Construct User Guide

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

February 2022
CMU-ISR-22-102

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

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

This report/document supersedes the following CMU-ISR Technical Reports:
CMU-ISR-21-102, "Construct User Guide", May 2021

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

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

Abstract

This technical report provides users and researchers information on the configuration and use of Construct version 5.3.X. Construct is the CASOS's agent-based simulation software for dynamic network and information diffusion in complex socio-technical systems. The report provides a quick start guide to Construct, a detailed discussion of its configuration, and use through a sample problem and virtual experiment configuration exemplar, and a set of appendices with additional useful information. This document is both an introduction to Construct for casual modelers as well as a reference guide for researchers, modelers, and simulationists.

Table of Contents

Table of Figures

Table of Tables

Construct User Guide

Introduction

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

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

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

Agent Based Models

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

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

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

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

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

More advanced models place agents in a socio-demographic space and separate the physical and the social space. In such models, very few have explicitly modeled the social network. Increasingly, however, researchers are incorporating more realistic network representations, such

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

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

Although frequently lumped together, ABMs vary widely in complexity and computational cost – some are extremely inexpensive (e.g., Swarm (Terna 1998)) 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 (Laird 2019) or ACT-R (Anderson 1993)). This increase in computational expense, however, is matched by construct validity to the actions of cognitively bounded humans: the least computationally expensive (per agent) simulations replicate the behavior of insects (specifically ants) while ACT-R has been able to replicate the brain activation patterns of children solving algebra problems and SOAR has replicated fighter pilot operations in concert with human pilots.

Although economics are an important consideration in picking an agent-based simulation, they should not be the only consideration; the specific phenomena of interest should impose its own set of criteria. For problems of traffic analysis or collision avoidance, swarm agents are particularly appropriate. However, in phenomena with significant cultural freight, such as those involving deception, leadership, participation in group activities, and/or compliance with group norms, these swarm-based technologies offer little useful insight to the policy analyst without additional (expensive) modification and incurring significant increases in computational cost. At the same time, not all group-based phenomena require the detail and expense imposed by high-fidelity models of individual agents. Construct, which can support hundreds and thousands of agents,

supports an appropriate middle ground. It also supports one of the only agent-based models which explicitly unites Herb Simon's dual requirement of bounded rationality, that rationality should be bounded both cognitively, and socially (Simon, 1957). Most of the highest-fidelity models constrain interaction to explicit messages, if at all, and many works entirely in isolation from other agents. Construct, thus, is less expensive and yet more useful for studying group phenomena.

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

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

Although not directly attributable to programming error, there may be other sources of surprising results that should be described. One, the model simulation is, at its core, not a sufficiently good model of the atomic primitive it represents; this is often the case when extending swarm agents beyond issues of traffic and navigation. Two, the experimental approach was not well matched to the empirical reality – if, for example, 75% of adults in the population are internet-literate, but the model assumes that only 10% of the agents will receive information from internet

sources, the model will significantly underestimate the prevalence of information from internet sources, and there may be further cascading effects of that error. Three, the results of Construct's outputs may simply not be well communicated. Relating accurately (and conservatively) the implications of models is itself a skill that must be polished.

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

Introduction to the Report

Construct Versions and This Report

Construct is, like all but end-of-life software, undergoing continuing development in both its capabilities and its implementation. This guide is for Construct version 5.3.X which can downloaded on CASOS's Construct Download page. Construct versions 5.4.X and later will be associated with an updated version of this user guide. Finally, experiment developers and designers should consider subscribing to the CASOS's ORA Google Group for ad-hoc and peer-to-peer assistance as well as assistance from students, staff, and faculty of CASOS.

Conventions Used in This Document

Where feasible, this document quotes a provided example of a Construct experiment configuration file. The sample file can be seen in Section Thoughts on Experimentation, using the courier new font in a reduced font size. This report uses the following typeface conventions:

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

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

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

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

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

Organization of This Overall Report

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

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

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

PART THREE: Construct API details how to create custom models.

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

A Motivating Example

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

It is recommended before continue that the reader contemplate on QoI they intend to answer using this work keeping in mind that Construct's roots lie in social networks and information diffusion. Without well defined QoI, it may be difficult to understand the necessity for many of the explanations this guide will go through. This motivating example will stay with this core capability and defer discussions of additional capabilities and experimental purposes to PART TWO: Construct in Detail.

Core Mechanisms

As previously mentioned, Construct is a framework which can be seen in Figure 1. In this framework we have nodes, networks, models, input, and output. Construct's primary function is to properly interface all these components together like the stairs and hallways in a house. Unlike

a regular house however, Construct can expand, and contract as needed to facilitate an end user's requests. Construct is able to handle an arbitrary number of node sets and networks and inputs for those areas. Construct is also able to handle one to an arbitrary number of models assuming there are no conflicts between models as well as many different types of output. The models and output however are limited to what is already built into Construct.

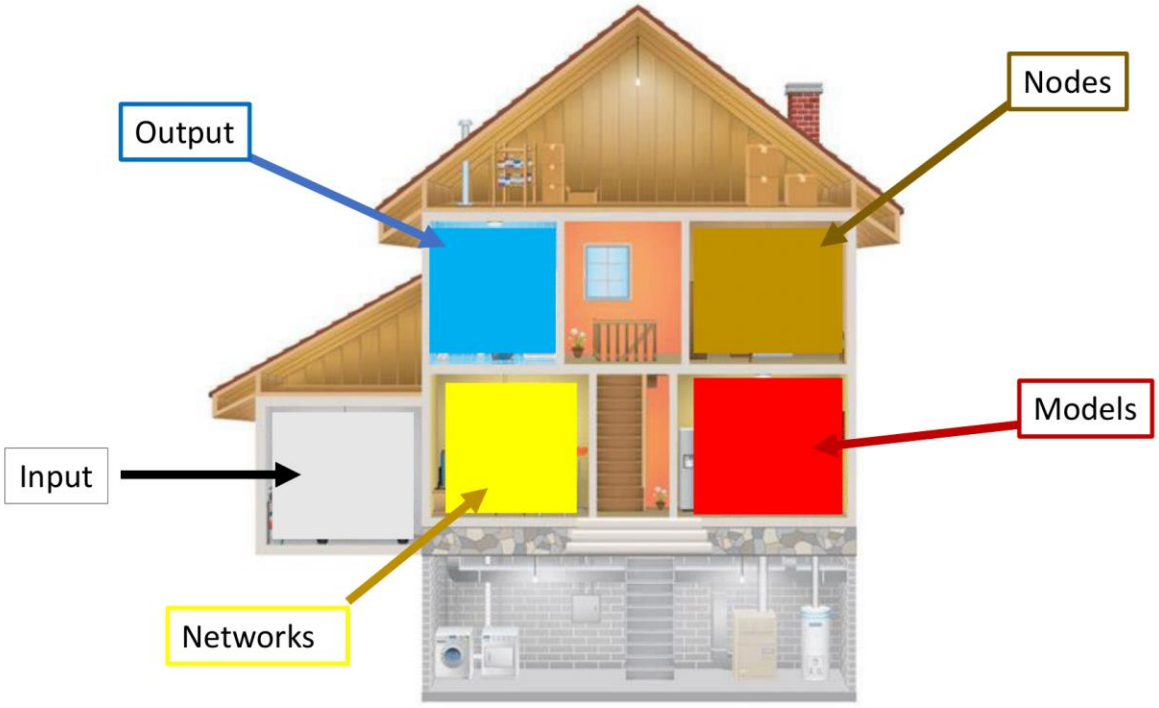

Figure 1: A visualization of the Construct framework as a house.

While Construct is a framework for agent based simulations, the models built into Construct are where the magic happens. The Standard Interaction model is Construct's signature model and the starting point for additional models and modifications. This model combines many different aspects such as decision making, technology restrictions, and information propagation. Agents make decisions about who to interact with, what information to transfer during an interaction, and which communication medium to use for that interaction. The communication mediums present technological limitations as mediums such as books, which are rich in information, but is only one way or face-to-face conversations, which happens instantaneously in both directions. Finally, when these interactions take place, the resulting information spread affects the decision making of each agent such as the search for more exclusive information as an agent's repertoire expands. PART TWO: Construct in Detail goes over the specifics of this and other Construct models.

A Scenario

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

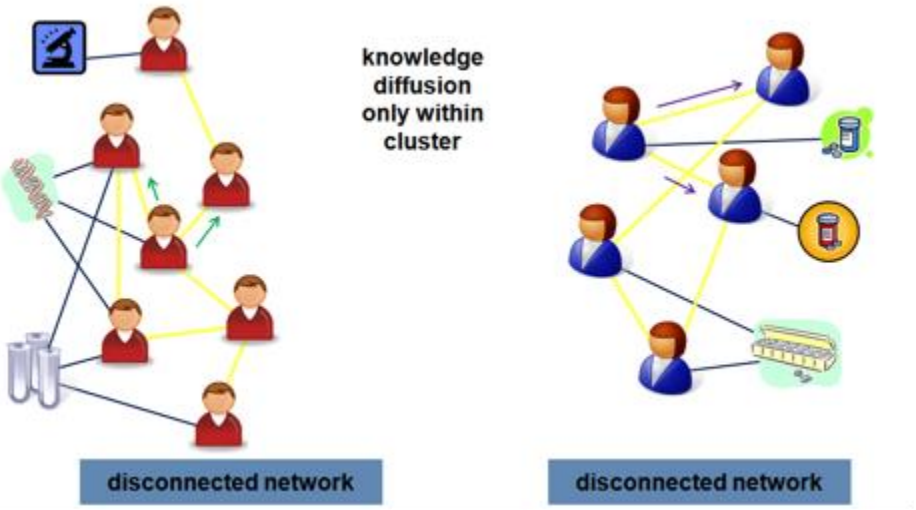

Figure 2. A depiction of two 'clean-room' teams of product developers.

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

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

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

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

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

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

These and other questions can be explored within the Construct framework. In Part 1, we will describe the entities and key relationships between those entities. The treatment in Part 1 is

intended to be useful towards further orienting a potential model builder or a model consumer. Part 2 describes mechanisms at a high-level of detail and is suitable to act as a reference even to a regular user of Construct.

PART ONE: Quick-Start Guide

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

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

The Input Deck

Construct is machine code which requires interpretation in order to properly interface with and give instructions to. The XML file format is the language of choice for interfacing with Construct. XML offers clearer labeling and easier viewing than json at the cost of larger file sizes. This cost is offset as it is not expected that files will not be exceedingly large for submitting instructions to Construct. The following example shows some of the key concepts required to build the various components that will be used in this document.

```
<book title="To Kill a Mockingbird" author="Harper Lee">
   <genre type="Southern Gothic"/>
   <genre type="Bildungsroman"/>
   <rating media="Goodreads" value="4.3"/>
   <rating media="Common Sense Media" value="5"/>
</book>
```

In this example, an XML element is created to represent the book "To Kill a Mockingbird". The overall object is a book, which defines the XML element's name. All XML elements need to be terminated by a forward slash. In this case the book element has sub-elements and is terminated by `</book>` after all other sub-elements have been added. Each element has a set of attributes of the form `[attribute_name]="[attribute_value]"`. Each attribute name must be unique. Sub-elements are used to display multiple similar pieces of information. In this case, the book fits into multiple genres and has multiple media ratings. Construct does not have a strict requirement on the order of sub-elements, however for some components, slightly different results can be yielded by a reordering.

The Objects

Construct organizes sets of objects into what are called node sets. Some examples are agents, knowledge, and time. A singleton example of each of these object classes is referred to (respectively) as an agent, knowledge bit, and a time step. Nodes are available globally in Construct and are used frequently in most areas of Construct. Below is an example of creating a node set.

```
<nodeset name="my nodes">
   <node name=”node 1”>
      <attribute name=”attribute 1” value=”value 1”/>
      <attribute name=”attribute 2” value=”value 2”/>
   </node>
</nodeset>
```

In this example, the node set is named “my nodes” and contains only one node. This node’s name is “node 1” and has two node attributes indicated by the two sub-elements. Node sets are required to contain at least one node. Required node attributes can vary based on the models used, which will be discussed below.

