter, the modeler must add a parameter using that value as a decision name and define the decision using scripting syntax. If the `value` attribute does not define the decision (as it does for `d1` and `d2` below), the modeler must include a `type` attribute with a `decision_name_list` value to tell Construct that the definitions of the decisions appear later in the input deck (see also decision `d3` below, which is composed of 2 1-bit decisions, `d4` and `d5`). A more specific example of this syntax is shown below.

```
<operation name="ReadDecisionOutput">
 <parameters>
<param name="output_filename" value="decision_outputs.csv"/>
<param name="output_format" value="csv"/>
<param name="run" value="all"/>
<param name="time" value="first,10,20,30,last"/>
<param name="verbose" value="false"/>
```

```
<param name="header_row" value="true"/>
<param name="applicable_agents" value="1,15,99"/>
<param name="decision_names" value="d1,d2,d3"/>
<param name="d1” value="getKnowledgeNetwork[agent,1]" with="agent"/>
<param name="d2" value="getKnowledgeNetwork[agent,20]" with="agent"/>
<param name="d3" value="d4,d5" type=”decision_name_list"/>
<param name="d$i$" value="getKnowledgeNetwork[agent,$i$]"
          with="agent,$i$=(4,5),$i$"/>
 <parameters>
</operation>
```

Reading the decision above, would sound like the follow:
“This is an operation to get the decision from agents 1, 15, and 99 for decision 1, decision 2, and decision 3, and to print the results of that retrieval at time 1 (the first time period), time 10, 20, 30 and the last time period of the simulation. Decision 1 is a represented by a single bit of knowledge, in column 1 of the Agent x Knowledge Network. Decision 2 is also a single bit of knowledge, in column 20. Decision3 is ____________. Decision 4 & 5, defined with a macro variable, are also single bits in the AxK network, and are with respect to Agents 4 & 5 only. The output format will be CSV with decision 1 in the first column, and decision 5 in the fifth column.”

Specifying Decisions

Construct does not have a defined limit on the number of decisions a modeler can define. Experience within CASOS and developmental testing indicate that no more than 200 decisions be created per simulation.

Modelers must define their decisions in the `decision_names` parameter or within a chain of decisions. One way of ensuring reachability is to have the `header_row` parameter set to true. If the decision of interest is in the header row, Construct is attempting to evaluate it.

A modeler can use all the scripting language capabilities and features available when specifying variables. The practical result is that a decision can use constants, mathematical and logical expressions, string operations, and conditional statements. There are an additional five (5) scripting features available to modelers when declaring and specifying decisions:
network getters
network setters
agent references
time period references
decision references

Network Getters

A very important part of the decision system in Construct is its ability to read and return a set of values from any network within the simulation. The general syntax is `getSomeNetworkName [row, col]`. The complete syntax to accomplish this functionality is in Appendix E Scripting in the Network Operations section.

Network Setters

A very important part of the decision system in Construct is its ability to a set of values within any network within the simulation. An example use case of this functionality could be an agent in the simulation learns of the existence of a web site from an interaction partner. This functionality could then set the interaction network row+column value to 1 between the agent and the website--knowledge of existence preceded ability to interact. The general syntax is `setSomeNetworkName [row, col, value]`. The complete syntax to accomplish this functionality is in Appendix E Scripting in the Network Operations section.

Agent References

Construct processes decisions iteratively for each of the agent values in the `applicable_agents` parameter. The modeler may frequently need to refer to the specific agent under evaluation, and can accomplish this by using the reserved word, `agent`, in the decision declaration.

Time Period References

The modeler may need Construct to refer to the current time period when processing decisions. This is possible using the reserved word `timeperiod` as part of the `with=""` attribute.

