SSIE 520 Spring 2019, Professor Sarah Lam

Toward an agent based modeling framework for investigating Command & Control in a Denied or Degraded Environment

Grady Kurpasi

[gkurpasi@hotmail.com](mailto:gkurpasi@hotmail.com)  
(213) 304-8541

|  |  |  |
| --- | --- | --- |
| A R T I C L E I N F O  7 May 2019  Keywords:  Python MESA  Agent Based Modeling  Command and Control in Denied Degraded Environment  C2D2E |  | A B S T R A C T  Command and Control (C2) is a competitive undertaking essential to the conduct of military operations. C2 refers to authorities, processes and technical systems that allow a force to coordinate information and action in space and time. Many C2 technical systems are exposed today to advesary capabilities to detect, interdict and or target electronic signatures. This paper document efforts to build an agent based modeling framework in Python using MESA. All development files can be found at:  https://github.com/GradyKurpasi/C2D2E-ABM |

1. Introduction

Command and Control, often referred to as ‘C2’, can loosely be described as the coordination of information and action in space and time. In the modern military environment, C2 is a complex, competitive undertaking layered on top of the tactical maneuver and logistic support of forces. Present C2 paradigms combine traditional visual and radio signals with high speed data networks and the internet to connect vast networks of knowledge and effort into unified entities capable of focused cognition and coordinated action. Simultaneously though, C2 networks present opportunities for adversaries to attack or impede coalition efforts.

Opposed forces have always endeavored to impede or interfere with advesary command and control. With the proliferation of electric / electronic C2 technologies throughout the 20th Century however, state and non-state actors have enjoyed wider, increasingly asymmetric opportunities to attack C2. The co-adaptation / co-evolution of military forces throughout the previous and current century have favored increasing speed and robustness of military capabilities, driving expanded requirements for information bandwidth. Accordingly, C2 architectures have trended toward increasing dependency on radio, satellite and digital communications. These technologies have given rise to new domains in warfare: Electronic Warfare and Cyber Warfare.

EW and Cyber capabilities present new challenges to military planners as they enable additional means to attack C2 infrastructure often without the physical danger of doing so in the traditional domains of land, sea and air. This makes attacking C2 increasingly lucrative in today’s network enabled / information centric age of warfare. Increased complexity of and reliance on C2 systems joined with increased exposure of these systems have redefined this problem set for military planners.

* + 1. C2D2E

To be precise, *Command* is a legal authority vested in an individual, and *Control* is both an authority and a function of coordinating the activity of subordinates (U.S Department of Defense, 2016). Command and Control refers collectively to the authorities and activities necessary to process information and direct action to achieve an objective.

Command and Control in a Denied or Degraded Environment (C2D2E) refers to the conduct of operations under information adverse conditions. That is, when an organization must operate “without its full complement of information and communications capabilities [as a result of ] adversarial action” (Bernier, Chan, Alberts, & Paul, 2013).

Command and Control in an information degraded environment is an enduring feature of conflict. Marine Corps Doctrinal Publication 1 (MCDP 1) describes warfare as being characterized by *Friction*, *Fluidity*, *Disorder* and *Uncertainty* (U.S. Marine Corps, 1997). The *Fog of War* refers to the incalculable, immemorial factors that make certainty in conflict a near impossibility. EW and Cyber present modern methods of directly attacking information; yet, although the technology is new, the effects are not, but they do present novel challenges to contemporary military planners.

The generally ubiquitous reliance of modern C2 architecture on electric / electronic communications means that modern military forces typically radiate electromagnetic energy to facilitate communication. (This paper focuses on EW capabilities and effects. The supporting model does not specifically address Cyber but can be extended to consider it). This makes modern C2 readily subject to

1. Jamming / Spoofing
2. Location Finding / Targeting.

This leads to the paradoxical effect of exercising C2 leading to increased exposure / vulnerability.

Modern planners describe this as the *Battle of Signatures*, where “to be detected, is to be targeted and killed” (U.S. Marine Corps, 2016). Again, the effect is not new, but the speed and range at which an advesary can exploit it is. Organizations like the U.S. Marine Corps have directed planners at all levels to address this vulnerability and develops plans and procedures to mitigate it.

* + 1. The United States Marine Corps and the Marine Operating Concept

The United States Marine Corps is the smallest U.S. military branch during peace time with a force of about 186,000 active duty and 35,000 reservists. The U.S. Marine Corps is mandated by law in Title 10 of the U.S. Code and is assigned duties by Title 10 and the National Security Act of 1947 (U.S. Government, 2006):

* Provide forces for seizure or defense of advanced naval bases and for the conduct of such land operations as may be essential to the prosecution of a naval campaign
* Perform such other duties as the President may direct
* Develop with the Army and the Air Force those phases of amphibious operations that pertain to the tactics, techniques, and equipment used by landing forces

Marine Corps Doctrinal Publication 1 defines *Maneuver Warfare* as its principal warfighting and organizational philosophy. Maneuver Warfare seeks to analyze and disable an advesary systemically, “shattering his cohesion through a variety of rapid, focused and unexpected actions which create a turbulent and rapidly deteriorating situation” (U.S. Marine Corps, 1997).

To address the current and assessed future global operating environment, the Marine Corps’ overarching strategic vision is currently defined in the Marine Operating Concept (MOC). The MOC “describes in broad terms how the Marine Corps will operate… in 2025 and beyond” (U.S. Marine Corps, 2016). The Marine Air-Ground Task Force (MAGTF) is the basic unit of Marine expeditionary operations. The MOC sets goals for a 21st Century MAGTF that “conducts maneuver warfare in the physical and cognitive dimensions of conflict to generate and exploit psychological, technological, temporal and spatial advantages… that embraces information warfare as indispensable for achieving complementary effects across five domains – air, land, sea, space and cyberspace” (U.S. Marine Corps, 2016). It envisions a highly networked, highly agile force, continuously forward deployed and maneuvering in physical and informational domains, always poised to provide expeditionary crisis response in everything from disaster relief to theater war.

1. The Problem

There is tension between the network enabled force the MOC describes, able to focus information and supporting arms down to the smallest, forward-most echelons, and its assessment of the future operating environment. The 5 Critical Tasks the MOC defines for future development includes *Operating with Resilience in a Contested Network Environment* (U.S. Marine Corps, 2016). Sub tasks simultaneously describe EM signature reduction and increased networking and processing power.

The MOC acknowledges this contradiction and provides guidance in navigating toward solutions. The Marine Corps already has time proven strategies for thriving in information deprived environments. These include decentralization of operations and exploiting relative tempo to generate advantage. The Marine Corps trains units at every level to decide and act, relying on shared understanding rather than explicit direction. This generates enormous advantage in responsiveness, but necessarily at the expense of precision.

There is every reason to believe that adherence to these tenets of maneuver warfare will continue to empower the Marine Corps to thrive amidst uncertainty and chaos when necessary. However there is also reason to believe that the Corps must make increasingly informed decisions about when to pursue greater decentralization (perhaps in the aim of reducing EM signature) and when to exercise centralized direction and information sharing to focus and mass effects. More so, the Corps must have the agility to switch between these paradigms fluidly and deliberately.

Research conducted by a series of NATO task groups on C2 characterized a number of C2 approaches by mapping them in a 3D descriptive space (Alberts, Bernier, Chan, & Manso, 2013). The axes of the description space were:

* Allocation of Decision Rights
* Distribution of Information
* Patterns of Interaction

Near the origin resides what most would consider traditional military control: Centralized authority and decision making; Hierarchical lines of communication, and Protected information.