Agents

Agents are the most important class of objects in Construct’s model library. 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 agency and make decisions based on input. Agents are typically treated as homogenous in that given the same inputs, all agents will perform a given action with the same probability as any other agents. The inputs themselves can give agents their identity and uniqueness. Because of this, most of an agent’s decision making comes from a general model which we discuss in later sections. While most models follow this methodology, other models can be created in which agents have fundamentally different decision logic.

Knowledge

A knowledge node represents information and any particular knowledge bit represented by a knowledge node typically represents a single atomic piece of information, such as “Sol is the name of the star at the center of our solar system”, or “Each water molecule is comprised of two hydrogen and one oxygen atom.” It is incumbent on a researcher to keep the stylized representation consistent in their experiments – one bit should not represent “How to pilot a 747-jumbo jet” while another bit represents ‘flight departed’, without proper modification to how those bits connect to the rest of the model.

Time

Many simulations will segment a timeline of events into many small slices referred to as time periods. These time periods are represented by a time node and indicate a specific point in time.

Beginning with the first time node to appear in the input, Construct will move from pointing to the current time node to next time node when a simulation cycle is completed. This continues until the simulation cycle finishes while pointing to the last time node to appear in the input. Construct assumes that all actions and events in a time period happen at the exact same time determined by the current time node being pointed to. In addition, Construct assumes all time nodes are evenly spaced in time.

It is usually good practice to attempt to identify, loosely, a length of time represented by each period. Time periods may be minutes, days, weeks, or months. This representation should be chosen relative to the type of actions being taken during each time period. It may be unrealistic for a human agent interact thousands or millions of times in a second. Likewise, for a human agent to interact only a handful of times in years.

Object Relations as a Network

In Construct, objects such as agent nodes or knowledge nodes can become connected in a network. Networks come in many different types and representation. A network can be represented as a dense matrix in the following example.

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

In this representation, a 1 indicates the presence of link. This can have different meanings in different contexts but for this example, they act as indication that a person is currently taking the specified class if a 1 is present in the element. For instance, Aba is taking Biology and Physics, while Jane is taking Physics and Sociology. Below is an example of creating the above network in Construct.

```
<network name="class network" edge_type="int" default="0">
   <source nodeset="agent" representation="dense"/>
   <target nodeset="class" representation="dense"/>

   <link src_name=”Aba” trg_name=”Biology” value=”1”/>
   <link src_name=”Aba” trg_name=”Physics” value=”1”/>
   <link src_name=”Jane” trg_name=”Physics” value=”1”/>
   <link src_name=”Jane” trg_name=”Sociology” value=”1”/>
   <link src_name=”Lu” trg_name=”Physics” value=”1”/>
   <link src_name=”Lu” trg_name=”Sociology” value=”1”/>
   <link src_name=”Raj” trg_name=”Biology” value=”1”/>
   <link src_name=”Raj” trg_name=”Sociology” value=”1”/>
   <link src_name=”Fred” trg_name=”Biology” value=”1”/>
```

```
</network>
```

The network XML element has three attributes, “name”, “edge_type”, and “default” in addition the various sub-elements. The name attribute is self-explanatory and acts as the key for storing and finding networks in Construct. Construct models will request specific data types which is specified by the edge type. The default attribute defines the value with which to initialize the network links.

After the initial declaration, there are two elements named “source” and “target”. This specifies the origin (the source node set) of a link and its destination (the target node set). An additional dimension will later be added in Part 2: Networks to create three dimensional networks. The “representation” attribute can only be “dense” or “sparse” and indicates the data structure with which that dimension is being stored. In a dense structure, an array is used which has constant lookup time for an index, but consumes memory for every index, even if there is not a link connecting to that index. In a sparse structure, a binary tree is used which has logarithmic lookup time but only consumes memory when the index is connected in a link. When to use which representation is discussed further in Part 2: Networks.

The source and target elements are the only required sub-elements for a network element. An optional number of “generator” and “link” sub-elements can be included in a network element. Generators are macros which can populate large networks with relatively few elements. Links are defined to connect a source node to a target node and can be created using the node’s name (`src_name`, `trg_name`) or the node’s index (`src_index`, `trg_index`). Each link also has a value that must be convertible from a string to the network’s `edge_type`. Additional information on types of generators and creating links can be found in Part 2: Network Generators.

The Knowledge Network

The knowledge network is a binary network connecting agent nodes to knowledge nodes. This defines “who knows what”. Similar to the example above, links in this network represent which agent knows which knowledge bits.

The Interaction Sphere

The interaction sphere is a binary network connecting agents to other agents and defines “who can find whom”. Agents can only initiate contact with other agents if they can find them. On a local level this translates to who knows whom. Agents may not know the agents they’re communicating to, however as symmetry is not required. As an example, newspapers or tv can allow certain agents to communicate information to vast number of individuals even though the broadcaster may not specifically know every individual they are broadcasting to.

Outputs

Researchers usually compare outputs of Construct simulations by examining files written over the course of the simulation. It is outside the scope of this quick start guide to offer in-depth

suggestions on how to deal with large quantities of simulation data. We will instead go over the basic tools available in this guide. Here is an example that we will start with followed by a break down for each component.

```
<outputs>
   <output name="csv">
      <parameter name="network name" value="knowledge network"/>
      <parameter name="output file" value="knowledge.csv"/>
      <parameter name="time periods" value="all"/>
   </output>
</outputs>
```

In this output, the network “knowledge network” is being outputted to “knowledge.csv”. In addition, all time steps will be output to the csv file. Additional types of output will be discussed in Part 2:

Output.

> *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.*

Models and Construct Program Flow

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

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

To accomplish this goal, all models have a similar structure. First models access all their required node sets and networks from Construct and adds a default networks for any optional network not included in the input deck. Each model then performs a standard set of model functions which can be seen in the Figure 3. Each model has a set of five functions that are called in the order shown that ensure models can create, manipulate, parse, and digest messages. Each function is completed by all the models before any model advances to the next step (i.e., each model completes the **Think** function before any model performs the **Update** function). Additionally, most models can be separated into interaction models, in which the primary purpose is to determine how interactions form and create messages to be sent between an interaction pair and manipulation models, in which the primary purpose to manipulate flow and content of messages. Below we describe in detail the five functions that allow the sufficient generality to achieve plug and play functionality of models.

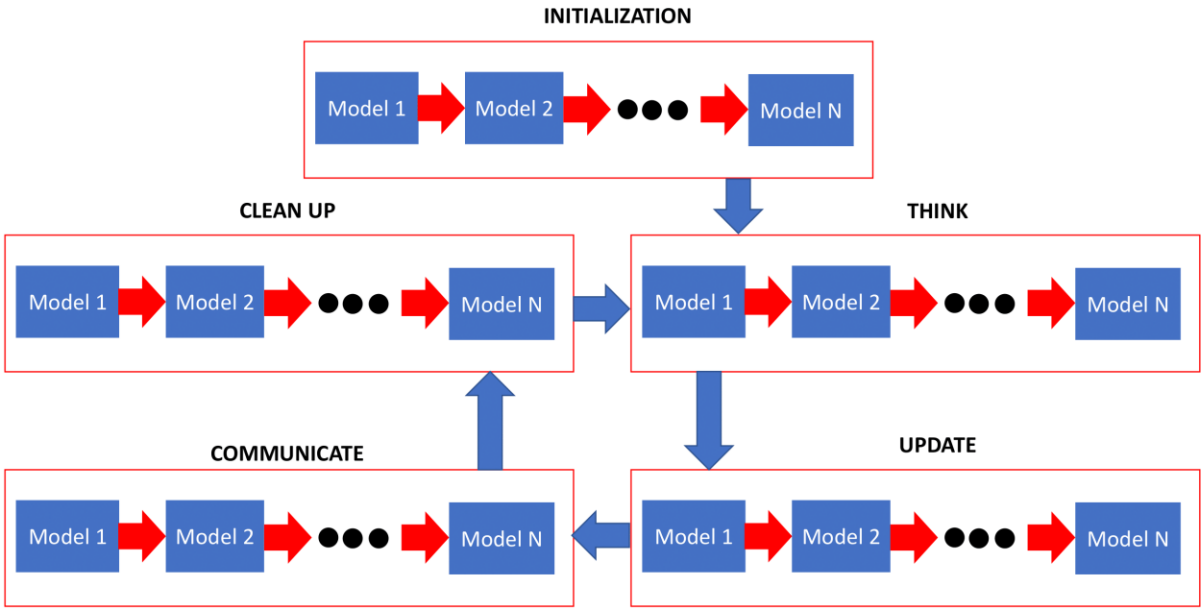

Figure 3. Model execution cycle: after INITIALIZATION executes once, each complete time period begins with all models performing THINK and ends with each completing CLEAN UP.

Initialize Function

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

Think Function

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

Update Function

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

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

Communicate Function

The **Communicate** function takes in each individual message and parses its contents. Each model is responsible for parsing the contents of messages it creates as well as any information it tacked onto another model's message. This allows each model the partition itself from each other without having to worry about additional content that may be in a message. This can be particularly useful as the node and message information space increases as previously existing models will not require modification. This function is the third executed during a time period.

Clean Up Function

The **Clean Up** function allows each model to update various strategies and characteristics based on the communicated messages in preparation for the next time period's **Think** function. This function is the last model function called in the execution cycle. After all models have completed their **Clean Up** function any output routines are processed.

Models

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

- Standard Interaction Model
  - The most fundamental version of Construct's interaction models which relies on proximity, similarity, and expertise to find well suited interaction partners.
- Knowledge Transactive Memory Model
  - An expansion on the Standard Interaction model, this model utilizes an error prone memory of who knows what to provide more realistic interaction seeking.
- Belief Interaction Model
  - An expansion of the Standard Interaction model, this model utilizes beliefs based on known knowledge to modify similarity comparison.
- Task Interaction Model
  - An expansion of the Standard Interaction model, this model utilizes tasks which can be completed by agents based on their known knowledge. Agents then prioritize seeking knowledge required to complete.
- Grand Interaction Model
  - An expansion of the Standard Interaction model that can optionally combine aspects of the Knowledge Transactive Memory Model, Belief Interaction Model, and Task Interaction Model. These affect the items created in a message,

and similarity and expertise between agents. Finally, a belief transactive memory can be enabled which updates beliefs based on influence-based calculations.

- Twitter Interaction Model
  - A model that describes how individuals use the Twitter social media platform to diffuse information. This includes a data structure for events, personal feed of unread events for each agent, and a follower network that affects the ordering of each agent's feeds.
- Facebook Interaction Model
  - A variation of the Twitter Interaction Model that can run simultaneously with the Twitter Interaction Model, replaces the "`twitter follower network`" with the "`facebook friend network`", and creates its own independent social media structure and feed.
- Location Interaction Model
  - Model where agents can learn knowledge based on the location the agent is at.
- Mail Model
  - Model that temporary stalls the transmission of messages based on the medium used.
- Knowledge Learning Difficulty Model
  - Model that makes learning can cause agents to stochastically not learn a knowledge bit when communicated.
- Knowledge Trust Model
  - Model that creates a trust for each knowledge bit which is updated based on other's trust in that knowledge.
- Forgetting Model
  - Model that simulates agents forgetting knowledge which disconnects the relevant link in the "`knowledge network`".
- Subscription Model
  - Model that forwards the content of messages made public based on medium to agents who subscribe to the message sender.

