NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

GENERATING REALISTIC MISSION PROFILES IN SUPPORT OF INTERACTIVE SYNTHETIC ENVIRONMENTS

by

Ryan M. Sohm

September 2021

Thesis Advisor:	Imre L. Balogh
Co-Advisors:	Perry L. McDowell
	Christian R. Fitzpatrick
Second Reader:	Glenn A. Hodges

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2021	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE GENERATING REALISTIC MISSION PROFILES IN SUPPORT OF INTERACTIVE SYNTHETIC ENVIRONMENTS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Ryan M. Sohm	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Modeling and simulations can determine how an upcoming technology will perform in the real world. An interactive synthetic environment (ISE) can accelerate technology development and improve the acquisitions process by providing immediate feedback on a specific technology or configuration. With the advent of artificial intelligence and machine learning, a model can be trained to choose the best performing configurations that fit a particular mission set. However, there is no current framework to develop these mission sets, or mission profiles, to allow the training to occur. Often, simulations are built, a specific mission or small set of missions is loaded, the simulation is run, and the results are examined. If a user could generate realistic, feasible mission profiles and run thousands of simulations with a single technology or specific configuration of a technology, the user would have more data for evaluation. This work aims to develop a framework that generates variable and feasible mission profiles. This framework provides a step towards a minimum viable product of ISE. Comparison of a large set of mission profiles between two competing technologies may provide a non-technical user a basis to decide which performs better. The result of this work will help advance modeling and simulations specific to acquisitions and wargaming, with the overall intent to expedite the acquisitions process.

14. SUBJECT TERMS interactive synthetic environment, ISE, mission, profiles, generation, modeling, simulation, Monte Carlo, autonomous, systems, wargaming, unmanned aerial system, UAS, vehicle, UAV, early synthetic prototyping, ESP	**15. NUMBER OF PAGES** 113
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

GENERATING REALISTIC MISSION PROFILES IN SUPPORT OF INTERACTIVE SYNTHETIC ENVIRONMENTS

Ryan M. Sohm
Captain, United States Marine Corps
BS, University of California–Los Angeles, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2021

Approved by: Imre L. Balogh
Advisor

Perry L. McDowell
Co-Advisor

Christian R. Fitzpatrick
Co-Advisor

Glenn A. Hodges
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Modeling and simulations can determine how an upcoming technology will perform in the real world. An interactive synthetic environment (ISE) can accelerate technology development and improve the acquisitions process by providing immediate feedback on a specific technology or configuration. With the advent of artificial intelligence and machine learning, a model can be trained to choose the best performing configurations that fit a particular mission set. However, there is no current framework to develop these mission sets, or mission profiles, to allow the training to occur. Often, simulations are built, a specific mission or small set of missions is loaded, the simulation is run, and the results are examined. If a user could generate realistic, feasible mission profiles and run thousands of simulations with a single technology or specific configuration of a technology, the user would have more data for evaluation. This work aims to develop a framework that generates variable and feasible mission profiles. This framework provides a step towards a minimum viable product of ISE. Comparison of a large set of mission profiles between two competing technologies may provide a non-technical user a basis to decide which performs better. The result of this work will help advance modeling and simulations specific to acquisitions and wargaming, with the overall intent to expedite the acquisitions process.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AAA	anti-aircraft artillery
AAF	Adaptive Acquisition Framework
AGI	Analytical Graphics, Inc.
AI	artificial intelligence
AO	area of operations
API	application programming interface
ARCIC	Army Capabilities Integration Center
BISim	Bohemia Interactive Simulations
CSV	comma separated values
DAS	Defense Acquisition System
DIS	distributed interactive simulation
DOD	Department of Defense
ESP	early synthetic prototyping
EW	electronic warfare
GUI	graphical user interface
IEEE	Institute of Electrical and Electronics Engineers
ISE	interactive synthetic environment
KPP	key performance parameters
MANPADS	man-portable air-defense systems
MCA	Major Capability Acquisition
ML	machine learning
MOVES	Modeling Virtual Environments and Simulations
M&S	modeling and simulation
MSL	mean sea level

MVP	minimum viable product
NAI	named area of interest
NPS	Naval Postgraduate School
NSW	Naval Special Warfare
OPFOR	opposing force
PDU	protocol data unit
PL	phase line
SBA	simulation-based acquisition
SDK	software development kit
SSAG	Space Systems Academic Group
STK	Systems Tool Kit
TAI	target area of interest
TCP/IP	transmission control protocol/internet protocol
UAS	unmanned aerial system
UDP	user datagram protocol
WARCOM	US Naval Special Warfare Command
WEZ	weapon engagement zone

Acknowledgments

I would like to thank my advisors Dr. Imre Balogh, Christian Fitzpatrick, and Perry McDowell, as well as my second reader Dr. Glenn Hodges. Your constant assistance and mentorship throughout this process made it possible. Thank you to Justin Davis for the introduction to ISE and the passion for it. Jonathan Berry and James Mulski, thank you for your early work on ISE which much of this thesis was built from. An enormous thank you to the students of computer science cohort 368-201 for an unforgettable few years. Finally, thank you to my family for the unyielding support. I'm happy to say that yes, it is now "done."

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 Motivation

The U.S. military has long enjoyed technological superiority over its adversaries. However, as nations such as China and Russia continue to modernize their militaries, this technological gap is quickly closing. The Marine Corps has recognized that in order to "maintain our warfighting overmatch," innovation and adaptation are necessary to thwart pacing threats [1]. The recently (2020) refreshed Defense Acquisition System (DAS) is in place to "acquire products and services that satisfy user needs with measurable and timely improvements to mission capability...at a fair and reasonable price" [2]. While new acquisition pathways offer a promising start, adversaries less burdened by bureaucracy may bring novel technological solutions to the battlefield faster. President Biden's Interim National Security Strategic Guidance recognizes that the "world's leading powers are racing to develop and deploy emerging technologies" [3]. However, completing a Major Capability Acquisition (MCA) as shown in Figure 1.1, especially with an unknown or unproven technology, requires significant time, cost, effort, and regular oversight. Modern Department of Defense (DOD) acquisitions programs must deliver major capabilities to the warfighter faster than any adversary, who need not conform to a similarly demanding process [4].

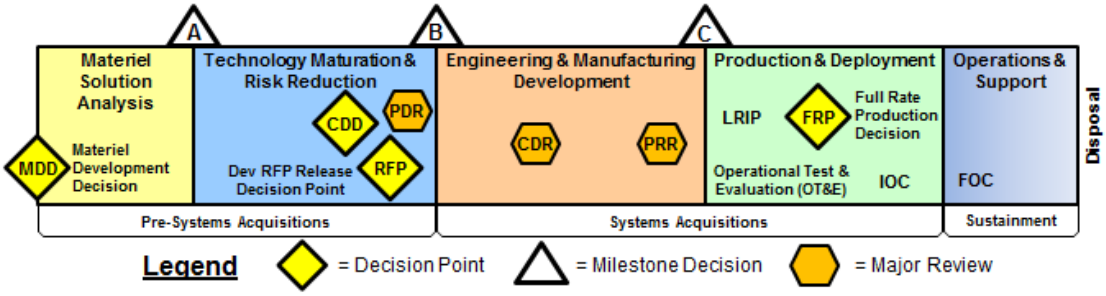

Figure 1.1. The MCA Process is considered the "traditional" acquisition approach. The phases and milestones ensure constant feedback, review, and decision making. Source: [5].

If it is assumed that the DOD acquisition process is a result of democratic and well-meaning policy, and reform is a "continuing process rather than an achievable end state," then the focus for improving the "speed to capability" should be within the project life cycle [6]. Similar to the MCA process in Figure 1.1, the life cycle defined by Heagney separates a project into phases: Concept, Definition, Planning, Execution, and Closeout as seen in Figure 1.2. However, Heagney posits that "projects seldom fail at the end. Rather they fail at the definition stage" [7]. In order to improve the acquisition of a major capability then, improving the "pre-systems acquisitions" phases, or the planning phases, would have greater impact than the "systems acquisitions" phases. If DOD can accelerate the concept, definition, and planning phases as in Figure 1.2, and improve the products that leave these phases, DOD can also expect a better end product.

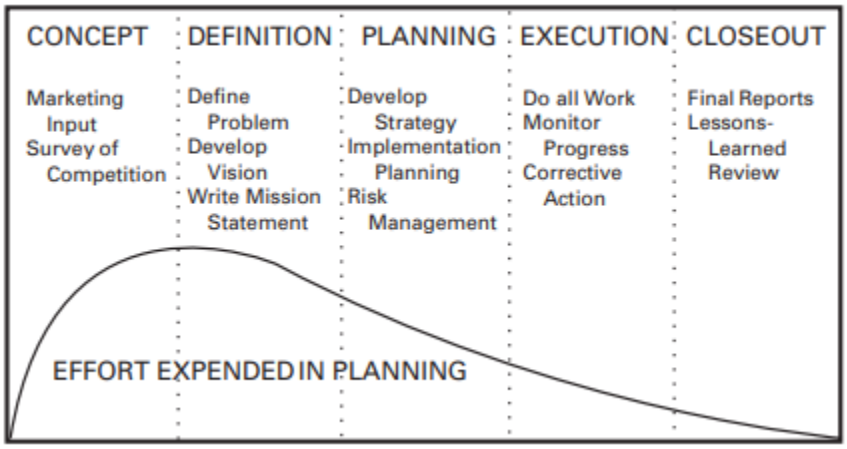

Figure 1.2. The appropriate project life cycle as defined by Heagney notes that the definition phase is critical to a project's success or failure. Source: [7].

Modeling and simulation (M&S) can provide solutions that allow for significant analytic capability at early stages of development. Much like a production prototype, a simulated prototype can ensure those iterative development processes occur and produce a successful outcome in production. M&S can impact the acquisition decisions and milestones early in a program's development, increasing efficiency and decreasing errors [8]. Early synthetic

prototyping (ESP) is a research project sponsored by the Army Capabilities Integration Center (ARCIC) dedicated to accelerating the feedback loop between engineers and soldiers. ESP pairs a process and tools that allow soldiers to assess a new technology within a simulated environment [9]. ESP's first product, Operation Overmatch, places soldiers in a virtual environment and allows them to choose vehicle's configuration and fight other soldiers serving as an opposing force (OPFOR) [10]. The feedback from the recorded session as well as soldiers' comments provide a test bed for novel configurations, some of which designers may not have considered [11].

Following the success of Operation Overmatch, in December 2019 the Naval Postgraduate School (NPS) Naval Special Warfare (NSW) Chair began formulating the concept of an interactive synthetic environment (ISE) for Warfighting Development. The goal was similar to ARCIC's ESP program: develop a realistic and immersive tactical simulation environment, for rapid development of future warfighting concepts and capabilities by non-technical users. ISE attempts to place the development of a novel unmanned aerial system (UAS), the possible replacement for the current ScanEagle UAS, into the hands of the end-user before production ever begins [12]. A non-technical user can then decide whether the new system outperforms the old. The motivation of this thesis is to provide a first step for ISE: create a baseline framework that allows for a direct comparison between one platform, UAS or otherwise, and its virtual competitor. Instead of comparing two platforms by evaluating key performance parameters (KPP)s, a non-technical user can compare and evaluate system performance by using the same set of thousands of realistic, generated mission profiles. The user can now make a comparison based on mission successes and failures and determine if the new technology is superior, equal, or inferior to existing capabilities. This thesis will focus on the generation of those mission profiles and their applicability towards a future ISE implementation.

1.2 Research Questions

The goal of this research is to develop the framework supporting a minimum viable product (MVP) implementation of ISE. It will focus entirely on the generation of mission profiles and how to create a comparison using those profiles. Research questions pursued in this work include:

- How can a user include historical data in generating mission profiles?
  - What data would a user include and utilize in a mission profile?
  - What is the minimum set of data to represent historical events for future missions?
  - Can a user represent historical enemy activity and determine if it will affect a mission?
- What inputs are required to generate full mission profiles for a particular aircraft?
- What outputs from a full mission simulation can be generated to evaluate aircraft performance?
  - Can you discover the limitations of a particular UAS or configuration through varying mission profiles?

1.3 Scope

While the ultimate goal for ISE is to answer "Is capability A better than capability B?" that answer is far beyond the scope of this thesis. Instead, the focus is on developing the framework behind that eventual comparison: generating the mission profiles that will provide a basis for comparison. This work will serve as a proof of concept and springboard for future development. It will show that thousands of mission profiles can be generated for use in a Monte Carlo-like simulation. The cumulative simulation results will then show whether a UAS successfully navigated a subset of those mission profiles, and show that a comparison can be made. This work will not focus explicitly on the validity of the comparison, but provide a pathway to validity. Future work may create mission profiles, using the code in this thesis, that reflects real world statistics and provide a more apt comparison.

1.4 Benefits of Study

The main beneficiary of this study will be the ISE project. This thesis hopes to serve as a framework to allow for further development of the project, or another interested party to continue its development. The ISE project goals are far more advanced than the goals of this thesis, but the work here hopes to prove or disprove that this methodology and comparison is valid. With the success of ISE or other projects like it, acquisitions and development of warfighting technology may occur faster and cheaper. This study hopes to assist those

in acquisition roles, as well as the warfighter, so that both parties can speak the common language of mission success or failure.

With the integration of this framework into a simulation environment, academia and industry are provided a system to validate technological solutions against existing solutions. If enough generalization is achieved in this study, the framework would require little work to tailor it toward a ground vehicle or dismounted infantry's movement in and out of an objective area. With a set of realistic mission profiles, artificial intelligence (AI) or machine learning (ML) solutions could validate a solution based on completion of an enormous variety of generated mission profiles. Such future improvement could remove the human from the feedback loop, allowing for much faster iteration and development of a prototype system.

1.5 Thesis Organization

A thorough technical background of most of the topics covered in this study will be covered in Chapter 2. Chapter 3 will review the methodology and tools used in the creation of mission profiles, as well as a code overview and concepts behind it. Chapter 4 will present the results and analysis gathered by generating mission profiles, along with the simulation results. This section will also cover any developmental issues or roadblocks, as well as how to analyze and use the generated data. Finally Chapter 5 will discuss any conclusions and future work that may use the developed framework.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Technical Background

2.1 Modeling and Simulation in Acquisition

DOD's Adaptive Acquisition Framework (AAF), seen in Figure 2.1, provides different acquisition pathways for a variety of hardware and software programs. The purpose of the "traditional" waterfall-type approach encapsulated in the MCA is "to acquire and modernize military unique programs that provide enduring capability" [13]. As depicted in Figure 2.1, the MCA pathway takes the longest, usually costs the most, and should be used in "major systems" and "complex acquisitions" [13]. When these programs fail the cost to taxpayers is measured in billions of dollars. One example of a failed MCA is the Boeing-Sikorsky RAH-66 Comanche program. Over the course of 20 years the Army invested roughly $6.9 billion in the Comanche's development. The long development time is attributed to shifting priorities after the end of the Cold War, "digitization" which brought about the programs renewal, and finally the new requirements of the Global War on Terror. While programs similar to the Comanche, namely the F-22A Raptor, only suffered a reduction in units, the Comanche was canceled due to a lack of political support, lack of cost effectiveness, and an overall shift in the strategic landscape [14].