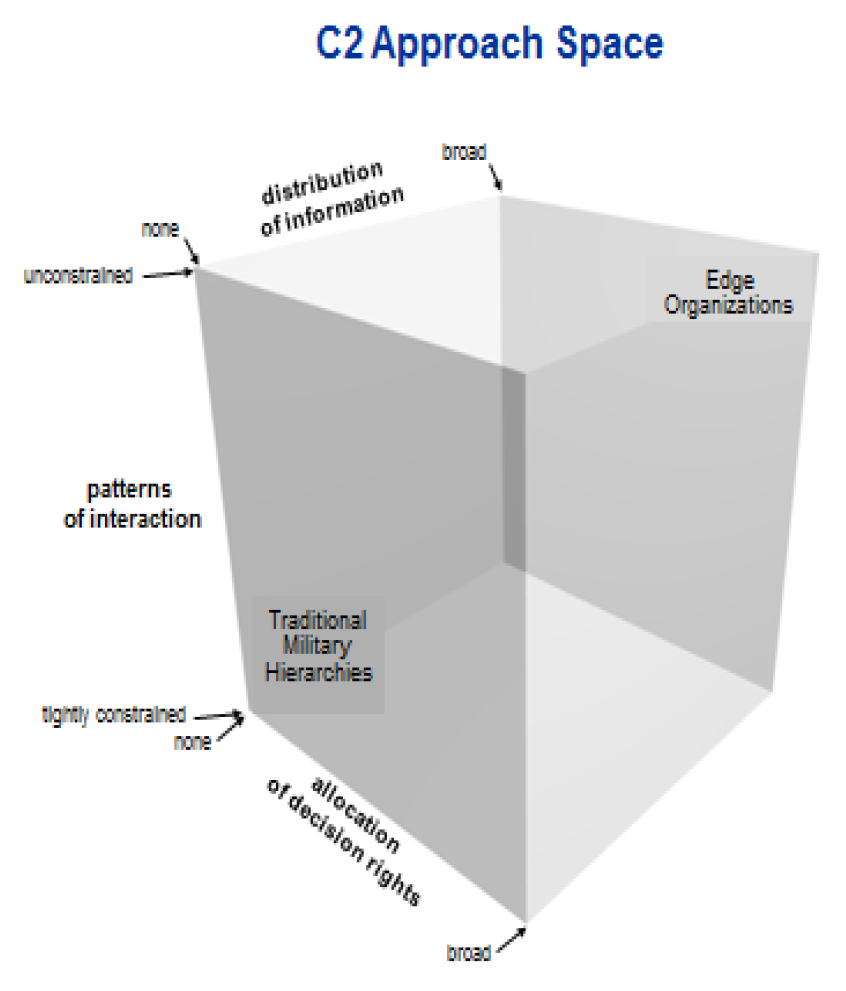


Figure 1 C2 Description Space

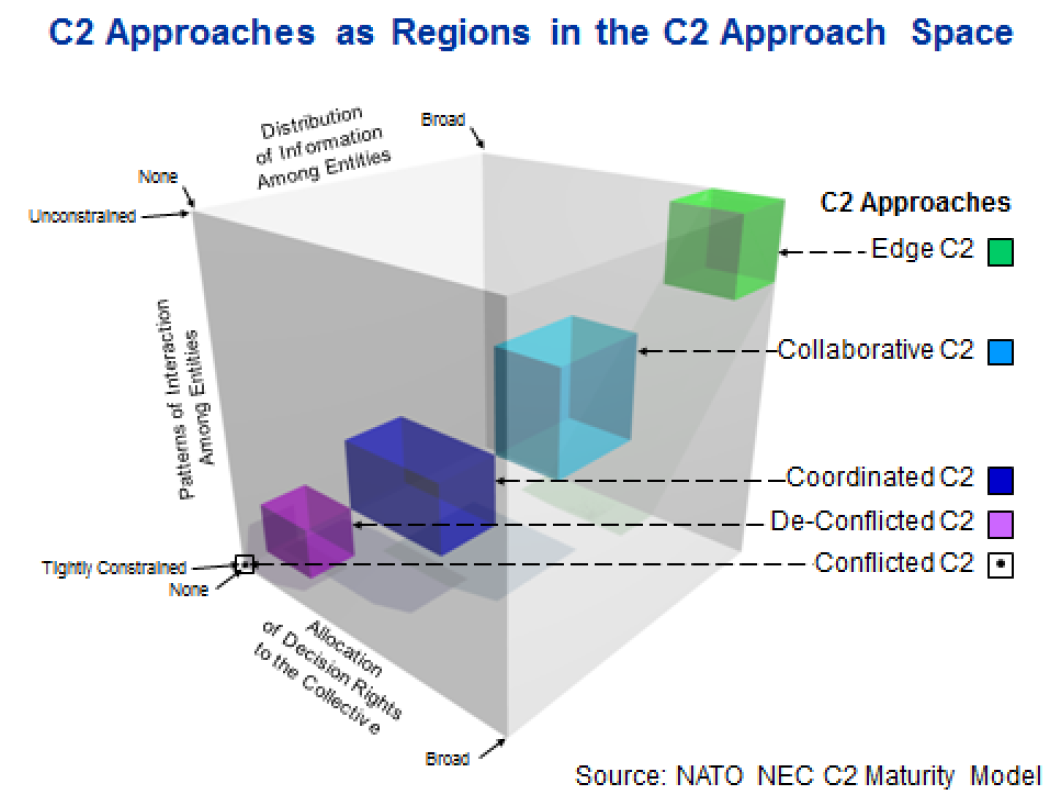


Figure 2 C2 Approaches within the C2 Description Space

Near the opposite, upper corner of the box, resides what they referred to as *Edge C2*. Edge C2 was characterized by a densely interconnected network with generally free information sharing. The researchers were investigating multi-national C2 structures and used a number of different simulations to do so. They found in many cases diminishing returns in operational effectiveness moving toward the extreme of Edge C2. However in a closely related study, found that the closer to Edge C2 an organization started in this space, the more resilient it tended to be when operating in a network constrained environment (Bernier, Chan, Alberts, & Paul, 2013).

This suggests, that the Marine Corps, which already exercises decentralization of decision authority and promotes both vertical and horizontal interaction and information sharing, should be resilient operating in a network / information constrained environment. In many of cases though the Marine Corps resorts to decentralization as a response to the break down of centralized command capabilities and the necessity of moving forward and maintaining tempo. Making this a deliberate, trained and rehearsed action, and identifying the battlefield conditions that would warrant shifting C2 stance are likely necessary for future success.

In the future operating environment, the questions are:

1. Is Deliberately Moving to Decentralized / Network Constrained modes of C2 a Viable Solution for EM Signature Reduction and operational flexibility.
2. What Are Critical Criteria to Shift C2 In One Direction Or Another?
3. Research Objectives

Ultimately, the objectives of this paper, the accompanying model and follow-on research are to:

* Explore the C2D2E solution space
* Evaluate relationships between reduction in C2 capabilities and mission performance indicators
* Identify critical criteria in battlefield conditions that might warrant a deliberate move toward reduced EM signature and increased decentralization of control.

Realistically, attainable goals for this paper and accompanying model are:

* Design an extensible modelling framework to investigate C2D2E
* Create and validate/invalidate an initial system design.

The system model and its implementation are intended to be an exploratory initial investigation aimed at determining if ABM is an effective means of investigating C2D2E.

Informal meta-KPIs will include ease of development, development time, ability to validate the model, and extensibility of the resulting framework.

As a vehicle to move toward future investigation, this iteration of a C2D2E model focuses on adversary (Red Force) targeting of coalition (Blue Force) communications. An agent based modelling framework was established to facilitate this study and 5000 runs were made to investigate the impact of:

* Rate of Red Force Targeting of Blue Force Communications
* Rate of Reduction in Blue Force Communications

on Mission Success and other key performance indicators. Its results are not predicted or intended to be quantitatively meaningful, but rather to qualitatively illuminate relationships between battlefield factors in a C2D2 environment.

* + 1. Assumptions / Constraints

3.1.1. EW & Cyber capabilities are highly sought after and highly guarded by modern forces. Specifics of both coalition and advesary capabilities are classified TS/SCI and so not available for inclusion to this study. As a result, instead of modeling specific capabilities, a range of capabilities (success rates) were modeled to fully explore the threat space and make assumptions about the rate to which Blue Force should desire to reduce Red Force effectiveness with mitigation measures.

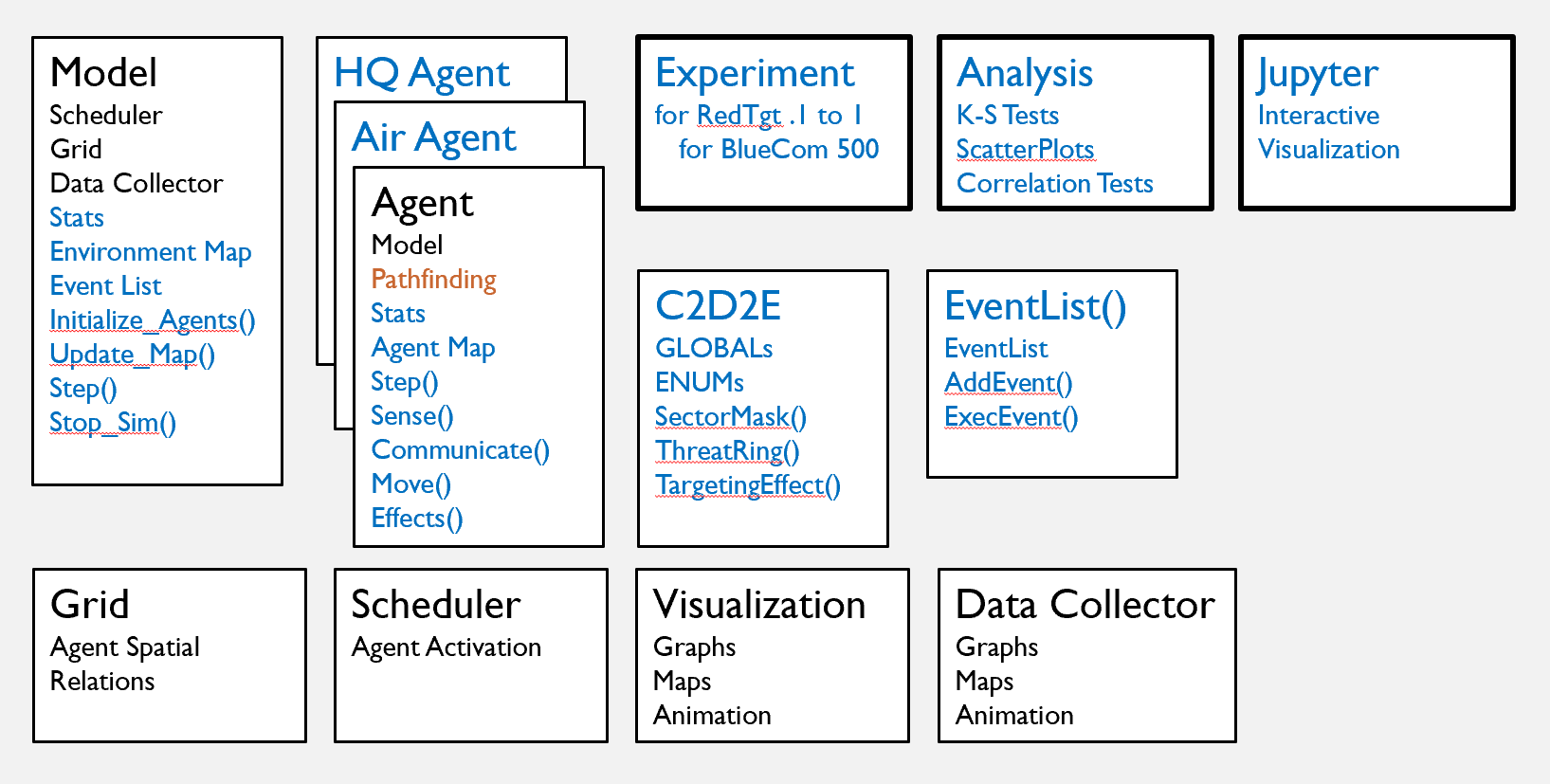
3.1.2. Access to Marine Corps Tactical Simulations such as Combat 21 on an unclassified network was not possible. Researchers only had access to mixed mode models (meaning they required human input and were generally intended for staff training). Initial aspirations of validating this model against existing simulations were not feasible and a design decision was made target a higher level of abstraction.

