The Matrix: An Agent-Based Modeling Framework for Data Intensive Simulations

Parantapa Bhattacharya
University of Virginia
parantapa@virginia.edu

Saliya Ekanayake
Lawrence Berkeley National Laboratory
esaliya@lbl.gov

Chris J. Kuhlman
University of Virginia
cjk8gx@virginia.edu

Christian Lebiere
Carnegie Mellon University
cl@cmu.edu

Don Morrison
Carnegie Mellon University
dfm2@cmu.edu

Samarth Swarup
University of Virginia
swarup@virginia.edu

Mandy L. Wilson
University of Virginia
alw4ey@virginia.edu

Mark G. Orr
University of Virginia
mo6xj@virginia.edu

ABSTRACT

Human decision-making is influenced by social, psychological, neurological, emotional, normative, and learning factors, as well as individual traits like age and education level. Social/cognitive computational models that incorporate these factors are increasingly used to study how humans make decisions. A result is that agent models, within agent-based modeling (ABM), are becoming more heavyweight, i.e., are more computationally demanding, making scalability and at-scale simulations all the more difficult to achieve. To address these challenges, we have developed an ABM simulation framework that addresses data-intensive simulation at-scale. We describe system requirements and design, and demonstrate at-scale simulation by modeling 3 million users (each as an individual agent), 13 million repositories, and 239 million user-repository interactions on GitHub. Simulations predict user interactions with GitHub repositories, which, to our knowledge, are the first simulations of this kind. Our simulations demonstrate a three-order of magnitude increase in the number of cognitive agents simultaneously interacting.

KEYWORDS

agent-based simulation; simulation framework; distributed simulation

ACM Reference Format:
Parantapa Bhattacharya, Saliya Ekanayake, Chris J. Kuhlman, Christian Lebiere, Don Morrison, Samarth Swarup, Mandy L. Wilson, and Mark G. Orr. 2019. The Matrix: An Agent-Based Modeling Framework for Data Intensive Simulations. In *Proc. of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019), Montreal, Canada, May 13–17, 2019,* IFAAMAS, 9 pages.

The research is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), via the Air Force Research Laboratory (AFRL). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the AFRL or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

Proc. of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019), N. Agmon, M. E. Taylor, E. Elkind, M. Veloso (eds.), May 13–17, 2019, Montreal, Canada. © 2019 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

1 INTRODUCTION

1.1 Background, Motivation, and Novelty

Background and motivation. Human decision-making is important in virtually all aspects of life, including forming habits; school, career, job, and hobby choices; and marriage decisions [6, 21, 40]. Moreover, social, psychological, neurological, emotional, normative, and learning factors, as well as age and education level, influence these decision-making processes. Social/cognitive computational models incorporating these factors are increasingly used to study how humans make decisions [7, 27, 36, 40, 53]. A result is that agent models, within agent-based modeling (ABM), have become more computationally demanding.

With the continuing pervasive use of social media, it is important to understand decision-making in these environments. Examples include instigation and perpetuation of social unrest [49][1], economic hardships within a country [19], and terrorism [25]. Yet, relatively little work exists that uses social and cognitive computational models to understand decision-making in large online environments *at scale*. Our approach is to develop an agent-based modeling and simulation (ABMS) system that incorporates social and cognitive agents to study online social platforms at scale.

Context and goals. One of our primary goals is to build, run, and evaluate cognitive and social agents that model online human decision-making processes, and to run simulations at scale where millions of these agents interact with each other and their environment. A second primary goal is to determine the trade-offs between fidelity of models, quality of results, and scalability.

We adopt an agent-based modeling approach for several reasons. A disaggregated representation of a population enables finer-grained inputs (e.g., heterogeneous inputs and agents) and outputs (e.g., evaluation of individual entities or sub-groups based on particular traits or interactions). ABMs allow finer-grained parametric and sensitivity studies to explore the effects of inputs on results. ABMs are also generative, and can thus provide more insights into the modeled system [16]. With detailed interaction structures among agents, establishing causality for particular behaviors may be easier [14]. Finally, the detailed representations afforded by ABM makes natural the process of exploring counterfactuals.

[1]This reference is for an entire special issue of the journal.

Table 1: Different types of agent-based models (ABMs) run within the Matrix simulation system.

Name	Model Type	Prog. Lang.
Freq-Stat	Frequentist statistical model	Python
Soc-Th	Social structure theory model	Python
CM-ANN	Artificial neural network model	C++
CM-Bayes	Bayesian cognitive theory model	R
CM-ACTR	ACT-R cognitive theory model	Common Lisp

Our approach separates model-building from simulation. Specifically, scientists and modelers design and build decision-making software agents, which are then managed by a simulation infrastructure responsible for all other aspects of the simulation (e.g., flow control, data management, distributed communications), but not agent behavior. In this way, many types of agent models can be run in a single simulation environment. *The focus of our paper is this agent-model agnostic simulation framework, which we call the **Matrix**.* [2] Our exemplar — the first of several studies — is human decision-making on GitHub. Modeling GitHub presents several unique challenges, but the Matrix distributed simulation platform is *far more general* and can easily be used to model other social media platforms such as Twitter, Facebook, and Reddit.

Novelty of work. There are several novel aspects of this work. *First*, we have produced a distributed framework for executing simulations involving heavyweight cognitive and social agents. *Second*, we demonstrate simulations with the Matrix that have *three orders of magnitude* more agents than the previous largest cognitive agent simulations. *Third*, the GitHub data set is large, encompassing 32 months of data, spanning 10 types of events, and containing hundreds of millions of users, repositories, and user-repository interactions. This is 2–5 orders of magnitude greater in numbers of users and repositories than any other works studying GitHub [29, 48, 51, 54]. *Fourth*, the agents we have run within the Matrix span agent models backed by artificial neural networks, cognitive reasoning frameworks (ACT-R), Bayesian models, social theory models based on communities, and simple statistical models that seek to replicate selected distributions found in the experimental data. This demonstrates the Matrix's flexibility in supporting many types of agents while itself being agent-model agnostic. See Table 1 for a list of agent models we have supported.

1.2 Requirements For Our Simulation System

The problem description above leads to the following requirements for the simulation platform.