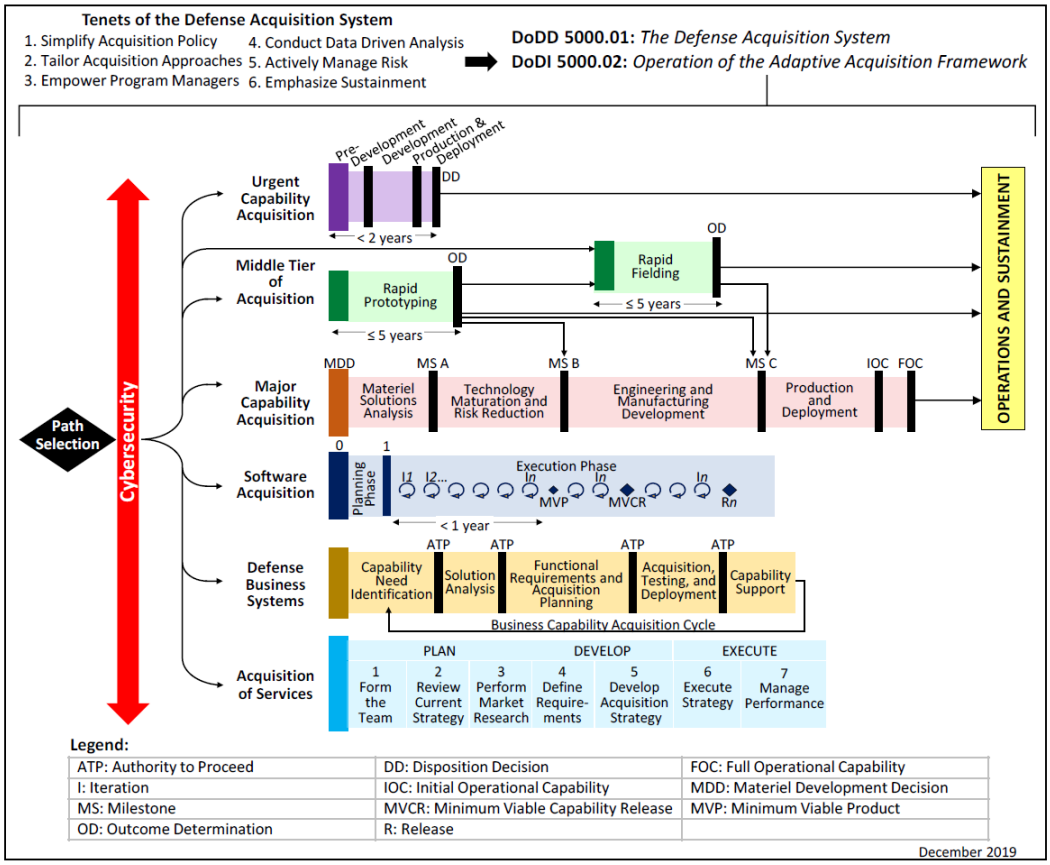

Figure 2.1. The AAF, developed in 2019, provides multiple acquisition pathways to support a variety of acquisitions programs. Source: [13].

By shortening the MCA pathway timeline, an acquisition program can deliver results to the customer faster, avoiding the delays caused by shifts in requirements. Simulation-based acquisition (SBA) was proposed in the early 2000s to reduce the time, resources, and risk associated with major acquisitions [15]. Programs could verify and validate models before any hardware prototypes were built. By making investments into M&S early in the acquisition life cycle the amount of risk in large programs can be drastically reduced. Additionally, "70% of downstream logistics cost is affected by design factors normally solidified early in the life cycle" [16]. SBA can implement design changes that affect future logistics resulting in lower operating and support budgets late in the program's lifecycle. Overall SBA presents a significant return on investment in MCAs, and paved the way

for more advanced techniques to support simulation in acquisition [16]. Early simulation studies can provide scope to requirements such as determining what sensor effectiveness is necessary for optimal coverage of a battlespace given multiple integrated platforms [17]. The results of SBA can also drive the number of items acquired by DOD. For example, a simulation study suggested that 14 non-lethal weapons creates the optimal balance between number of lethal shots fired and the number of non-lethal weapons carried by a Marine platoon [18]. Both studies aimed to prevent wasted time and money in acquisitions.

2.2 Early Synthetic Prototyping

ESP is an effort to use a physics-based game to assess how technologies might be employed on the battlefield [10]. The Army explored ESP utilizing game environments to evaluate new designs early in the acquisition life cycle [9]. The use of a game environment allows for lower development costs as well as increased accessibility to computer-literate but non-acquisition savvy users. Targeted towards the soldier, ESP created a set of processes and tools that enabled quick feedback to acquisitions professionals. DOD acquisition professionals and industry contractors could then refine the concept to better fit the needs of the warfighter. Including the end user also allowed for radical experimentation and on-the-fly development by those personnel who use the end product [19]. As depicted in Figure 2.2, the ESP construct involved a large feedback loop between Red and Blue team users, the non-technical end-users, and the acquisitions side of Government, Industry, and Science and Technology personnel. Acquisitions professionals (1 in Figure 2.2) would create scenarios (2) to answer questions, perform measurements, or verify models, which are simulated and wargamed by Red and Blue teams (3-7). Play data and user feedback are collected and sent back to acquisitions professionals for analysis and further scenario development (8-11). This iterative form of development more closely aligned with the agile-like software development of the software acquisition pathway.

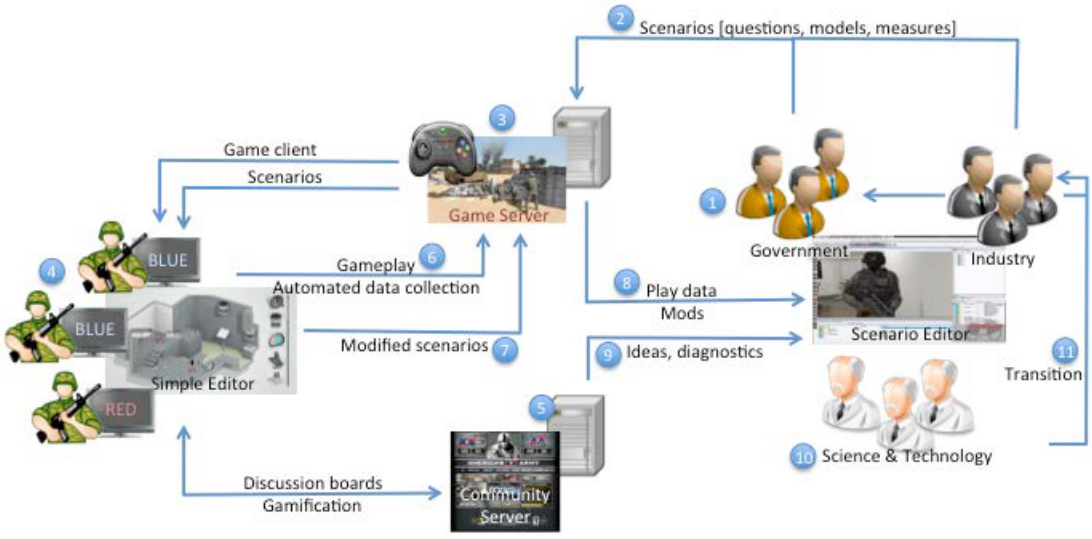

Figure 2.2. Early Synthetic Prototyping schematic describing interactions between Government and Industry, Science and Technology, and Users. Source: [19].

While SBA attempted to simulate issues in the entire life cycle of a program, ESP focused on much earlier stages of development. ESP aimed to capitalize where the cost to change a program is low: prior to developing a physical prototype. In this way, lessons learned through ESP are incorporated into a program's schedule and budget before significant money and labor is spent. The "change demand," which usually occurs after testing and production when the system is in the hands of the end users, is shifted earlier. Figure 2.3 highlights the focus areas of ESP and SBA inside a traditional MCA program. One attempt at implementing ESP and bringing the warfighter into the decision loop of acquisitions is Operation Overmatch.

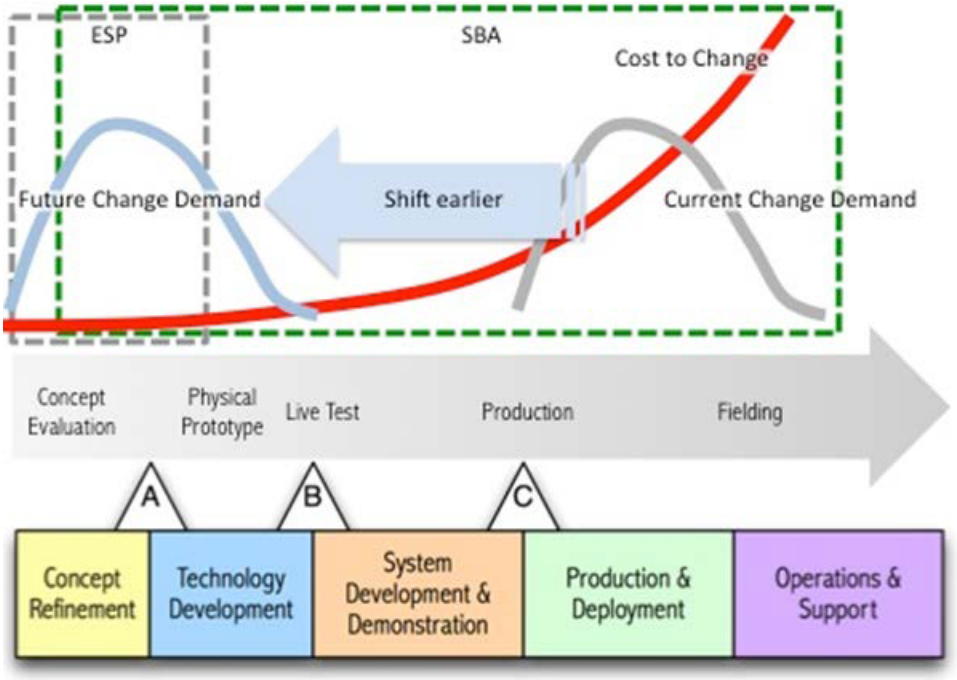

Figure 2.3. Shift of change demand with ESP and the relative cost to change during each stage of an MCA. Source: [19].

2.3 Operation Overmatch

Operation Overmatch hoped to tap into the large amount of time soldiers were spending playing video games; a survey from an ESP study at Fort Bliss indicated roughly a million hours of off-duty play a month. To leverage this large pool of time and ability and benefit Army acquisitions, a game was created to prove and employ emerging technologies prior to expending development dollars. Soldiers are able to experiment with utilizing and defending against emerging technology and tactics while scientists and engineers ingest the telemetry and feedback to improve acquisitions [11]. The excerpt below describes a future vision of Operation Overmatch:

> After everyone logs in, the team receives a virtual budget and must first choose its base vehicle from three options ... This team opts for the hacked taxi. Next, players move to the virtual garage to kit out their vehicle using their remaining

> virtual cash. The cadets decide against adding armor to their already slow taxi and instead choose soft exoskeletons to wear when they dismount. ... Still, the enemy will create surprises, as the opposing force is played by another group of Soldiers. [11]

In the 2018 iteration of Operation Overmatch, soldiers could select from both red and blue force vehicles and select different loadouts for those vehicles. A soldier could swap out munitions, systems, sensors, and defenses as seen in Figure 2.4. Using the feedback from the above hypothetical scenario, scientists, engineers, and concept and capability developers may decide where to focus their development dollars. Given that the most effective loadout in the prescribed scenario, which reflects some real world locale, is a taxi and not a heavily armored tracked vehicle, Army acquisitions may be less inclined to invest in heavier armored technology. ESP systems such as Operation Overmatch might justify the fielding of smaller numbers of vehicles and weapons tailored to a specific mission or area [20].

Figure 2.4. Gameplay from Operation Overmatch shows red and blue team selections along with varying modules on each vehicle. Adapted from [21].

2.4 Interactive Synthetic Environment

ESP's goals are similar to that of ISE. While ESP focuses on the acquisition-side of the arena, ISE aims to address both acquisition and operational requirements. The ScanEagle, a smaller block two UAS primarily used for intelligence, surveillance, and reconnaissance, was

originally developed to track tuna and dolphins from fishing boats [22]. US Naval Special Warfare Command (WARCOM) and its acquisition professionals aim to create a replacement for the ScanEagle; ISE may serve as the tool to aid collaboration between acquisitions and the warfighter. Similar to Operation Overmatch, ISE hopes to enable development of new employment strategies and development ideas. Creating the replacement for the ScanEagle in ISE, through constant iteration and virtual prototyping, may also result in significant time and money savings [12].

Another goal of ISE is to assist in the warfighter's operation of a ScanEagle replacement. This future UAS may consist of swap-able modules that change the performance characteristics of the system. Traditional KPPs may not serve to accurately represent this modular UAS's capabilities if it is scalable to fit the needs of the warfighter. The complexity of many different modules for many different missions may be daunting for a non-technical user. If a ground commander wishes to employ this hypothetical next-generation UAS for a specific mission, then s/he would require an in-depth technical knowledge of every module, how the aircraft performs with each module, and whether the configured aircraft will fit the needs of the mission plus any contingencies. To remedy this situation, ISE can serve as a mission planner, which chooses the ideal aircraft configuration for a particular scenario through wargaming [12].

While this thesis will focus on the technical development of a potential ISE framework, the Master's thesis entitled "Other Transaction Authority (OTA) Application for Warfighting Development" by J. Berry and J. Mulski covers ISE's initial acquisition efforts. Because a specific vendor has not been decided, this thesis attempts to create a general solution using commercial off-the-shelf simulation products.

2.5 Simulation Environments

The thesis requirements for selecting a simulation solution are described in Chapter 3. There were three commercial simulation products that stood out as useful for this thesis. These were selected primarily for their relevancy to the Berry and Mulski thesis, availability of subject matter experts, and extensibility through modules and add-ons. While the framework will choose a subset of these simulators to work in, there will be minimal effort in converting this solution into other simulation products.

2.5.1 VBS4

VBS4 is a simulation that touts "whole-earth virtual and constructive simulation" developed by Bohemia Interactive Simulations [23]. VBS3 has long been used by the Marine Corps to train call for fire, close air support, and fire support team integration [24]. VBS4 emphasizes ease-of-use, efficient workflows, and whole-earth data. The sensor viewpoints (seen in Figure 2.5) and course-of-action analysis might directly relate to the goals of this thesis; a user can plan a mission, allow the VBS Control AI to take control of the friendly and enemy forces, then assess if that plan meets the commander's goals. The VBS Simulation software development kit (SDK), which grants access to source code and application programming interface (API)s to create custom applications, requires a separate purchase. The SDK would be required to perform any sort of integration or modification of the core simulation components, which may be required to reach the thesis goals [23]. Ultimately VBS4 was not utilized in this thesis since VR-Forces presented a simpler interface to place units and waypoints, granted easier access to distributed interactive simulation (DIS) packets, and Systems Tool Kit (STK) was less graphics-intensive with large numbers of simultaneous simulations. However, later iterations of this project might integrate directly with VBS4 for more seamless and high-fidelity experiences.