Figure 3 C2D2E ABM Framework (Extensions to MESA in Blue)

3.1.2. Estimates of Red Force/Blue Force attrition rates and combat service support capabilities are high. Although the rates are not entirely outlandish, they are high and were made so to accentuate Key Performance Indicators. Destruction / Attrition rates of 30% are by no means unheard of in peer conflict but are not something a unit would recover quickly from.

3.1.4. Current level of abstraction is appropriate. As mentioned in 3.1.2. a design decision was made not to try to closely replicate tactical realism. The undertaking was estimated to be beyond the scope of this study and likely would detract the investigation.

1. Methodology

Python / MESA was chosen as a development platform to leverage the speed of Python development and the number of mature API resources available in the language. Python libraries relied on by this project included Scipy & Numpy which provided ndarray data types as well as random number generators and statistical analysis tools <https://docs.scipy.org/>. Pandas provided the dataframe object which was used to analyze the large amount of data generated by the model as well provide bridge to Microsoft Excel https://pandas.pydata.org/. Matplotlib provided additional visualization tools for analysis of data results https://matplotlib.org/.

MESA is an ABM library built in Python intended to replicate the functionality of MASON and NetLogo. MESA is not yet a fully mature library but was the most robust ABM tool available in Python. Available documentation and support extended the learning curve, but the tool was effective in meeting project requirements. MESA can be found at

<https://github.com/projectmesa/mesa/>.

Some code snippets found on StackOverflow were also incorporated and are discussed in the model implementation below.

* + 1. MESA

The MESA framework provides everything necessary to build and run an agent based model. The top level MESA class is the Model object. The MESA Model class represents a model overall. It encapsulates and coordinates the other classes in the object model. Models are run by instantiating a Model object and calling its step() function.

Agent objects are spawned by the Model object. Each model can have a theoretically unlimited number of agents. The Model object advances time in the model by increasing a step counter and calling each Agent object’s step() function every ‘turn’. Each agent must be registered with the Grid and Scheduler objects.

The Grid object is also instantiated by the Model object. The Grid object manages spatial relations between agents. The default Grid is a two dimensional map that can be toroidal. MESA provides several versions of the Grid object including a grid whose cells are occupied by exactly one agent, one whose cells can be occupied by many agents, a hex grid, and a network grid.

The Scheduler object is likewise instantiated by the Model object. The Scheduler object that controls how agents are activated each turn. Activation can be sequential, random, all-at-once, or a hybrid that incrementally activates agents’ step() function over the course of a turn. The pattern of agent activation can have significant effects on the behavior of a model.

The DataCollector object collects data about both the model and individual agents. It is instantiated by the Model object and can call custom functions to build agent level or model level statistics. It can be activated each turn and or at specific points in a model run. The DataCollector object saves its data in a Pandas DataFrame object which in turn can write data to an Excel or comma-separated-file.

Finally, MESA provides a Visualization object that can produce HTML browser animations of model runs and or live graphs of model statistics.

* + 1. C2D2E Model Implementation

The following extensions were made to the MESA framework to create the C2D2E Model.

The Model object was given an environment map, implemented as 2D Numpy ndarray. This was necessary because the Grid object’s cells could not be extended with additional characteristics. The environment map provides model level information about the state of the model world. Additionally, per the framework, the Model object’s step() and \_\_init\_\_() functions were overridden with implementation specific behavior. C2D2E specific Model code is found in model.py.

The Agent object was given an effectiveness rating, short for combat-effectiveness. This attribute abstracts a number of factors that determine how an agent might fare in combat including casualties, ammunition levels, fuel levels, food & water supplies, etc. The Agent object was also given a map analogous to the Model object’s global environment map. Agent maps represent each agent’s knowledge of the world. The Agent’s step() function was overridden and sense(), move(), communicate(), and effects() functions were added. These are detailed in the C2D2E Model discussion below. Finally, the Agent model was subtyped to provide Ground, Air, and Headquarters agents. C2D2E specific Agent code is found in agent.py.

An EventList class was created which can be attached to either the Model object or Agent objects. The EventList provides is a 2D array of function names and parameter references that can be scheduled to execute at specific step numbers. This allows the model and agents to look forward in time and schedule events / interactions. It also allows input data modeling according to a distribution similar to a discrete event simulation. The EventList class definition is found in eventlist.py.

A number of domain specific helper functions as well global parameters are defined in C2D2E.py.

Jupyter Notebooks was used to run interactive model runs that employed MESA’s Visualization object and tools. Primarily this was used to assist in validating model behavior.

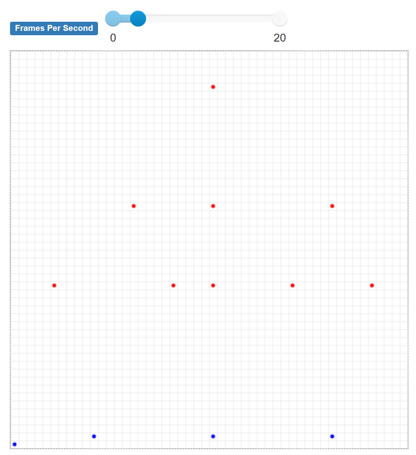
 The model experiment eventually consisted of 5000 runs that explored different combinations of Red Force Targeting Effectiveness and Blue Force Com Risk Acceptance. These model runs were not interactive and did not employ visualization. The code controlling them is found in experiment1.py.

Figure 4 Initial Model Disposition

Finally, the data analysis conducted on model results was provided by Scipy/Numpy libraries as well as Pandas. Matplotlib was used was used to display histograms and scatter plots. This code is found in analysis.py.

* + 1. C2D2E Model Design

This model intends to investigate the C2D2 Environment by examining relations between battlefield factors. The model presents a coalition force (BlueForce) and an advesary force (RedForce) that engage in an abstraction of ground combat operations. The model does not attempt to replicate tactical realism and the level of abstraction present in the model is intended to facilitate investigation of factors at the operational-level of war. Specifically, the model focuses on

* Red Force Targeting of Blue Force Communications
* Reduction in Blue Force Communications

Because optimal values for these variables are either unknown or highly classified, the model investigates their impact across a full range (0-100%) of effectiveness. The model looks for correlation between these factors and key performance indicators (KPIs) of mission performance, including:

* Accomplishment of the objective
* Speed
* Combat Losses
* Readiness for follow-on tasking.

The model world is a 50 x 50 grid. World size was originally 100 x 100 but had to be reduced due to processing time. BlueForce is roughly intended to be an infantry regiment. A regimental headquarters exists as an instance of the Headquarters Agent class and is stationary in the lower left corner of the map throughout the model run. Three BlueForce infantry battalions are Ground Agents that maneuver through the map during model runs. A RedForce Ground Agent is placed at the top of the map and defined as the objective for the BlueForce agents. Across the center of the map a RedForce defense-in-depth is represented as a band of reinforcing red Ground Agents (See Figure 4).

RedForce agents are currently stationary throughout model runs (although the framework supports RedForce maneuver). Each RedForce agent is able to inflict ranged damage and close combat damage on BlueForce agents within their threat rings (See Figure 5).

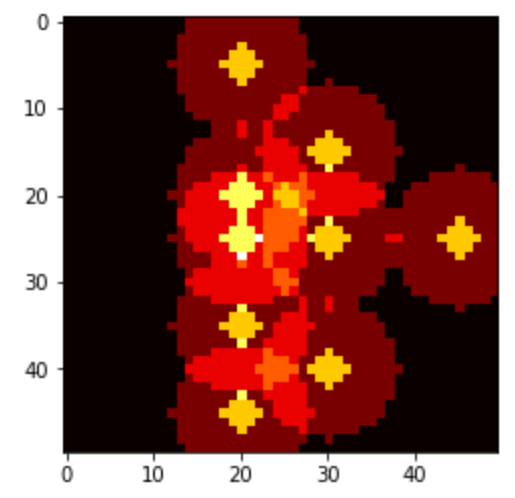
 The RedForce defense is intended to add complexity for the BlueForce and create conditions that may necessitate friendly communications.