(1) Ability to quickly store, update, and query large amounts (hundreds of gigabytes) of overall system state data consisting of both active agent states and passive object states that the agents require to make decisions.
(2) Ability to run large scale distributed simulations without access to Remote Direct Memory Access (RDMA) backed networks such as commodity (multicore) clusters, and popular cloud computing platforms [39] such as Amazon EC2, Google Compute Cloud, and Microsoft Azure [5, 52].
(3) Ability for agents to be written in different programming languages, including, but not limited to, Python, R, Lisp, C, and C++.
(4) Ability to use Compute Unified Device Architecture (CUDA) based GPU units, popular deep neural network libraries (such as TensorFlow[3], PyTorch[4], and LENS [42]), and cognitive system libraries (like ACT-R [47]) for writing agent models.
(5) Ability to run heterogeneous simulations incorporating different classes of agents in a single simulation.
(6) Ability to rapidly incorporate new data sources and prototype new agent models.

In the following sections, we describe how we meet these requirements. Related Work addresses how these requirements led us away from existing solutions.

1.3 Contributions

1. Dynamical systems-based formal simulation model. We use graph dynamical systems [31], a discrete dynamical systems formulation, to represent GitHub system dynamics. It explicitly represents active agents in the system (GitHub users), passive objects (e.g., GitHub repositories, pull requests, and issues), and the Matrix infrastructure. This formalism makes clear the role of the Matrix in performing control flow, data transmission, and data management. This allows us to reason about the system from an abstract perspective, and to separate dynamic interactions within GitHub from software design and implementation.

2. Design and development of the Matrix-distributed simulation framework. The Matrix conforms to a bulk synchronous parallel model [50]. The principal components are the *controller* that manages control flow within a Matrix instance, *state store* that provides data to agents in formats amenable to their needs, reducing their burden of querying the system state as they compute behaviors, and *data distribution system* to pass messages between the distributed processes. These elements, design choices, and system extensibility are discussed, and APIs are provided.

3. Descriptions of agents run in the Matrix. Agents run as standalone processes and communicate with the Matrix through well-defined APIs. Several different classes of models have been built and integrated into the Matrix, as described in Table 1. We provide ABM APIs and describe agent construction.

4. Performance evaluation of the implemented system. To evaluate the Matrix platform, we provide scaling studies using the ACT-R based cognitive theory inspired model. Our evaluations show that the Matrix platform can scale to millions of agents, a three-order of magnitude improvement in the state of the art for this class of cognitive models.

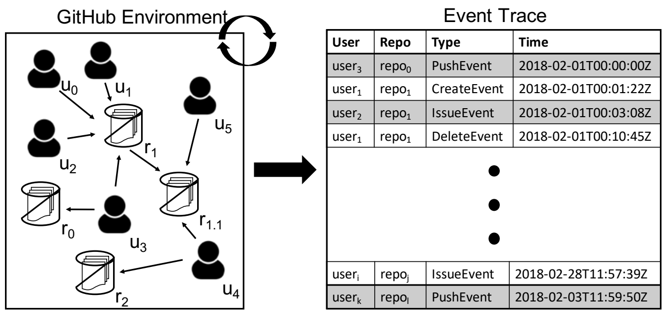
1.4 Exemplar Problem Description of GitHub

The GitHub system is represented in Figure 1. The system we study has 9 million users, 44 million repositories (repos), and 797 million user-repo interactions (events) over a 32-month window that were

[2]The source code of the Matrix framework is available at following URL https://github.com/NSSAC/socioneticus-matrix

[3]https://www.tensorflow.org/

[4]https://pytorch.org/



User	Repo	Type	Time
$user_3$	$repo_0$	PushEvent	2018-02-01T00:00:00Z
$user_1$	$repo_1$	CreateEvent	2018-02-01T00:01:22Z
$user_2$	$repo_1$	IssueEvent	2018-02-01T00:03:08Z
$user_1$	$repo_1$	DeleteEvent	2018-02-01T00:10:45Z
$user_i$	$repo_j$	IssueEvent	2018-02-28T11:57:39Z
$user_k$	$repo_l$	PushEvent	2018-02-03T11:59:50Z

Figure 1: Schematic of the GitHub system. Edges between users and repos represent event tuples (interactions) and other data (e.g., aggregated data) being transmitted, while the event trace contains all event tuples.

collected by Leidos[5]. Users interact directly with repos by initiating events on them. For example, a user may fork a repo (ForkEvent) to make personal modifications to an existing project, and, after the modifications are complete, the same user may request that her code be incorporated into the main-branch code base (PullRequestEvent). Each event type changes the state of GitHub by changing the state of a repo, and may change the internal state of user agents.

The fundamental unit of information is the *event tuple*, e_t. It is a 4-tuple $e_t = (u, r, e, t)$ where u is the user initiating the event, r is the repo on which the event is performed, e is the event type, and t is the date and time of the event. This is illustrated in Figure 1.

Cognitive modelers and social theorists used these event tuples, along with user and repo profile information, to develop agent-based models (ABMs). These models were, in turn, submitted to the Matrix simulation engine in order to simulate decision-making over model-defined time intervals.

Finally, the capabilities of the Matrix go well beyond GitHub; however, this problem provides a representative example of a concrete use of the Matrix.

Paper organization. The remainder of the paper is organized as follows: Section 2 contains related work. The formal mathematical model on which the Matrix is based is presented in Section 3. The Matrix is described in Section 4, and the state store and agents are presented in Section 5. Computational experiments are provided in Section 6. In the last sections, we go over the limitations of our system and conclude.

2 RELATED WORK

Agent-based simulation (ABS) platforms. The many ABS frameworks include Repast and Repast HPC [13], FLAME and FLAME GPU [12] [24], MIRAGE [37], Swarm [30], Mason [28], AnyLogic [20], and NetLogo (e.g., [13, 24]). Some of these emphasize usability (e.g., through graphics capabilities); others emphasize performance and/or simulation features. Systems use discrete time, event-driven, or both paradigms for controlling agent execution. These and other systems are described in several surveys (e.g, [26]). Books on ABM cover details of simulation systems design and implementation [17] and applications [18]. Most simulators are produced by writing custom software for agent behavior to meet a particular need; compiling the code; and linking the agent code to the appropriate framework (e.g., from the list of frameworks mentioned above).

The Matrix differs from this approach in that one or more agents, at the developer's discretion, execute as a separate process (and there can be any number of such processes), thus providing looser coupling between agents and simulation infrastructure. This means, for example, that agents can be written in any programming language (PL), and no PL bindings are required. Agents conform to a small application programming interface (API) to exchange messages with the Matrix infrastructure. Also, the Matrix specifically accommodates agents that require large amounts of data (as in the case of GitHub in our exemplar study) through a unique, flexible, and customizable state store component that is described in Section 5. To alleviate communication congestion, state stores are replicated across compute nodes (e.g., one instance per hardware node of a cluster). The Matrix, like Repast HPC, can implement interactions based on network representations of interacting agents (i.e., graph nodes, such as Twitter followers/followees) and spatial representation and movement of agents.