Decision References

It is possible that a modeler needs to evaluate decisions both independently and in some form of combined output. Construct supports the chaining of decisions to allow a modeler to meet this need. In the example below the modeler has two independent decisions (XX, YY) and needs to also model the combined XX || YY. Specifically, the modeler is trying to determine if `agent` ever talked with agent 0, or if agent 0 ever talked with `agent`, or if either happened in previous execution of this decision script.

```
<operation name="ReadDecisionOutput">
<parameters>
<param name="decision_names" value="everTalkedTo0" type="decision_name_list"/>

<param name="everTalkedTo0"
value="getInteractionNetwork[agent,0] ||
                                        getInteractionNetwork[0,agent]  ||
 previousResult:bool}" with="agent"/>
</parameters>
</operation>
```

The `previousResult` reserved word shown above allows a modeler to retrieve, as a `construct::stringvar` the result of this decision during the previous time period.

Decisions using `with` statements

A modeler can use all the with functionality described in Part 2 Variables. The authors in fact strongly encourage the use of the `with=”verbose”` attribute and value to help enable more effective debugging of input decks.

The example script above shows one example of the `with=”agent”` attribute that allows Construct to move through the `applicable_agents` list.

The example script at the top of this section also shows an example of the use of macros inside the `with=””` attribute. In this example, Construct creates decisions `d4` and `d5` as a result of the `$i$` macro in the `with` attribute.

The decision parsing differs slightly from the parsing of regular Construct variables. Construct can interpret regular variables that contain non-reserverd words, not contained in single quotation marks (‘) as stringvars. This is not true for decisions, and modelers must use single quotation marks (‘) stringvars to delimit string variables.

Common Gotchas

If Construct is unable to open an input file, it will exit and close. There are times when an error message is not present to the user in this situation! Users should ensure that if they have opened any output files (e.g., in Excel to view the files), they should either close the file or use an application that does not place a file-level lock on the file (e.g., Notepadd++).

APPENDIX D Construct 'Generators'

This section discusses various generators. Several have previously been mentioned, but will be covered here as well.

Constant

This generator is used to set a constant value across a wide-range of tables – it assumes that matrix fill-in should occur from the 'first' cell to the 'last', all inclusive.

Constant generators are frequently used to set a default value, which is then elaborated on with other generators.

```
<generator type="constant">
      <rows first="0" last="nodeclass::agent::count_minus_one" />
      <cols first="0" last="nodeclass::knowledge::count_minus_one" />
      <param name="constant_value" value="1.0" />
</generator>
```

This generator sets every cell in an Agent x Knowledge network to '1'. Note that "nodeclass::agent::count_minus_one" reduces to a single number - there is no check that this is actually an Agent by Knowledge network. Common errors include using the wrong node type.

There is also a Constant3D Generator – the same principle applies, but is applied across 3 dimensions.

```
<generator type="constant3d">
      <rows first="0" last="nodeclass::agent::count_minus_one" />
      <inners first="0" last="nodeclass::agent::count_minus_one" />
      <cols first="0" last="construct::intvar::comms_media_email_id" />
      <param name="constant_value" value="1.0" />
      <param name="symmetric_flag" value="false" />
</generator>
```

This is intended to set values in a 3D matrix, in this case an Agent x Agent x CommunicationMedium matrix.

Random Uniform

This generator is used to set values, using a uniform distribution, between a minimum and maximum value. Every value in between is equally probable.

As with the constant generator, this generator sets every cell between the first and last given cell value across both dimensions, inclusive.

```
<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="0.0" />
      <param name="max" value="1.0" />
</generator>
```

This generator sets values for all agents to all agents between a minimum of 0.0 and a maximum of 1.0.

Random Binary

This generator is used to set values of either 0 or 1 across a range of cells. As with a constant generator, all cells between ‘first’ and ‘last’, inclusive, are filled in. The mean argument identifies the target proportion of 0s and 1s. A value of .3 means that roughly 70% of the cells should have 0s, and 30% should have 1s.

```
<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=".5"/>
      <!-- This is commented out
            <param name="symmetric_flag" value="true"
      -->
</generator>
```

This generator sets values for all agents to all agents so that roughly half (.5) the values will be 0, and the other half 1. If the symmetry flag was not commented out, then early cell value writes would then be overwritten to be consistent with later cell value writes. Since it’s not included, V_{ij} may be 1, while V_{ji} is 0 (or vice versa).