Figure 5 RedForce Initial Threat Rings

Overall model behavior is really encapsulated in the Agent step() function. The Model object calls each agent’s step() function every ‘turn’. The Agent step() function in turn calls the Agent sense(), move(), communicate() and effects() functions.

Every turn, each agent first senses the world around it according to a SENSOR\_RANGE parameter, which for these model runs were equivalent between BlueForce and RedForce. Sensing the world around it allows an agent to determine the threat rating of nearby map squares.

Following, if an agent is allowed to maneuever (currently only BlueForce battalions) the agent conducts pathfinding based on its objective and its knowledge of the threat map and the takes one step along this path. The agent will re-sense the world next turn and re-calculate pathfinding every turn. An A\* pathfinding algorithm was used from a pathfinding library found at: <https://github.com/brean/python-pathfinding> (pathfinding requires future optimization if larger simulations are desired).

After moving, the agent makes a determination of whether to send or receive a number of messages. Establishment of communication is the heart of this model. Communications is representative of C2 in this model and presents both potential benefits to agents as well as risks. Messages an agent can send/receive include:

* Position Report
* Situation Report
* Combat Service Support Request
* Call For Fire
* Intelligence Update

BlueForce agents determine whether or not to send/receive each of these messages each turn. A BlueForce agent’s decision to establish communications is primarily governed by the BLUE\_COM\_RISK\_THRESHOLD model parameter which is explored by the model in a uniform range of 0 to 100%. Additionally, the agent’s state can further modify the risk threshold for specific messages.

An agent’s decision to establish communication is governed by a ‘die-roll’. That is, a randomly generated value from a standard uniform distribution is matched against the agent’s risk threshold. If it falls within it, the agent establishes communications.

Every time an agent decides to establish communications, RedForce has a chance to target the blue agent. RedForce’s ability to detect, locate, and target BlueForce communication is abstracted into a single RED\_TARGETING\_EFFECTIVENESS parameter. The model explores the range of RedForce targeting effectiveness discretely at .1, .2, .3,… 1 (although future iterations of this model will sample continuously from this range). RedForce com targeting is not dependent location of agents. RedForce always gets a chance to target Blue coms.

Targeting is also governed by a ‘die-roll’ and ‘damage’ done to a BlueForce agent’s effectiveness attribute is governed by the RED\_CFF\_THREAT parameter. Damage is additionally stochastic and every ‘damage roll’ is a percentage of potential threat determined by a randomly generated value from a beta distribution with parameters (1.5, 5).

A Blue agent’s Position Report message sends its map position to its Headquarters agent. The BLUE\_COM\_WINDOW parameter weights the agent’s decision to send a position report if the number of steps since an agent last communicated with higher headquarters is greater than the designated communications window. Position Reports are generally short transmissions and thus decrease the ability of RedForce to target this message.

The SituationReport message sends an agent’s worldmap to its higher headquarters replacing areas of the headquarters map with regions of the map that an agent has sensed. SitReps are often longer transmissions and thus increase the ability of RedForce to target this message.

The Combat Service Support request message restores a BlueAgent’s effectiveness rating according to a BLUE\_CSS magnitude parameter. A Blue agent’s decision to send this message is modified by the BLUE\_COM\_EFFECTIVENESS\_THRESHOLD parameter, below which an agent will send a combat service support request in order to avoid being destroyed.

The Call For Fire message allows a Blue agent to call supporting arms (artillery, air, etc.) on a nearby Red agent. ‘Damage’ done to that agent is governed by the BLUE\_CFF\_THREAT parameter.

The Intelligence Update message updates an agent’s map with the higher headquarters map. This transmission is generally longer and increases RedForces change to target this message.

After sensing the world, moving one space and determining which messages to send, agents calculate net effects for the round. As the agent moves and communicates, effects are queued in an Agent curr\_effects list. These are calculated cumulatively at the end of the turn and it is determined if an agent will die at the end of its turn.

Additionally, there is a single BlueForce Air Agent whose arrival is determined according to an exponential distribution with mean 10 (steps) that is scheduled via the Model object’s EventList. The agent does little more than arrive, conduct a sensor sweep, report its map to higher and depart. It was included primarily to prove functionality of Model (and thereby Agent) event scheduling.

* + 1. Experiment 1

The C2D2E model was run 5000 times in Spyder. Jupyter Notebooks was only used for visually ‘debugging’ (i.e. validating) model behavior via the visualizations and the Spyder environment saw a small speed advantage over VisualStudio Code.

The experiment looped through 10 values of RED\_TARGETING EFFECTIVENESS at .1, .2, .3,… 1. At each discrete value of RED\_TARGETING\_EFFECTIVENESS, 500 runs were made with BLUE\_COM\_RISK\_THRESHOLD being randomly sampled from a standard uniform distribution.

Each turn the DataCollector object gathered statistics directly from every agent as well as aggregate statistics from the model. Agent statistics were written each turn to an Excel file. Model statistics were written at the end of the experiment to an excel file.

At the end of the model run the Pandas, Scipy/Numpy and Matplotlib libraries were used to conduct statistical tests. Pandas was first used to conduct K-S tests on the 500 Sample blocks conducted at each interval of RED\_TARGETING\_EFFECTIVENESS. This was to ensure that BLUE\_COM\_RISK THRESHOLD was indeed sampled uniformly. Histograms were generated with Matplotlib to back up the K-S tests.

Following that Scipy was used to conduct Pearson Correlation tests on the relation of the independent variables (RED\_TARGETING\_EFFECTIVENESS, BLUE\_COM\_RISK\_THRESHOLD) and the individual KPIs. Matplot lib was used to generate scatter plots.

1. Results

RED Targeting Effectiveness = 0.1

Adjusted Test Statistic 1.02

KSTest Test for Standard Uniform

Distribution at Alpha.05

Failed to Reject H0



RED Targeting Effectiveness = 0.2

Adjusted Test Statistic 0.85

KSTest Test for Standard Uniform

Distribution at Alpha.05

Failed to Reject H0



RED Targeting Effectiveness = 0.3

Adjusted Test Statistic 1.50

KSTest Test for Standard Uniform

Distribution at Alpha.05 -

**Reject H0**



RED Targeting Effectiveness = 0.4

Adjusted Test Statistic 0.70

KSTest Test for Standard Uniform

Distribution at Alpha.05 -

Failed to Reject H0



RED Targeting Effectiveness = 0.5

Adjusted Test Statistic 0.51

KSTest Test for Standard Uniform

Distribution at Alpha.05 -

Failed to Reject H0



RED Targeting Effectiveness = 0.6

Adjusted Test Statistic 1.30

KSTest Test for Standard Uniform

Distribution at Alpha.05 -

Failed to Reject H0



RED Targeting Effectiveness = 0.7

Adjusted Test Statistic 0.58

KSTest Test for Standard Uniform

Distribution at Alpha.05 -

Failed to Reject H0



RED Targeting Effectiveness = 0.8

Adjusted Test Statistic 0.66

KSTest Test for Standard Uniform

Distribution at Alpha.05 -

Failed to Reject H0



RED Targeting Effectiveness = 0.9

Adjusted Test Statistic 1.18

KSTest Test for Standard Uniform

Distribution at Alpha.05 -

Failed to Reject H0



RED Targeting Effectiveness = 1.0

Adjusted Test Statistic 1.15

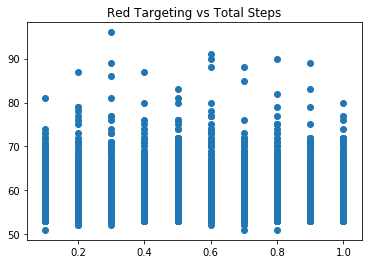
KSTest Test for Standard Uniform

Distribution at Alpha.05 -

Failed to Reject H0

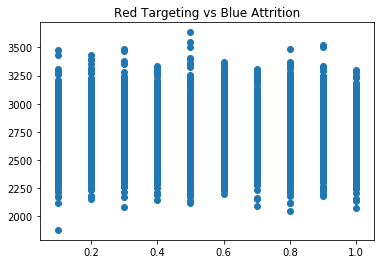


In [ ]:



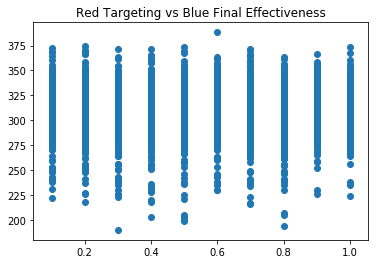
(Pearson) Correclation Coef =

0.0007986346142643681



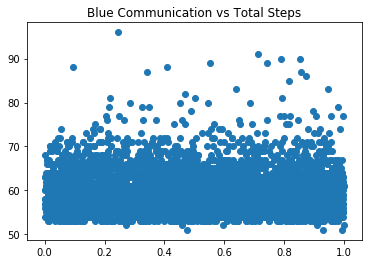
(Pearson) Correclation Coef =

-0.009471222811023076



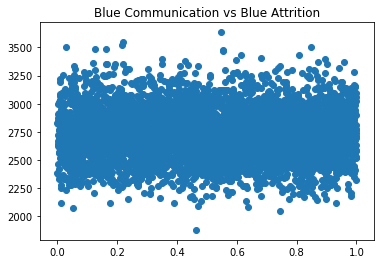
(Pearson) Correclation Coef =

0.01858087829536106



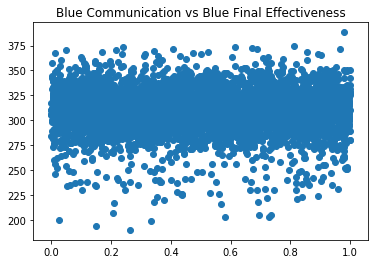
(Pearson) Correclation Coef =

-0.0016514861511249981



(Pearson) Correclation Coef =

-0.016642733012641676



(Pearson) Correclation Coef =

-0.016587908956367605

1. Conclusions

With regard to this iteration of this model, results were, as expected, inconclusive. Model results appeared to indicate that neither RedForce targeting of BlueForce Coms, nor BlueForce restriction of communication had any bearing on the mission success, speed, combat losses and readiness for follow-on missions.