Figure 2.5. VBS provides sensor views that can be tuned using VBS Simulation SDK. Source: [25].

2.5.2 VR-Forces

VR-Forces, a simulation solution developed by MAK technologies, is currently used by the Modeling Virtual Environments and Simulations (MOVES) Institute aboard NPS. It provides a simple scenario editing tool that allows users to position forces and create waypoints. The simulator is also packaged with the DISSpy application which allows for the inspection of DIS packets. All of these simulation environments are DIS compatible, but VR-Forces exposed the DIS packets to the end user more than any other. VR-Forces also used a less graphically intensive user interface to plan and create scenarios. The simpler interface resembled a map with standard military symbology, as seen in Figure 2.6, rather than a high-fidelity 3D interface.

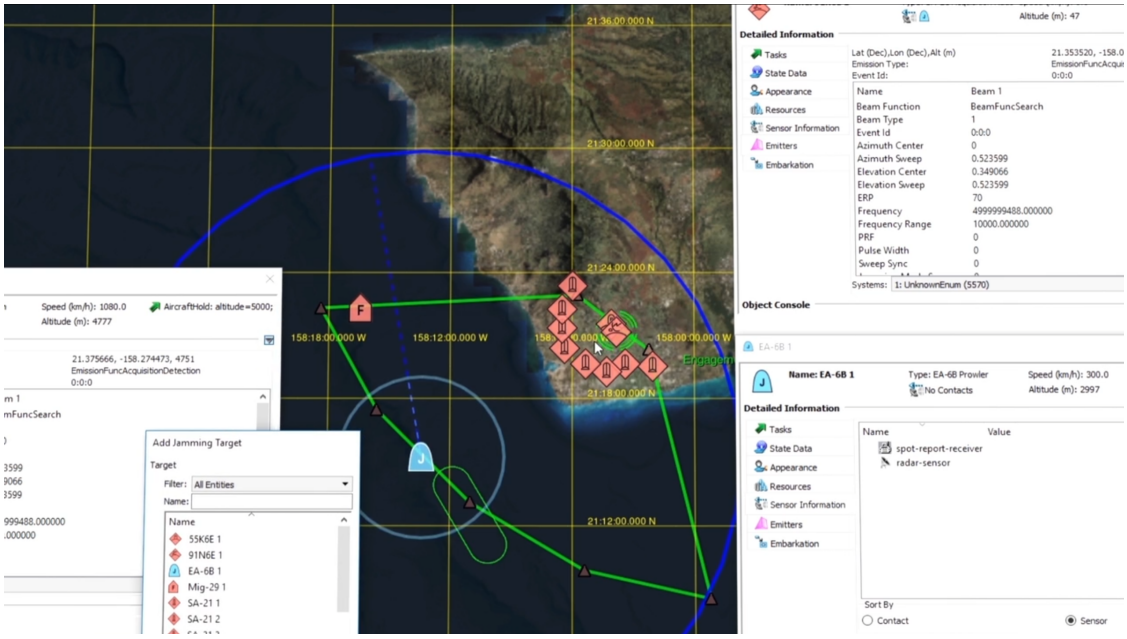

Figure 2.6. VR-Forces provides a simple user interface that exposes DIS packet information. Source: [26].

2.5.3 Systems Tool Kit

The Analytical Graphics, Inc. (AGI) developed STK is also utilized extensively by the Space Systems Academic Group (SSAG) at NPS. STK offers a large library of specialized modules, some of which are directly applicable to this thesis and ISE. STK Aviator and Aviator Pro offer aircraft physics simulation, as seen in Figure 2.7, valuable in calculating aerodynamics, fuel consumption, and wind/atmospheric effects in a future version of this framework [27]. The STK Integration module also grants access to the STK interfaces through a variety of programming languages and tools. To support the proof-of-concept nature of this thesis, the base STK version allows users to input STK Connect code (plain-text instructions) directly into the simulation, bypassing the need for purchasing additional modules. The simulation engine also lends itself to simulate hundreds of objects simultaneously; this capability would be required to see the results of hundreds of mission profiles at the same time [28]. Finally, the wealth of knowledge aboard NPS on STK, as well as available licenses, suggest STK would be the most useful in developing this thesis work.

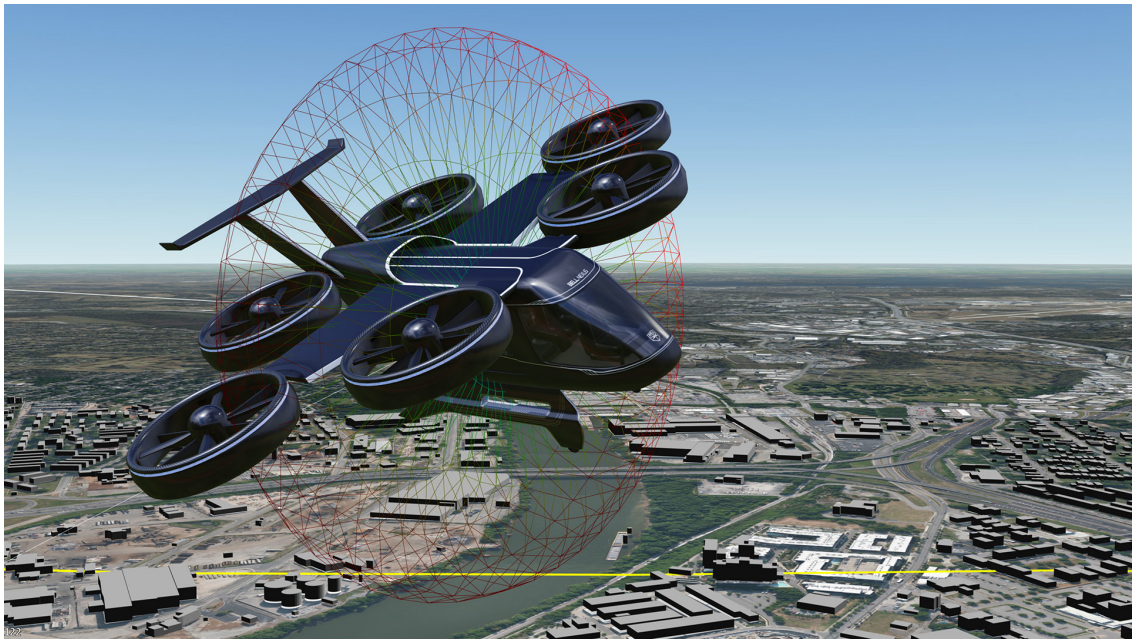

Figure 2.7. STK using the Urban Communications module and a prototype Bell Nexus aircraft. Source: [27].

2.6 Python

Python was selected as the primary language to develop the initial ISE framework. Aside from the use of Python in the NPS computer science curriculum, Python's many modules and extensive API meant a straightforward selection for use in this thesis. Taking into account the possible future work for this thesis, Python's ML, statistics, and big-data libraries allow for substantial room to develop the framework. STK fully supports Python through their Integration module and Gears Studio, the integrated development environment used by Bohemia Interactive Simulations and VBS4, claims to integrate with Python in the near future [29].

CHAPTER 3:
Methodology

3.1 Creating Mission Profiles

3.1.1 Separation Into Phases and Events

The joint Army, Marine Corps, Navy, and Air Force publication entitled *UAS Multi-Service Tactics, Techniques, and Procedures for the Tactical Employment of Unmanned Aircraft Systems* describes the planning and execution phases of UAS operations. This publication also discusses the many employment considerations present when incorporating UAS in a combatant environment including environmental considerations, communications, spectrum management, airspace management, hand overs between units, and lost links. Execution of UAS missions generally involves three top-level phases: pre-mission, mission execution, and post-mission. In the pre-mission phase UAS operators receive tasking to support a unit. This phase will last until mission communications is established with the supported unit and the UAS crew can receive direct updates and information. During the mission execution phase, coordination occurs directly with the supported element (i.e., the UAS is on station) until the supported unit releases the UAS or it receives higher priority tasking. Finally, in the post-mission phase, the UAS ceases communication with the supported unit and either returns to base or moves to another supported unit location and another pre-mission phase [30]. The reconnaissance tasks required by a UAS, whether route, area, or zone reconnaissance, usually shift at or near specific boundaries on a phase line (PL), named area of interest (NAI), target area of interest (TAI), or area of operations (AO) as seen in Figure 3.1.

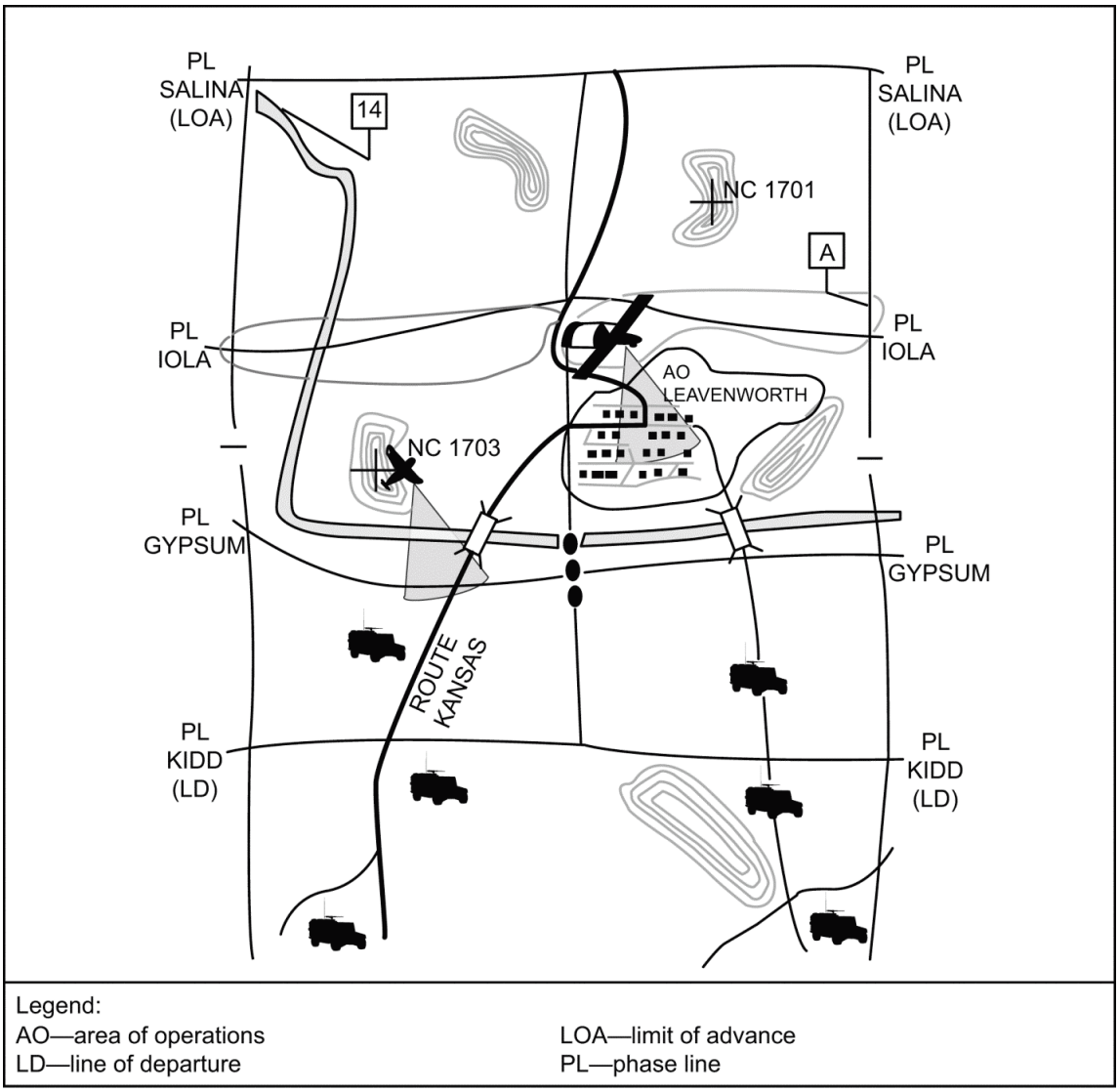

Figure 3.1. Two UAS conduct area and zone reconnaissance of an NAI (PL Gypsum and Route Kansas) and an AO (Leavenworth), respectively. Source: [30].

Knowing that locations or waypoints generally guide different phases of an operation, and that one mission may shift into multiple mission execution phases, the developed framework should focus on simulating phases that are then chained together for a complete mission

profile. A phase will also determine what possible events may occur, as the proposed task usually changes with the phase. During a pre-mission phase, when contact has not yet been established with the supported unit, events might include a link loss, UAS handover, retasking, or airspace management and deconfliction. In contrast, mission phase events might include enemy detection, surveillance, reconnaissance, or providing fires [30]. While these events are not exhaustive or exclusive to those phases, their relative likelihoods, driven by historical data, may provide some insight on the overall risk of UAS during each phase and what mitigation may be necessary. Therefore, the proposed framework should support the historical input of events by phase and allow for the chaining of phases into a single mission profile. Figure 3.2 shows the three top-level phases followed by a further breakdown of specific launch, ingress, actions, egress, and landing phases. Below each phase, the figure also displays some example events that may occur during each phase with some historical likelihood. As the simulated UAS travels through a single mission profile, it generates a subset of those events during each phase.

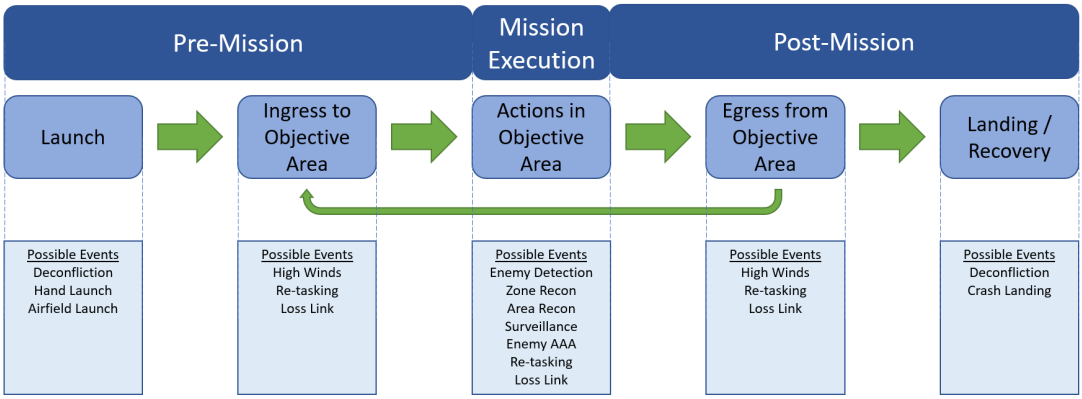

Figure 3.2. The three top-level phases broken down further into potential phases of a mission, with example events that may occur in each phase.