CSV

This is a very useful generator – used for taking externally generated or inferred networks from outside processes (such as ORA) that have been exported into a CSV file.

It’s important, prior to loading, that the indexes of the external file match the Construct file.

CSV network files come in two common types, link-lists and network matrices:

- Link-lists are sparse network files, where every row defines a single edge. They are smaller (in size) when networks are not very dense.

```
            Source, Sink, Value
                  0,    1,    .3
                  0,    2,    .8
                  1,    0,    .1
                  2,    1,    1.0
```

- Network Matrices are dense network files, where every row defines all the connections the row node has. They may have header rows and node labels, or not. They are smaller and more efficient (in size) when the networks are dense.

```
,     0,    1,    2
0,    0.0,  0.3,  0.8
1,    0.1,  0.0,  0.0
2,    0.0,  1.0,  0.0
```

The first generator we will discuss is the “csv_binarize” generator. This is used when you need the final values to be 0s or 1s, but may want to use a weighted network as the input.

```
<generator type="csv_binarize">
      <cols first="0" last="nodeclass::knowledge::count_minus_one"/>
      <rows first="0" last="nodeclass::agent::count_minus_one"/>
      <param name="filesystem_path" value="knowledge_network.csv"/>
      <param name="skip_first_col" value="false"/>
      <param name="skip_first_row" value="false"/>
      <param name="csvrow" value="construct::stringvar::agent_list"/>
      <param name="csvcol" value="construct::stringvar::knowledge_list"/>
      <param name="load_style" value="dense"/>
      <param name="binarization_threshold" value="0.0"/>
</generator>
```

This is an agent x knowledge network. This generator has a bunch of special arguments so we’ll cover them in some detail:

- filesystem_path is the location of the network file – it can be either an absolute or relative path
- skip_first_col – if the file has a header row, set this value to true – this is usually used with dense matrices with a node id to the right.
- skip_first_row – if the file has a header column, set this value to true
- csvrow – this is a list of indices from the row group (agents, in this case) from the construct deck that will be set (0,1,2,3,4,5, etc)
- cvscol – this is a list of indices from the column group (knowledge, in this case) from the construct deck that will be set.
- load_style – this is how you tell Construct what kind of network file the file is, “dense” means the network file is a matrice, “sparse_to_dense_convert” means that the file was a link-list
- binarization_threshold – the threshold used to identify the minimum value, exclusive, that will be set as a ‘1’ – this is used in the csv_binarize generator

The csv and csv_binarize generator are very similar – every argument in the csv generator has already been discussed in the csv_binarize discussion. You use a CSV generator when you want the values listed in the network to be placed untransformed into Construct input.

```
<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_fname" />
      <param name="skip_first_row" value="true" />
      <param name="csvrow" value="construct::stringvar::agent_list" />
      <param name="csvcol" value="construct::stringvar::timeperiod_list" />
      <param name="load_style" value="sparse_to_dense_convert" />
</generator>
```

Transactive Memory Generators

Transactive Memory is an inherently three dimensional concept, agents have their own representations of other agents (called alters) on some third axis. In Construct, that third axis is usually knowledge, but there is belief TM, and one could imagine tasks as a useful supplement.

We have two kinds of transactive memory generators for knowledge as currently defined in modern Construct code:

1) Perception-Based Generators
2) Group-Based Generators

In perception-based generators, each agent is granted a (usually flawed) view of ground-truth (i.e., what agents actually know), and uses that to set values for their own personal alter x knowledge matrix.

```
<generator type="perception_based">
      <ego first="0" last="node class::agent::count_minus_one"/>
      <alter first="0" last="node class::agent::count_minus_one"/>
      <transactive first="0" last="node class::knowledge::count_minus_one"/>
      <param name="false_positive_rate" value="0.0"/>
      <param name="false_negative_rate" value="0.0"/>
      <param name="rounding_threshold" value="0.0"/>
      <param name="verbose" value="true"/>
</generator>
```

The “false_positive_rate” indicates the chance that an alter will be thought to have knowledge they do not actually have. The “false_negative_rate” indicates the chance that an alter will be thought to not have knowledge they do actually have. Both values should range between 0 to 1, and usually should be much closer to 0. Keep in mind that Construct’s homophily drive works based on ‘positive’ homophily, a preference to interact with people that are similar to us. Agents with high false negatives will tend to interact only with themselves.