Below is an example of including a Construct model. Note that some models have required or optional parameters. See each model for a list of such parameters.

```
<model name="Standard Interaction Model">
   <!-- Insert parameters here  -->
</model>
```

Thoughts on Experimentation

In this guide, we have discussed how to create nodes, networks that connect those nodes, models that dictate how nodes interact with each other, and output for networks. Combining all of these aspects we get the example input deck for a basic simulation in Construct.

```
<construct>

  <models>
    <model name=”Standard Interaction Model”/>
  </models>

  <nodesets>

    <nodeset name="medium">
      <node name="face to face">
        <attribute name=“maximum message complexity" value="1"/>
        <attribute name=“maximum percent learnable" value="1.0"/>
        <attribute name=“time to send" value="1"/>
      </node>
    </nodeset>

    <nodeset name="agent">
      <generator type=”constant”>
        <count value="50"/>
       <attribute name="can send knowledge" value="true"/>
        <attribute name="can receive knowledge" value="true"/>
      </generator>
    </nodeset>

    <nodeset name="knowledge">
      <generator type=”constant”>
        <count value=”20”>
      </generator>
    </nodeset>

    <nodeset name="time">
      <generator type=”constant”>
         <count value=”10”/>
      </generator>
    </nodeset>

  </nodesets>

  <networks>
    <network name="interaction sphere network" edge_type="int" default="1">
      <source nodeset="agent" representation="sparse"/>
      <target nodeset="agent" representation="sparse"/>
    </network>

    <network name="knowledge network" edge_type="int" default="0">
      <source nodeset="agent" representation="dense"/>
      <target nodeset="knowledge" representation="sparse"/>
      <generator type="random binary">
        <param name="density" value="0.2"/>
      </generator>
    </network>

  </networks>

  <outputs>

    <output type="dynetml">
      <param name="network names" value="interaction network,knowledge network"/>
      <param name="output file" value="output.xml"/>
      <param name="time periods" value="all"/>
    </output>

  </outputs>
```

```
</construct>
```

This example brings together many of the concepts already discussed and presents a few new concepts that will be elaborated upon in the following section. From here many modifications can be made. Many default networks can be explicitly declared to give the simulation additional depth. An example might be to create a super spreader of information that can interact with many people each time step rather than the default of one. Another might be to include additional communication mediums and restrict agent's access to certain mediums. These are all parameters that can easily be modified by the user.

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

In addition, ORA can be used to create and manipulate nodes and networks which can be imported into Construct. This is expanded upon in the section on the DyNetML Network Generator. In addition, many models expect attributes in a node set. In ORA, this can be done by importing attributes from a text file, or by adding attributes to existing node sets and editing the attribute values with tools such as Transform Attribute Values to manipulate attribute values. See the ORA manual for additional details regarding how to create nodes, node attributes, and complex network structure.

PART TWO: Construct in Detail

This section of the report, to some degree, repeats information provided in Part 1: Construct Essentials. This is a deliberate choice by the authors. Part 2 provides in-depth details of the workings of Construct. In this section, a more in-depth discussion will be held on nodes, networks, models, and output.

Parameters

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

```
<construct_parameters>
   <param name="[name 1]" value="[value 1]"/>
   <param name="[name 2]" value="[value 2]"/>
</construct_parameters>
```

Parameter names are limited to those predefined by Construct and are all optional parameters.

Seed

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

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

Verbose Initialization

Verbose initialization provides additional details when loading construct entities (nodes, networks, models, output).

```
<param name="verbose initialization" value="[true | false]"/>
```

Verbose Runtime

Verbose runtime provides additional details about the process of models performing their functions.

```
<param name="verbose runtime" value="[true | false]"/>
```

Working Directory

Working directory specifies the path to which output should be saved. This will obviously be dependent of the operating systems one is using.

```
<param name="working directory" value="[path to dir]"/>
```

Nodes

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

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

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

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

Below is an example for creating a node set.

```
<nodeset name=”agent”>
   <generator type=”constant”>
      <count value=”20”/>
      <attribute name= “can send knowledge” value= “true”/>
      <attribute name= “can receive knowledge” value= “true”/>
   </generator>
   <node name= “Sam”>
      <attribute name= “can send knowledge” value= “false”/>
      <attribute name= “can receive knowledge” value= “true”/>
   </node>
</nodeset>
```

In this example, twenty-one agents are created with the agent at index 20 being named “Sam”. The first twenty agents can send and receive knowledge whereas Sam can only receive knowledge. In this way, many nodes can be created without having to individually specify each node’s attributes. Required attributes are determined by the models included in the XML input.

Additionally if there exists a “[your node set name]” in a DyNetML file, you can import that node set using the “dynetml” generator seen below.

```
<nodeset name=”[your node set name]”>
   <generator type=”dynetml”>
      <param name=”file name” value=”[your file name].xml”/>
   </generator>
</nodeset>
```

This will import the node set in the DyNetML file that matches the name of the node set defined in the input deck. The imported node’s id is used for its name and the list of “property” elements are imported as attributes with “id” being the attribute name and “value” being the attribute value.

Agent Node set

The “agent” node set represents the actors in the simulation. Agents interact with each other, exchange messages that contain information, and make decisions based on interactions. This node set is used most often in Construct’s library of models.

Knowledge Node set

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

Medium Node set

Just like light or sound, communication requires a medium, and different mediums have different properties which effect the entity that moves through it. The “medium” node set has the following set of required node attributes:

1. “maximum message complexity” is an integer that gives an upper limit on the information content of a message. In practice, this means that when the number of items attached to a message is larger than the “maximum message complexity”, items are removed randomly until the number returns to the upper bound. Additionally, adding an item after a message has been created will cause the message to randomly remove an existing item to make room for the new item.
2. “maximum percent learnable” is a float that sets an upper bound on the link strength between an agent and knowledge node in the knowledge network. The stronger that link strength, the more difficult it is to be broken in models like the Forget model. This attribute has the range [0,1].

3. "time to send" is an integer that dictates how many time periods a message must wait before being delivered.

Time Node set

Nodes in the "time" node set represent an instantaneous point in time that all events during a time period occur. The length of the simulation is represented by the number of nodes in this node set. If no time node set is given, the simulation completes one cycle and exits.

Other Node sets

While the node sets listed above are standard node sets in construct models, Construct is not limited to these node sets. The user may create additional node sets for any custom models they wish to create in PART THREE: Construct API.

Node attributes

Some models may require node attributes. Note: Construct models do put restrictions on the number of node attributes a node can have. These attributes act as static properties of a node such as gender, age, or other characteristics. Non-static properties such as activity are instead stored in networks with the time node set as the target dimension. As indicated in the example above to add an attribute to a node, the <attribute> element must be present. These attributes are all unique and repeating the same attribute name more than once will return a runtime error. Below is a list of node attributes (which are case sensitive) for node sets, which models directly require these attributes, what C++ data type these attributes are converted to, and the expected range for these attributes.

Table 1. Node attributes used in Construct.

Attribute Name	Node set	Data Type	Range	Models Used In
can receive beliefs	agent	bool	{true,false}	Grand Interaction Model
can receive beliefTM	agent	bool	{true,false}	Grand Interaction Model
can receive knowledge	agent	bool	{true,false}	Standard Interaction Model, Twitter Interaction Model, Facebook Interaction Model, Location Interaction Model
can receive knowledge trust	agent	bool	{true,false}	Twitter Interaction Model, Facebook Interaction Model, Knowledge Trust Model
can receive knowledgeTM	agent	bool	{true,false}	Knowledge Transactive Memory Interaction Model
can send beliefs	agent	bool	{true,false}	Grand Interaction Model
can send beliefTM	agent	bool	{true,false}	Grand Interaction Model
can send knowledge	agent	bool	{true,false}	Standard Interaction Model, Twitter Interaction Model, Facebook Interaction Model, Location Interaction Model
can send knowledge trust	agent	bool	{true,false}	Twitter Interaction Model, Facebook Interaction Model, Knowledge Trust Model

can send knowledgeTM	agent	bool	{true,false}	Knowledge Transactive Memory Interaction Model
Facebook add follower scale factor	agent	float	[0,∞)	Facebook Interaction Model
Facebook auto follow	agent	bool	{true,false}	Facebook Interaction Model
Facebook charisma	agent	float	[0,1]	Facebook Interaction Model
Facebook post density	agent	float	[0,∞)	Facebook Interaction Model
Facebook quote probability	agent	float	[0,1]	Facebook Interaction Model
Facebook reading density	agent	float	[0,∞)	Facebook Interaction Model
Facebook remove follower scale factor	agent	float	[0,∞)	Facebook Interaction Model
Facebook reply probability	agent	float	[0,1]	Facebook Interaction Model
Facebook repost probability	agent	float	[0,1]	Facebook Interaction Model
influence	agent	float	[0,∞)	Grand Interaction Model
learning rate	agent	float	[0,1]	Forget Model
maximum message complexity	medium	unsigned int	[0,∞)	Standard Interaction Model, Location Interaction Model
maximum percent learnable	medium	float	[0,1]	Standard Interaction Model, Location Interaction Model
susceptibility	agent	float	[0,1]	Grand Interaction Model
time to send	medium	unsigned int	[0,∞)	Standard Interaction Model, Location Interaction Model
Twitter add follower density	agent	float	[0,∞)	Twitter Interaction Model
Twitter auto follow	agent	bool	{true,false}	Twitter Interaction Model
Twitter charisma	agent	float	[0,1]	Twitter Interaction Model
Twitter post density	agent	float	[0,∞)	Twitter Interaction Model
Twitter quote probability	agent	float	[0,1]	Twitter Interaction Model
Twitter remove follower scale factor	agent	float	[0,∞)	Twitter Interaction Model
Twitter reply density	agent	float	[0,1]	Twitter Interaction Model
Twitter repost density	agent	float	[0,1]	Twitter Interaction Model

Networks

Networks are the primary data structures for input and output in Construct. Like node sets, networks must also be uniquely named. Table 2 shows all Construct networks (case sensitive), the associated node sets for that network, the data type for links, and all models that use that network. Descriptions of how a network is used are available in the corresponding models.

Table 2. Network relations to node sets.

Network Name	Source, Target, (and Slice) Node sets	Data Type	Models Used In
agent active time network	agent x time	bool	Location Interaction Model, Standard Interaction Model, Twitter Interaction Model, Facebook Interaction Model
agent current location network	agent x location	bool	Location Interaction Model
agent group belief network	agent group x belief	float	Grand Interaction Model
agent group knowledge network	agent group x knowledge	float	Knowledge Transactive Memory Interaction Model, Grand Interaction Model
agent group membership network	agent x agent group	bool	Knowledge Transactive Memory Interaction Model, Grand Interaction Model
agent initiation count network	agent x time	unsigned int	Standard Interaction Model
agent location preference network	agent x location	float	Location Interaction Model
agent mail usage by medium network	agent x medium	float	Mail Model
agent reception count network	agent x time	unsigned int	Standard Interaction Model
belief knowledge weight network	belief x knowledge	float	Belief Interaction Model
belief message complexity network	agent x time	unsigned int	Grand Interaction Model
belief network	agent x belief	float	Belief Interaction Model
belief similarity weight network	agent x time	float	Belief Interaction Model
belief transactive memory network	agent x agent x belief	float	Grand Interaction Model
communication medium access network	agent x medium	bool	Standard Interaction Model
communication medium preferences network	agent x medium	float	Standard Interaction Model
facebook friend network	agent x agent	bool	Facebook Interaction Model
interaction knowledge weight network	agent x knowledge	float	Standard Interaction Model, Location Interaction Model
interaction network	agent x agent	bool	Standard Interaction Model
interaction probability weight network	agent x agent	float	Standard Interaction Model
interaction sphere network	agent x agent	bool	Standard Interaction Model
knowledge expertise weight network	agent x time	float	Standard Interaction Model
knowledge forgetting prob network	agent x knowledge	float	Forget Model

Network Name	Source, Target, (and Slice) Node sets	Data Type	Models Used In
knowledge forgetting rate network	agent x knowledge	float	Forget Model
knowledge learning difficulty network	agent x knowledge	float	Knowledge Learning Difficulty Model
knowledge message complexity network	agent x time	unsigned int	Standard Interaction Model
knowledge network	agent x knowledge	bool	Standard Interaction Model, Forget Model, Location Interaction Model, Twitter Interaction Model, Facebook Interaction Model, Knowledge Trust Model
knowledge priority network	agent x knowledge	float	Standard Interaction Model
knowledge similarity weight network	agent x time	float	Standard Interaction Model
knowledge strength network	agent x knowledge	float	Forget Model
knowledge transactive memory network	agent x agent x knowledge	bool	Knowledge Transactive Memory Interaction Model
knowledge trust network	agent x knowledge	float	Twitter Interaction Model, Facebook Interaction Model, Knowledge Trust Model
knowledge trust transactive memory network	agent x agent x knowledge	bool	Twitter Interaction Model, Facebook Interaction Model, Knowledge Trust Model
learnable knowledge network	agent x knowledge	bool	Standard Interaction Model
location knowledge network	location x knowledge	bool	Location Interaction Model
location learning limit network	agent x location	unsigned int	Location Interaction Model
location medium access network	location x medium	bool	Location Interaction Model
location network	agent x location	bool	Location Interaction Model
mail check probability network	agent x time	float	Mail Model
medium knowledge access network	medium x knowledge	bool	Standard Interaction Model
physical proximity network	agent x agent	float	Standard Interaction Model
physical proximity weight network	agent x time	float	Standard Interaction Model
public propensity network	agent x time	float	Subscriber Model
social proximity network	agent x agent	float	Standard Interaction Model
social proximity weight network	agent x time	float	Standard Interaction Model