3.1.2 Scenarios and Campaigns

The breakdown of a mission profile into phases facilitates the "Monte Carlo-like" nature of creating many possible mission outcomes from one proposed mission. For clarity, the

initial mission that will generate all of the mission profiles will be called a Scenario. This nomenclature aligns with what most simulation environments define as a scenario, a small vignette of a larger battlespace usually involving a single mission or objective [31]. VBS4, a simulation solution created by Bohemia Interactive Simulations (BISim), VR-Forces from MAK, and STK by AGI all use "scenario" to describe the initial state where simulation begins, as seen in Figure 3.3. Similarly, in this framework, scenario will represent the initial proposed mission that generates the random set of mission profiles.

If the user of ISE is an acquisition professional or contractor, they may want to create the best possible UAS given multiple scenarios. To facilitate this functionality in the framework, the term Campaign is used to define a set of scenarios. The tactical user might also wish to configure a potential UAS for a variety of specific scenarios. This eventual ISE use case could determine what specific modules or configurations the tactical user brings out on operations enabling specific capabilities. In summary, a campaign consists of multiple scenarios, scenarios generate a set of mission profiles, each profile has a set of phases, and each phase has events that may take place inside it.

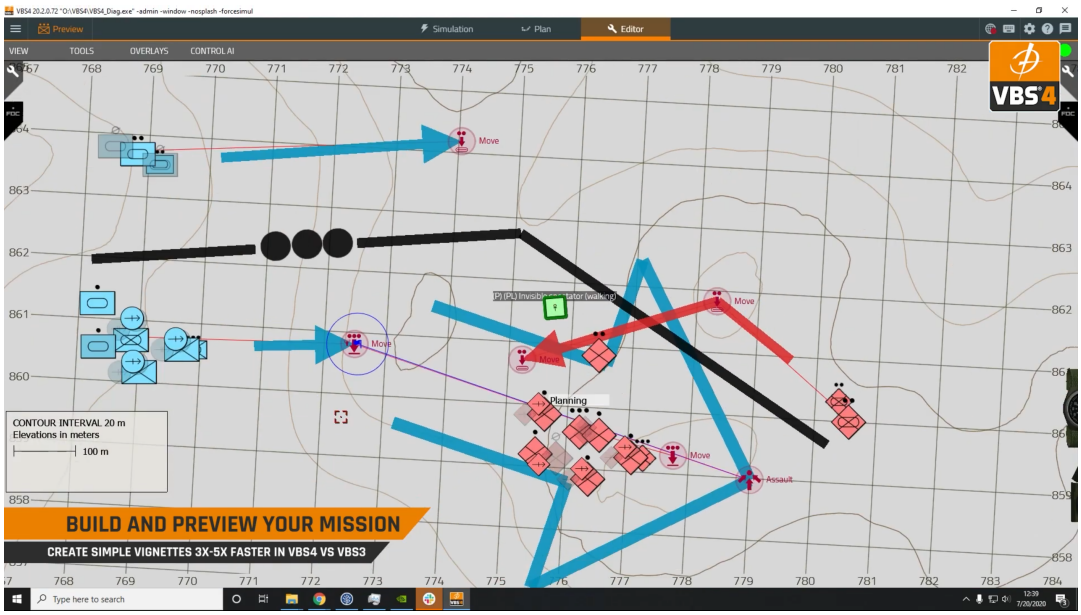

Figure 3.3. Scenarios represent a single vignette in a larger battlespace. They are often the initial state of a simulation representing a single mission or objective. Source: [31].

3.2 Requirements

Following the campaign, scenario, phase, and event breakdown there were three primary design goals in developing the initial ISE framework.

1. Allow for relatively easy input of historical data from a non-technical user.
2. Generate a set of mission profiles into "mission strings," an intermediate output to facilitate any simulation environment.
3. Convert mission strings into a format readable by a simulation environment.

Since the end users of ISE may range from an acquisition professional to an 18-year old Marine, and the validity of a simulation may depend on historical data, the input of data must be approachable for most levels of computer literacy. The format of the historical data should be easily translatable from some other source format (e.g., avoid a proprietary format), capable of generation by hand if no historical data exists, and able to accept updates by the end user.

The use of intermediate mission strings ensures that the proposed framework is not dependent on a particular simulation solution. If the framework only output to a particular simulator, such as STK, and the acquisitions side of the ISE selected BISim with VBS4 as an industry partner, the framework would be largely useless without significant rework. By using mission strings, if another contractor was selected (or if multiple were selected), the only coding effort would be to translate the mission strings into the preferred input format. Intermediate mission strings also enable debugging and verification before injection into the preferred simulation environment. Development can continue on mission profile generation even if a simulation environment is unavailable or cost-prohibitive at the time.

The final step is to translate mission strings into a format that is readable by the preferred simulation environment. The framework should support any integration between it and a simulation, whether translation involves converting mission strings into another simulation-recognizable string, or directly making calls to the simulation through Python (or any other language). Simulators like STK provide connections through Python, Java, or MATLAB using their STK Object Model. These connections provide a more efficient pathway to exchange information with the simulation environment [32].

3.2.1 Implicit Requirements

Preliminary research into these primary requirements yielded some clear pathways to solve them. While there may be some changes to the implicit requirements going forward, these choices best fit the proof-of-concept target for this thesis.

- Use of Microsoft Excel to input historical data.
- DIS-compatible simulation software to create scenarios.
- Use of Python as an intermediate language to generate profiles.
- Final proof-of-concept simulation in STK.

Personal experience dictated Microsoft Excel as the obvious choice of historical data entry. Excel boasts a straightforward interface for input and almost universal familiarity. Any user who has basic computer literacy should have some exposure to spreadsheets or Excel. The user can edit or input historical data with ease and developers can make a future transition into a full-fledged database without a significant code overhaul. Excel also supports input from various data formats such as comma separated values (CSV), which may represent the output of current UAS event and mishap tracking.

DIS-compatible simulation software was necessary to future-proof this framework. As with the decision to use intermediate mission strings, using a DIS-compatible solution prevents locking in to one particular simulator. Since DIS is an Institute of Electrical and Electronics Engineers (IEEE) standard for non-real-time and real-time wargaming, any program that can interpret DIS packets can communicate with DIS simluators [33]. This standard is used by VBS4, VR-Forces, and STK to represent entities, their positions, and velocities.

Python's robust libraries, popularity especially in the big data realm, and compatibility with VBS4 and STK made it a clear choice to develop in [34]. Python's libraries and big data popularity allow for future extensions utilizing AI and ML. Additionally, there is a Python branch of Open-DIS, a free and open-source implementation of the DIS IEEE standard [35]. This will drastically reduce the amount of effort involved in examining, understanding, and parsing DIS packets from different simulation environments. Utilizing Open-DIS also brings about a pathway for additions and extensibility into other DIS elements beyond the concept of an Entity.

Licensing expensive simulators became a common issue in this thesis work and will be

discussed later on. However, STK and NPS's use of it in other departments, particularly the SSAG, provided the most cost effective solution for this thesis work. STK also provided an alternative form of processing commands: STK Connect. Connect allowed for interaction with the simulation using strings over a network connection. This ability to feed STK strings of information meant less of a code effort. The intermediate strings could be converted into Connect strings and run inside STK's simulation environment. The process also saved on cost as STK Professional, which enabled outside integration with the STK Object Model, required an additional license to function [28]. Figure 3.4 provides an example of how an external application would interact with STK through their Connect model. While this solution requires copying Connect strings from an output file and pasting them into STK, this works fine for a proof of concept.

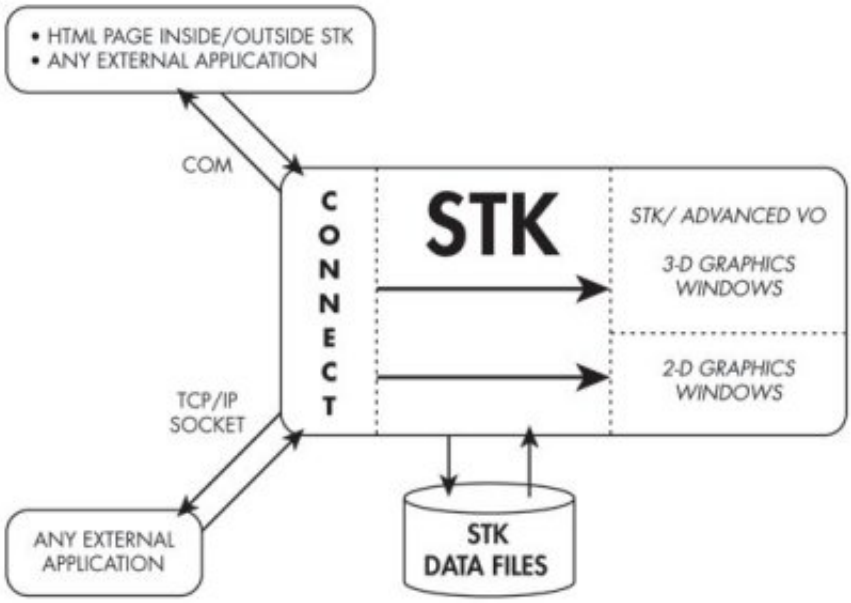

Figure 3.4. A TCP/IP connection can be utilized to send Connect commands over a specific port, eliminating the need for a language specific connection through Java or Python. Source: [28].

3.3 Code Architecture

By identifying the primary requirements of the mission profile framework, three core components emerged. The first is an input component, responsible for gathering historical data, metadata relevant to particular scenarios, and DIS input from a running simulation environment. The second is the mission profile generator, which converts all the inputs into the equivalent campaign, scenarios, missions, phases, and events. After organizing the input data, the resultant mission strings are generated by introducing variability in the form of waypoint fuzzing and event probabilities. This ensures no two missions profiles are the same and accounts for unforeseen variation and inaccuracy during a mission. Finally, the mission strings are converted into the proper output format, in this case STK Connect strings. If additional or replacement output modules are needed, the difference is a single line of code that instantiates a new output generator.

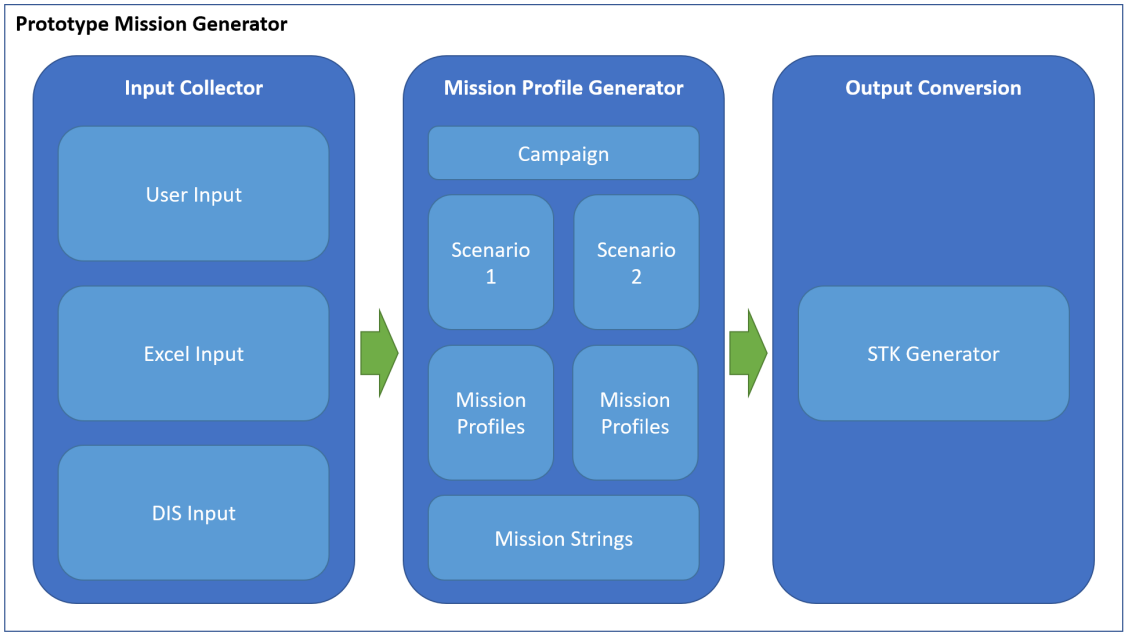

Figure 3.5. A prototype architecture for generating mission profiles. Note that the Output Conversion only contains an STK Connect generator but can accommodate multiple outputs.

Figure 3.5 shows a high-level breakdown of how the prototype mission generator functions. Each section will be further broken down to serve as documentation along-side the code. The top-level component that serves as the main entry point for the program is `PrototypeGenerator.py`. It performs basic system option parsing to select test mode or specify a seed for easily reproducible results. Table 3.1 displays the full list of Python classes used in developing the framework. While this code is by no means "done" and efficiency was not a goal, it represents a first draft attempt at a framework. A list of software libraries used in development can also be found in Appendix A.

Table 3.1. A list of the Python classes created during development and the functions they perform.

Class	Description
Campaign	Contains all scenarios and their metadata. `generate()` method returns a full list of mission strings.
DISInput	Gathers DIS input by listening for DIS-compatible simulators. `collect()` method populates .units and .waypoints
Event	Describes a single event, its probability, and where it may occur.
ExcelInput	Gathers historical data from an Excel spreadsheet, passed as a parameter. `parse()` method gathers phases, events, and event waypoints.
InputCollector	Collects input using DISInput, ExcelInput, and user input at command prompt. `collect()` method gathers all relevant data and returns a Campaign.
Mission	Represents a single mission, its waypoints, and any events that took place.
Phase	Represents a single phase, its start and end waypoint, and any events in that phase. `generate_events()` generates a list of events that occurred during this phase, for a single mission.
PrototypeGenerator	Main top-level class for generating mission profiles. Use '-t' for testing mode and '-s <seed>' to use a specific seed. `run()` generates mission profiles and outputs Connect strings.
Scenario	Holds the specific phases and units for a given scenario. `generate()` creates the requested number of profiles for this scenario. Returns the mission strings and a counter for the mission number.
STKConnectGenerator	Creates STK Connect strings for simulation. `add_aircraft()`, `add_facility()`, `add_waypoint()`, and `compute_access()` generate respective STK strings.
WaypointFuzzer	Generates a random x,y,z waypoint within a normal distribution given horizontal and vertical standard deviation. `fuzz_single()` takes a single waypoint and `fuzz_group()` takes a list of waypoints.

3.4 Input

The input portion of the architecture consisted of the `InputCollector.py` class. After gathering all the relevant input data, it will build out a campaign to generate mission profiles from. It incorporates the `ExcelInput.py` class to collect the historical data, the `DISInput.py` class to sniff[1] DIS specific packets, and the `Campaign.py`, `Scenario.py`, `Phase.py`, and `Event.py` classes to build a campaign.

3.4.1 DIS Input

It was decided early on to take advantage of a DIS-compatible simulator to prevent a large coding effort for the initial input scenarios; a user could utilize a well-developed graphical user interface (GUI) to pick and place units and waypoints. By offloading the GUI and DIS packet generation to a simulation environment, this thesis could focus more on the generation of profiles from those scenarios. Additionally, by using DIS, this thesis can stay agnostic to the front-end simulation environment used. The first challenge then was to find a way of parsing DIS packets without investing the time to make sense of every byte moving across the wire.