Social and social media modeling. There are several surveys about ABM in social research (e.g., [10]). In particular, [9] cites works over a broad range of social topics that include reciprocity, reputation, punishment, trust, and conventions, among many others. ABM has been used, for example, in the study of segregation [43], and is widely used to study Twitter [44] and Facebook [33] networks. Equally important, there are many models of social media behavior that can be converted to ABMs; e.g., [19, 32, 46]. In contrast, we found no GitHub-based social modeling works, making unique our particular application of at-scale interacting social and cognitive agents.

Cognitive modeling. Fourteen human decision-making architectures are explored across five dimensions (cognitive, affective, social, normative, learning) in [7]. They start with rule-based systems, and proceed through belief-desire-intention (BDI), normative, cognitive, psychological, and neurological architectures. They also overview dozens of modeling frameworks that space precludes us from addressing here. Tradeoffs in model fidelity versus computational efficiency are discussed in [1]; we take this approach when using a streamlined cognitive agent modeling environment, ACT-UP [41], rather than the heavier-weight ACT-R model. We also construct agents with LENS [42], an artificial neural network (ANN) software library, that has been used in a number of cognitive studies (e.g., [38]). ABM relating to social psychology appears in [45]. Beyond software, there are vast amounts of literature on conceptual models of cognition that have greatly influenced computational modeling (e.g., [35, 36]). The process of inferring properties of cognitive models from data is addressed in [22]. We found no works on cognitive modeling of users on GitHub.

Numbers of agents in at-scale cognitive simulations. In simulations of a housing market, [15] uses 203 agents. A hybrid simulation is reported in [11], where there are between 500 to 2500 light-weight agents and eight cognitive agents. Simulations of academic publishing [34] involve 3000-4000 cognitive agents, but at any given time in a simulation, at most ten agents are active. Cognitive modeling of pedestrian movements used up to 447 agents [23].

[5]https://www.leidos.com/

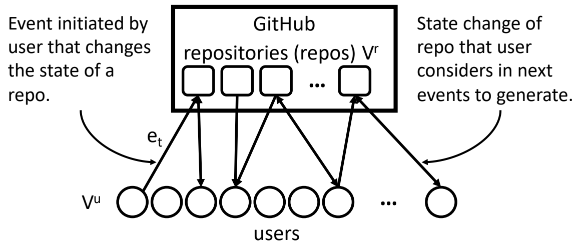

Event initiated by user that changes the state of a repo.
GitHub repositories (repos) V^r
State change of repo that user considers in next events to generate.
e_t
V^u
users

Figure 2: Conceptual view of GitHub consistent with the Graph Dynamical System (GDS) formal model. The user V^u and repo V^r node sets are shown, as is an illustrative event tuple e_t.

One thousand cognitive agents were simulated in [41]. By comparison, in this work, we simultaneously simulate up to 3 million agents. This includes the entire corpus of data mentioned above, fullfilling our goals of data-intensive computations at scale.

3 MODELS

This section is divided into two parts. First, a mathematical model called graph dynamical systems (GDSs) is introduced. The GDS formalism is general, and can simulate a Turing machine for certain complexity classes [8]. We also comment on specializations of this dynamical systems description for the GitHub system (space prevents a full treatment). Mathematical models such as GDS enable reasoning about dynamical systems (and simulations) while avoiding software implementation details. For example, unlike Facebook or Twitter, where models use a homogeneous population of users, it is more natural in the GitHub system to use two sets of nodes—one for users and one for repos—and sequence their dynamics accordingly. Block sequential loading within GDS captures precisely this needed behavior; it is needed both in the model and in the software implementation. Second, selected elements of the GDS are reformulated for the Software System Implementation Model (SSIM). Note that this transformation from a purely mathematical model to an SSIM is often done to achieve computational efficiencies.

3.1 Graph Dynamical System Model

A graph dynamical system [31] S is a discrete dynamical system defined by $S(G, K, F, W)$. Here, $G(V, E)$ is a *network* with vertex set V and edge set E, K is the set of *vertex states*, F is a sequence of *vertex* or *local functions* (one per vertex), and W is an *update scheme* that describes the execution order or sequencing of the vertex functions. GitHub can be represented as a graph, as in Figure 2, where $V = V^u \cup V^r$ and edges E denote channels for events.

Let $v_i \in V$ be a vertex in G, $x_i \in K$ be its state, and $n[v_i]$ be the sequence of vertices in the 1-neighborhood of v_i. The *restricted state* of v_i is the sequence of states of its 1-neighborhood vertices, $x[v_i] = (x_{n[v_i](1)}, x_{n[v_i](1)}, \ldots, x_{n[v_i](d_i+1)})$, $n[v_i](j)$ denotes the j'th vertex of the 1-neighborhood and d_i is the degree of v_i. The *system state* is the sequence of all vertex states, denoted $x = (x_1, x_2, \ldots, x_n) \in K^n$, where $n = |V|$.

The dynamics of the system are governed by $F = (f_i)_{i=1}^n$, the sequence of vertex functions, where f_i is the vertex function for v_i, and W, the update scheme, respectively. Function $f_i : K^{d_i+1} \longrightarrow K$ maps the current state of v_i at time t into the next state at time $(t + 1)$, i.e.,

$$x_i(t + 1) = f_i(x[v_i](t)) \ . \tag{1}$$

A vertex function for a user $v_i \in V^u$ will produce e_t's, where e_t is part of the vertex state x_i.

In this work, the *block-sequential* update scheme is used, meaning that vertices are grouped into a sequence of q blocks $(B_1, B_2, \ldots, B_q)$ and Equation (1) executes as

$$(F_{B_k}(x(t + 1)))_i = \begin{cases} f_i(x[v_i](t)), & \text{if } v_i \in B_k \\ x_i(t), & \text{otherwise .} \end{cases} \tag{2}$$

For GitHub, $B_1 = V^u$, the set of users, and $B_2 = V^r$, the set of repos. Over all blocks, Equation (2) captures the system dynamics

$$x(t + 1) = \mathbf{F}(x(t)) \ , \tag{3}$$

where $\mathbf{F}: K^n \longrightarrow K^n$ is defined as $\mathbf{F} = F_{B_q} \circ F_{B_{q-1}} \circ \ldots \circ F_{B_1}$, with $\circ$ denoting function composition. That is, at each time t, all vertex functions in the first block execute in parallel, then all vertex functions in the second block execute in parallel (using the updated output from all preceding blocks), and so on, until all blocks execute. Further details are found in [2, 31].