In group-based generators, agents are identified as members of groups, and they are assigned knowledge based on what is known by the majority of the members of the groups to which they belong.

```
<generator type="group_based">
      <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"/>
      <param name="group_flip_to_positive_rate" value="0.0"/>
      <param name="group_flip_to_negative_rate" value="0.0"/>
      <param name="agent_flip_to_positive_rate" value="0.0"/>
      <param name="agent_flip_to_negative_rate" value="0.0"/>
      <param name="verbose" value="true"/>
</generator>
```

Group members are assumed to have knowledge that the majority of their group members share based on ground-truth. The ‘flip’ variables indicate the chance that the perception of the group or alter (respectively) will be more or less generous.

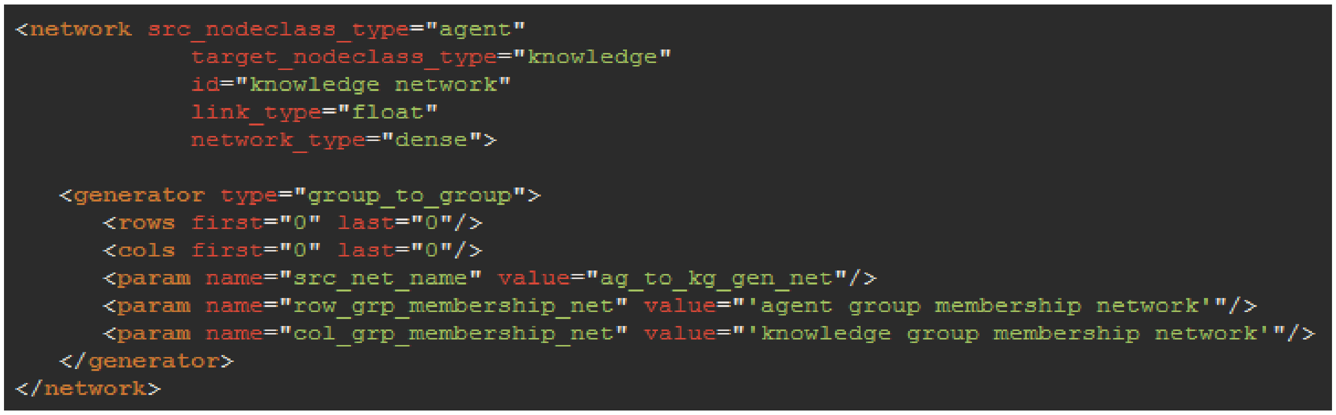
Group to Group Generators

Group to Group Generators are intended to support detailed configurations where nodes that set to be set with a common value may not be contiguous in the node list. All of the previous generators have been what are sometimes called ‘box’ generators, where values are set for a given rectangle of the matrix space defined by the row and col arguments. So nodes that need to be set with a specific value need one box generator per contiguous listing – thus, you may end up needing thousands of box generators for large simulations.

Reading CSV input is the other common way to handle this problem, but some process needs to manage the creation of a composite image – group generators may be a more elegant solution.

An example use-case is you have agents in two groups – you want to assign them to groups randomly, and not define it previously. Using box-generators for this could be very difficult – you’d probably need a pre-processor that creates the CSV input. Instead, you assign each agent to a group at random and then use group generators to set values of interest.