Thankfully, Open-DIS is a solution actively developed by the MOVES Institute at NPS. Open-DIS provides Java, C++, Python, Javascript, and C# implementations of the DIS protocol [35]. Figure 3.6 shows the Open-DIS Python example that the `DISInput.py` code base was built from. The code creates a standard user datagram protocol (UDP) socket on port 3000, listens for UDP packets, creates a protocol data unit (PDU) from a received packet, then parses out latitude, longitude, and altitude from a specific type of packet.

[1]Packet sniffing is the practice of collecting packets that pass through a computer network regardless of where the packet is addressed to. All of the sniffed packets, or a subset, are then analyzed for a variety of purposes such as monitoring or data extraction [36].

```python
#!python

__author__ = "mcgredo"
__date__ = "$Jun 25, 2015 12:10:26 PM$"

import socket
import time
import sys
import array

from opendis.dis7 import *
from opendis.RangeCoordinates import GPS
from opendis.PduFactory import createPdu

UDP_PORT = 3000

udpSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udpSocket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
udpSocket.bind(("", UDP_PORT))

print("Listening for DIS on UDP socket {}".format(UDP_PORT))

gps = GPS();

def recv():
    data = udpSocket.recv(4056) # buffer size in bytes
    pdu = createPdu(data);
    pduTypeName = pdu.__class__.__name__

    if pdu.pduType == 1: #PduTypeDecoders.EntityStatePdu:
        loc = (pdu.entityLocation.x, pdu.entityLocation.y, pdu.entityLocation.z)
        lla = gps.ecef2lla(loc)
        print("Received {}. Id: {}, Location: {} {} {}"\
            .format(pduTypeName, pdu.entityID.entityID, lla[0], lla[1], lla[2]))
    else:
        print("Received {}, {} bytes".format(pduTypeName, len(data)), flush=True)


while True:
    recv()
```

Figure 3.6. A tutorial example provided by Open-DIS as a receiver. This code formed the basis of the `DISInput.py` class [37].

Line 30 in Figure 3.6 specifies that only a PDU of type 1, representing an "Entity State," has its location printed out. There were only two items required for generating mission profiles as a proof of concept, units and waypoints. An enumeration of PDU types can be found

in the `open-dis-python/opendis/PduFactory.py` file, however, per the IEEE 1278.2-2015 standard, none represent a waypoint. To overcome this small hurdle, waypoints are input as any other unit, except they must have the "Waypoint" string in their name (name is referred to as "marking" in Open-DIS), as seen in Figure 3.8. By doing this, the code only cares about type 1 PDUs and can ignore all others. However, additional functionality could be added by adding in other PDU types in the `DISInput.py` class. After waiting a configurable number of seconds, defaulted to eight, `DISInput.py` returns all of the units and waypoints that were sniffed from the network. `InputCollector.py` then stores both for later use, as seen in Figure 3.7.

```
Open a DIS-friendly simulation (e.g. VR Forces) and input scenario parameters including waypoints and units. P
Listening for DIS on UDP port 3000
===== Waiting 8 seconds to collect DIS information. =====

Discovered Waypoint: "Waypoint 5" Id: 9, Location: 36.608599263885445 -121.89435091913558 1523.9999996777624
Discovered Waypoint: "Waypoint 4" Id: 7, Location: 36.5965809520834 -121.87603438790408 1536.5469153383747
Discovered Waypoint: "Waypoint 1" Id: 5, Location: 36.621604222503166 -121.81715669319516 1626.5606792727485
Ignoring PDU: PointObjectStatePdu, 1264 bytes
Ignoring PDU: ElectronicEmissionsPdu, 100 bytes
Discovered Waypoint: "Waypoint 3" Id: 8, Location: 36.587818727600556 -121.84746442302591 1578.304816050455
Ignoring PDU: NoneType, 226 bytes
Ignoring PDU: DataPdu, 192 bytes
Discovered Waypoint: "Waypoint 4" Id: 7, Location: 36.5965809520834 -121.87603438790408 1536.5469153383747
Discovered Waypoint: "Waypoint 1" Id: 5, Location: 36.621604222503166 -121.81715669319516 1626.5606792727485
Discovered Unit: "ScanEagle" Id: 1, Location: 36.609088924809825 -121.87610598775198 30.47999965120107
Discovered Waypoint: "Waypoint 5" Id: 9, Location: 36.608599263885445 -121.89435091913558 1523.9999996777624
Discovered Waypoint: "Waypoint 2" Id: 6, Location: 36.596640836485214 -121.8291796345506 1587.4566782470793
Ignoring PDU: PointObjectStatePdu, 1264 bytes
Ignoring PDU: ElectronicEmissionsPdu, 100 bytes
Ignoring PDU: NoneType, 226 bytes
===== Listening complete. =====

Units Discovered: {'ScanEagle': (36.609088924809825, -121.87610598775198, 30.47999965120107)}

Waypoints Discovered: {'Waypoint 5': (36.608599263885445, -121.89435091913558, 1523.9999996777624), 'Waypoint
36.5469153383747), 'Waypoint 1': (36.621604222503166, -121.81715669319516, 1626.5606792727485), 'Waypoint 3':
.304816050455), 'Waypoint 2': (36.596640836485214, -121.8291796345506, 1587.4566782470793)}
```

Figure 3.7. DISInput.py class waits 8 seconds to sniff DIS packet information from a running simulator such as VR-Forces. This run discovered five waypoints and one ScanEagle unit. All other discovered PDUs are ignored.

Figure 3.8. Using VR-Forces as input to simulate a scenario. Units are any entity, and waypoints are any entity with the "Waypoint" string contained in its name.

VR-Forces was selected as the simulation environment of choice for unit and waypoint input. VR-Forces provided familiarity in that the MOVES program at NPS has used it in instructional courses. Additionally, VR-Forces is prepackaged with `DISSpy` which allows a user to sniff DIS packets with no coding effort. Figure 3.8 also highlights the VR-Forces interface with built-in streaming maps and a much smaller graphical burden than competing products (no 3D rendering). Choosing a different input and output simulator also highlights how the mission generator framework need not be tied to a specific simulation environment. One sample use case for this framework involves generating missions in parallel with an existing simulation environment. This enables another computer on the network to take the computational load of generating potentially thousands of missions and streaming them to multiple simulation environments.

3.4.2 Historical Input

An Excel spreadsheet was the input method of choice because a convenient library for reading Excel files already existed. The `pandas` Python Data Analysis Library contained the `read_excel` method which reads in an Excel file and outputs a `pandas DataFrame` [38]. Table 3.2 represents an example file containing phases, events, and their relative probabilities. `ExcelInput.py` parses the list of phases from the "Phase" column and each event associated to their phase. The output data structure is a dictionary of phase names as keys and all the possible events and their properties as values. Events that require some calculation of sight lines (or "access" in STK terminology) also require a latitude X, longitude Y, and altitude Z. As an enemy location may move, this location will be fuzzed by a user determined amount in `WaypointFuzzer.py`, discussed in the Waypoint Fuzzing section. While the events in Table 3.2 are not particularly interesting, the framework is built to allow more significant event processing, which is discussed in the Future Work section.

Table 3.2. Notional historical data saved as `past_mission_data.xlsx` to draw phases and events from. Note that this is only example data to support the proof of concept. "Ending" marks whether the event will cause the scenario to end, unless some mitigation is incorporated. Locations are only input when access (sight) must be computed between the aircraft and some event, like Enemy anti-aircraft artillery (AAA).

Event	Phase	Probability	Ending	Loc X	Loc Y	Loc Z
Airfield	Launch	0.75	0			
Boat	Launch	0.25	0			
Hand	Launch	0.1	0			
Crash	Launch	0.05	1			
Link Loss	Ingress	0.2	0			
Enemy Discovery	Ingress	0.3	0			
HVT Appears	Actions	0.1	0			
Link Loss	Actions	0.1	0			
Retasking	Actions	0.3	0			
Enemy AAA	Egress	0.1	1	36.58	-121.92	42.08
Link Loss	Egress	0.2	0			
Enemy Discovery	Egress	0.3	0			
Airfield Landing	Recovery	0.1	0			
Hard Landing	Recovery	0.3	0			
Crash	Recovery	0.05	1			
Friendly Aircraft Delay	Recovery	0.1	0			
Link Loss	Recovery	0.05	0			

3.4.3 Metadata Input

The first iteration of this prototype used a separate class to collect the user input and metadata. This data included the campaign name, scenario names, what weights to give each scenario (e.g. 90% of missions generated from one scenario and 10% from another), and which DIS-collected units and waypoints to use for each phase. However as the code matured,

the extra class was removed and code relocated to `InputCollector.py`. The simple user prompts did not warrant a completely separate class for what amounted to a dozen lines of code. After collecting the user inputs, units and waypoints from `DISInput.py`, and phases and events from `ExcelInput.py`, `InputCollector.py` must now associate phases with waypoints. The user is prompted for a start and end waypoint for the first phase, then end waypoints for each next phase, until all phases are exhausted. There is no required number of phases or waypoints; a phase can have the same waypoint for its start and end. Generally, for `n` phases there will be `n+1` waypoints as shown in Figure 3.9.

Figure 3.9. Each phase has a starting and ending waypoint. Phase transitions occur once the next waypoint has been reached. Arrows and phase names added to represent example phases.

Once all the relevant information has been collected, `InputCollector.py` creates a campaign with every generated scenario in it. Each scenario will hold one input mission profile with the units, phases, waypoints, and events that are possible. `InputCollector.py` returns a compiled campaign for use by the `PrototypeGenerator.py` class. However, before a mission profile is generated, fuzzing and events need to be introduced.

3.4.4 Waypoint Fuzzing

While this exercise does not represent true "fuzz testing," or injecting invalid data to improve fault tolerance [39], the goal is similar. By injecting variability in "perfect" waypoints, the framework may be able to account for sensor inaccuracies, operator error, or the general fog of war. This fuzzing may expose configuration errors (e.g., sacrificing fuel for an extra radio) that result in mission failure or operation too close to performance envelopes. Given a particular waypoint as the mean, the user should be able to introduce a Gaussian distribution and draw random samples from both the latitude (X) and longitude (Y) values. To create the desired multivariate Gaussian distribution much like in Figure 3.10, the `numpy` library and `numpy.random.normal` function was used [40]. Latitude and longitude used the same standard deviation for a circular distribution, while altitude used a much smaller vertical standard deviation. An online tool was used to determine what default standard deviations were reasonable for the horizontal and vertical cases [41]. As it stands, a default horizontal standard deviation of 0.001 results in roughly 50-300m separation from the original waypoint. A default vertical standard deviation of 0.0001 equates to 1-10m difference in altitude. However, this is another situation where historical data would yield the most realistic variations.

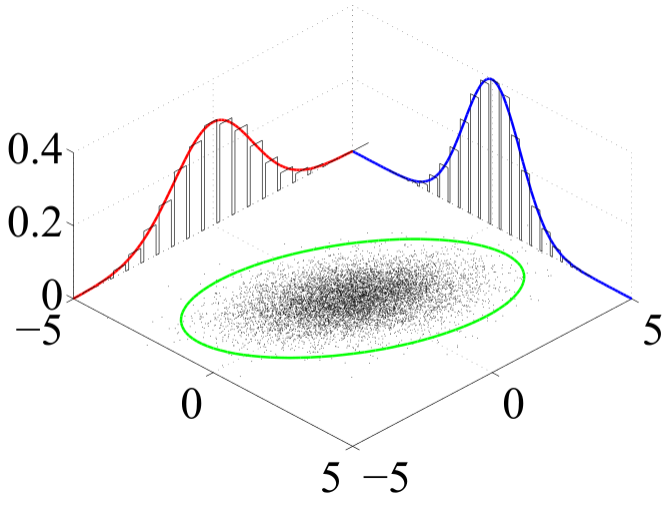

Figure 3.10. Example multivariate Gaussian distribution with mean of X and Y at 0 but different standard deviations. Source: [42].

3.4.5 Event Generation

After events have been loaded into their respective phases, a decision needs to be made whether that event will occur during the particular mission profile. Inside the `Scenario.py` class's `generate()` method, before a mission is instantiated, `generate_events()` is called through the `Phase.py` class. This method of the `Phase.py` class takes in another standard deviation for event waypoint generation (defaulted to the same horizontal deviation in the above waypoint fuzzing). `generate_events()` utilizes the SciPy library's `scipy.stats.bernoulli` function to generate a Bernoulli discrete random variable [43]. This random variable determines whether the event will occur (returning 1 or 0), based upon the probability loaded from `ExcelInput.py`. If the event also contains a location, `generate_events()` will fuzz that waypoint using `WaypointFuzzer.py` to ensure some variability. The events that occur in a phase are returned to `Scenario.py` and added to the mission string.

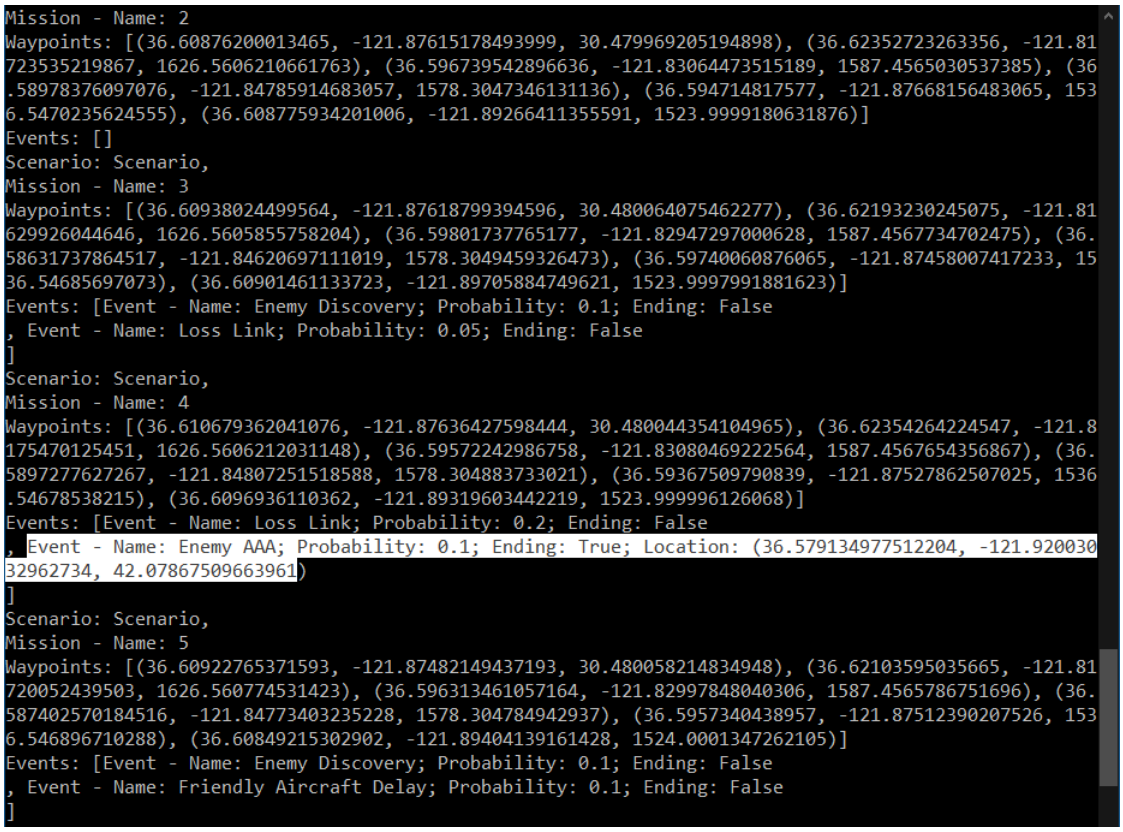

Figure 3.11. A random event occurs with probability loaded from Excel. If the event occurs, the historical waypoint is fuzzed and added to the phase's event list. Here a particular mission profile generates an Enemy AAA event that can affect the mission's outcome.