3.2 Software System Implementation Model

The Matrix is the infrastructure of the SSIM. The combination of the Matrix and ABMs produces an agent-based modeling and simulation (ABMS) system. This is discussed in Sections 4 and 5.

Here, we provide an implementation model that preserves the network representation of GitHub and the block-sequential update scheme of the GDS, but regroups the vertex functions and vertex states. It also provides a higher level representation of the vertex functions that assists in the software design. For example, note that in Section 3.1, the same event tuples $e_t \in E_t$ are part of both user vertex states K^u and repo vertex states K^r. While perfectly valid and useful in reasoning with a mathematical model, it is of course often inefficient to store information redundantly.

Also, only state updates are passed within a distributed system to avoid transmitting redundant information because system state updates are typically much smaller than the entire system state at each time step t of a simulation. Thus, in software, a distinction is made between a system state $x \in S$ and an update of a system state $x^u \in U$. Note that the set of all system states S equals K^n and the set of all system state updates U equals K^n.

The Matrix provides a framework for writing discrete time simulations. A stochastic discrete time simulation can be modeled as a stochastic function $g_{sim} : S \longrightarrow S$, such that given a system state at time t, $x(t) \in S$, $x(t)$ is transformed to $x(t + 1) = g_{sim}(x(t))$, the next system state. Then a run of the simulation can be thought of as successive application of the simulation function g_{sim} starting with the initial system state $x(0)$, analogous to Equation (3).

To utilize distributed computing hardware to speed up the simulation, first, the monolithic function g_{sim} is decomposed into component functions that can be computed in parallel. In our model, we first decompose g_{sim} into two functions $g_{act} : S \longrightarrow U$ and $g_{red} : S \times U \longrightarrow S$. Conceptually, g_{act} is responsible for computing the state updates $x^u(t + 1)$ that need to be applied to the current

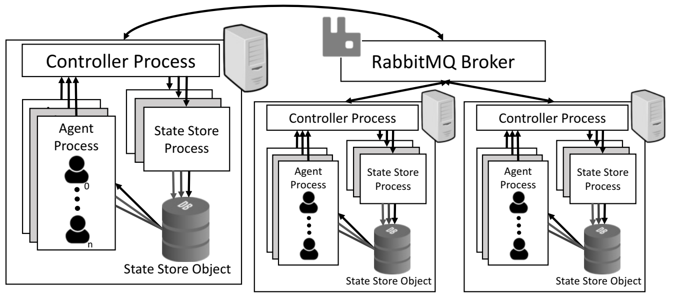


Figure 3: Matrix simulation framework architecture, denoting the controller, message broker, state store, and agents.

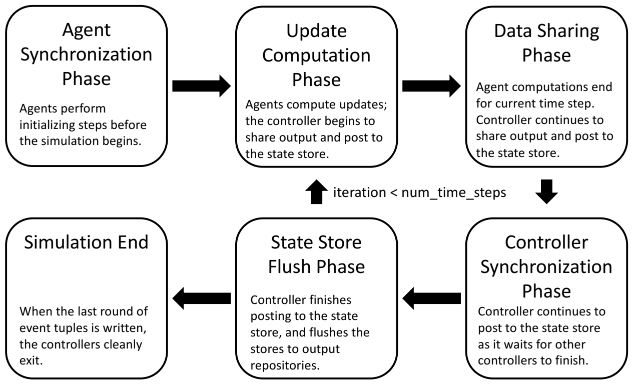

Agent Synchronization Phase
Agents perform initializing steps before the simulation begins.
Update Computation Phase
Agents compute updates; the controller begins to share output and post to the state store.
Data Sharing Phase
Agent computations end for current time step. Controller continues to share output and post to the state store.
iteration < num_time_steps
Simulation End
When the last round of event tuples is written, the controllers cleanly exit.
State Store Flush Phase
Controller finishes posting to the state store, and flushes the stores to output repositories.
Controller Synchronization Phase
Controller continues to post to the state store as it waits for other controllers to finish.

Figure 4: Steps in the simulation process of the Matrix.

state $x(t)$, and g_{red} is responsible for applying those updates to the current state to produce $x(t+1)$. Using g_{act} and g_{red}, g_{sim} can be written as

$$x(t+1) = g_{sim}(x(t)) = g_{red}(x(t), g_{act}(x(t))) \ . \tag{4}$$

The similarity between Equations (4) and (3) is obvious.

In an agent based simulation, g_{act} is responsible for computing all the necessary state updates that need to be made on behalf of all the agents present in the system. If we, however, assume that within a given time t, the agents can compute their actions independently, then the function g_{act} can be further decomposed as

$$g_{act}(x(t)) = \bigcup_{v_j \in V} g_{act}^{v_j}(x(t)) \tag{5}$$

Here, V is the set of agents in the system, and $g_{act}^{v_j}$ is assumed to compute updates only on behalf of agent v_j. Note that the independence here must respect the sequencing of the user state updates with the repository state updates, as embodied in Equation (2).

In the decomposition of the simulation function g_{sim} described above, the function g_{act} produces an *unordered* set of updates for the system state s_i. This requires some consideration in developing g_{red}, because if we were to naively apply the unordered updates to the system state, it can introduce unwanted system-introduced non-determinism.

This problem, however, can be easily remedied if there exists a total ordering among all the elements of update set U, which is often the case in practice. In such cases g_{red} can be decomposed as

$$g_{red}(x(t), x^{u^*}) = \hat{g}_{red}(s_i, (x^{u_1}, x^{u_2}, \ldots, x^{u_k})) \tag{6}$$

Here, $(x^{u_1}, x^{u_2}, \ldots, x^{u_k})$ is the ordered version of the unordered set of messages x^{u^*} which is the output of g_{act}.

4 MATRIX DESIGN AND IMPLEMENTATION

In this section, we describe the overall design and implementation of the Matrix ABM system.

The Matrix ABM System builds on the decomposition of the simulation computation as described in the previous section. To implement agent-based simulations using the Matrix, one needs to implement two programs: the agent program ($g_{act}^{a_j}$) and the state store program ($\hat{g}_{red}$).