Network Name	Source, Target, (and Slice) Node sets	Data Type	Models Used In
sociodemographic proximity network	agent x agent	float	Standard Interaction Model
sociodemographic proximity weight network	agent x time	float	Standard Interaction Model
subscription network	agent x agent	bool	Subscriber Model
subscription probability network	agent x agent	float	Subscriber Model
task assignment network	agent x task	bool	Task Interaction Model
task availability network	task x time	bool	Task Interaction Model
task completion network	agent x task	unsigned int	Task Interaction Model
task guess probability network	task x knowledge	float	Task Interaction Model
task knowledge importance network	task x knowledge	float	Task Interaction Model
task knowledge requirement network	task x knowledge	bool	Task Interaction Model
transactive belief message complexity network	agent x time	unsigned int	Grand Interaction Model
transactive knowledge message complexity network	agent x time	unsigned int	Knowledge Transactive Memory Interaction Model
twitter follower network	agent x agent	bool	Twitter Interaction Model
unused knowledge network	agent x knowledge	bool	Forget Model

To add network to the input deck, the appropriate sub-elements should be added to the `<networks>` element. An example networks can be seen below.

```
<networks>
   <network name="[your network name]" edge_type="[bool | int | unsigned
   int | float | string]" default="[your chosen default value]">

      <source nodeset="[source node set]" representation="[dense |
   sparse]"/>

      <target nodeset="[target node set]" representation="[dense |
   sparse]"/>

      <!-- Insert links here -->
      <!-- Insert generators here -->
   </network>
```

```
   <network name="[your network name]" edge_type="[bool | int | unsigned
   int | float | string]" default="[your chosen default value]">

      <source nodeset="[source node set]" representation="[dense |
   sparse]"/>

      <target nodeset="[target node set]" representation="[dense |
   sparse]"/>

      <slice nodeset="[slice node set]" representation="[dense |
   sparse]"/>

      <!-- Insert links here -->
      <!-- Insert generators here -->
   </network>
</networks>
```

The top example is a two dimensional (2d) network while the bottom is an example of a three dimensional (3d) network. A 3d network is identified solely by the existence of the `<slice>` sub-element. The representation is a free parameter for the source and target, but the slice representation is required to match the expected representation given by a specific model. In a dense representation, indexes are stored in an array which has a constant look up time. The sparse representation stores indexes in a binary tree, which saves memory, but increases the look up time for a link.

Choosing a representation can drastically change the demands on a simulation. If speed is a concern, making all networks dense will provide the fastest results. While this does seem advantageous, it is a waste of memory if the network is a trivial one with all values being the same. In a sparse representation, links are only stored in memory if their value differs from the default value. If a link is queried and it is not stored in memory, the network assumes that link has the default value. If all values in a network are the same, the network can be initialized with a sparse representation for the source and target dimensions, the default value is then set to the homogenous value, and no links need to be included. In this way, we can represent a trivial network with minimal resources. In addition, if only a few links differ, they can be explicitly defined, while having minimal impact on computation time.

Three dimensional networks typically represent transactive memory. Due to how models handle three dimensional networks the slice dimension representation is required to match the model's specification. These specifications are:

- `"knowledge transactive memory network"` – dense slice dimension representation
- `"knowledge trust transactive memory network"` – sparse slice dimension representation
- `"belief transactive memory network"` – sparse slice dimension representation

Links can be defined in one of the following examples.

```
<link src_name="[src node name]" trg_name="[trg node name]" value="[your
value]"/>
```

```
<link src_index=”[src node index]” trg_index=”[trg node index]”
value=”[your value]”/>

<link src_name=”[src node name]” trg_name=”[trg node name]” slc_name=”[slc
node name]”value=”[your value]”/>

<link src_index=”[src node index]” trg_index=”[trg node index]”
slc_name=”[slc node index]”value=”[your value]”/>
```

Here, the first two examples are for 2d networks, while the latter two examples are for 3d networks. In these examples, either the name or the index is required to identify the corresponding node. Names and indexes cannot be mixed in a link. Node indexes begin at zero and the last node in that node set has index equal to the size of the node set minus one. Using a node’s index produces faster results as finding a node by its name takes logarithmic time with the size of the node set.

Network Generators

Generators allow non-trivial networks to be generated either through importing from another file or using stochastic methods to create links. Generators are applied successively based on their ordering in the input deck and can overwrite links created by previous generators. Below are examples of a generator.

```
<generator type="[type]">
   <rows begin=”first” end=”1”/>
   <cols begin=”1” end=”last”/>
   <param name=”param 1” value=”value 1”/>
</generator>
<generator type="[type]">
   <cols begin=”1” end=”last”/>
   <slcs begin=”first” end=”last”>
   <param name=”param 1” value=”value 1”/>
</generator>
```

In the top example for a 2d network, the generator is only applied to the bounding box defined by the `<rows>` and `<cols>` elements. The bottom example is similar but for 3d networks which includes the `<slcs>` element for the now bounding cube. `“first”` and `“last”` correspond to the index of the first node and last node, respectively. The `“begin”` and `“end”` attributes define the bounds for a given dimension with both being inclusive on the submitted index. This means having a bounding which goes from `“first”` to `“last”` for a dimension will include all nodes in the node set. The `<rows>`, `<cols>`, and `<slcs>` elements are all optional and when not present default the `“begin”` attribute to `“first”` and the `“end”` attribute to `“last”`.

CSV Generator

This generator imports the network from a CSV file. It is expected that the only contents of the file are link values for each corresponding indexes. Files should not have row or column headers. Traditional CSV files are used for 2d networks. 3d networks are imported based on their slice representation. If the slice representation is dense, each element should contain a comma separated

array enclosed by curly brackets (ex. {0,1,0,1,1,0}). If the slice representation is sparse, a dictionary is instead used with the index first and link value second (ex. {4:1,5:-2,9:7}). This difference in implementation is small technical difference between a dense and sparse slice representation. Row, column, slice sizes in the CSV file must match the corresponding node set sizes for the network. This generator does not use any bounding boxes. Below is an example of calling this generator.

```
<generator type="csv">
   <param name=”file” value=”[your csv file].csv”>
</generator>
```

Perception Generator

Transactive memory is built upon many previous interactions, however the initialization of this memory by definition has no previous interactions to rely on. Instead, the memory is initialized by copying elements in another network and adding noise to ensure the transactive memory is similar, but not equivalent. For more information on transactive memory and its usage, see the example used in the Knowledge Transactive Memory Interaction Model. Adding noise is slightly ambiguous when lacking context on what type of variable the noise is being added to. For this reason, there exists different implementations based on the data type of links as well as multiple choices for noise production based on the on type of link value. Below are two examples for using this generator.

```
<network name="[your network name]" edge_type="bool" default="[your default
value]">
   <source nodeset="[source node set name]" representation="[dense |
sparse]"/>
   <target nodeset="[target node set name]" representation="[dense |
sparse]"/>
   <slice nodeset="[slice node set name]" representation="[dense |
sparse]"/>

   <generator type="perception">
      <param name="perception network" value="[your perception network]"/>
      <param name="influence network" value="[your influence network]"/>
      <param name="density" value="[your value]"/>
      <param name="false positive rate" value="[your value]"/>
      <param name="false negative rate" value="[your value]"/>
   </generator>

</network>
```

In this example, a network is created with an edge type of “bool”. The network that the generator is basing the memory off of is the perception network and must have matching “edge_type”. The perception network’s source and target node set must match the example’s target and slice node set respectively. Transactive memory is typically limited to only a small portion of the population even if those nodes interact with other nodes outside of their influence network. The influence network dictates which target nodes are known by the source nodes with its “edge_type” always being “bool”, regardless of the perception network “edge_type”. The

influence network's source and target node set must match with the example's source and target node set, respectively.

The "density" parameter which is required to be in range [0,1] determines what fraction of the "perception network" is copied in the transactive memory. The density parameter is optional and defaults to one if the XML element isn't present. The parameters "false positive rate" and "false negative rate" indicate how error prone the copying process is. A high false positive rate will create more connections than actually exist in the perception network, while a high false positive rate will create fewer. Both parameters are in the range [0,1] with zero creating a perfect copy and one creating an exactly opposite copy of the perception network.

```
<network name="[your network name]" edge_type="float" default="[your default
value]">
   <source nodeset="[source node set name]" representation="[dense |
sparse]"/>
   <target nodeset="[target node set name]" representation="[dense |
sparse]"/>
   <slice nodeset="[slice node set name]" representation="[dense |
sparse]"/>

   <generator type="perception">
      <param name="perception network" value="[your perception network]"/>
      <param name="influence network" value="[your influence network]"/>
      <param name="density" value="[your value]"/>
      <param name="noise implementation" value="[normal | unit normal]"/>
      <param name="variance" value="[your value]"/>
   </generator>

</network>
```

Similar to the previous example, this generator is instead applied to a network with edge type of float. The perception and influence network perform similar roles as previously and also follow the same rules regarding dimension node sets and edge types. The critical difference is how noise is introduced as a simple negating of a float does not produce the same effect in this instance. Here, the parameter "noise implementation" determines how the noise is applied to the values found in the perception network. "unit normal" adds noise such that the resulting values stay in the range [0,1]. This requires that the initial range for the values are in [0,1] to begin with. "normal" does not have any restrictions on range and both initial values and values after adding noise are in the range (-∞,∞). The initial value is used as the mean for a normal distribution along with the parameter "variance" as the variance of the distribution. The resulting value is then sampled from this normal distribution.

The unit normal implementation takes the corresponding perception network value which are in the range [0,1] and transforms them to the range (-∞,∞). This transformation is $\mu = -\ln\left(\frac{x}{1+x}\right)$, where x is the perception network value. This value is then used as a mean in order to sample from a normal distribution with the variance coming from the parameter "variance". The sampled value is then transformed back to the original range of [0,1] by $x' = \left(1 + e^{-\psi}\right)^{-1}$. The normal

implementation does not require a range transformation, so the copied value is sampled from normal distribution with the mean equal to the corresponding value in the perception network and the variance again coming from the variance parameter.

> *This generator can only be added to 3d networks.*

Random Binary Generator

For each link in the bounding box, a one is entered as its value with probability equal to the "density" parameter and zero otherwise.

```
<generator type="random binary">
   <param density="your value"/>
</generator>
```

Random Uniform Generator

For each link in the bounding box, a random uniform value is assigned with a lower bound dictated by the "min" parameter, and an upper bound by the "max" parameter. Both bounds are inclusive bounds. Only a percentage of the links are assigned a value if the "density" parameter is present and assigned to a value less than one. Any link not assigned a value will continue to be its default value.

```
<generator type="random uniform">
   <param name="min" value="[your min value]"/>
   <param name="max" value="[your max value]"/>
</generator>
```

DyNetML Network Generator

Links are created based on the "file" parameter which is expected to be of the DyNetML format. This coincides with the format that ORA uses to save its networks. This generator begins at the element "DynamicMetaNetwork" → "MetaNetwork" → "networks" and searches for a network with name from the parameter "network name". This search only takes place in the first dynamic meta network's first meta network. For each link in the file's network "source", "target", and "value" attributes are parsed to create links. The source and target attributes indicate the names of the nodes in their respective node sets. Each link value is assigned based on the value attribute. If no value attribute is found, the link value is assigned to be 1 converted to the network's C++ data type.

```
<generator type="dynetml">
   <param name="file" value="[your file name].xml"/>
   <param name="network name" value="[network name in your xml file]"/>
</generator>
```

This generator can only be added to 2d networks with default value 0.