3.5 Profile Generation

With the completed campaign and scenarios, the next goal is to generate the set of mission strings, or intermediate output, to send to the chosen simulation environment. The `Campaign.py` class contains a `generate()` method that creates a set of mission profiles for each scenario in the campaign. The `Scenario.py` class contains all the phases, with their respective waypoints, events, and units, along with its own `generate()` method. Given a number of `mission_iterations` and a standard deviation for fuzzing waypoints the `generate()` method will:

1. Create and increment the mission number (name)
2. For each phase
   (a) If first phase, fuzz the start waypoint and add it the waypoint list
   (b) Fuzz the ending waypoint and add it to the waypoint list
   (c) Generate events for this phase
3. Create a mission with the name, waypoints, events, and scenario it belongs to

After a single mission profile is generated it is added to the scenario's list of mission profiles in `Scenario.py`. After a single scenario completes generating its missions, it is added to the campaign's list of mission profiles. The entire list will have unique mission numbers to allow users to highlight specific missions of interest.

3.6 Output

The output from the generation process is a list of missions, where each element is an instance of a `Mission.py` class. Each element contains the mission number, fuzzed waypoints, generated events, and the name of the scenario it belongs to. Figure 3.12 displays the mission strings that will be translated into simulator specific input. Each represents a "run" of a scenario for an aircraft (or any other unit) to simulate. The completed data structure of missions is returned by `Campaign.generate()` to `PrototypeGenerator.py`.

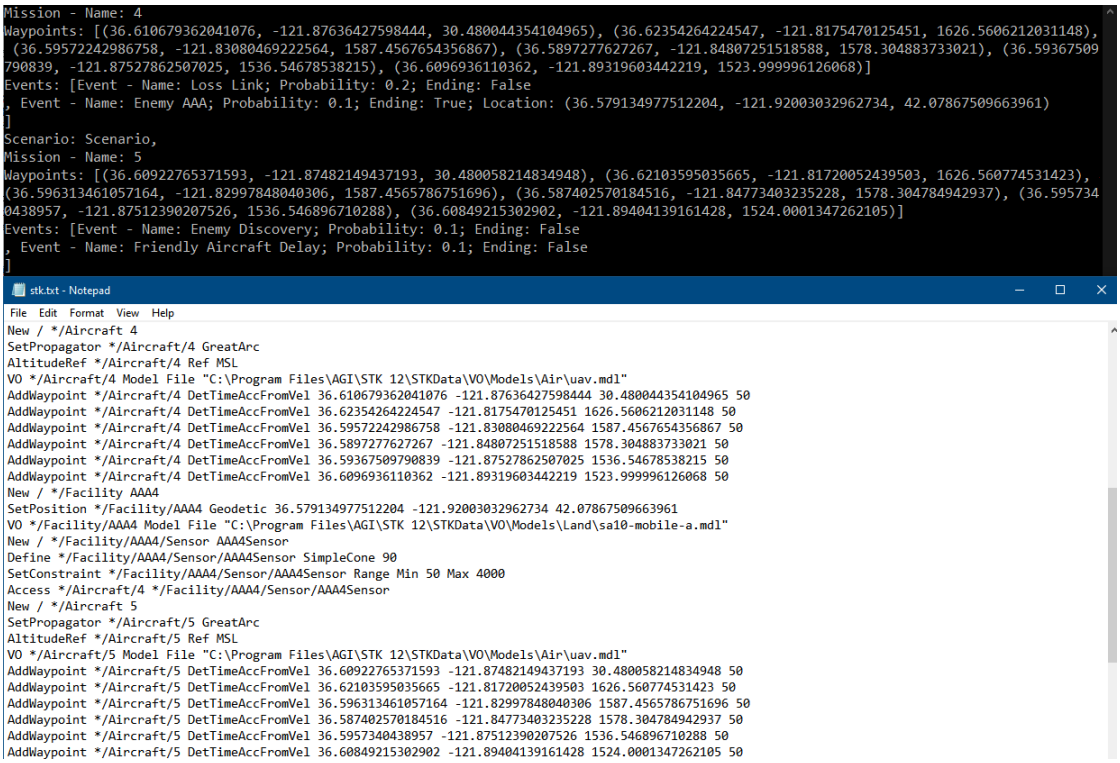

Figure 3.12. A set of mission strings (top half) used as the intermediate output of the prototype mission profile generator. These strings will be converted into simulator specific commands (bottom half).

3.6.1 Mapping to STK Connect

In order to simulate the generated mission strings in STK, this thesis made use of the STK Connect library. The library contains every function available through the GUI or STK Object model along with ample documentation on how to use each command. For example, creating a new entity is performed through the command: `New <ApplicationPath> <ClassPath> <NewObjectName> {NewOptions}`. Creating a new UAS to simulate is done with four commands: 1) creating a new aircraft entity (Aircraft 4 in the example below), 2) setting a `GreatArc` propagator (how the entity moves in the simulated world, GreatArc is used for aircraft, vehicles, and ships), 3) setting the altitude reference to mean

sea level (MSL), and 4) setting which graphical model is used by the aircraft [28].

```
New / */Aircraft 4
SetPropagator */Aircraft/4 GreatArc
AltitudeRef */Aircraft/4 Ref MSL
VO */Aircraft/4 Model File "C:\...\STK 12\STKData\VO\Models\Air\uav.mdl"
```

The next step is to add the waypoints from the mission string. The `AddWaypoint` command adds individual waypoints to a specific entity and whether the time is dependent on velocity, velocity and acceleration, or time: `AddWaypoint <VehicleObjectPath> {AddMethod} <Parameters> [<TurnRadius>]`

```
AddWaypoint */Aircraft/4 DetTimeAccFromVel 36.6 -121.9 30.5 50
```

After the waypoints are sent to the specific entity, access also needs to be calculated between AAA and the aircraft entity. This is modeled through a `Facility` object in STK. The below code will:

1. Instantiate a new Facility called AAA4
2. Set the position of the facility in geodetic coordinate space
3. Indicate which model to use in the simulation
4. Create a new sensor at the facility
5. Define the sensor to be a cone with a 90 viewing degree angle (hemisphere)
6. Set the sensor to have a minimum range of 50 meters and maximum range of 4000 meters
7. Calculate the access between the aircraft and the sensor

```
New / */Facility AAA4
SetPosition */Facility/AAA4 Geodetic 36.6 -121.9 42.1
VO */Facility/AAA4 Model File "C:\...\VO\Models\Land\sa10-mobile-a.mdl"
New / */Facility/AAA4/Sensor AAA4Sensor
Define */Facility/AAA4/Sensor/AAA4Sensor SimpleCone 90
SetConstraint */Facility/AAA4/Sensor/AAA4Sensor Range Min 50 Max 4000
Access */Aircraft/4 */Facility/AAA4/Sensor/AAA4Sensor
```

The above example represents the minimum code required in order to instantiate a single mission profile with a AAA threat. Once the code is input into STK, the user will have access to the full 2D and 3D interfaces, access interval computations and graphs, and the ability to modify any aspect of the simulation [28]. After generating the mission strings in `PrototypeGenerator.py`, the `STKConnectGenerator.py` class was used to generate the STK Connect strings. The `add_aircraft()`, `add_multiple_waypoints()`, `add_facility()`, and `compute_access()` methods were called to generate the relevant Connect code given the mission string as input. If another simulation environment was used e.g., VBS4 the parsing of the mission strings would occur in the same place in the code.

3.7 Simulation

As discussed in the implicit requirements, STK was primarily chosen due to integration and licensing issues with other simulators. Integrating with third-party code is considered a "professional" feature that usually costs more or is not available in a demo license. This was also true with STK as communication with Python required the STK Integration package [44]. STK Connect (to send strings over transmission control protocol/internet protocol (TCP/IP)) also required the STK Professional license to communicate with third-party applications. However, STK allowed for input of STK Connect commands directly into the STK environment. STK Connect strings could be generated in Python and pasted into STK without paying for a license fee. However, if the correct licenses were acquired there is little required effort in sending STK Connect strings over TCP/IP [28]. Instead of saving the Connect strings to a file in `PrototypeGenerator.py`, an integration license would allow for Python code to send all the Connect commands at once.

CHAPTER 4:
Results and Analysis

4.1 Results

4.1.1 Notional Mission

A notional scenario was created to establish that the code produces reasonable results. A commander aboard a boat in Monterey Bay wishes to conduct an area reconnaissance in support of seizing Objective Airfield, as seen in Figure 4.1. The commander consults an ISE terminal to determine if a ScanEagle UAS is capable of conducting the reconnaissance. Intelligence assesses that there are SA-7 (surface-to-air) man-portable air-defense systems (MANPADS) in the vicinity of `10S EF950510`, marked on the map, which have a maximum effective range of about four kilometers [45]. The commander also has access to historical data from similar missions as seen in Table 4.1. This is simplistic data meant to verify the results of 100 mission profiles simulated in STK. Given the notional accuracy of the intelligence reporting and ScanEagle sensors (set to defaults as stated in Chapter 3), the commander wants to know how often the UAS is exposed to the AAA threat and for how long each exposure lasts. The framework should allow the commander to examine the results and make an informed decision whether s/he can use ScanEagle for the actual mission.

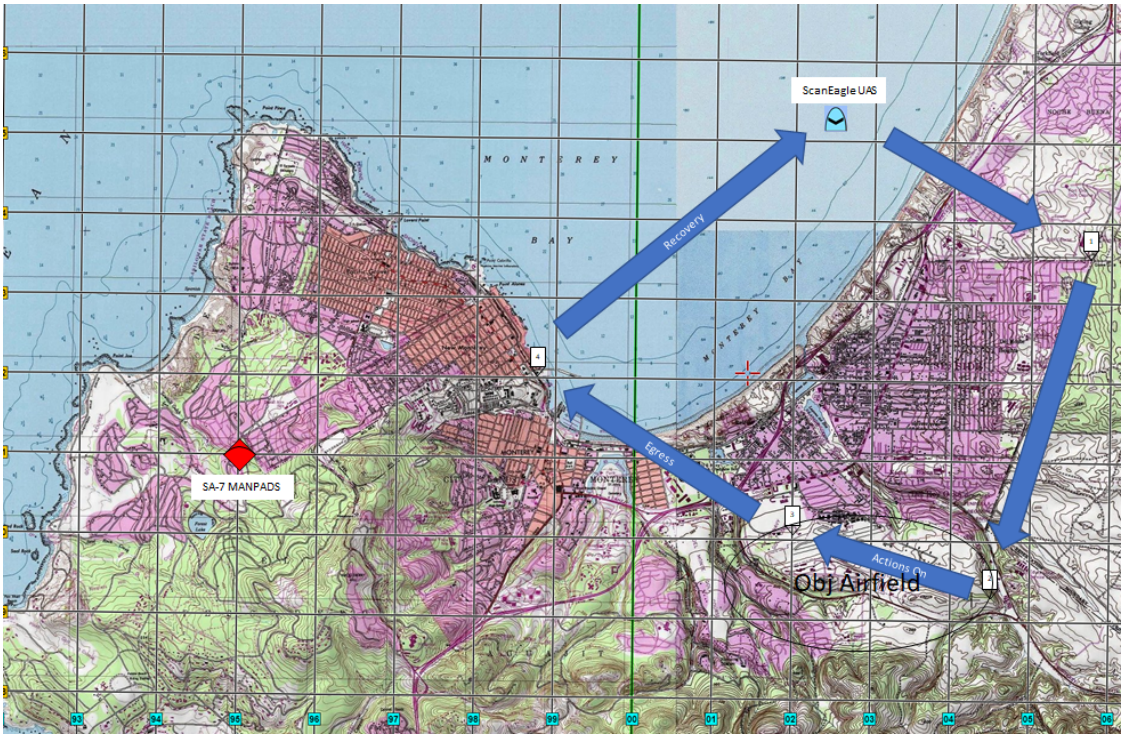

Figure 4.1. Unit positions, waypoints, and a templated enemy threat for a notional area reconnaissance mission of Objective Airfield. Adapted from [46].

Table 4.1. Data used to verify that the results from 100 mission profiles reflects input probabilities. Each phase only has one or two events with an easily calculated probability. Note the probability of a "Boat" launch is 1 to reflect the scenario and Enemy AAA is 0.5.

Event	Phase	Probability	Ending	Loc X	Loc Y	Loc Z
Boat Launch	Launch	1	0			
Link Loss	Ingress	0.1	0			
Retasking	Actions	0.3	0			
Enemy AAA	Egress	0.1	1	36.6	-121.936	42.08
Crash	Recovery	0.05	1			
Boat Capture	Recovery	0.8	0			

4.1.2 Mission Profiles

Given the scenario and inputs described, the set of mission strings generated should reflect the initial probabilities. Of the 100 mission profiles generated, Table 4.2 shows the number of events that occurred. The only outlier is the Link Loss event, which had 16 events instead of the expected 10. Unsurprisingly, with more mission profiles generated, the event counts will more closely reflect their initial probabilities. As discussed in Chapter 3, the waypoints of both the ScanEagle and the AAA threat have been fuzzed in each mission profile as seen in Figure 4.2. The final distribution of waypoints can be seen in Figure 4.4; the cluster of points inside the teal sphere and the points at the vertices of the "rainbow" polygon represent the varied AAA and waypoint locations respectively.

Table 4.2. After generating 100 mission profiles, the event occurrences in mission strings reflected their initial probabilities as entered in Table 4.1.

Event	Initial Probability	Event Count	Count/100
Boat Launch	1	100	1
Link Loss	0.1	16	0.16
Retasking	0.2	22	0.22
Enemy AAA	0.5	52	0.52
Crash	0.05	5	0.05
Boat Capture	0.8	78	0.78

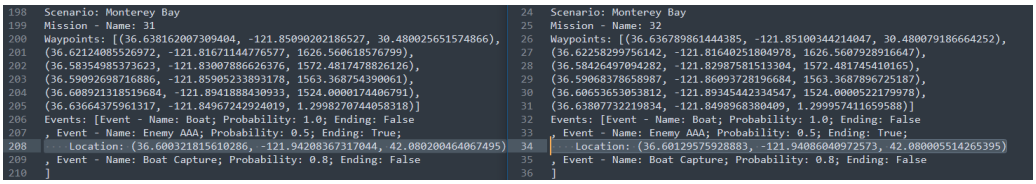

Figure 4.2. Output mission strings of two successive missions. Each waypoint is unique and fuzzed within roughly 100 meters (in the latitude and longitude) of the original waypoint.