Figure 3 shows the overview of the Matrix platform. Every compute node/machine in the simulation cluster runs the Matrix *controller* process. Also present on every compute node is the state store object which contains the complete current state of the simulation. Agent programs written to work within the Matrix ecosystem are designed as individual programs; an agent process is an instantiation of the agent program and computes updates on behalf of the agents it is responsible for. The computed updates are then passed on to the locally available controller by the agent process, which is responsible for sharing the updates with the controller processes running on the other compute nodes. The computation model of the Matrix platform conforms to a BSP model [50], with control and communications being the responsibility of a controller, one instance residing on each hardware node.

Figure 4 shows the states of operation that the Matrix controller process goes through. When the controller process receives updates it passes them on to the local state store process. The state store process keeps track of the updates it receives, maintaining them in a consistently sorted order. At the end of each time step, the controllers on all the compute nodes signal their respective state store processes to apply the updates to their local state store objects so the agent processes in the next time step can use the newly simulated data.

For our GitHub simulations, the state store objects are deployed as SQLite3 database files stored either on SSD-backed filesystems or in memory using the memory-backed filesystem ("tmpfs" on Linux-based operating systems). Furthermore, the controller processes communicate with each other using an intermediate RabbitMQ message broker system. The controller processes are implemented in Python 3 and utilize "asyncio", which is Python 3's native implementation of green threads, for handling large numbers of network connections — TCP connections from the agent processes and state store processes, and AMQP connections with the RabbitMQ message broker. The technologies required to interface with the Matrix ABM system are JSON and TCP/IP; these were chosen because of their relatively low overhead, and because libraries for using JSON and TCP/IP are readily available across a large range of programming languages.

The agents in the Matrix ABM system communicate with the controller process via remote procedure call (RPC) methods exposed by the controller that rely on JSON RPC over TCP/IP using the loopback interface. The two main RPC methods are CanWeStartYet and RegisterUpdates. All the RPC methods exposed by the Matrix controller are synchronous and blocking; this is done to simplify the logic of writing agents and state stores.

An agent process signals its readiness to process agents and produce updates on their behalf by calling the RPC method CanWeStartYet from its local controller. This method call blocks until all of the agent processes on all of the compute nodes are ready, and all of the state store processes have updated their state store objects with updates from the previous time step. Once the call to the CanWeStartYet method returns, the agent process then iterates over the agents it is responsible for, and computes updates. When the agent process is finished computing updates for a given agent, it hands the updates to the controller using the RegisterUpdates RPC call.

In the current Matrix ABM system, each agent process is expected to be running on its own CPU core. Thus, it is a separate task to distribute the agents operating inside the simulation among the available agent processes. Note that, due to the design of the simulation framework, it does not matter in which compute node the computation of the given agent process takes place, as all the information required to generate updates on behalf of the given agent is present in the system store object, which is replicated on every compute node. The Matrix ABM system does not enforce any particular method for distributing agents among the agent processes; they may either be distributed statically at the beginning of the simulation or dynamically via an agent distribution mechanism.

To receive updates from the controller, the state store process calls the exposed RPC method GetUpdates over TCP/IP using the loopback device. This is a blocking call which either returns a set of one or more updates, or an instruction for applying cached updates to the state store object. If GetUpdates returns a set of updates, the state store program is expected to update a cache of updates with the newly obtained ones; this can happen multiple times over the course of a time step. At the end of each time step, GetUpdates will return the "apply update" signal, triggering the state store process to push updates from its cache to the state store object in a consistent order.

One of the requirements for our ABM system was that it be able to run on commodity clusters, as opposed to HPC clusters backed by RDMA-based high throughput and low latency networks such as InfiniBand and Omni-Path; this is why we chose the more robust message brokering system RabbitMQ over MPI. The abstraction used by the Matrix system, however, allows us to easily swap out the backend technologies for an alternative inter-node communication technology without modifying any agent or state store code. Our future plans include creation of a version of the Matrix ABM platform that uses MPI instead of RabbitMQ for inter-node communication on HPC platforms.

Evolution of time within Matrix simulations: To evolve time in simulations, we use a combined discrete time and discrete event approach which we refer to as a *hybrid* approach. In particular, we use discrete time steps (specified *a priori*) and synchronize agent states at time step boundaries per the BSP model. However, within a time step, agents can generate any number of event tuples (and internal state updates) at any time. The assumption of this modeling approach is that state updates that take place within a time step are independent among agents. Given the latest time t_{sync} at which synchronization of time occurred, each agent can use all (event) information at times $t \leq t_{curr}$. Thus, this time-stepping approach is a conservative scheduling approach, i.e., it will never need to back-track in time as is the case with optimistic scheduling methods.

Algorithm 1 Agent process logic

```
procedure AgentProcess($I, C, S$)
    while $i \leftarrow$ CanWeStartYet($I, C$) do
        while $a_j \leftarrow$ GetNextAgent($I, C$) do
            $u_{j,i} \leftarrow$ ComputeAgentUpdates($a_j, S$)
            RegisterUpdates($u_{j,i}, C$)
```

In our work, this is advantageous because each model has different requirements for the time steps. For example, a cognitive model may prefer small discrete times (e.g., one-hour increments) to minimize the time over which the independence assumption holds. However, a statistical model may produce better results over a larger time step where stochasticity has a longer window in which to operate. These trade-offs are still being evaluated, but the Matrix is adaptable to these needs.

5 AGENTS AND STATE STORES

In this section, we describe the process of writing agent and state store programs that can work together with the Matrix.

5.1 Agent processes in the Matrix

Algorithm 1 shows the overall workings of a generic agent process logic. In general, an agent process is instantiated per available cpu core on every available node on the cluster. An agent process, on instantiation, is passed three arguments: its own agent process id I, a connection to the local Matrix controller process C, and a handle to the local system state S. The agent process then runs a two-level nested loop. The outer loop runs once per round (i.e., time step) of the simulation, and begins with the RPC call CanWeStartYet. This method returns the current round i during the simulation, and Null when the simulation is over. When the agent receives a Null response from CanWeStartYet, the outer loop is terminated, ending the agent process.

Within each iteration of the inner loop, the agent process fetches an agent a_j using the RPC call GetNextAgent and produces updates $u_{j,i}$ on its behalf, using ComputeAgentUpdates. ComputeAgentUpdates implements $g_{act}^{a_j}$ (Eq. 5), encapsulates the simulation logic, and varies across different simulations. Once the updates $u_{j,i}$ have been computed, the agent process hands it over to the Matrix controller using the RegisterUpdates RPC call. The RPC call GetNextAgent returns Null when all agents have been distributed to agent processes for computing updates. On receiving Null response from GetNextAgent the inner loop terminates, thus ending the current round for the current agent process.