Interaction Models

Interaction Models are the backbone of Construct models. They provide the rules in the simulation for who interacts with whom, and what happens when said interaction occurs. While not strictly required that one of these models be included in the input deck, a lack of an interaction model produces a trivial simulation as no interactions occur and all outputs are equal to inputs. Some models inherit other models in order to modify a particular behavior of the base model. This inheritance makes a model mutually exclusive with the inherited model. The figure below shows the inheritance web of Construct's models. Only the models displayed are mutually exclusive with each other.

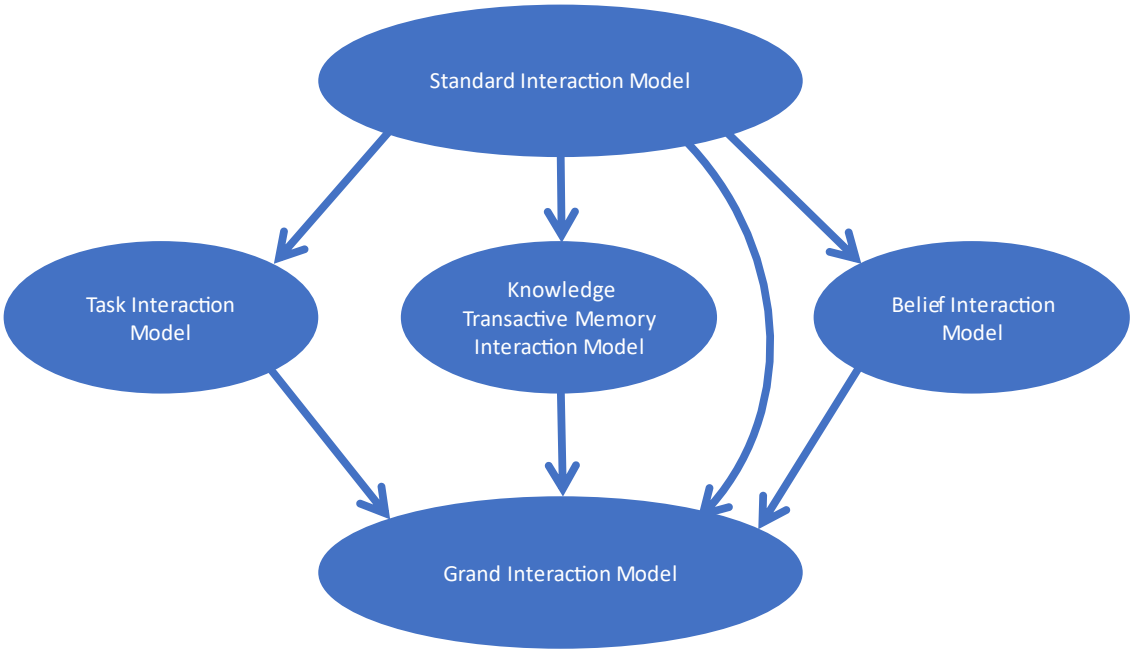

Figure 4: The inheritance of various Construct models. Models with a common source are mutually exclusive and cannot both be present in the input deck.

Models are added to the `<models>` element with the only required attribute being the model's name. Some models may have optional or required parameters. The below example shows how to add a model to the input deck.

```
<model name="[your model's name]">
   <param name="parameter 1" value="value 1"/>
   <param name="parameter 2" value="value 2"/>
</model>
```

Each model description begins with the list of networks that are used by the model. Networks can be required or optional. Required networks must be explicitly declared in the input deck. Optional networks do not need to be declared, are still created by the model, and can be used in an output element. In addition, some networks are only intended as output only. These networks are not expected to be included in the input deck, are reset at the beginning of each time step, and are meant only to output intermediate calculations a researcher may be interested in. For optional networks, the default value for the network is also displayed. The "Range" column indicates the range link values that the model expects. If a link is outside this range, an error may occur or result in undefined behavior. Finally, the access type refers to the methods that are used to access a link value. An iterative access iterates over a dimension with the dimension representation having minimal effect on performance. Random access will access random elements during the simulation applying a computation time penalty for the usage of a sparse representation. Note that this penalty is avoided if links are not held in memory with the network being homogenous. Note that which node sets are required are defined by each model's set of networks.

Standard Interaction Model

Table 3: Networks used by the Standard Interaction Model

Network Name	Required/Optional	Default value	Range	Access Type
agent active time network	Optional	1	{true,false}	iterative access, random access
agent initiation count network	Optional	1	[0,∞)	iterative access
agent reception count network	Optional	1	[0,∞)	iterative access, random access
communication medium access network	Optional	1	{true,false}	random access
communication medium preferences network	Optional	1	[0,∞)	random access
interaction knowledge weight network	Optional	1	[0,∞)	iterative access
interaction network	Output Only	0	{true,false}	random access
interaction probability weight network	Output Only	0	[0,∞)	random access
interaction sphere network	Optional	1	{true,false}	iterative access, random access
knowledge expertise weight network	Optional	1	[0,∞)	random access
knowledge message complexity	Optional	1	[0,∞)	random access
knowledge network	Required		{true,false}	iterative access
knowledge priority network	Optional	1	[0,∞)	iterative access
knowledge similarity weight network	Optional	1	[0,∞)	random access
learnable knowledge	Optional	1	{true,false}	random access
medium knowledge access network	Optional	1	{true,false}	random access
physical proximity network	Optional	1	[0,∞)	random access
physical proximity weight network	Optional	1	[0,∞)	random access
social proximity network	Optional	1	[0,∞)	random access
social proximity weight network	Optional	1	[0,∞)	random access
sociodemographic proximity network	Optional	1	[0,∞)	random access
sociodemographic proximity weight network	Optional	1	[0,∞)	random access

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

The Standard Interaction Model requires two networks: the "knowledge network" and the "interaction sphere network". The knowledge network represents which knowledge a given knows through the links in the network. The interaction sphere network dictates which agents are known to an agent and is not strictly symmetric. An agent can only initiate interactions with agents connected in the interaction sphere with the initiating agent being the source. This does not preclude agents from receiving an interaction initiation from another agent. Next, we'll discuss the various factors impacting the calculation of probability weights which are stored in the "interaction probability weight network".

Proximity between two agents ($PX_{i,j}$) is comprised of three factors, the "physical proximity network" ($PP_{i,j}$), "social proximity network" ($SP_{i,j}$), and "sociodemographic proximity network" ($DP_{i,j}$). Each factor has an associated weight to determine importance in calculating proximity, "physical proximity weight network" ($PPW_i(t)$), "social proximity weight network" ($SPW_i(t)$), and "sociodemographic proximity weight network" ($DPW_i(t)$). The overall proximity is then,

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

The other two factors for determining probability weights are knowledge similarity ($KS_{i,j}$) and knowledge expertise ($KE_{i,j}$). Using K_i^* as the set of knowledge that agent i knows we calculate knowledge similarity and expertise as,

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

where $KW_{i,k}$ is the "knowledge weight network" which is the importance for agent i on agent j being connected to knowledge node k. These two factors are weighted by the "knowledge similarity weight network" ($KSW_i(t)$), and the "knowledge expertise weight network" ($KEW_i(t)$).

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

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

These are probability weights are not calculated for all agents. The "agent active time network" defines the time steps an agent is active and, when not active, the agent cannot interact, and the associated probability weights are set to zero. Additionally, $P_{i,j} = 0$ if the corresponding link in the interaction sphere network is also zero. Agents also only have so many times they can both initiate and be initiated upon in each time step which set by the "agent initiation count network" and "agent reception count network", respectively. $P_{i,j} = 0$ if either agent i's initiation count is zero or agent j's reception count is zero. Finally, both agents require a link to a common communication medium node through the "communication medium access network". Under these rules agents can interact with themselves. This would be equivalent to a person refreshing their memory on a topic.

As mentioned earlier, interaction pair formation is dynamic and as pairs form the interaction probabilities within a set of agents can change. First, an agent with remaining available initiations is chosen randomly with equal probability of selection. The initiator's number of available initiations is then decremented if a receiver is found, and pair formed. Using the probability weights discussed above, an agent is chosen with remaining available receptions, which as with initiations, is then decremented if the pair is formed. Agents can self-interact, in which case reception count is not decremented. Agents can only interact with another agent once and which agents interact with whom is recorded in the "interaction network" with the initiators in the source dimension. This process continues until no more initiators are available.

A number of potential cases can cause the loop to become infinite. A couple examples are an agent with remaining initiation available, but no other agents with remaining receptions available, or an agent that does not have access to a common medium with agents with remaining receptions. An internal counter keeps track of how many times an attempt was made to create an interaction pair. If the counter goes beyond the threshold, the pair formation process prematurely exits, and the simulation continues as normal. This threshold can be set via the optional model parameter "rejection limit" and its default value is the number of agents squared.

Once a potential interaction pair is selected, it is not formally formed until at least one interaction message is created. Both the initiator and initiated create interaction messages that they send to the other. Interaction messages are transmitted of a communication medium represented by the communication medium nodeset. The communication medium node is selected by the initiator via the "communication medium preference network". These network elements act as probabilities, but because not all communication mediums may be common between the initiator and receiver, probabilities are always normalized after excluding invalid combinations. It is expected that if an agent has access to a communication medium node, that the agent's preference for that medium is greater than zero.

From here, the content of a message is constructed. A message minimally contains information about the sender, receiver, and communication medium, but additional information, knowledge in this case, is contained in a set of items attached to the message. First, a check is done on the sending agent for the node attribute "can send knowledge", on the receiving agent for the node attribute "can receive knowledge", and on the "knowledge message complexity network" link for the sending agent and current time step. If the values are "true", "true", and non-zero respectively, knowledge items are added based on the "knowledge priority network" with higher link value increasing the chance a particular knowledge bit is added first. Agents can only add knowledge bits to a knowledge item if they possess that knowledge in the knowledge network. Lastly, knowledge is restricted by the "medium knowledge network" and "learnable knowledge network". A link is required from the message medium to the knowledge bit and a link is required from the receiving agent to that knowledge bit. Once the set of knowledge items are selected are added to a message in a randomized order. When a message is created if the number of items is larger than the medium's "maximum message complexity" node attribute, items are removed to ensure the maximum message complexity is enforced.

Finally, both messages are checked to see if they contain a non-zero number of items. If either message count is non-zero, the interaction pair is formally created. Due to the large amount of complexity that can arise from heterogenous initial conditions, this step is the only step in the model in which progression is not deterministic. If the interaction pair is not created, the process continues with no change to simulation state. Once the interaction pair has been formed its message is added to Construct's central message queue.

Knowledge is parsed by the model "Knowledge Parsing Model". This model is not callable via the input XML file and is automatically created by the Standard Interaction Model and its various variants. Its purpose is to ensure that the knowledge in messages are only parsed once.

Knowledge Transactive Memory Interaction Model

Table 4: Networks used by the Knowledge Transactive Memory Interaction Model.

Network Name	Required/Optional	Default value	Range	Access Type
agent group knowledge network	Output Only	0	[0,1]	iterative access
agent group membership network	Optional		{true,false}	iterative access
knowledge transactive memory network	Optional	0	{true,false}	random access
transactive knowledge message complexity network	Optional	1	[0,∞)	random access

The "Knowledge Transactive Memory Interaction Model" model is an expansion of the "Standard Interaction Model" and inherits and modifies some or all of the model's functions. As this model is executing a modified version of its functions it is mutually exclusive with the "Standard Interaction Model". The primary modification is the addition of a transactive memory (Wegner, 1987) for the knowledge of each agent. Knowledge Transactive Memory

(KTM) is a data storage in the "knowledge transactive memory network" for each ego agent on what knowledge an alter agents know. This memory is incomplete and error prone as it relies on recording previous interactions to populate what agents know about each other. This model builds upon the "Standard Interaction Model" and adopts many of the functions used therein. Because of this, the Knowledge Transactive Memory Interaction Model is mutually exclusive with the "Standard Interaction Model". All required networks in the "Standard Interaction Model" are also required for the Knowledge Transactive Memory Interaction Model and similarly for optional networks.