The output of 100 generated mission profiles can be seen in Figure 4.3. Each aircraft is labeled with its respective mission number and each AAA event labeled with the mission number it was generated for (e.g., AAA5 is generated for and only interacts with ScanEagle 5). Each AAA event has a resultant minimum and maximum effective range sphere, however, only the nearest is shown for clarity in Figure 4.3. Figure 4.4 shows the full mission profiles of all 100 missions. Figure 4.4 also highlights the distribution of the waypoints, AAA events, and aircraft with a 0.001 standard deviation in latitude and longitude. However, this output does not provide much value outside of a three-dimensional map study. The value may come from STK's ability to compute visibility ("access" in STK terminology) between a mission's AAA event and its aircraft.

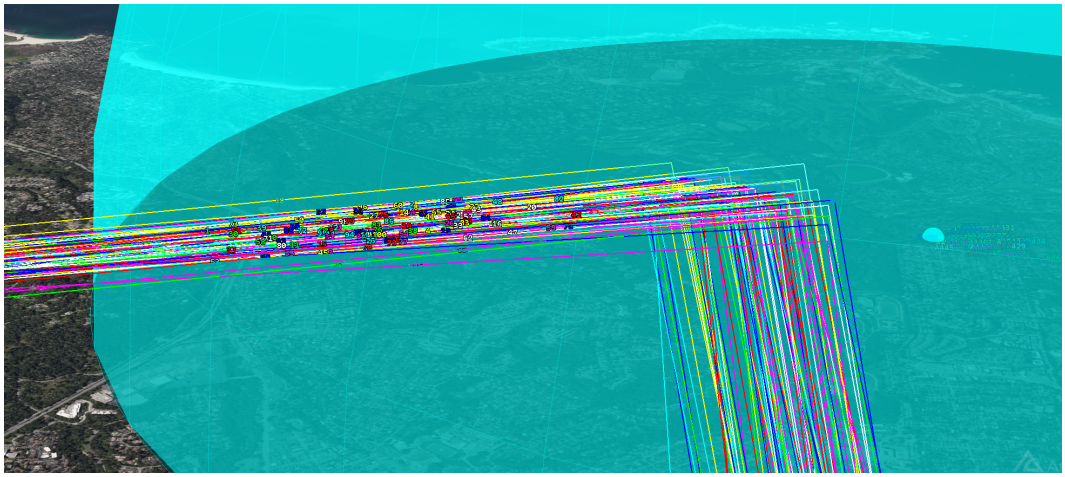

Figure 4.3. 100 mission profiles simulated simultaneously in STK. Each line represents a single mission; each number a ScanEagle with its respective mission number. In the far right, each AAA<number> represents a single AAA event generated for that numbered mission profile. Only one "threat ring" for the closest AAA event is highlighted, however, each has a minimum and maximum effective range.

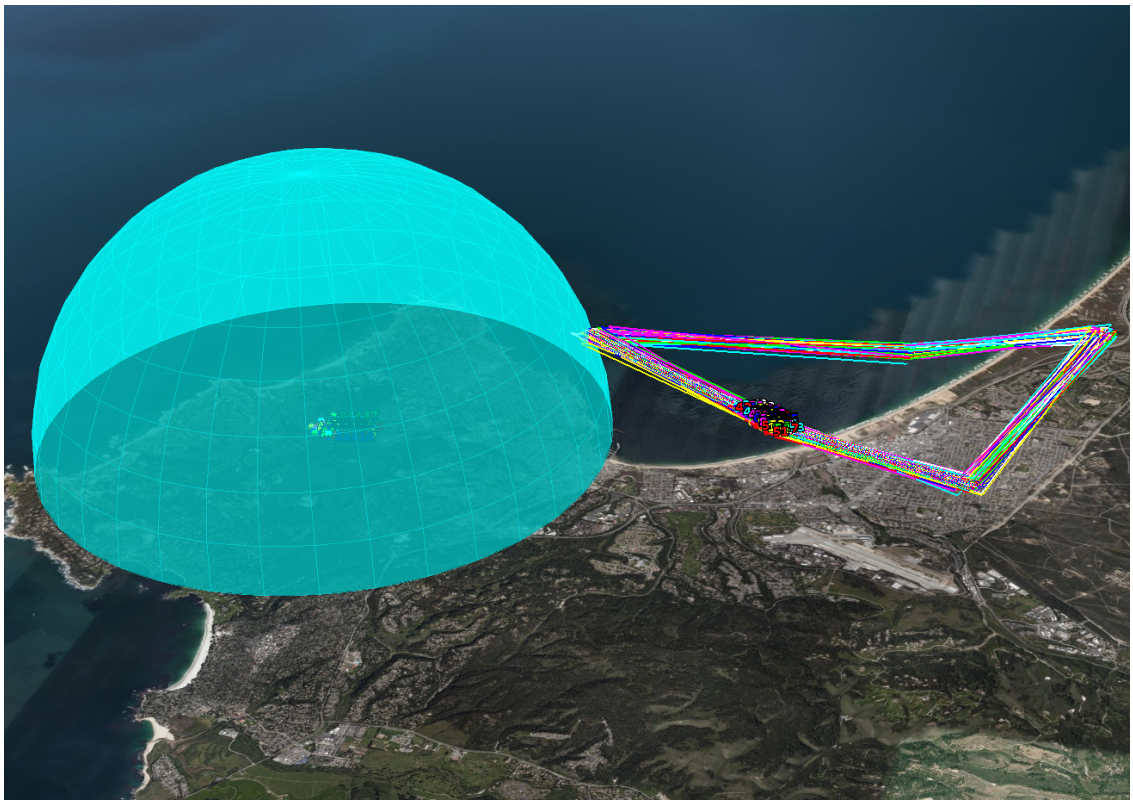

Figure 4.4. The full output of 100 mission profiles. Only one AAA threat ring is highlighted for clarity. The location distribution of waypoints and events generated is clearly seen.

4.1.3 Computing Access

Of the 52 Enemy AAA events, most aircraft were outside of the AAA threat's maximum range. Only nine resulted in access between the AAA and the ScanEagle and most were only for a few seconds. Figure 4.5 shows where, how long, and when the AAA could see the ScanEagle during mission profile 19. The ScanEagle was "visible" to the AAA threat for a total of 5.4 seconds at a range of roughly 4000 meters. The longest access occurred during mission 85, where the AAA had almost 19 seconds to acquire and shoot down the ScanEagle.

Start: 26 Jul 2021 19:00:00.000 UTCG
Stop: 26 Jul 2021 19:06:35.412 UTCG
Step: Using object's default time points | Show Step Value

18 Aug 2021 20:03:19

Facility-AAA19-Sensor-AAA19Sensor-To-Aircraft-19: Inview Azimuth, Elevation, & Range

AAA19Sensor-To-19 - AER reported in the object's default AER frame

	Time (UTCG)	Azimuth (deg)	Elevation (deg)	Range (km)
	26 Jul 2021 19:04:47.189	76.876	21.222	3.999863
	26 Jul 2021 19:04:49.869	75.412	21.709	3.910538
	26 Jul 2021 19:04:52.549	74.579	20.625	3.999874
Global Statistics				
Min Elevation	26 Jul 2021 19:04:52.549	74.579	20.625	3.999874
Max Elevation	26 Jul 2021 19:04:49.957	75.362	21.725	3.907659
Mean Elevation			21.186	
Min Range	26 Jul 2021 19:04:49.957	75.362	21.725	3.907644
Max Range	26 Jul 2021 19:04:52.549	74.579	20.625	3.999874
Mean Range				3.970092

18 Aug 2021 20:14:02

Facility-AAA19-Sensor-AAA19Sensor-To-Aircraft-19: Access Summary Report

AAA19Sensor-To-19

	Access	Start Time (UTCG)	Stop Time (UTCG)	Duration (sec)
	1	26 Jul 2021 19:04:47.189	26 Jul 2021 19:04:52.549	5.359
Global Statistics				
Min Duration	1	26 Jul 2021 19:04:47.189	26 Jul 2021 19:04:52.549	5.359
Max Duration	1	26 Jul 2021 19:04:47.189	26 Jul 2021 19:04:52.549	5.359
Mean Duration				5.359
Total Duration				5.359

Figure 4.5. Computed access between mission 19's ScanEagle and AAA event. The access lasted only 5.4 seconds at a range of 4000m.

4.1.4 Single Mission Breakdown

The long access duration of mission 85 warranted a closer look as to what caused the increased risk in this mission profile. As seen in Figure 4.6, neither the mission profile or enemy AAA threat deviates drastically from the average locations. Because of the relatively slow aircraft speed, the small venture into the AAA weapon engagement zone (WEZ) was still significant enough for a long exposure. The ground commander responsible for the ScanEagle asset could decide that this 19 second exposure worst-case scenario is reasonable enough considering the intelligence gathered. Alternatively, the ground commander could

decide that this exposure is too great and modify the mission profile, possibly increasing the speed of the aircraft in the vicinity of Waypoint 4, to reduce the chance of an exposure. This analysis might also serve as a suggestion of go/no-go criteria for the actual mission; if the AAA threat is identified too far east or too far north of a specific point, the commander could issue a return to base command to the UAS. Summarizing the locations of the mission profiles with significant exposures would establish the no-go criteria to prevent damage to the aircraft.

Figure 4.6. Mission number 85 resulted in a 19 second access duration between the AAA and the ScanEagle, despite no drastic change to either's position.

4.2 Analysis

Considering the presented scenario, a commander might assess the use of a ScanEagle as low risk, with a reasonable chance of mission success. Of the 100 missions, only 9 entered the WEZ of the AAA threat, and most exposures were for a few seconds. Additionally, all of the paths took by the ScanEagle entered the WEZ at near the maximum effective range of the SA-7 MANPADS threat. While this analysis is simplistic and compares to a simple map study by the commander, the simulation can yield much greater precision and accuracy. For example, drawing a 4000 meter threat ring around the SA-7 threat in

4.1 would suggest that the aircraft's path is safely outside of the WEZ. However, a drawn threat ring does not account for elevation, terrain mitigation, sensor inaccuracies, or the mobility of the threat. Small errors in the aircraft's position along with roughly 100 meters of movement in the threat results in aircraft inside the WEZ. With accurate sensor data and a reasonable intelligence assessment on how much the threat can actually move, the simulator can produce more accurate results. The improved accuracy over a drawn threat ring also allows a commander to operate closer to the margins of safety without compromising the mission. As stated previously, these results only serve to validate the framework; future work that can be built off of this framework will be discussed in Conclusions.

4.2.1 Simulation Limitations

While the events currently have little use if no location is given, adding the functionality of STK Aviator, which will be discussed in Future Work, may allow the simulation of communication links, aerodynamics, fuel, and performance characteristics. For example, if the "Link Loss" occurs, it may trigger a return to base functionality which can impact mission success, fuel requirements, or trigger additional events. Unfortunately the Aviator module proved too cost prohibitive for this thesis, but the future addition is highly recommended. This thesis shows that working outside of and interfacing with commercial simulation engines is possible; however, this type of simulation will perform better natively inside a single simulation engine. Access to the underlying engine would allow the inclusion of many more features, elimination of redundancy between mission profiles, and increased efficiency in generating individual missions.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusions and Future Work

5.1 Conclusions

This framework provides one step towards what the ISE of the future may look like. By generating mission profiles, a non-technical operator is offered a basis of comparison between two competing vehicles or configurations beyond KPPs. While this thesis does not attempt to create actual comparisons using mission profiles, it does provide the framework to make those future comparisons. The original goal was to provide a proof-of-concept and basic framework to build from. The value from this thesis will come from future work and iterations. However, building the framework also exposed other use cases beyond the original scope.

5.1.1 Interactive Simulation Environment

The most obvious use case is what this thesis was designed for: generating mission profiles in support of ISE. To that end, a set of profiles was generated and access was computed between randomly generated events and an aircraft. If two competing aircraft are given the same large set of mission profiles, a simulation environment that accurately simulates physics and configuration, and the results of each mission can be compared between aircraft, then this may provide an MVP for ISE. The most challenging part of that equation is the accurate simulation environment. However, STK provides products that may facilitate both the simulation and the comparison. Discussed in Future Work, STK Aviator and Professional products may be the next step in a full ISE proof-of-concept. Acquiring and inputting actual historical data, whether it be mishap information or successful mission information, may also bring validity to this mission generation approach. Contributing historically accurate horizontal and vertical standard deviations to programmed waypoints can also increase the fidelity of this model.

5.1.2 Additional Use Cases

One potentially interesting use of generated mission profiles is the ability to perform distributed, large-scale simulations with multiple environments. Using one environment, such as VBS4, to simulate a single scenario will produce one outcome on one computer. However, generating mission profiles and sending them to multiple instances of VBS4 across a large number of computers will allow for parallel completion of multiple scenarios. This framework could be used as an intermediate step between one scenario predicting multiple outcomes across many computers. Interpreting the output of a large number of simulations will be discussed in Future Work, which may also be of use to facilitate this use case. Similarly, given a set of mission profiles, acquisitions professionals or contractors could evaluate the capability of different simulators by translating the exact same scenarios to each environment. The federated output of all the simulation environments might expose additional information that may not otherwise be accessible. This work can also form the basis for a federated simulation environment that decides which simulators provide the most accurate results. A program could send mission profiles to multiple simulators, compare the results, remove outliers, and average out alike results for a "better" result. Alternatively, a human could choose the most promising results generating more trust in that simulator's output for future work.

5.2 Future Work

5.2.1 Evaluation With Historical Data

It was clear early in the creation of this thesis that acquiring UAS historical data would be an effort on its own. Negotiating what might be proprietary or protected information did not fit in the timeline of also developing mission profiles. However, if generating profiles is paired with accurate historical information, or the input method of the historical information is developed to exactly fit the output of UAS ground control stations, the mission profiles may become much more useful to ISE or industry. An alternative strategy is to attach the idea of mission profile generation to a new generation of UAS currently in development. Playback of a real mission from a ground control station may not have a common format between UAS platforms. A desire to generate duplicate mission profiles for training, ISE, or reinforce simulation may provide an incentive to produce common output formats.

5.2.2 Events

Events in this framework are little more than a notification or an access computation between two objects. If the event does not have a location, it provides no actual input to the simulation. The first step to bring "Ending" events into usefulness is to also add "Mitigation" properties to the UAS. If the UAS experiences an ending event e.g., Enemy AAA without a mitigation e.g., electronic jamming or flares, then the mission ends and is considered a failure. Paired with an accurate event probability, a decision could be made about whether it is worth carrying electronic warfare (EW) payloads. Another example involves a chance of break-apart on landing. If the probability of break-apart is high due to terrain or environment, a navigational sensor's variance is incorporated into the waypoint fuzzing's standard deviation, and there is a small landing zone to negate break-apart on landing, a user could estimate how many potential landings would result in break-apart. A user could also increase the size of the landing zone until the risk of break-apart is negated. Bringing this type of functionality to the framework would require a negligible coding effort.