Algorithm 2 State store process logic

```
procedure StateStoreProcess($C, S$)
    $U \leftarrow \emptyset$
    loop
        $u \leftarrow$ GetUpdates($C$)
        if $u$ = SIMEND then
            break
        else if $u$ = FLUSH then
            ApplyUpdates($S, U$);    $U \leftarrow \emptyset$
        else
            InsertStored($U, u$)
```

5.2 State store processes in the Matrix

Algorithm 2 gives a high level overview of the procedure followed by a generic state store process. Every compute node runs a single instantiation of the state store process. On instantiation, the state store process is passed two arguments: a connection to the local Matrix controller process C, and a handle to the local system state S. The state store process begins by initializing its cache for storing updates, U. The state store process then runs a single loop, starting with the RPC call GetUpdates. This method has three possible return values: SIMEND, FLUSH, or a list of updates. On receiving a list of updates, the state store process inserts them into its cache of update messages U, while maintaining the sorted order of the updates in U. The GetUpdates method returns FLUSH at the end every simulation round; upon receipt of the FLUSH message, the state store process is expected to use the ApplyUpdates method to update the state store object S with all of the updates in its cache. This is followed by reinitialization of an empty cache. ApplyUpdates is the method that implements $\hat{g}_{red}$ (Eq. 6) and encapsulates the logic for maintaining the state store object, and varies in tandem with the instantiation of ComputeAgentUpdates which implements $g^{a_j}_{act}$ (Eq. 5). The GetUpdates method returns SIMEND at the end of the simulation, at which point the main loop of the state store process terminates.

6 BENCHMARK APPLICATION AND EXPERIMENTS

We refer to Table 1 for a listing of all the models that have been implemented and run in the Matrix. Here, we focus on one of these models to illustrate the scalability of the Matrix ABMS platform.

6.1 CM-ACTR: An ACT-R model for Simulating GitHub System

The ACT-R based cognitive model (CM-ACTR) makes use of the ACT-R cognitive architecture [3], a high-fidelity cognitive architecture that has been used to develop hundreds of human behavior models from simple cognitive psychology experiments to complex task environments[6]. ACT-R models can reproduce every aspect of human behavior from sub-second response times, all types of errors and biases, motor actions and eye movements, and even patterns of neural activation.

In order to scale to orders of magnitude more agents and longer time scales than typical ACT-R models, our approach follows the accountable modeling principle [41], which states that models should focus the use of cognitive mechanisms on key aspects of the task and parameterize less important parts of the task. The CM-ACTR model leverages the long-term memory learning and retrieval mechanisms, specifically the statistical activation process [4] and its components such as base-level learning (to capture the power laws of decay and practice), partial matching (to generalize across situations) and blending (to generate aggregate decisions).

A preprocessing stage decomposes the user population into clusters of about 1,000 users. Past events (or a subset of the more recent ones) performed by those users are loaded into memory. To generate new events, the model generates from memory events composed of a user, repo, type and time interval, matched sequentially in increasing order of specificity. Those memory retrievals reflect past event distributions and generalize across similarities between users, reflecting their past action patterns. The computational load generated by the model is approximately linearly proportional to both the number of past events represented in memory and the number of future events generated.

6.2 Scaling study using CM-ACTR model

To analyze the scalability of the Matrix platform, we used the CM-ACTR model, and ran the same simulation task using a varying number of compute cores. For this experiment, we chose four different population sizes: 300 thousand users, 1.2 million users, 2.1 million users, and 3 million users. The CM-ACTR model was used to simulate events produced by these users for a period of two weeks. The compute nodes used for the simulations had 32 core dual processor configuration, with each processor being a 16 core Haswell-EP E5-2698 v3 2.30GHz processor. Each node had 128 GB of RAM. The state store objects were implemented as SQLite3 files and were kept in memory using on a Linux TMPFS backed partition. The CM-ACTR model was itself implemented in Common Lisp (CCL). On each compute node 30 of the 32 cpu cores were dedicated to agent model computations, while the other two cpu cores were reserved for use by the operating system and the Matrix platform code. The experiments were run on one to 32 compute nodes, which translates to 30 to 960 CPU cores being used for the agent simulation code.

Figure 5 shows the decrease in simulation runtime when increasing the number of compute cores for a given fixed-sized simulation task. Each combination of population and compute cores was run 30 times and the mean runtime is plotted in Figure 5.[7] The graph shows an approximately linear decrease in runtime on a log-log scale. This demonstrates the suitability of the Matrix platform for seamlessly distributing compute load across varying numbers of compute nodes.

One of the concerns in distributing computations across multiple compute nodes is a potential degradation in performance due to network use and synchronization issues. In Figure 6, we plot the number of updates produced per second (which is a good measure of cpu use efficiency) when using varying numbers of compute cores.

[6]see the ACT-R web site at http://act-r.psy.cmu.edu/ for models and publications

[7]Note, that the error bars (standard error) are not shown in Figure 5, as they are barely visible when plotted.

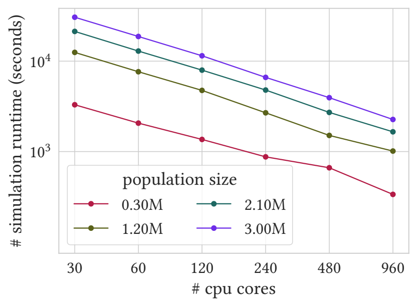


Figure 5: Reduction in simulation runtime of CM-ACTR simulation with increasing number of cpu cores, for different population sizes of GitHub agents.

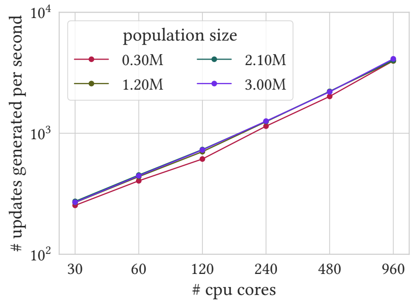


Figure 6: Number of updates generated per second by CM-ACTR simulation with increasing number of cpu cores.

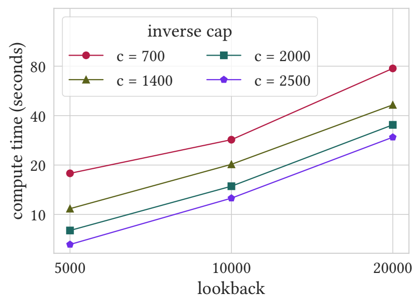


Figure 7: Time required by the CM-ACTR model to compute events on behalf of 265 active users with varying numbers of lookback events, and an inverse event generator cap, on a single CPU core.