Examining the data, there were no cases in which BlueForce did not successfully defeat the objective. This simply isn’t realistic. Additionally there were only 100 runs in which a BlueForce agent was destroyed, and no runs in which more than one BlueForce agent was destroyed. Finally, each BlueForce agent was able to restore an average of 32 times its starting effectiveness through Combat Service Support (CSS) requests. For perspective, per model design, that represents a battalion of roughly 1500 Marines replenishing itself, its ammunition, its vehicles, fuel and other supplies, in full over 30 times while enroute to a single objective. This isn’t even remotely possible.

It is estimated that this factor, the effectiveness of BlueForce CSS obscured the impact of communications targeting and communications reduction on mission success by making BlueForce agents nearly invincible (98% survivability rate).

Although the model was never intended to generate quantitative analysis, these figures necessarily invalidate this run of the model.

Luckily though, this behaviour is governed by global parameters that are easily changed in future experiments and these figures don’t necessarily invalidate the modelling framework.

Next steps for the framework are to move the model onto a network of higher classification and plug in more realistic parameters, perhaps generated by existing tactical sims.

With regard to meta KPIs for the framework, development was conducted fairly quickly and inexpensively. As well because it was developed in Python, there is effectively open-ended capability to incorporate data from other models and or to computationally validate/verify model performance against other sources.

1. References

Alberts, D. S., Bernier, F., Chan, K., & Manso, M. (2013). C2 Approaches: Looking for the “Sweet Spot”. *18th ICCRTS.*

Bernier, F., Chan, K., Alberts, D. S., & Paul, P. (2013). Coping with Degraded or Denied Environments in the C2 Approach Space. *18th ICCRTS.* U.S. Army Research Laboratory.

U.S Department of Defense. (2016). Department of Defense Dictionary of Military and Associated Terms. *JP 1-02*. DOD.

U.S. Government. (2006). Armed Forces, 10 U.S.C. Subtitle C, Part I, Chap 507 § 5063. U.S. Congress.

U.S. Marine Corps. (1997). *Marine Corps Doctrinal Publication 1 - Warfighting.* Heaquarters Marine Corps.

U.S. Marine Corps. (2016). *Marine Operating Concept.* Headquarters Marine Corps.

1. Python Source Code

model.py

#model.py

from C2D2E import \*

from agent import \*

from eventlist import \*

class C2D2EModel(Model):

"""A model with some number of agents."""

def \_\_init\_\_(self, N=1):

#init vars / maps

self.num\_agents = N

self.grid = MultiGrid(MAP\_WIDTH, MAP\_HEIGHT, False) #MESA grid object

self.schedule = RandomActivation(self) #MESA scheduling object

self.environment\_map = np.zeros((MAP\_WIDTH, MAP\_HEIGHT)) #secondary internal map for storing map weights

self.step\_count = 0

self.model\_events = EventList()

self.waiting\_agents = []

#init stats

self.stat\_red\_total\_attrition = 0

self.stat\_red\_units\_destroyed = 0

self.stat\_blue\_total\_attrition = 0

self.stat\_blue\_total\_css = 0

self.stat\_blue\_units\_destroyed = 0

self.stat\_blue\_final\_effectiveness = 0

self.stat\_mission\_accomplished = False

self.stat\_total\_steps = 0

#add starting agents

self.\_\_initialize\_agents()

#add data collector

self.datacollector = DataCollector(

model\_reporters={"Red Total Attrition" : "stat\_red\_total\_attrition",

"Red Units Destroyed" : "stat\_red\_units\_destroyed",

"Blue Total Attrition" :"stat\_blue\_total\_attrition",

"Blue Total CSS" : "stat\_blue\_total\_css",

"Blue Units Destroyed" : "stat\_blue\_units\_destroyed",

"Blue Final Effectiveness" : "stat\_blue\_final\_effectiveness",

"Mission Accomplished" : "stat\_mission\_accomplished",

"Total Steps" : "stat\_total\_steps"},

agent\_reporters={"POS" : "pos",

"Effectiveness": "effectiveness",

"CFF Damage" : "stat\_cff",

"Attrition Self" : "stat\_attrition\_self",

"Attrition OpFor" : "stat\_attrition\_opfor",

"CSS" : "stat\_css",

"COM Made" : "stat\_com\_made",

"COM Targeted" : "stat\_com\_targeted",

"Update Sent" : "stat\_map\_update\_sent",

"Update Received" : "stat\_map\_update\_received",

"Path Length" : "stat\_path\_length",

"Current Effects" : "curr\_effects",

"CFF Targets" : "stat\_cff\_targets",

"Range Targets" : "stat\_range\_targets",

"Close Targets" : "stat\_close\_targets"})

#for MESA visualization

self.running = True

#schedule future events

#Future AIR agent and events scheduled in self.Initialize\_Agents

def place\_red\_unit(self, unique\_id, coord):

# agent init function

#def \_\_init\_\_(self, unique\_id, model, agent\_type, color=AgentColor.BLUE, should\_move=False, objective=None, higher=None):

should\_move = False

a = UnitAgent(unique\_id, self, AgentType.GROUND, AgentColor.RED, should\_move)

self.schedule.add(a)

self.grid.place\_agent(a, coord)

return a

def place\_blue\_unit(self, unique\_id, coord, objective, higher = None):

# agent init function

#def \_\_init\_\_(self, unique\_id, model, agent\_type, color=AgentColor.BLUE, should\_move=False, objective=None, higher=None):

should\_move = True

a = UnitAgent(unique\_id, self, AgentType.GROUND, AgentColor.BLUE, should\_move, objective, higher)

self.schedule.add(a)

self.grid.place\_agent(a, coord)

if a.higher: a.higher.add\_subordinate(a) #if higher != to None establish command relationship

return a

def \_\_initialize\_agents(self):

# agent init function

#def \_\_init\_\_(self, unique\_id, model, agent\_type, color="blue", should\_move=False, objective=None, higher=None):

#debug stuff

#self.place\_blue\_unit("v12", (0,0), (19,19))

#self.place\_red\_unit("red", (10, 19))

HQ = HeadquartersAgent("2d Marines", self, AgentColor.BLUE, False, (25,45))

self.schedule.add(HQ)

self.grid.place\_agent(HQ, (0,0))

self.place\_blue\_unit("V12", (10, 1), (25,45), HQ)

self.place\_blue\_unit("V22", (25, 1), (25,45), HQ)

self.place\_blue\_unit("V32", (40, 1), (25,45), HQ)

self.place\_red\_unit("red1", (5, 20))

self.place\_red\_unit("red2", (20, 20))

self.place\_red\_unit("red3", (25, 20))

self.place\_red\_unit("red4", (35, 20))

self.place\_red\_unit("red5", (45, 20))

self.place\_red\_unit("red6", (15, 30))

self.place\_red\_unit("red7", (25, 30))

self.place\_red\_unit("red8", (40, 30))

self.place\_red\_unit("Goal", (25, 45))

#schedule future agent / events

#currently only used to demonstrate capability

# agent init function

#def \_\_init\_\_(self, unique\_id, model, agent\_type, color="blue", should\_move=False, objective=None, higher=None):

#interarrrival = max(int(np.random.exponential(10)), 1)

interarrrival = 0

agt = UnitAgent("Black Knight 1-1", self, AgentType.AIR, AgentColor.BLUE, False, None, HQ)

self.model\_events.add\_event(interarrrival, self.schedule.add, (agt, ))

self.model\_events.add\_event(interarrrival, self.grid.place\_agent, (agt, (10,10)))

self.model\_events.add\_event(interarrrival + 1, self.schedule.remove, (agt, ))

self.model\_events.add\_event(interarrrival + 1, self.grid.remove\_agent, (agt,))

def \_\_update\_map\_weights(self):

#generates current map weights

#map weights = (potential) damage taken by blue units when moving through map

#all map weights are zeroed and re-calculated each step according to current location of units

self.environment\_map = np.ones((MAP\_WIDTH, MAP\_HEIGHT)) #secondary internal map for storing map weights

#generate (potential) threat to blue units based on location of red unitws

for agent in self.schedule.agents:

#for all red agents, create threat rings centered on red units

#reflect threat rings as map weights

if agent.color == AgentColor.RED:

x, y = agent.pos

#close combat threat ring

self.environment\_map += generate\_threat\_ring((x, y), RED\_CLOSE\_RANGE, RED\_CLOSE\_THREAT)

#ranged threat ring

self.environment\_map += generate\_threat\_ring((x, y), RED\_RANGE, RED\_RANGE\_THREAT)

def stop\_sim(self, success=False):

self.running = False

self.stat\_mission\_accomplished = success

self.stat\_total\_steps = self.step\_count

for agent in self.schedule.agents:

if agent.color == AgentColor.BLUE : self.stat\_blue\_final\_effectiveness += agent.effectiveness

def step(self):

self.\_\_update\_map\_weights()

self.model\_events.exec\_events(self.step\_count)

self.schedule.step()

self.datacollector.collect(self)

self.step\_count += 1

if self.step\_count > MAX\_STEPS : self.stop\_sim(False)

def run\_model(self, n):

for i in range(n):

self.step()

agent.py

#agent.py

from C2D2E import \*

from eventlist import \*

def damage\_mod():

return np.random.beta(5, 1.5)

class UnitAgent(Agent):

""" A Military unit agent """

def \_\_init\_\_(self, unique\_id, model, agent\_type=AgentType.GROUND, color=AgentColor.BLUE, should\_move=False, objective=None, higher=None):

super().\_\_init\_\_(unique\_id, model)

self.agent\_type = agent\_type

self.color = color

self.should\_move = should\_move

self.objective = [objective] #LIFO list of objectives, next = objective[-1], final = objective[0]

self.higher = higher

self.pos = (-1,-1)

self.last\_com=0 #last turn had com w higher

self.effectiveness = 100.0 #hybrid of statistic of unit health/supply/combat effectiveness

self.agent\_map = np.ones((MAP\_WIDTH, MAP\_HEIGHT)) #what each agent knows about the world

#Blue-For agents do not explictly know positions of Red-For agents

#Red-For agents do not currently move and can be identified by map weights

#stats for data collector, zeroed at beginning of every step

self.stat\_cff = 0 #Blue-For call for fire effects

self.stat\_attrition\_self = 0 #Blue-For 'damage' taken

self.stat\_attrition\_opfor= 0 #Blue-For 'damage' dealt

self.stat\_css = 0 #Blue-For 'health' restored

self.stat\_com\_made = False #Blue-For established communications with higher

self.stat\_com\_targeted = False #Blue-For communications detected and targeted by Red-For

self.stat\_map\_update\_sent = False #Blue-For sent intel update

self.stat\_map\_update\_received = False #Blue-For received intel update

self.stat\_path\_length = 0 #Number of steps required to reach objective

self.curr\_effects = [] #FIFO list of effects each turn, gathered throghout turn and applied by self.effects()

self.stat\_cff\_targets = []

self.stat\_range\_targets = []

self.stat\_close\_targets = []

#NOTE: agent.pos not defined until model.grid.add\_agent is called. agent.pos cant be accessed in agent.\_\_init

def zero\_stats(self):

#stats for data collector, zeroed at beginning of every step

self.stat\_cff = 0 #Blue-For call for fire effects

self.stat\_attrition\_self = 0 #Blue-For 'damage' taken

self.stat\_attrition\_opfor= 0 #Blue-For 'damage' dealt

self.stat\_css = 0 #Blue-For 'health' restored

self.stat\_com\_made = False #Blue-For established communications with higher

self.stat\_com\_targeted = False #Blue-For communications detected and targeted by Red-For

self.stat\_map\_update\_sent = False #Blue-For sent intel update

self.stat\_map\_update\_received = False #Blue-For received intel update

self.stat\_path\_length = 0 #Number of steps required to reach objective

self.curr\_effects = [] #FIFO list of effects each turn, gathered throghout turn and applied by self.effects()

self.stat\_cff\_targets = []

self.stat\_range\_targets = []

self.stat\_close\_targets = []

def print\_stats(self):

#stats for data collector, zeroed at beginning of every step

print("")

print("")

print(self.model.step\_count, self.unique\_id, self.pos, self.agent\_map[self.pos])

print("CURR EFFECTS ", self.curr\_effects) #FIFO list of effects each turn, gathered throghout turn and applied by self.effects()

print("CFF TARGETS ", self.stat\_cff\_targets)

print("RANGE TARGETS ", self.stat\_range\_targets)

print("CLOSE TARGETS ", self.stat\_close\_targets)

print("CFF ", self.stat\_cff) #Blue-For call for fire effects

print("ATTR SELF ", self.stat\_attrition\_self) #Blue-For 'damage' taken

print("ATTR OTHER ", self.stat\_attrition\_opfor) #Blue-For 'damage' dealt

print("CSS ", self.stat\_css) #Blue-For 'health' restored

print("COM MADE ", self.stat\_com\_made) #Blue-For established communications with higher

print("COM TARGETED ", self.stat\_com\_targeted) #Blue-For communications detected and targeted by Red-For

print("UPDATE SENT ", self.stat\_map\_update\_sent) #Blue-For sent intel update

print("UPDATE RECEIVED ", self.stat\_map\_update\_received) #Blue-For received intel update

print("PATH LENGTH ", self.stat\_path\_length) #Number of steps required to reach objective

print("CURR EFFECTIVENESS", self.effectiveness)

def sense(self):

#replaces a section (radius) of agent map with an update from environement map

#occurs every step/turn with radius = SENSOR\_RANGE

mask = generate\_sector\_mask(self.pos, SENSOR\_RANGE)

self.agent\_map[mask] = self.model.environment\_map[mask]

return

def communicate(self):

#determines whether not to send one or more communication messages

#each message can be detected and targeted by Red-For

#if com detected, Blue-For unit takes effects = current map weight

rnd = random.uniform(0, 1) #chance to decide to send comms this step

if self.agent\_type == AgentType.HEADQUARTERS : return

if self.color == AgentColor.BLUE :

if self.agent\_type == AgentType.AIR :

#single air unit is used primarily to demonstrate functionality of agent/event scheduling

#only exists for one step/turn, makes sensor sweep, reports map to higher

mask = self.agent\_map > 1 # portion of map that agent has sensed

self.higher.agent\_map[mask] = self.agent\_map[mask]

return

#POSREP

if rnd < BLUE\_COM\_RISK\_THRESHOLD or self.model.step\_count - self.last\_com > BLUE\_COM\_WINDOW :

#send Position Report to higher if risk allows or BLUE\_COM\_WINDOW has passed

self.higher.subordinates\_lastpos[self.unique\_id] = self.pos

if RedTargetingEffects(MessageType.POSREP) : self.stat\_com\_targeted = True

self.last\_com = self.model.step\_count

self.stat\_com\_made = True

#SITREP

if rnd < BLUE\_COM\_RISK\_THRESHOLD:

#send Situation Report (i.e. push map updates to higher)

mask = self.agent\_map > 1 # portion of map that agent has sensed

self.higher.agent\_map[mask] = self.agent\_map[mask]

if RedTargetingEffects(MessageType.SITREP) : self.stat\_com\_targeted = True

self.last\_com = self.model.step\_count

self.stat\_map\_update\_sent = True

self.stat\_com\_made = True

#CSS

if rnd < BLUE\_COM\_RISK\_THRESHOLD or self.effectiveness < BLUE\_COM\_EFFECTIVENESS\_THRESHOLD:

#send Combat Service Support request to higher if risk allows or effectiveness below threshold

#CSS message is a catch all for MEDEVAC, Resupply, Combat Replacements, etc.

#will increase effectiveness

self.curr\_effects.append((EffectType.CSS, BLUE\_CSS))

if RedTargetingEffects(MessageType.CSS) : self.stat\_com\_targeted = True

self.last\_com = self.model.step\_count

self.stat\_com\_made = True

#CFF

if (rnd < BLUE\_COM\_RISK\_THRESHOLD and self.agent\_map[self.pos] > 1) or (self.agent\_map[self.pos] > RED\_RANGE\_THREAT) :

#request Call For Fire if near Red-For and risk allows OR if in close combat with Red-For unit

#proximity to Red-For unit currenlty implied from map\_weights

#will need to be updated if Red-For updated to maneuver

self.curr\_effects.append((EffectType.CFF, BLUE\_CFF\_THREAT))

if RedTargetingEffects(MessageType.CFF) : self.stat\_com\_targeted = True

self.last\_com = self.model.step\_count

self.stat\_com\_made = True

#INTEL UPDATE

if rnd < BLUE\_COM\_RISK\_THRESHOLD :

#request Intelligence Update (i.e. pull map updates to higher)

mask = self.higher.agent\_map > 1 # portion of map that has been sensed

self.agent\_map[mask] = self.higher.agent\_map[mask]

if RedTargetingEffects(MessageType.INTEL\_UPDATE) : self.stat\_com\_targeted = True

self.last\_com = self.model.step\_count

self.stat\_map\_update\_received = True

self.stat\_com\_made = True

if self.color == AgentColor.RED:

#future modification should allow for Blue-For targeting of RedFor coms

pass

if self.stat\_com\_targeted :

#add effect if Red-For successfully targets Blue-For communication

self.curr\_effects.append((EffectType.IDF, RED\_CFF\_THREAT))

def move(self):