You need a reference to the group membership network Uses a reference to a second network for the mappings as well as references to the two group networks



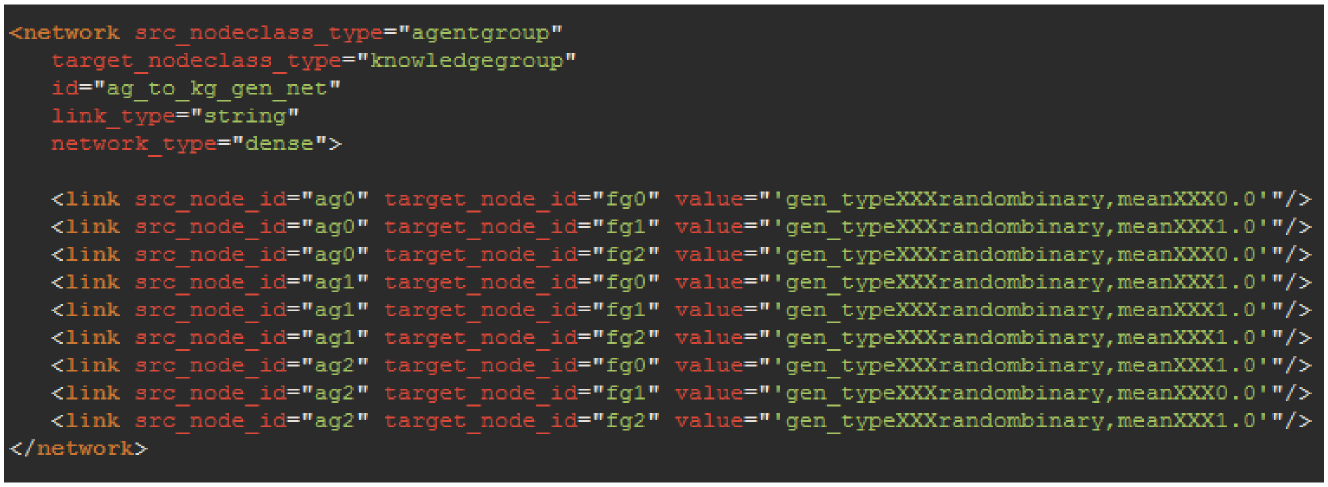
```
<network src_nodeclass_type="agent"
            target_nodeclass_type="knowledge"
            id="knowledge network"
            link_type="float"
            network_type="dense">

   <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 tells the generator where the group to group mapping is found. 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 call this network whatever you want, but src_net_name must have correct name, whatever you choose.

row_grp_membership_net
- This tells the generator where the membership network for the row groups is at. 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 agentgroups.
- Construct uses the “agent group membership network” for its own purposes, and you can use it in this case, but you can also use your own node to nodegroup network. Just make sure you give the correct name in the row_grp_membership_net.

col_grp_membership_net
- This tells the generator where the membership network for the col groups is at. 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 nodeset 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">
       <rows first="0" last="2"/>
       <cols first="0" last="0"/>
       <param name="mean" value="1.0"/>
       <param name="symmetric_flag" value="false"/>
   </generator>

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

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

- Maps nodes to groups.
- Groups must exist in their own nodeset.
- 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 nodeset of agent groups
- Example groups: finance_dept, advertising_dept, friendlist
- Example: knowledgegroup is a nodeset 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 E Scripting

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.

agent
bool
construct
delay_interpolation
details
else
error
get
if
preserve_all_white_space
preserve_spaces_only
preserve_white_space
random
randomBinary
randomNormal
randomUniform
return
set*
spaces_only
static_if
timeperiod
verbose

Note that any string beginning with the word “get” or “set” is reserved for use with referencing networks, and thus the remaineder of the network reference must be one of the networks that Construct is aware of, specified in CamelCase. string that begins with the word “get” or “set” will be treated as a network reference. In addition, any and 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. Finally, one should note that 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`.

Having stated what should not be used 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.

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> 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 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 string
§ **concatenation:** `<Expr>,<Expr>`

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

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 quickly and concisely create comma-separated lists of integers without having to utilize a loop. An example of its usage is 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 here for convenience. In the case that one of the operands is a string, all of 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 provided here together 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>)

These two functions allow for the creation of random numbers. In both 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. dom 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 random normal 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. In order 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.

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 ifs 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 and to a final (and necessary) 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, one 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 { }
```

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, in order 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 or not 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

§ **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. When

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 7 below with the variable result. As can also be seen in Table 7 below, it is necessary to include a return statement in the script in order to specify which result will be returned. If the end of a script does not contain a return statement, the parser will error.