We see that for all four population sizes, there is an approximately linear increase in number of updates produced per compute core on a log-log scale. This demonstrates that the cost of distribution of computation on up to 32 compute nodes is minimal, when using the Matrix.

6.3 Case study: Trade-off between accuracy and computation time in the CM-ACTR model

For compute-heavy agent-based simulations, such as those intended to run within the Matrix, it is necessary to take into consideration accuracy versus performance tradeoffs that can be configured in the model, and to systematically understand their impact, to obtain the best results. In the CM-ACTR GitHub simulation, we identified two primary parameters that could be used to trade off model accuracy and performance.

The *first* is the *lookback* parameter, which is the number of past events that a agent looks at, to initialize its state for future event generation. Using a smaller history length, or lookback, is a standard approach for reducing the model's compute complexity, since older events are less relevant as ACT-R also has time-based decay to capture that phenomenon.

The *second* is an upper limit or cap on the number of events generated per simulation time step. The effect of reducing this upper limit is mainly visible for highly active agents in the system that tend to produce orders of magnitude more events than the average agent in the system. Nevertheless, the impact of this parameter is significant. For actual model parameterization, we use *inverse cap* c, which specifies the upper limit as $\frac{\delta_t}{c}$ where δ_t is the size of the simulation timestep in seconds. Theoretically, for ACT-R's core memory mechanisms (partial matching, blending), the time complexity of the computation is linear in both the lookback and inverse cap $\frac{1}{c}$.

Figure 7 shows the effects of lookback and inverse cap on performance, measured in the number of seconds required to generate one month's worth of events for a group of 265 active users, using a single CPU core. As can be seen, the performance is approximately linear in lookback, and as the inverse cap c decreases, the model takes more time to execute, also approximately linearly.

7 LIMITATIONS

As discussed in the Related Work section, there are multiple alternative ABMS platforms available to modelers who want to do agent based modeling. We created the Matrix ABMS platform because the existing systems did not meet our requirements (as described in Section 1). However, there may be scenarios where existing platforms are better choices than the Matrix.

One of the major benefits of using the Matrix platform is that it supports code written in any programming language, allowing reuse of existing libraries in those languages. However, if programming language or existing library flexibility is not an issue, frameworks such as Repast HPC [13] that use fast-compiled languages such as C++ may be more beneficial.

The Matrix ABM system was built to utilize compute facilities on commodity clusters, and popular cloud computing platforms such as Amazon EC2, Google Compute Cloud, and Microsoft Azure. Unlike high performance computing facilities, these platforms, as of the time of writing, do not provide RDMA-based backends, such as those provided by Infiniband or Intel OmniPath. Thus, if model authors wanted to make use of these high speed network backends, an HPC-based ABM platform such as Repast HPC would be a better solution.

8 CONCLUSION AND FUTURE WORK

This paper describes the design, implementation, and execution of a novel agent-based simulation platform called the Matrix. Scalability was demonstrated, among other features. An outstanding issue is evaluation of the trade-off among model fidelity and accuracy of models.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable suggestions.

This work is partially supported by the Defense Advanced Research Projects Agency (DARPA), via the Air Force Research Laboratory (AFRL) (Grant No. FA8650-18-C-7826), and by the Defense Threat Reduction Agency (DTRA) (Grant No. HDTRA1-17-0118).

REFERENCES

[1] M. Abrams. 2013. A moderate role for cognitive models in agent-based modeling of cultural change. *Complex Adaptive Systems Modeling* (2013), 1–33.

[2] A. Adiga, C. J. Kuhlman, H. S. Mortveit, and S. Wu. 2015. Effect of Graph Structure on the Limit Sets of Threshold Dynamical Systems. In *AUTOMATA*. 59–70.

[3] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. 2004. An integrated theory of the mind. *Psychological review* 111, 4 (2004), 1036.

[4] J. R. Anderson and L. J. Schooler. 1991. Reflections of the environment in memory. *Psychological science* 2, 6 (1991), 396–408.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58.

[6] R. L. Axtell. 2016. 120 Million Agents Self-Organize into 6 Million Firms: A Model of the U.S. Private Sector. In *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*. 806–816.

[7] T. Balke and N. Gilbert. 2015. How Do Agents Make Decisions? A Survey. *Journal of Artificial Societies and Social Simulation* (2015), 1–30.

[8] C. Barrett, H. B. Hunt III, M. V. Marathe, S.S. Ravi, D. J. Rosenkrantz, and R. E. Stearns. 2011. Modeling and analyzing social network dynamics using stochastic discrete graphical dynamical systems. *Theoretical Computer Science* 412 (2011), 3932–3946.

[9] F. Bianchi and F. Squazzoni. 2015. Agent-based models in sociology. *WIREs Comput Stat* 7 (2015), 284–306.

[10] E. Bruch and J. Atwell. 2013. Agent-Based Models in Empirical Social Research. *Sociological Methods & Research* 44 (2013), 186–221.

[11] A. Caballero, J. Botia, and A. Gomez-Skarmeta. 2011. Using cognitive agents in social simulations. *Engineering Applications of Artificial Intelligence* 24 (2011), 1098–1109.

[12] S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough. 2012. Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework. In *IEEE 14th International Conference on High Performance Computing and Communications*. 538–545.

[13] N. Collier and M. North. 2013. Parallel agent-based simulation with Repast for High Performance Computing. *SIMULATION* 89, 10 (2013), 1215–1235. https://doi.org/10.1177/0037549712462620

[14] R. Conte and M. Paolucci. 2014. On agent-based modeling and computational social science. *Frontiers in Psychology* 5 (2014), 1–9.

[15] V. Dignum, J. Tranier, and F. Dignum. 2010. Simulation of intermediation using rich cognitive agents. *Simulation Modelling Practice and Theory* 18 (2010), 1526–1536.

[16] J. M. Epstein. 2007. *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton U. Press.

[17] R. M. Fujimoto. 2000. *Parallel and Distributed Simulation Systems*. John Wiley & Sons.

[18] N. Gilbert. 2007. *Agent-Based Models*. Sage.

[19] S. Gonzalez-Bailon, J. Borge-Holthoefer, A. Rivero, and Y. Moreno. 2011. The dynamics of protest recruitment through an online network. *Scientific Reports* (2011), 1–7.

[20] J. Huang, L. Liu, and L. Shi. 2016. Auction Policy Analysis: An Agent-Based Simulation Optimization Model of Grain Market. In *Winter Simulation Conference (WSC)*. 3417–3428.