The primary differences are a modification for how similarities and expertise are calculated, an additional type of item that can be added to a message called a KTM item, and additional parsing to handle this additional type of message item. For this new type of message item, rather than being the sender sharing a piece of knowledge to the receiver, the sender instead shares the information that another alter agent knows a piece of knowledge, which we will refer to as a KTM item. When an ego agent receives either this type of item or a knowledge item, that agent adds that information to their transactive memory only if the alter agent is in the ego agent's interaction sphere. Each agent then has a memory about what other agents knows. This memory is then used instead of the "knowledge network" to calculate the similarity and expertise values for the "Standard Interaction Model" probability weights.

This memory is not perfect however, as any secondhand information is not guaranteed to still be true. One can imagine a game of telephone (AKA Chinese whispers) where a chain of individuals secretly communicates a message to the next person in the chain in hopes of preserving the message. In a perfect system, this would be achievable, but it is almost a certainty that in a real setting that an ego agent will eventually send an item about an alter agent, that the ego believes to be true, but is not. The existence of this divide between reality and perception allows Construct agents to better match social theory and real-world behaviors. For example, Ren et al. (2006) used Construct's transactive memory mechanisms to show evidence that people trained on a task in a group setting are better able to solve a problem than those trained individually and then forced into a group setting.

If an alter is not in an ego agent's transactive memory, a generalized other can be used. By default, this generalized other is the entire population of agents. The probability that the ego agent believes an alter outside of the interaction sphere knows a knowledge bit is equal to the percentage of agents in the node set that know that knowledge bit. This can further be divided in generalized other groups. The creation of these groups is optional and is done so by the inclusion of the "agent group" node set. The "agent group membership network" is a required if and only if the agent group node set is present. This generalized group is then used if an agent is a member of a group in a similar way to the generalized other. If an agent belongs to multiple groups, a group is chosen at random. The percentage of agents in a group that know a knowledge bit is stored in the "agent group knowledge network" and is used as the probability an agent in that group knows a knowledge bit.

In addition to adding knowledge items to a message in the exact same way as the Standard Interaction Model, KTM items are added in a similar way using "can send knowledgeTM" and "can receive knowledgeTM" from a node's attributes. KTM items however have no knowledge priority. The number of KTM items that can be added to a message is restricted by the "transactive knowledge message complexity network". These items are then combined and randomly shuffled with the knowledge items and added to the message. As before if the medium's "maximum message complexity" node attribute is less than the number of items added, items are removed from the message to meet this requirement.

Knowledge items are parsed in the same way except it is also added to the receiver's transactive memory with the alter agent being the sender of the message if the sender is in the ego agent's interaction sphere. KTM items are added to the receiver's transactive memory with the alter agent coming from the item the sender attached rather than the sender being the alter agent. As an example, Agent A may send a message to Agent B that Agent C knows knowledge K. Agent A will then add that Agent C knows knowledge K into their transactive memory. Some obvious situations are avoided when sending a KTM item. An agent cannot send a KTM item about themselves or the intended receiver as both would have perfect memory about what knowledge they know.

Belief Interaction Model

Table 5: Networks used by the Belief Interaction Model.

Network Name	Required/Optional	Default value	Range	Access Type
belief knowledge weight network	Required		(-∞,∞)	iterative access
belief network	Optional	0	(-∞,∞)	iterative access
belief similarity weight network	Optional	1	[0,∞)	random access

The "Belief Interaction Model" is an expansion of the Standard Interaction Model and inherits and modifies some or all of the model's functions. As this model is executing a modified version of its functions, it is mutually exclusive with the Standard Interaction Model. This model uses the belief network which uses the belief node set to describe how strongly agents believe or disbelieve a belief node. A belief is determined by the knowledge bits an agent knows weighted by the "belief knowledge weight network" and is calculated during the **Clean Up** function.

This model then directly modifies the Standard Interaction Model by applying an additive factor $BSW_i(t)\exp(\Sigma_b B_{i,b}B_{j,b})$ to $P_{i,j}$, with B being the "belief network" and $BSW_i(t)$ being the "belief similarity weight network". Those with beliefs that strongly align create very large additions while those with beliefs that conflict create small additions with the exponential ensure all values remain positive.

Task Interaction Model

Table 6: Networks used by the Task Interaction Model.

Network Name	Required/Optional	Default value	Range	Access Type
task assignment network	Optional	1	{true,false}	iterative access
task availability network	Optional	1	{true,false}	iterative access
task completion network	Output Only	0	[0,∞)	random access
task guess probability network	Optional	0	[0,1]	random access
task knowledge importance network	Optional	1	[0,∞)	random access
task knowledge requirement network	Required		{true,false}	iterative access

The `"Task Interaction Model"` is an expansion of the Standard Interaction Model and inherits and modifies some or all of the model's functions. As this model is executing a modified version of its functions it is mutually exclusive with the Standard Interaction Model. In this model agents, in addition to performing interactions as described in the Standard Interaction Model, attempt to complete tasks which are represented as task nodes.

Agents attempt to complete tasks during the **Clean Up** function and can only attempt to complete a task is the task is available and if the task is assigned to them, which comes from the `"task availability network"` and the `"task assignment network"`, respectively. Connections to one or many knowledge nodes may be required which comes from the `"task knowledge requirement network"`. If an agent does not possess a required knowledge node connection, the agents can make a guess for each missing connection. The probability that a guess is correct comes from the `"task guess probability network"` and if the agent successfully guesses each missing connection, the task is completed. Each task that is completed by each agent is recorded in the `"task completion network"`.

During interactions, ego agents will prioritize interacting with alter agents that have knowledge the ego agent is lacking that they require to complete their assigned tasks for that time step. The `"task knowledge importance network"` is used in place of the `"interaction knowledge weight network"` for calculating knowledge expertise. If an agent has no available tasks that they are assigned to, the knowledge expertise portion of the interaction probability weights is calculated as normal by the Standard Interaction Model.

Grand Interaction Model

Table 7: Networks used by the Grand Interaction Model.

Network Name	Required/Optional	Default value	Range	Access Type
agent group belief network	Output Only	0	(-∞,∞)	random access
agent group membership network	Optional		{true,false}	iterative access
belief message complexity network	Optional	1	[0,∞)	random access
belief transactive memory network	Optional		(-∞,∞)	random access
transactive belief message complexity network	Optional	1	[0,∞)	random access

The "Grand Interaction Model" is an expansion of the Standard Interaction Model, Knowledge Transactive Memory Interaction Model, Belief Interaction Model, and Task Interaction Model and inherits and modifies some or all of the models' functions. As this model is executing a modified version of its functions it is mutually exclusive the indicated models. Each model except for the Standard Interaction Model has to be specifically enabled by setting model parameters to "true":

- "beliefs enabled" to enable the Belief Interaction Model
- "tasks enabled" to enable the Task Interaction Model
- "knowledge transactive memory enabled" to enable the Knowledge Transactive Memory Interaction Model.

All network and node requirements for only enabled models are also required for this model. An additional model parameter "belief transactive memory enabled" enables a transactive memory for beliefs when set to "true" in a similar way knowledge transactive memory is utilized. The parameter "belief rate of change" is required if "belief transactive memory enabled" is enabled. This parameter is required to be in the range [0,1] and is saved in the variable α.

Because this model is hybridization of many models, each component of calculating the interaction probability weights can be modified based on which models are currently active. If beliefs are enabled, the additive factor described in the Belief Interaction Model is still applied, however if belief transactive memory (BTM) is enabled, the transactive memory of the ego agent is instead used to calculate belief similarity. The other two models still apply their modifications to how interaction probabilities weights are calculated. The only overlap between the two exists when calculating expertise based on assigned tasks. Here, knowledge transactive memory is used instead of directly observing the knowledge links of alter agents.

Each model also applies their specific modifications to how interaction messages are created. If BTM is enabled, beliefs and BTM's can be sent in as items in messages with the maximum number of items for each determined by the belief message complexity network and the transactive belief message complexity network respectively. Beliefs can only be sent by and received by agents if their node attributes "can send beliefs" and "can receive beliefs" are "true", respectively. Similarly, agents require the node attributes "can send beliefTM" and "can receive beliefTM" to be set to "true" in order to send or receive BTM items. Which beliefs and BTM's that are included in a message is chosen uniformly random. Each message must still respect the overall message complexity of a medium. If the number of items to be added is larger than this message complexity, items are chosen uniformly random to be removed.

When parsing messages in the **Communicate** function, each model that is enabled parses the same message with their own **Communicate** function. If BTM is enabled, belief and BTM items

are also parsed and added to the ego agent's BTM. The belief network does not become modified when parsing a belief item, unlike when parsing knowledge. Beliefs continue to be updated in the **Clean Up** function.

The **Clean Up** function performs similar actions depending on which components are enabled such as updating group knowledge and group beliefs. The notable exception is when BTM is enabled. Beliefs changes become impacted by the belief of others. Agents weight how important others' beliefs are based on how much the agent wants to interact with them. However, not all agents with equal interaction probability weights with will necessarily influence the ego agent equally. Finally, beliefs should not erratically change over time and the agent's feeling of what the belief should be should also impact this calculation. Beliefs are then updated in the following manner,

$$B_{i,b}(t+1) = (1-S_i)\left((1-\alpha)B_{i,b}(t) + \alpha \sum_{k \in K_i^*} V_{b,k}\right) + S_i \frac{\sum_j P_{i,j} I_j BTM_{i,j,b}}{\sum_j P_{i,j} I_j},$$

where B is the `"belief network"`, V is the `"belief knowledge weight network"`, P is the `"interaction probability network"`, BTM is the `"belief transactive memory network"`, S is the agent's node attribute `"susceptibility"` which is in the range [0,1] that determines how important other's beliefs are, I is the agent's node attribute `"influence"` which is in the range [0,∞) that weights how important an agent's beliefs are when the interaction probabilities are similar, and α is the model parameter `"belief rate of change"` and affects the rate at which beliefs change based on the knowledge the ego agent knows.

Twitter Interaction Model

Table 8: Networks used by the Twitter Interaction Model.

Network Name	Required/Optional	Default value	Range	Access Type
agent active time network	Optional	1	{true,false}	iterative access
knowledge network	Required		{true,false}	iterative access
knowledge trust network	Optional	0.5	[0,1]	iterative access, random access
knowledge trust transactive memory network	Optional	0.5	[0,1]	iterative access, random access
twitter follower network	Required		{true,false}	iterative access, random access

The `"Twitter Interaction Model"` aims to simulate the Twitter environment. Twitter-style communications has frequently been modeled in the world of network analysis and agent based simulations. To simulate Twitter, we need two primary concepts, how do agents decide when to create a tweet, reply, quote, or retweet and how do agents decide which tweets, replies, quotes, or retweets to read. To facilitate simulating this, event entities will be constructed rather than creating various proxy functions to emulate the environment without saving the information in a tweet like

format. Additionally, methods are constructed to maintain long-term stability and provide a soft cap on the memory usage of the simulation, as well as implementing twitter followers. The two primary entities this model attempts to investigate is how does the follower network evolve in this environment and how do agent's trust of knowledge change as agents exchange information.

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

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

Most events follow a similar structure for creating an event. The event contains an id, user, timestamp, and a single piece of knowledge. When a tweet is generated, a random piece of knowledge is selected. If the `"Knowledge Trust Model"` is active, the tweet's author attaches their trust of that piece of knowledge to the tweet. Trust is how likely an agent/user believes a piece of knowledge is factually true and is in the range [0,1]. Each agent's trust of a piece of knowledge is stored in the `"knowledge trust network"` ($T_{i,k}$). Any event may also mention other users. This will only become significant when examining how users decide which events to read.

The other types of events: replies, retweets, and quotes, are created in response to reading an event. When an event is read, the reading user has probability based on the agent's node attribute `"Twitter reply probability"` to reply to an event, `"Twitter repost probability"` to retweet a tweet, and `"Twitter quote probability"` to quote an event. When an event is responded to, the response will contain/be about the same piece of knowledge. This is meant to signify that tweets are typically short and primarily about one topic/piece of knowledge. Quotes attach the same trust as the parent event as the quote is a simple forwarding mechanism. Replies and retweets contain the responding user's trust of the parent event's piece of knowledge.