#this method relies on external pathfinding API

#https://github.com/brean/python-pathfinding

#not optimized for models with large maps or many agents

#uses pathfinder to plot optimal course based on curent state of map

#then moves 1 position toward objective

#will recalculate move next turn

#In future - optimize to not calculate full path

if not self.objective or not self.should\_move: return #if objective list is empty or unit shouldn't move: return

#(should raise handled error in future)

if self.pos != self.objective[-1]: #if current location not equal to next objective (i.e. last element of list)

#create pathfinding grid from MESA grid

#MESA grid and pathfinding grid are oriented differently, requires transposition of array => map.T

grid = Grid(matrix=self.agent\_map.T)

x, y = self.pos

start = grid.node(x, y)

x, y = self.objective[-1]

end = grid.node(x, y)

finder = AStarFinder(diagonal\_movement=DiagonalMovement.always)

path, runs = finder.find\_path(start, end, grid)

self.stat\_path\_length = len(path)

if len(path) > 1: #if there is a solution path, then take first move

x, y = path[1]

self.model.grid.move\_agent(self, (x, y)) #move

if self.pos == self.objective[-1] : self.objective.pop() # if have arrived at objective, set next objective

def effects(self):

if self.color == AgentColor.BLUE:

if self.agent\_type == AgentType.AIR : return

cff\_effect = 0

#com effects

for effect in self.curr\_effects: #curr\_effect list is a type, (EffectType, scalar magnitude)

if effect[0] == EffectType.CFF: #Blue-For Call for Fire

cff\_effect += effect[1]

elif effect[0] == EffectType.CSS: #Blue-For request for Combat Service Support

self.effectiveness += effect[1]

self.effectiveness = 100 if self.effectiveness > 100 else self.effectiveness #max effectiveness = 100

self.stat\_css += effect[1]

else: #everything else is damage

damage = damage\_mod() \* effect[1] #'damage' is stochastic, % of THREAT/EFFECT, skewed toward .8

self.effectiveness -= damage

self.stat\_attrition\_self += damage

#NOTE: Red-For CFF effects included in effects list as EffectType.IDF

#ground combat effects -< CFF, Ranged, Close >- Effects all Stack

#for simplicity BLUE/RED effect ranges are currently equal

#CFF Effects

for RedAgent in self.model.grid.get\_neighbors(self.pos, True, True, BLUE\_RANGE + 1):

#+1 range accounts for Blue-For CFF request made in agent.Communicate() which occurs before agent.Move()

if RedAgent.color != AgentColor.RED : continue

self.stat\_cff\_targets.append((RedAgent.unique\_id, RedAgent.pos))

#Blue-For only, Red-For CFF effects applied above in Com Effects

damage = damage\_mod() \* cff\_effect #'damage' is stochastic, % of THREAT/EFFECT, skewed toward .8

RedAgent.effectiveness -= damage

self.stat\_cff += damage

self.stat\_attrition\_opfor += damage

#Ranged Effects

for RedAgent in self.model.grid.get\_neighbors(self.pos, True, True, BLUE\_RANGE):

if RedAgent.color != AgentColor.RED : continue

self.stat\_range\_targets.append((RedAgent.unique\_id, RedAgent.pos))

#Red-For effects

damage = damage\_mod() \* RED\_RANGE\_THREAT #'damage' is stochastic, % of THREAT/EFFECT, skewed toward .8

self.effectiveness -= damage

self.stat\_attrition\_self += damage

#Blue-For effects

damage = damage\_mod() \* BLUE\_RANGE\_THREAT #'damage' is stochastic, % of THREAT/EFFECT, skewed toward .8

RedAgent.effectiveness -= damage

self.stat\_attrition\_opfor += damage

#Close Effects

for RedAgent in self.model.grid.get\_neighbors(self.pos, True, True, BLUE\_CLOSE\_RANGE):

if RedAgent.color != AgentColor.RED : continue

self.stat\_close\_targets.append((RedAgent.unique\_id, RedAgent.pos))

#Red-For effects

damage = damage\_mod() \* RED\_CLOSE\_THREAT #'damage' is stochastic, % of THREAT/EFFECT, skewed toward .8

self.effectiveness -= damage

self.stat\_attrition\_self += damage

#Blue-For effects

damage = damage\_mod() \* BLUE\_CLOSE\_THREAT #'damage' is stochastic, % of THREAT/EFFECT, skewed toward .8

RedAgent.effectiveness -= damage

self.stat\_attrition\_opfor += damage

if self.effectiveness < 0: #agent dies at end of turn if effectiveness falls below 0

self.model.grid.remove\_agent(self)

self.model.schedule.remove(self)

if self.color == AgentColor.BLUE : self.model.stat\_blue\_units\_destroyed += 1

if self.color == AgentColor.RED : self.model.stat\_red\_units\_destroyed += 1

if self.unique\_id == "Goal" : self.model.stop\_sim(True)

#update remaining stats

self.model.stat\_red\_total\_attrition += self.stat\_attrition\_opfor

self.model.stat\_blue\_total\_attrition += self.stat\_attrition\_self

self.model.stat\_blue\_total\_css += self.stat\_css

def step(self):

self.zero\_stats()

self.sense()

self.communicate()

self.move()

self.effects()

#debug

#if self.color == AgentColor.BLUE : self.print\_stats()

class HeadquartersAgent(UnitAgent):

#headquarters units must be instantiated prior to subordiantes

#so command relationship can be created

def \_\_init\_\_(self, unique\_id, model, color=AgentColor.BLUE, should\_move=False, objective=None, higher=None):

super().\_\_init\_\_(unique\_id, model, AgentType.HEADQUARTERS, color, should\_move, objective, higher)

self.subordinates = [] #in future, this will be generic to UnitAgent

self.subordinates\_lastpos = {}

def add\_subordinate(self, agent):

#establishes 'command' relationships between agents and a HQ agent

#this must be called after 'agent' object has been added to grid

#otherwise reference to agent.pos will fail

#add agent to subordinates

self.subordinates.append(agent)

#ensure reciprocal command relationship

agent.higher = self

#add last known position

self.subordinates\_lastpos[agent.unique\_id] = agent.pos

def command(self):

#future: coordinate movement and effects of agents

pass

C2D2E.py

#C2D2E.py

import random

import numpy as np

import pandas as pd

from enum import Enum, auto

import matplotlib.pyplot as plt

from mesa import Model, Agent

from mesa.datacollection import DataCollector

from mesa.time import RandomActivation

from mesa.space import MultiGrid

from mesa.visualization.modules import CanvasGrid

from mesa.visualization.ModularVisualization import ModularServer

from mesa.visualization.modules import ChartModule

from pathfinding.core.diagonal\_movement import DiagonalMovement

from pathfinding.core.grid import Grid

from pathfinding.finder.a\_star import AStarFinder

#Pathfinding functions used from https://github.com/brean/python-pathfinding.

#Agent & Environment Maps must have positive values for agents to move.

#In this API, map (i.e. array) values <= 0 equate to obstacles

#GLOBAL 'CONSTANTS' - not wrapped in Enums so that they can be manipulated by Batch Runs

MAX\_STEPS = 500

MAP\_HEIGHT = 50

MAP\_WIDTH = 50

SENSOR\_RANGE = 15 #range at which Red & Blue units can detect each other

RED\_TARGETING\_EFFECTIVENESS = .5 #Red Side's effectiveness of detecting and targeting Blue Side coms

RED\_RANGE = 8 #Red-For ranged combat range (radius)

RED\_RANGE\_THREAT = 10 #Red-For ranged combat threat magnitude

RED\_CLOSE\_RANGE = 2 #Red-For close combat range (radius)

RED\_CLOSE\_THREAT = 30 #Red-For close combat threat magnitude

RED\_CFF\_THREAT = 30

MSG\_MODIFIER\_POSREP = -.2 #modifiers to RED\_TARGETING\_EFFECTIVENESS

MSG\_MODIFIER\_SITREP = .1 #based IRL on message duration, required RF Freq / Output, etc.

MSG\_MODIFIER\_CSS = 0

MSG\_MODIFIER\_CFF = 0

MSG\_MODIFIER\_INTEL\_UPDATE = .2

BLUE\_COM\_RISK\_THRESHOLD = .5 #Blue Side's risk acceptance for establishing coms

BLUE\_COM\_WINDOW = 10 #mandated frequency of Blue Side communication with higher (in Steps)

BLUE\_COM\_EFFECTIVENESS\_THRESHOLD = 70 #effectiveness threshold below which Blue Side will establish com regardless of risk

BLUE\_RANGE = 8 #Red-For ranged combat range (radius)

BLUE\_RANGE\_THREAT = 10 #Red-For ranged combat threat magnitude

BLUE\_CLOSE\_RANGE = 2 #Red-For close combat range (radius)

BLUE\_CLOSE\_THREAT = 30 #Red-For close combat threat magnitude

BLUE\_CFF\_THREAT = 30

BLUE\_CSS = 30 #amount Blue Side effectiveness raised when requesting support

#GLOBAL Enums

class AgentColor(Enum):

BLUE = auto()

RED = auto()

UNKNOWN = auto()

class AgentType(Enum):

GROUND = auto()

AIR = auto()

HEADQUARTERS = auto()

class EffectType(Enum):

CSS = auto()

CFF = auto()

GROUND = auto()

AIR = auto()

IDF = auto()

class MessageType(Enum):