[21] J. W. Kable, M. K. Caulfield, M. Falcone, M. McConnell, L. Bernardo, T. Parthasarathi, N. Cooper, R. Ashare, J. Audrain-McGovern, R. Hornik, P. Diefenbach, F. J. Lee, and C. Lerman. 2017. No Effect of Commercial Cognitive Training on Neural Activity During Decision-Making. *Journal of Neuroscience* (2017).

[22] A. Kangasrääsiö, K. Athukorala, A. Howes, J. Corander, S. Kaski, and A. Oulasvirta. 2017. Inferring Cognitive Models from Data Using Approximate Bayesian Computation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI)*. 1295–1306.

[23] P. M. Kielar and A. Borrmann. 2018. Spice: a cognitive agent framework for computational crowd simulations in complex environments. *Auton Agent Multi-Agent Syst* 32 (2018), 387–416.

[24] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, and C. Greenough. 2010. FLAME: Simulating Large Populations of Agents on Parallel Hardware Architectures. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 1633–1636.

[25] J. Klausen. 2015. Tweeting the Jihad: Social Media Networks of Western Foreign Fighters in Syria and Iraq. *Studies in Conflict & Terrorism* 38 (2015), 1–22.

[26] K. Kravari and N. Bassiliades. 2015. A Survey of Agent Platforms. *Journal of Artificial Societies and Social Simulation* (2015), 1–18.

[27] W. Li, Q. Bai, M. Zhang, and T. D. Nguyen. 2018. Modelling Multiple Influences Diffusion in On-Line Social Networks. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '18)*. 1053–1061.

[28] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. MASON: A Multi-Agent Simulation Environment. *Simulation: Transactions of the society for Modeling and Simulation International* 82 (2005), 517–527.

[29] N. McDonald, K. Blincoe, E. Petakovic, and S. Goggins. 2014. Modeling Distributed Collaboration on GitHub. *Advances in Complex Systems* 17 (2014).

[30] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. 1996. *The Swarm simulation system: A toolkit for building multi-agent simulations*. Technical Report 96-06-042. Santa Fe Institute.

[31] H. Mortveit and C. Reidys. 2007. *An introduction to sequential dynamical systems*. Springer Science & Business Media.

[32] S. A. Myers and J. Leskovec. 2012. Clash of the Contagions: Cooperation and Competition in Information Diffusion.. In *ICDM*. 539–548.

[33] H. R. Nasrinpour, M. R. Friesen, and R. D. McLeod. 2016. An Agent-Based Model of Message Propagation in the Facebook Electronic Social Network. ArXiv. (2016).

[34] I. Naveh and R. Sun. 2006. A cognitively based simulation of academic science. *Comput Math Organiz Theor* 12 (2006), 313–337.

[35] B. R. Newell and A. Broder. 2008. Cognitive processes, models and metaphors in decision research. *Judgment and Decision Making* 3 (2008), 195–204.

[36] M. G. Orr, C. Lebiere, A. Stocco, P. Pirolli, B. Pires, and W. G. Kennedy. 2018. Multi-scale Resolution of Cognitive Architectures: A Paradigm for Simulating Minds and Society. In *Social, Cultural, and Behavioral Modeling*. 3–15.

[37] B. H. Park, M. R. Allen, D. White, E. Weber, J. T. Murphy, M. J. North, and P. Sydelko. 2017. MIRAGE: A Framework for Data-Driven Collaborative High-Resolution Simulation. In *Advances in Geocomputation*, D. A. Griffith, Y. Chun, and D. J. Dean (Eds.). Springer International Publishing, 395–403.

[38] D. C. Plaut and A. K. Vande Velde. 2017. Statistical learning of parts and wholes: A neural network approach. *Journal of Experimental Psychology: General* 146 (2017), 318–336.

[39] L. Qian, Z. Luo, Y. Du, and L. Guo. 2009. Cloud Computing: An Overview. In *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 626–631.

[40] S. Reimers, E. A. Maylor, N. Stewart, and N. Chater. 2009. Associations between a one-shot delay discounting measure and age, income, education and real-world impulsive behavior. *Personality and Individual Differences* 47 (2009), 973–978.

[41] D. Reitter and C. Lebiere. 2010. Accountable Modeling in ACT-UP, a Scalable, Rapid-Prototyping ACT-R Implementation. In *In Proceedings of the 10th International Conference on Cognitive Modeling (ICCM)*.

[42] D. L. T. Rohde. 1999. *LENS: The light, efficient network simulator*. Technical Report CMU-CS-99-164. Carnegie Mellon University.

[43] T. C. Schelling. 1971. Dynamic Models of Segregation. *Journal of Mathematical Sociology* 1 (1971), 143–186.

[44] E. Serrano, C. Á. Iglesias, and M. Garijo. 2015. A Novel Agent-Based Rumor Spreading Model in Twitter. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*. 811–814.

[45] E. R. Smith and F. R. Conrey. 2007. Agent-Based Modeling: A New Approachfor Theory Building in Social Psychology. *Personality and Social Psychology Review* 11 (2007), 87–104.

[46] E. Sun, I. Rosenn, C. Marlow, and T. Lento. 2009. Gesundheit! Modeling Contagion through Facebook News Feed. In *International AAAI Conference on Weblogs and Social Media*.

[47] N. A. Taatgen and C. Lebiere J. R. Anderson. 2006. Modeling Paradigms in ACT-R. In *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, R. Sun (Ed.). Cambridge University Press, 29–52.

[48] F. Thung, T. F. Bissyande, D. Lo, and L. Jiang. 2013. Network Structure of Social Coding in GitHub. In *17th European Conference on Software Maintenance and Reengineering*. 323–326.

[49] Z. Tufekci and D. Freelon. 2013. Introduction to the Special Issue on New Media and Social Unrest. *American Behavioral Scientist* 57 (2013), 843–847.

[50] L. G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33 (1990), 103–111.

[51] B. Vasilescu, D. Posnett, B. Ray, M. G. J. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. 2015. Gender and Tenure Diversity in GitHub Teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 3789–3798.

[52] C. Vecchiola, S. Pandey, and R. Buyya. 2009. High-Performance Cloud Computing: A View of Scientific Applications. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*. 4–16.

[53] X. Wang and M. P. Wellman. 2017. Spoofing the Limit Order Book: An Agent-Based Model. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '18)*. 651–659.

[54] Y. Yu, H. Wang, V. Filkov, and B. Devanbu, P.and Vasilescu. 2015. Wait for It: Determinants of Pull Request Evaluation Latency on GitHub. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. 367–371.