When reading an event, the reading user has the option to follow the event's author. The user decides whether to follow that user based on the event author's node attribute `"Twitter charisma"`. This node attribute has a value in the range of [0,1] and represents the probability the

reading user to follow the event's author. If the event author's node attribute of "Twitter autofollow" is "true", will reciprocate the act of following by the reading user. This probability is modified if the "Knowledge Trust Model" is active. The probability increases based on how similar the reading user's trust of knowledge is compared to the reading user's transactive memory of the event's author. The probability that agent i follows agent j when reading their event is,

$$PF_{i,j} = C_j\left(1 - \frac{1}{|R^*_{i,j}|}\sum_{k \in R^*_{i,j}} \left|R_{i,j,k} - T_{i,k}\right|\right),$$

where C_j is charisma, $T_{i,k}$ is entry from the "knowledge trust network", $R_{i,j,k}$ is the entry from the "knowledge trust transactive memory network" $R^*_{i,j}$ is the set of items agent i has in their knowledge trust transactive memory for agent j.

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

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

Posts are separated into these three priority levels of replies and mentions, follower posts, and all other posts. Within each level, the posts are ordered by their popularity, which we define as how large a post's cascade is divided by how old the post is. In this definition, we only consider

the cascade below a post. Even if a reply is a part of a large cascade, if that reply does not itself have a large resulting cascade, we do not consider that reply as being "popular". Lastly, random posts are sprinkled through the feed after this ordering to promote the agent to branch out and explore new topics/agents. This is achieved by swapping two posts position in a feed. Ten percent of a feed is swapped in this fashion.

Agents read posts during the **Think** function after generating original post. The number of posts an agent reads is determined by the agent's node attribute `"Twitter reading density"`. Multiplying this density by the `"interval time duration"`, gives the mean value of the exponential distribution that determines via sampling the number posts an agent will read in a given time period. When an agent reads a post, all content is put into a message which likewise enters the global message queue. This allows for modification models to apply changes to what type of information agents absorb. Like creating original posts, agents will only read posts if that agent is active via the agent active time network. Agents read posts in order in their feed and only read posts once. Therefore, when a post is read is removed from an agent's feed and can never reenter an agent's feed. With this restriction, posts are still capable of affecting an agent over multiple time periods.

When an agent reads a post, they gain the knowledge that post is communicating. In addition, the reading agent will also absorb information from the local sphere of posts around the read post. Agents do not reply to any posts in this local sphere of posts, nor do they follow any of the posts' users. When reading a root or original post, all direct replies to that post are read. For retweets, the retweeted post is parsed as well as any of the retweet's direct replies. The same is true for quotes. Finally, replies to a reply are parsed as well as the post the parent post to the reply. In the case where the reply is replying to something other than the root post, the root post is also parsed.

When a post is read, knowledge is only added to messages sent to the central queue if the post's author's node attribute `"can send knowledge"` is set to `"true"` and the reading user's node attribute `"can receive knowledge"` is set to `"true"`. Similar node attributes `"can send knowledge trust"` and `"can receive knowledge trust"` are used to add knowledge trust to messages only if the `"Knowledge Trust Model"` is active.

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

As a final note, communication mediums typically take the form of books, billboards, email, conversations, etc. Broadly, Twitter would be considered an internet medium if implemented in the Standard Interaction Model, however, to ensure proper cooperation with other interaction and modification models, the Twitter Interaction model creates its own communication medium which is obscured from other models. Many features in the **Update** and **Communicate** functions operate

regardless of the communication medium used. To avoid potential unintended conflicts functions that do depend on communication medium will simply ignore this extra communication medium. An obvious example would be potential conflicts with the Mail model which can delay a message being sent.

Facebook Interaction Model

Table 9: Networks used by the Facebook Interaction Model.

Network Name	Required/Optional	Default value	Range	Access Type
agent active time network	Optional	1	{true,false}	iterative access
facebook friend network	Required		{true,false}	iterative access, random access
knowledge network	Required		{true,false}	iterative access
knowledge trust network	Optional	0.5	[0,1]	iterative access, random access
knowledge trust transactive memory network	Optional	0.5	[0,1]	iterative access, random access

The "`Facebook Interaction Model`" utilizes most of the same networks and node attributes as the Twitter Interaction Model. This model uses the "`facebook follower network`" instead of the twitter follower network and uses identical model parameters. In addition, required node attributes that begin with Twitter replace that string with Facebook instead. Agents use the same trust of knowledge and transactive memory across both the Twitter and Facebook models. This model however allows an individual to operate significantly differently on two different media platforms.

Location Interaction Model

Table 10: Networks used by the Location Interaction Model.

Network Name	Required/Optional	Default value	Range	Access Type
agent active time network	Optional	1	{true,false}	iterative access
agent current location network	Required		{true,false}	iterative access, random access
agent location preference network	Optional	1	[0,1]	iterative access
communication medium access network	Optional	1	{true,false}	random access
communication medium preferences network	Optional	1	[0,∞)	random access
interaction knowledge weight network	Optional	1	[0,∞)	iterative access
knowledge network	Required		{true,false}	iterative access, random access
knowledge priority network	Optional	1	[0,∞)	iterative access
location knowledge network	Optional	1	{true,false}	iterative access
location learning limit network	Optional	1	[0,∞)	random access
location medium access network	Optional	1	{true,false}	random access
location network	Optional	0	[0,1]	iterative access, random access

The "Location Interaction Model" creates an environment where agents can learn knowledge based on their current location and without interacting with others. Examples might be an archaeologist at a dig site, a child at a playground, or a scientist simulating agent based models. In each case, agents are learning information from the environment rather than from another agent.

Each agent begins at exactly one location given by the "agent current location network". Each location has a set of knowledge available to be learned given by the "location knowledge network". The "location learning limit network" puts a cap on how many knowledge bits can be learned in one time step. During the **Think** function, agents create messages that are sent to themselves if they are active given by the "agent active time network". While the transfer of information could be self-contained, it is important to pass the knowledge through a message to ensure other models can observe and interact with the process of learning the knowledge.

A medium is required to be used for creating this message. The "communication medium access network" provides for which communication medium an agent can use to create a message. The "communication medium preference network" dictates the probability weight the agent will use that medium. In addition, the "location medium access network" further restricts which mediums can be used at a location. Knowledge items will then be added to the message based on the available knowledge at that location. Knowledge is prioritized based on the "knowledge priority network". Because messages are restricted by the location's learning limit and the medium's "maximum message complexity" node attribute, knowledge with large priority are more likely to be added to a message if either of these restrictions are met. If the Task Interaction Model is enabled, instead of attempting to learn all possible knowledge at a location, only the knowledge required for completing assigned tasks are learned. These messages are then parsed in the **Communicate** function and knowledge links are created appropriately.

The model ends in the **Clean Up** function with agents deciding on their next location. Agents are capable of moving locations stochastically with the probability to move to a specific location coming from the "location preference network". The "location network" determines which locations are accessible based on an agent's current location.

Modification Models

These models do not inherently generate new messages. Rather a modification model aims at manipulating, removing, or in some cases generating new messages in response to existing messages. An example is the Subscription Model, which can copy messages and forward them different agents. Some models parse messages to record statistics or modify networks and even have **Clean Up** function usage. As with the interaction models, all networks associated with the model are presented at the beginning of the model description.

Mail Model

Table 11: Networks used by the Mail Model.

Network Name	Required/Optional	Default value	Range	Access Type
agent mail usage by medium network	Optional	1	[0,1]	random access
mail check probability network	Optional	0.5	[0,1]	iterative access

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

Knowledge Learning Difficulty Model

Table 12: Networks used by the Knowledge Learning Difficulty Model.

Network Name	Required/Optional	Default value	Range	Access Type
knowledge learning difficulty network	Required		[0,1]	random access

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

Knowledge Trust Model

Table 13: Networks used by the Knowledge Trust Model.

Network Name	Required/Optional	Default value	Range	Access Type
knowledge network	Required		{true,false}	iterative access, random access
knowledge trust network	Optional	0.5	[0,1]	random access
knowledge trust transactive memory network	Optional	0.5	[0,1]	random access

The "`Knowledge Trust Model`" adds a trust to knowledge items based on an agent's trust in the knowledge bit based on the "`knowledge trust network`". Trust is a value on the range [0,1] which represents the probability that the agent thinks the piece of knowledge is factual and otherwise, the knowledge is considered a falsehood. This entity operates as a mechanism to simulate agents deciding between fact and misinformation. When agents learn a new knowledge,

the corresponding trust is set to 0.5. The addition of trust does not affect the parsing of the knowledge item.

Knowledge trust is also parsed by this model and that information is added to the "knowledge trust transactive memory network". An agent's trust is updated during the **Clean Up** function based on the trust of the agent's alters in their transactive memory. Agents slowly change their trust of a piece of knowledge based on the model parameter "relax rate" which is a value in the range [0,1]. The updated knowledge trust T' is then calculated as follows,

$$T'_{i,k} = (1 - rr)T_{i,k} + \frac{rr}{|R^*_{i,j}|} \sum_{k \in R^*_{i,j}} R_{i,j,k}$$

where rr is the relax rate $R_{i,j,k}$ correspond to elements in the "knowledge trust transactive memory network" and $R^*_{i,j}$ is the set of all knowledge bit indexes ego agent i has in their trust transactive memory for alter agent j.

Forgetting Model

Table 14: Networks used by the Forgetting Model.

Network Name	Required/Optional	Default value	Range	Access Type
knowledge forgetting prob network	Optional	0.1	[0,1]	random access
knowledge forgetting rate network	Optional	1	[0,1]	random access
knowledge network	Optional		{true,false}	iterative access
knowledge strength network	Optional	0	[0,1]	iterative access
unused knowledge network	Output Only	1	{true,false}	random access

When messages are parsed during the **Communicate** function, the "Forgetting Model" can cause links in the "knowledge network" to be removed if the associated knowledge hasn't been used in a time period. This process only happens if the "knowledge network" is already loaded as a network. The function also increases the corresponding link for that knowledge bit in the "knowledge strength network" by the receiving agent's "learning rate" node attribute. Then during the **Clean Up** function, if an agent has neither sent nor received a piece of knowledge, the corresponding value in the "knowledge strength network" is decremented with probability from the "knowledge forgetting prob network". The amount that is decreased comes from the "knowledge forgetting rate network". If the knowledge strength between an agent node and knowledge node reaches the lower bound of zero, the corresponding link is removed from the "knowledge network". The knowledge strength is initially checked to ensure that all links in the "knowledge network" are correspond to non-zero values in the "knowledge strength network" as well as lack of links corresponding to zeros. Note that the knowledge network is unaffected by this initialization; only the "knowledge strength network" is modified if a discrepancy occurs.

Subscription Model

Table 15: Networks used by the Subscription Model.

Network Name	Required/Optional	Default value	Range	Access Type
public propensity network	Optional	0.01	[0,1]	random access
subscription network	Optional	0	[0,1]	iterative access
subscription probability network	Optional	0.01	{true,false}	random access

When a message is parsed by the "`Subscription Model`" during the Communicate function, that message is added to an internal public queue with probability from the sender's link in the public propensity network. This public queue is only accessible by the Subscription model. In the Clean Up function, agents will subscribe to the sender of a message in the public queue with probability from the "`subscription probability network`". Finally, during the Think function, public messages from the previous time step are copied and forwarded to all subscribing agents.

Output

There are three output methods currently provided by Construct, each of which uses a different file format. Each output routine follows a similar structure of a the `<output>` XML element with a list of `<parameter>` sub-elements. The type of output is determined by the "`type`" attribute in the output element. Below is an example.

```
<output type="[your output type]">
   <parameter name="[your parameter 1]" value="[value 1]"/>
   <parameter name="[your parameter 2]" value="[value 2]"/>
</output>
```

CSV

The CSV output routine uses "`csv`" for the type attribute. This output requires three parameters; "network name" which specifies the name of the network to be recorded, "output file" which indicates the name of the output file, and "`time periods`" which designates whether all or just the last time step is recorded. Any network that is created during the network loading process, or during model construction (before a model's **Initialization** function) can be used as output. Any file or file path can be used to designate the output file with the only requirement being the ".csv" extension be at the end of the string. Finally, three possible values are allowed for "`time periods`".