Table 7. Examples of `foreach` loops

Variable	Value

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

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

§ **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 in order to make sure that "impossible" conditions within the code are actually never hit. If a string expression is not give, a default message of "<no error message provided>") will be returned. Note that the string expression can, of course, 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 encoldes 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 sequences 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 work return, but must only have a trialing 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 thuse 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 – see Figure 18 below for examples. First, within the script, an identifier (name) for the macro is placed within 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 effect the text of the expression. So, for example, in Table 8, 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 8. Examples of macros

Variable	Value
<var name="x1” value="$i$:int+1" with="$i$=1"/>	"2"
<var name="x2" value="construct::intvar::x$i$" with="$i$=1"/>	"2"
<var name="x3" value="$2*i:int$+1" with="$i$=1"/>	"3"
<var name="x$i$" value="$i$" with="$i$=(4,5)"/>	x4="4" x5="5"

In most cases, variables should be defined inside of the with tag, as is the case with x1, x2, 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 the course of 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 of Figure 18 above gives us x4 and x5, not a variable with the name x$i$.

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. In order 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 importatnt 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.

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

§ set aggregate network values: `set<NetworkName>[`

<RowExpr>,<ColExprString>,<ValueExpr>]

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 indicies listed in the list ColExprString, a string containing a series of comma-separated integers. The value treturned 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].

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 of 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 IO is extremely time-intensive, and thus should be used with care.

APPENDIX F Construct in High Performance Computing (HPC) Environments

In many ways, the resource we are concerned when we do simulation shifts from man-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 which 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 [IRS_Intervention]. In this project, we were faced with approximately 2000 runs, each of a population of 4000 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 have the ability to 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 [IRS_Intervention], 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 “map-reduce” framework [Map_Reduce], answers the question of how we can “black-box” both the distribution of simulations and the coallation of their output to various machines which can be potentially interrupted at any time. In its most basic definition, the map-reduce framework “maps” out simulations to different machines, ensuring in some way that we will receive output from each machine, and “reduces” all of out output to a single format which we can specify.

Several open-source packages exist to rapidly install the map-reduce framework on computers that researchers have available to them. Importantly, such a framework allows for the researcher to be ignorant of the number of processors he or she has available – the map-reduce concept works exactly the same (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 CONDOR cluster software [Condor] to connect machines in our center, and their DAGMan [Dagman] application, along with some straightforward scripting, to implement the map-reduce framework.

The map-reduce 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 which hide all implementation details of the map-reduce framework and give cheap access to an unlimited supply of machines, such as Amazon's EC2 cluster. Indeed, workflow technologies like SORACS [SORACS] are rapidly evolving to allow for this full workflow to be completed without a research having acess 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 CONDOR cluster framework [Condor] implemented at CASOS. The first step, of course, is to install CONDOR onto machines in your cluster- this step is not covered here, but is described in detail in the CONDOR setup manual, located at [Condor].

Once installed properly, a machine with CONDOR installed on it and a user with submission priviledges 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 of 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 CONDOR. 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 who's 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 (version 2.7) to run the script. For information on how to download python version 2.7 and run a script, consult the Python documentation at [Python].

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 condor 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 man-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 CONDOR will use. Though we do not detail in depth the details of CONDOR 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 CONDOR 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 condor 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 CONDOR programs, such as `condor_q` to check the status of your runs - for full details, see [Condor].

APPENDIX G Construct in Research Literature

Below are some brief descriptions of projects that Construct has been used in. Links to the full publications and project sites are provided below the project title and authors.

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

http://www.irs.gov.edgesuite-staging.net/pub/irs-soi/10rescon.pdf#page=162

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: Kathleen M. Carley, Craig Schreiber

http://brimsconference.org/archives/2007/papers/07-BRIMS-054.pdf

Carley and Schreiber 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

http://www.pnas.org/content/99/suppl.3/7257.short

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

http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA477116

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

http://www.css.gmu.edu/~maksim/pdf/TsvetovatCarley_2003_OfficialReprint.pdf

In order 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.