POSREP = auto()

SITREP = auto()

CSS = auto()

CFF = auto()

INTEL\_UPDATE = auto()

class PhysicalObstacle(Enum):

#not currently in use

#future use in map definition

UNPASSABLE = 0

RIVER = - 1

MOUNTAIN = -2

def RedTargetingEffects(MsgType):

#calculates chance of Red-For targeting of Blue-For communications

#returns True if detected, else False

AdjustedEffectiveness = RED\_TARGETING\_EFFECTIVENESS

if MsgType == MessageType.POSREP : AdjustedEffectiveness += MSG\_MODIFIER\_POSREP

elif MsgType == MessageType.SITREP : AdjustedEffectiveness += MSG\_MODIFIER\_SITREP

elif MsgType == MessageType.CSS : AdjustedEffectiveness += MSG\_MODIFIER\_CSS

elif MsgType == MessageType.CFF : AdjustedEffectiveness += MSG\_MODIFIER\_CFF

elif MsgType == MessageType.INTEL\_UPDATE : AdjustedEffectiveness += MSG\_MODIFIER\_INTEL\_UPDATE

return True if random.uniform(0, 1) < AdjustedEffectiveness else False

def sector\_mask(shape,centre,radius,angle\_range):

#re-used from StackExchange

#https://stackoverflow.com/questions/18352973/mask-a-circular-sector-in-a-numpy-array?noredirect=1

#used to mask areas of a 2D array to reflect threat rings (circles)

"""

Return a boolean mask for a circular sector. The start/stop angles in

`angle\_range` should be given in clockwise order.

"""

x,y = np.ogrid[:shape[0],:shape[1]]

cx,cy = centre

tmin,tmax = np.deg2rad(angle\_range)

# ensure stop angle > start angle

if tmax < tmin:

tmax += 2\*np.pi

# convert cartesian --> polar coordinates

r2 = (x-cx)\*(x-cx) + (y-cy)\*(y-cy)

theta = np.arctan2(x-cx,y-cy) - tmin

# wrap angles between 0 and 2\*pi

theta %= (2\*np.pi)

# circular mask

circmask = r2 <= radius\*radius

# angular mask

anglemask = theta <= (tmax-tmin)

return circmask\*anglemask

def generate\_sector\_mask(coord, radius, angle\_range=(0,360)):

#returns a mask of the environment\_map array (i.e. [0][0]=True, [0][1]=False, etc.)

#by default returns a circle of radius RADIUS, centered on COORD

#can return a sector

x, y = coord

mask = sector\_mask((MAP\_WIDTH, MAP\_HEIGHT), (x, y), radius, angle\_range)

return mask

def generate\_threat\_ring(coord, radius, threat) :

#returns a map with a threat ring as weights, intended to be added to existing environment map

#threat ring weigts = THREAT, circle size = RADIUS, centered on COORD

map\_update = np.zeros((MAP\_WIDTH, MAP\_HEIGHT))

mask = generate\_sector\_mask(coord, radius)

map\_update[mask] = threat

return map\_update

eventlist.py

#EventList.py

def printme(st = "ASDFASDFASDFASDFASD"):

print(st)

return st

def addme(x=0, y=0):

print(x+y)

return(x+y)

class EventList:

def \_\_init\_\_(self):

self.events = []

def add\_event(self, step, action, params=()):

self.events.append((step, action, params))

#when adding events with 1 parameter

#parameter must be added as a tuple with trailing ','

# e.g. ('asdf', )

#otherwise exec\_events won't unpack param correctly

#when adding class methods/functions pass parent object to action

# e.g. add\_event(1, some\_object.some\_action, (params,))

def show\_events(self):

print(self.events)

def exec\_events(self, step):

for events in self.events:

if events[0] == step:

x = events[1]

y = events[2]

x(\*y)

#myevents = EventList()

#myevents.add\_event(1, printme, ("ASDF",))

#myevents.add\_event(1, addme, (1, 5))

#myevents.add\_event(2, myevents.show\_events)

#myevents.show\_events()

#myevents.exec\_events(2)

runinteractive.py

#run.py

# Interactive Visualization

# show environment map and agents for

from C2D2E import \*

from model import \*

from agent import \*

def agent\_portrayal(agent):

portrayal = {"Shape": "circle",

"Filled": "true",

"Layer": 0,

"Color": agent.color.name,

"r": 0.5}

return portrayal

grid = CanvasGrid(agent\_portrayal, MAP\_WIDTH, MAP\_HEIGHT)

chart = ChartModule([{"Label": "Blue Total Attrition",

"Color": "Blue"},

{"Label": "Red Total Attrition",

"Color": "Red"}],

data\_collector\_name='datacollector')

server = ModularServer(C2D2EModel,

[grid, chart],

"Unit Model",

{"N": 7})

server.port = 8521 # The default

server.launch()

experiement1.py

from C2D2E import \*

from model import \*

from agent import \*

#vary risk acceptance of friendly com

Experiment1 = None

n = 0

#degree of RED\_TARGETING\_EFFECTIVENESS = independent variable

for i in range(1, 11, 1):

#Sim will look at 10 values of Red-For targeting [.1, .2, .3, ... 1]

RED\_TARGETING\_EFFECTIVENESS = i \* .1

for j in range(500):

#at each value of RED\_TARGETING, 500 samples / runs will be measured for Blue Com

BLUE\_COM\_RISK\_THRESHOLD = np.random.uniform()

c2model = C2D2EModel()

while c2model.running :

c2model.step()

filename = "Ex1RedTgt" + str(RED\_TARGETING\_EFFECTIVENESS) + "BlueCom" + str(BLUE\_COM\_RISK\_THRESHOLD)

agent\_stats = c2model.datacollector.get\_agent\_vars\_dataframe()

agent\_stats.to\_excel(filename + "agents.xlsx")

model\_stats = c2model.datacollector.get\_model\_vars\_dataframe()

lastrow = model\_stats.tail(1)

lastrow["RedTgt"] = RED\_TARGETING\_EFFECTIVENESS

lastrow["BlueCom"] = BLUE\_COM\_RISK\_THRESHOLD

if n == 0 :

Experiment1 = pd.DataFrame(lastrow)

else:

Experiment1 = Experiment1.append(lastrow)

print(n, i, j)

n += 1

Experiment1.to\_excel("Experiment1.xlsx")

print("ALL DONE")

analysis.py

import pandas as pd

import numpy as np

from scipy import stats

import matplotlib.pyplot as plt

def AdjustTestStatistic(n, D):

return (np.sqrt(n) + .12 + (.11/np.sqrt(n))) \* D

df = pd.read\_excel('Experiment1.xlsx', index\_col=None)

list(df)

# Conduct K-S Tests for each sample (n=500) to ensure standard uniform distribution of Blue Com Risk

for i in range(1, 11, 1):

if i == 3:RED\_TGT\_EFF = .3

else: RED\_TGT\_EFF = i \*.1

df2 = df.loc[df.RedTgt==RED\_TGT\_EFF]

bluecom = np.array(df2.BlueCom)

Dn = stats.kstest(bluecom, 'uniform')[0] #scipy unniform distribution defaults to standard U(0,1)

AdjD = AdjustTestStatistic(500, Dn)

print("RED Targeting Effectiveness = ", RED\_TGT\_EFF)

print("Adjusted Test Statistic ", AdjD)

KSTest = "Failed to Reject H0" if AdjD <= 1.358 else "Reject H0"

print("KSTest Test for Standard Uniform Distribution at Alpha.05 - ", KSTest)

print("")

plt.hist(bluecom)

plt.title("RED\_TGT\_EFF = " + str(RED\_TGT\_EFF))

plt.show()

#conduct scatter plot analysis of Red Targeting Effectiveness to KPIs

plt.scatter(df.RedTgt, df['Total Steps'])

plt.title("Red Targeting vs Total Steps")

plt.show()

print("(Pearson) Correclation Coef = ",df.RedTgt.corr(df['Total Steps']))

print("")

plt.scatter(df.RedTgt, df['Blue Total Attrition'])

plt.title("Red Targeting vs Blue Attrition")

plt.show()

print("(Pearson) Correclation Coef = ",df.RedTgt.corr(df['Blue Total Attrition']))

print("")

plt.scatter(df.RedTgt, df['Blue Final Effectiveness'])

plt.title("Red Targeting vs Blue Final Effectiveness")

plt.show()

print("(Pearson) Correclation Coef = ",df.RedTgt.corr(df['Blue Final Effectiveness']))

#conduct scatter plot analysis of Red Targeting Effectiveness to KPIs

plt.scatter(df.BlueCom, df['Total Steps'])

plt.title("Blue Communication vs Total Steps")

plt.show()

print("(Pearson) Correclation Coef = ",df.BlueCom.corr(df['Total Steps']))

print("")

plt.scatter(df.BlueCom, df['Blue Total Attrition'])

plt.title("Blue Communication vs Blue Attrition")

plt.show()

print("(Pearson) Correclation Coef = ",df.BlueCom.corr(df['Blue Total Attrition']))

print("")

plt.scatter(df.BlueCom, df['Blue Final Effectiveness'])

plt.title("Blue Communication vs Blue Final Effectiveness")

plt.show()

print("(Pearson) Correclation Coef = ",df.BlueCom.corr(df['Blue Final Effectiveness']))

print("")