- "`initial`" will output only the initial state of the network before the simulation begins.
- "`last`" will output the final state of the network at the end of the simulation as well as the initial state of the network.
- "`all`" will output the initial network and the network at the end of each time period in the in simulation.

The structure of the CSV file contains the source node set on the rows and target node set for the columns. Node names are not included to reduce file size; rather the row number corresponds to the source node index and the column number corresponds to the target node index. In the CSV file, an empty line indicates the transition from one time step to the next. The row number is then reset at the empty line for determining which row corresponds to which node index. When a 3d network is used as output, at each element braces contain data for the slice dimension. For a dense representation, the elements appear as comma separated values such as `{v0,v1,v2,…,vN}`. For a sparse representation, the elements appear in a dictionary format such as `{i0:v0,i3:v3,…,iN:vN}`.

DyNetML

The DyNetML output routine uses `"dynetml"` for the type attribute. This output requires three parameters; `"network names"` which is a comma separated list (`"net_name1, net_name2, net_name3"`) of the names of all networks to be recorded, `"output file"` which indicates the name of the output file, and `"time periods"` which designates whether all or just the last time step is recorded. Networks can be any network that is created during the network loading process, or during model construction (before a model's **Initialization** function). Any file or file path can be used to designate the output file with the only requirement being the ".xml" extension be at the end of the string. The output XML file is consistent with the DyNetML format (DyNetML is an XML derivative language for exchanging rich social network data) and can be directly loaded in XML parsing software such as ORA. Finally, two possible values are allowed for `"time periods"`.

- `"initial"` will output only the initial state of each network before the simulation begins.
- `"last"` will output each network's final state at the end of the simulation as well as the initial state of each network.
- `"all"` will output the initial state of each network and the state of each network at the end of each time period in the in simulation.

Messages

The Messages output routine uses `"messages"` for the type attribute and has only one parameter `"output file"`. Any file or file path can be used to designate the output file with the only requirement being the ".json" extension be at the end of the string. The JSON representation in the output file contains all messages sent each time step. Messages contain the sending agent index, receiving agent index, the name of the communication medium used, and a list of message items. Each message item contains a set of attributes, indexes, and values. Below is an example of a message in JSON format.

```
{
   "sender" : 93,
```

```
    "receiver" : 0,
    "medium name" : "CommunicationMedium_0",
    "Items" : [
       {
          "attributes" : {"belief"],
          "indexes" : {
             "belief" : 3
          },
          "values" : {
             "belief" : 4
          }
       }
    ]
}
```

In the above example, agent 93 sent a message to agent 0 using "CommunicationMedium_0". The message contains only one item. The "belief" attribute indicates this item contains a belief that is to be sent to the receiver. The belief in indexes indicates the belief node index being communicated is 3. The belief in values indicates the value of the sender's corresponding belief link value. This gives a complete list of all messages sent using Construct's central messaging system

PART THREE: Construct API

The Construct Application Programming Interface (API) exists in the Consturct-API repository on the CASOS GitHub. The API consists of an executable for each operating system that can call the API functions, a header and statically linked library to allow the use of Construct's classes, functions, and namespaces, and the source files for the Construct API which produces a dynamically linked library (DLL). The executables seen in this repository differ from the introductory executable used as they require the file Construct_DLL.dll in order to execute. This section is geared towards those wishing to develop their own model. The implementation and example language used is C++. How this API can be used to create custom models, output, and unique users for social media models is discussed below. This section focuses on overall concepts and the detailed API documentation can be found on the CASOS Construct main page as well as through the GitHub repository.

Creating Custom Models

Construct has, throughout its history, constantly evolved as development continues to improve the underlying code base. Rather than develop for every possible case a modeler may require, the ability to create custom models was implemented. Models can be created that are completely stand alone and do not interact with any other part of Construct. Models can also be created to interact with the shared content between models. Finally, through class inheritance, models can copy components of an existing model and apply alterations to those components. This section will go over all the requirements a Construct model must meet to eliminate undefined behavior.

To create a custom model, a new class or struct must be created that inherits from the `Model` class. This can be seen in the `Template` class in the Template.h file. In addition, Template.cpp also contains many examples of using various Construct functions and classes. Classes that inherit from the `Model` class can reimplement five virtual functions that correspond to the steps of Construct simulation cycle detailed in Models and Construct Program Flow. If the base virtual functions are not replaced, they do not have any effect on the Construct simulation, but will output warnings when Construct parameter `"verbose runtime"` is set to true and when using the DEBUG executable version. Finally, all Construct model constructors require as input the pointer to the `Construct` class which is which is subsequently passed to the `Model` constructor. This will initialize many of a model's member variables including the now saved Construct pointer (`Model::construct`) and pointers to the Node (`Model::ns_manager`), Graph (`Model::graph_manager`), and Random (`Model::random`) managers. A Construct model's constructor can also accept a `dynet::ParameterMap` as input and allows for model parameters to be passed to a model.

Simply creating the class however is insufficient to allow Construct to create the model when requested via the input deck. The entry point for a custom model is in the function `dynet::create_custom_model`, which is defined in Supp_Library.h and Supp_Library.cpp This

function is called by the Model Manager class and passes the Construct pointer (`construct`), the model's parameters (`parameters`), and the name of the model the manager is trying to load (`model_name`). In the function is a series of if-else statements used to select appropriate model constructor. To allow a custom model to be created, the if-else statement checks if the `model_name` is equivalent to each model's name. The string conditioned on in this statement should match the string submitted to the `Model` constructor, otherwise an assertion will be raised. The contents of this statement should allocate the custom model's pointer using `new`. The created pointer should then be immediately returned by the function. Once a `Model` pointer is returned from the function, the Model Manager takes ownership (the model pointer will be deallocated by the Model Manager and should not be deallocated by any other entity). Once the pointer ownership has been transferred and no exceptions have been raised, Construct will automatically call the appropriate functions of the custom model during the simulation cycle.

Construct throws `dynet::construct_exception` as exceptions which is derived from `std::exception` as well as other exceptions derived from `dynet::construct_exception`. Construct's exceptions protects for the many possible ways the end user can include potentially problematic input such as setting a float parameter to "duck". Exceptions that are not Construct exceptions indicate possible bugs and should be reported to the ORA google group. Additionally, when using the DEBUG compilation flag and the corresponding executable in the Debug folder, Construct will check various conditions and raise assertions upon failure. These assertions contain only the assertion message, which begin with the phrase "Construct has raised an assertion".

Creating Custom Output

As with models, the API allows for the creation of custom output routines. A custom output may be more advantageous than coding an output into a model as the logging of output is guaranteed to happen after all models have completed their clean up function. There are many similarities between `Models` and `Output` in terms of their injection in Construct. All outputs inherit from the `Output` class and the Ouput Manager calls the function `dynet::create_custom_output` which takes as input a similar set as `dynet::create_custom_model`. Similarly, a series of if-else statements are used to select the correct constructor, but using the input `output_name` instead of `model_name`. The classes and return statement also follow a similar structure except the Output Manager takes ownership of `Output` pointers. Classes that inherit from the `Output` class have no strict requirements on the form of its constructor and should reimplement `Output::process` which is called by the Output Manager after models have completed their clean up functions.

Creating Custom Social Media Users

In both the Twitter Interaction Model and Facebook Interaction Model, a `media_user` class is used to dictate the decisions an agent can make in a social media, such as whether to reply, what to add to a reply, or whether to follow another agent. As with the previous two methods, `dynet::create_custom_media_user` expects a `media_user` pointer to be returned. An else if statement can again be used with condition depending on the submitted node iterator's attributes.

It is up to the developer as to what new attribute to use to signal the condition. Unlike the previous two methods, a NULL pointer should not be returned. See the API documentation to determine what methods can be modified and the possible effects of those modifications.

Full Control of a Custom Construct Construction

Developers confident in their knowledge of C++ and of Construct can create an instance of Construct using its constructor. This constructor only initializes empty managers and sets the random seed based on the submitted seed value. All Construct parameters, nodesets, networks/graphs, models, and outputs require manual creation. This obviously allows for significantly increased customization, but also allows for pointers outside of Construct to be added to models or outputs as they can be constructed outside of Construct. Once all the components are loaded, the simulation can be started by calling `"Construct::run"`, which initializes all models and begins the simulation cycle. A try statement should surround the section of code that loads all the Construct components to catch any string exceptions thrown. The `"Construct::run"` function is already wrapped internally with the appropriate try and catch statements and returns false if any exceptions is thrown. `"Construct::run"` can only be called once otherwise, an assertion is raised.

References

Anderson, J. R. (1993). Rules of mind. *Psychology Press*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Laird, J. (2019). Soar Cognitive Architecture. *MIT Press*.

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

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

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

Terna, P. (1998). Simulation Tools for Social Scientists: Building Agent Based Models with SWARM. *Journal of Artificial Societies and Social Simulation, 1(2)*. http://jasss.soc.surrey.ac.uk/1/2/4.html.

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

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

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

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

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

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

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

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

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

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

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

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

Appendices

Appendix A A History of Construct

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

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

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

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

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

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

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

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

world data. This was used to assess revolutionary activity during the Arab Spring, and so to predict revolutionary behavior given changes in what was covered in the news (Joseph, Carley, et al., 2014).

In 2020, Construct-TM returned to its original name of Construct with many of the components that were tied into one model, compartmentalized into separate models. Along with utilizing default values, the overall complexity for new users was drastically reduced. Many of the plethora of input options including an in-string scripting language and output routines were removed in place of an Application Programming Interface (API). The increase in popular and accessible scripting languages like Python for data analysis, removes the necessity this extra complexity. This decrease in complexity allows for a lower barrier of entry for those wishing to use the software. Another critical advance is the utilization of a network data structure that can be dense or sparse along a dimension depending on the user's needs. The ability to freely decide whether to sacrifice memory space to increase speed, or sacrifice speed in exchange for less memory space used allows users far more control and thus access to much larger simulations than previous possible.

These modifications allowed for the addition of a social media model to model medias such as Twitter and Facebook. These models simulate the actual media structure using events and feeds rather than using a proxy for transmitting the contents of tweets and posts. In addition, a mechanism was added to allow others to display their trust in a piece of knowledge. For example, I trust the statement "The earth is round" or I mostly distrust the statement "This person is innocent". Finally, the capability was implemented using the Construct API to allow custom social medias to be created as well as customization of how the participating agents function in the existing or custom social media models.

Appendix B Construct in High Performance Computing (HPC) Environments

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

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

interested user, and then give examples of how to utilize the tools for HPC environments employed at CASOS.

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

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

Several open-source packages exist to implement the MapReduce framework on computers that researchers have available to them. Importantly, such a framework allows the researcher to be ignorant of the number of processors he or she has available – the MapReduce concept works in the same way (though with obvious time increases) on a single core as it does on the millions of cores used by companies such as Google. We use the HTCondor (formerly Condor) High Throughput Computing (HTC) software (https://research.cs.wisc.edu/htcondor/) to connect machines in our center, and their DAGMan (Directed Acyclic Graph Manager) (https://research.cs.wisc.edu/htcondor/dagman/dagman.html) application, along with some straightforward scripting, to implement the MapReduce framework.

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

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

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

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

The first column of the file simply labels the condition being changed - though this is not necessary (we will never tell Construct to look at this value), it is naturally useful in keeping track of which lines of your parameters file refer to which condition. Once this parameter file has been specified, we need some way to submit (in this case) three different runs to multiple machines via HTCondor. To do so, we need to complete three further steps.

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

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

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

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

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

As we know from the above sections, this tells Construct to read the agent_count variable from the first (zeroth) row and the second (zero-indexed) column of the csv file "params.csv". Once we have done this, we can add our XML file to the directory we are working in (i.e., at the same level as conditions_file.csv). Note that this implementation will only require us to have a single model file, which is desirable with respect to person-hours required to change the model and the amount of space needed to store results.

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

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

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

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

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

Appendix C Construct in Research Literature

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

Predicting Intentional and Inadvertent Non-compliance

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

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

Agent Interactions in Construct: An Empirical Validation using Calibrated Grounding

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

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

Computational organization science: A new frontier

By: Kathleen M. Carley (Carley, 2002)

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

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

By: Kathleen M. Carley (Carley, 2006)

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

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

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

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

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

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

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