A similarly low coding-effort to produce useful results involves creating events for every type of enemy AAA. If a particular AO has specific AAA threats along with previous and predicted positions, the framework can take a proposed route and determine if there is a significant danger to aircraft. With mission profiles, the user can also determine whether any deviations from the proposed route would cross into AAA weapon engagement zones. If the number of mission profiles that cross into AAA danger zones is high, based upon previous mission data, then pilots might opt for a specific type of mitigation, dependent on the threats faced. Data from these mission profiles can also be sent to AI or ML solutions that would suggest specific payloads or configurations to mitigate risk.

5.2.3 Extension Beyond DIS Entities

The current framework only supports the "Entity" PDU. Adding in other PDUs may also enhance the quality and usefulness of mission profiles. The "Collision" PDU can be utilized to determine if multiple autonomous aircraft operating within the accuracy of their sensors have a high probability of collision. Introducing an accurate standard deviation to the waypoint fuzzing algorithm based on sensor accuracy, running thousands of mission profiles on both aircraft, and determining if any collisions occur could provide some value to

prevent collisions in dense UAS swarms. For example, if 600/1000 simulations have a collision, then the UAS may require sensors with a higher degree of accuracy. Other useful PDUs include the "Transmitter" and "Receiver," which could determine if a particular transmitter is strong enough for relay capabilities. If 1000/1000 similar missions show constant signal between the transmitter and receiver, then that particular transmitter and receiver combination should be sufficient. Implementation of other PDU types only requires modification of the `DISInput.py` class.

5.2.4 Interpreting Output

To bring more usefulness out of generating mission profiles, a method of aggregating the large amount of output is required. The simplest determination of success is whether the UAS reached the final waypoint. If two UAS are simulated on the same set of 1000 missions, and one reached the last waypoint 500 times, while the other only 100 times, then one might conclude that the first is better than the second. This is a simplistic example but might provide a starting point for more thorough examination of what defines "mission success" in each mission profile. Interpreting output will be an essential part of the ISE project as mission success or failure will provide the constructive or destructive feedback into the system's choices.

5.2.5 STK Aviator

Acquiring a license for STK Aviator will increase the usefulness of this work. Aviator provides many high fidelity aircraft models and physics simulation [27]. Paring wind, atmospheric effects, gravity, and maneuver simulation with mission profiles results in more events to simulate, accuracy in the model, and relevancy to a particular aircraft. A user could generate a thousand mission profiles, select an existing model from Aviator, and import a new model in development, then determine whether the new model outperforms an existing model. If not, the differences between individual mission success and failure may explain the performance differences. The cost of the license may be offset by the usefulness this could provide to an acquisitions professional.

5.2.6 Machine Learning

A very long-term goal incorporates ML to make decisions to improve an aircraft. If a number of profiles are generated and a model's fitness is determined by the number of successful missions, an ML model could design the "perfect" aircraft to maximize success. The model needs to change the aircraft by swapping out engines, wing types, payloads, or configurations, but after enough time, the aircraft will have explored all the possible configurations to maximize success. With more variation in scenarios and missions, the user could create the best general UAS, or with less variation, configure a UAS for a specific mission. These scenarios represent an ideal endstate for ISE; a non-technical ground commander can input a desired mission and an AI can configure the best UAS for mission success. Ultimately, the usefulness of generating mission profiles will be determined by its future work.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Software Libraries

A.1 Libraries

pandas

https://pandas.pydata.org/

Used for reading in from Excel and creating DataFrames to support result.

OpenPyXL

https://openpyxl.readthedocs.io/en/stable/

Used by pandas to read Excel files.

NumPy

https://numpy.org/doc/stable/index.html

Create distributions to sample random variables from. Used in waypoint fuzzing.

Scapy

https://scapy.net/

Used for packet investigation and manipulation for DIS.

Open-DIS

https://github.com/open-dis/open-dis-python

Filtered and parsed DIS packets for initial input information.

A.2 Environments

Anaconda

https://www.anaconda.com/

Allowed for easy package management and environment setup through command line.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Source Code

Note that code is posted here with no guarantee of validation or verification. While this code may serve as a proof-of-concept, significant work is required for use in any sort of larger program. Source code can be downloaded from: https://gitlab.nps.edu/ryan.sohm/sohm_thesis

B.1 Campaign.py

```
import Scenario as scenario
import numpy as np
from scipy.stats import bernoulli

class Campaign:
        """
        Description:
        Contains all scenarios and their metadata. generate()
            function returns a full list of mission strings.
        """
    def __init__(self, name = "Campaign", scenarios = [],
        scenario_weights = []):

        self.name = name
        self.scenarios = scenarios
        if (len(scenario_weights) == 0):
            for _ in range(len(scenarios)):
                scenario_weights.append(1)
        if (len(scenario_weights) != len(scenarios)):
            raise Exception("Distribution␣parameter␣must␣match␣
                number␣of␣scenarios.")
        self.scenario_weights = scenario_weights
        self.next_mission_number = 1

    def add_scenario(self, scenario, weight = 1):
        self.scenarios.append(scenario)
        if (weight > 0):
```

```
        self.scenario_weights.append(weight)

def __str__(self):
    return "Campaign - Name: " + self.name + " ==========\n" +
       str(self.scenarios) + "\nScenario Weights: " + str(self.
       scenario_weights)

def __repr__(self):
    return self.__str__()

def calculate_weights(self, weights = [], iterations = 1):
    # Calculate the total weights
    total_weight = 0
    for weight in weights:
        total_weight += weight

    # Calculate the cutoffs for each scenario
    # Not yet implemented

    return cutoffs, np.random.uniform(0,total_weight,iterations)

def generate(self, scenario_iterations = 1, fuzzing_std_dev =
   0.001):
    # Calculate the weight distributions
    #distribution = self.calculate_weights(self.scenario_weights
       , scenario_iterations)
    # add in distrubiton for weights, not yet implemented
    #for _ in distribution:

    # bernoulli distribution https://www.askpython.com/python/
       examples/probability-distributions
    # plug into scenarios[x] return from bernoulli rounded
    #which = bernoulli.rvs(size = 1, p = )

    # Generate missions for each scenario
    missions = []
    for scen in self.scenarios:
        new_missions, last_no = scen.generate(
           scenario_iterations,fuzzing_std_dev,self.
           next_mission_number)
```

```
            missions.extend(new_missions)
            self.next_mission_number = last_no
        return missions



    def test(self):
        """ Test method for Campaign """
        print("Testing Campaign class")
        print("Campaign Name:",self.name)
        print("Campaign Scenarios:",self.scenarios)
        print("Scenario Weights:",self.scenario_weights)



if __name__ == "__main__":
    s1 = scenario.Scenario()
    s2 = scenario.Scenario()

    scenarios = [s1,s2]

    c = Campaign("camptest",scenarios,[1,2])
    c.test()
```

B.2 DISInput.py

```
import socket
import datetime
from opendis.dis7 import *
from opendis.RangeCoordinates import GPS
from opendis.PduFactory import createPdu
from opendis.PduFactory import PduTypeDecoders


ENTITY_PDU_TYPE = 1

class DISInput:
    """
    Description:
    Gathers DIS input by listening for DIS-compatible simulators.
       collect() method populates .units and .waypoints
    """
    def __init__(self, udp_port = 3000, buffer_size = 2048,
       wait_time = 8):
        self.udp_port = udp_port
        self.buffer_size = buffer_size
        self.udp_socket = -1
        self.units = {}
        self.waypoints = {}
        self.wait_time = wait_time

    def collect(self):
        # Setup DIS listener
        self.udp_socket = socket.socket(socket.AF_INET, socket.
           SOCK_DGRAM)
        self.udp_socket.settimeout(self.wait_time)
        self.udp_socket.setsockopt(socket.SOL_SOCKET, socket.
           SO_BROADCAST, 1)
        self.udp_socket.bind(("", self.udp_port))

        print("Listening␣for␣DIS␣on␣UDP␣port",self.udp_port)
        self.gps = GPS()

        start = datetime.datetime.now()
        end = start + datetime.timedelta(0,self.wait_time)
```

```
        print("=====␣Waiting␣{}␣seconds␣to␣collect␣DIS␣information.␣
            =====\n".format(int(self.wait_time)))
        while start < end:
            self.recv()
            start = datetime.datetime.now()
        print("=====␣Listening␣complete.␣=====\n")
        print("Units␣Discovered:",self.units,"\n")
        print("Waypoints␣Discovered:",self.waypoints)


    def recv(self):
        data = self.udp_socket.recv(self.buffer_size)
        pdu = createPdu(data)
        pduTypeName = pdu.__class__.__name__

        # Handle Entity State PDU
        if ((pduTypeName != "NoneType") and (pdu.pduType ==
            ENTITY_PDU_TYPE)):
            location = (pdu.entityLocation.x, pdu.entityLocation.y,
                pdu.entityLocation.z)
            lla = self.gps.ecef2lla(location)
            #print("Received {}. Id: {}, Location: {} {} {}".format(
                pduTypeName, pdu.entityID.entityID, lla[0], lla[1],
                lla[2]))

            # Parse out the user marking
            marking = pdu.marking.charactersString()

            if ("Waypoint" in marking):
                self.waypoints[marking] = lla
                print("Discovered␣Waypoint:␣\"{}\"␣Id:␣{},␣Location:
                    ␣{}␣{}␣{}".format(marking, pdu.entityID.entityID,
                     lla[0], lla[1], lla[2]))
            else:
                self.units[marking] = lla
                print("Discovered␣Unit:␣\"{}\"␣Id:␣{},␣Location:␣{}␣
                    {}␣{}".format(marking, pdu.entityID.entityID, lla
                    [0], lla[1], lla[2]))

        else:
```

```
            print("Ignoring␣PDU:␣{},␣{}␣bytes".format(pduTypeName,
                len(data)), flush=True)

    def test(self):
        """ Test method for DISInput """
        self.collect()
        return

if __name__ == "__main__":
    dis = DISInput()
    dis.test()


'''
Original code from https://github.com/open-dis/open-dis-python/tree/
   master/examples
#!python

__author__ = "mcgredo"
__date__ = "$Jun 25, 2015 12:10:26 PM$"

import socket
import time
import sys
import array

from opendis.dis7 import *
from opendis.RangeCoordinates import GPS
from opendis.PduFactory import createPdu

UDP_PORT = 3000

udpSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udpSocket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
udpSocket.bind(("", UDP_PORT))

print("Listening for DIS on UDP socket {}".format(UDP_PORT))

gps = GPS();
```

```
def recv():
    data = udpSocket.recv(4056) # buffer size in bytes
    pdu = createPdu(data);
    pduTypeName = pdu.__class__.__name__

    if pdu.pduType == 1: #PduTypeDecoders.EntityStatePdu:
        loc = (pdu.entityLocation.x, pdu.entityLocation.y, pdu.
            entityLocation.z)
        lla = gps.ecef2lla(loc)
        print("Received {}. Id: {}, Location: {} {} {}".format(
            pduTypeName, pdu.entityID.entityID, lla[0], lla[1], lla
            [2]))
    else:
        print("Received {}, {} bytes".format(pduTypeName, len(data))
            , flush=True)


while True:
    recv()
'''
```

B.3 Event.py

```
import math

class Event:
    """
    Description:
    Describes a single event, its probability, and where it may
       occur.
    """

    def __init__(self, name = "Event", probability = 1, ending =
       False, locx = math.nan, locy = math.nan, locz = math.nan):
        self.name = name
        if ((probability >= 0) and (probability <= 1)):
            self.probability = probability
        else:
            raise Exception("Probability␣parameter␣must␣be␣between␣0
               ␣and␣1␣inclusive.")
        self.ending = ending
        self._location = None
        if (not math.isnan(locx)) and (not math.isnan(locy)) and (
           not math.isnan(locz)):
            self._location = (locx,locy,locz)

    @property
    def location(self):
        return self._location

    @location.setter
    def location(self, value):
        self._location = value

    def __str__(self):
        if self._location != None:
            return "Event␣-␣Name:␣" + self.name + ";␣Probability:␣"
               + str(self.probability) + \
            ";␣Ending:␣" + str(self.ending) + ";␣Location:␣" + str(
               self._location) + "\n"
        else:
            return "Event␣-␣Name:␣" + self.name + ";␣Probability:␣"
```

```
                + str(self.probability) + \
            "; Ending: " + str(self.ending) + "\n"

    def __repr__(self):
        return self.__str__()

    def test(self):
        """ Test method for Event """
        print("Testing Event class")

if __name__ == "__main__":
    e = Event()
    e.test()
```

B.4 ExcelInput.py

```
import pandas as pd
import Event as event

class ExcelInput:
    """
    Description:
    Gathers historical data from an Excel spreadsheet, passed as a
       parameter. parse() method gathers phases, events, and event
       waypoints.
    """

    def parse(self, filename):
        # Parse the excel file into a pandas dataframe
        df = pd.read_excel(filename)

        # Grab the list of phases from the phase column
        phases = list(df["Phase"].unique())

        # Group the events by phase
        grouped = df.groupby("Phase")

        # Create dictionary with phase as key and (phase, events) as
            value
        parsed_data = {}
        for single_phase in phases:
            phase_events = grouped.get_group(single_phase)
            events = []
            for row in phase_events.itertuples(index=False):
                e = event.Event(name=row[0],probability=float(row
                   [2]),ending=bool(row[3]),locx=float(row[4]),locy=
                   float(row[5]),locz=float(row[6]))
                events.append(e)
                #print(e)
            parsed_data[single_phase] = events
        return parsed_data

if __name__ == "__main__":
    ep = ExcelInput()
    parsed_data = ep.parse("past_mission_data.xlsx")
```

```
print(parsed_data)
```

B.5 InputCollector.py

```
import ExcelInput as excelinput
import DISInput as disinput
import Scenario as scenario
import Phase as phase
import Campaign as campaign
import Event as event

class InputCollector:
    """
    Description:
    Collects input using DISInput, ExcelInput, and user input at
       command prompt. collect() method gathers all relevant data
       and returns a Campaign.

    """

    def __init__(self):
        """
        Description:

        Arguments:
        name:
        scenarios:
        distribution:
        """
        self.units = {}
        self.waypoints = {}
        self.excel_parsed = {}

    def collect(self):
        # Collect campaign details from user
        answer = input("Name␣of␣campaign␣(default:␣Campaign):␣")
        if answer == "":
            answer = "Campaign"
        c = campaign.Campaign(answer)

        answer = input("Number␣of␣scenarios␣to␣simulate␣within␣
           campaign␣(default:␣1):␣")
        if answer == "":
```

```
    answer = 1
num_scenarios = int(answer)

# Collect scenario details from DIS
for _ in range(num_scenarios):
    print("============")
    answer = input("Name␣of␣scenario␣(default:␣Scenario):␣")
    if answer == "":
        answer = "Scenario"
    scenario_name = answer

    input("\nOpen␣a␣DIS-friendly␣simulation␣(e.g.␣VR␣Forces)
       ␣and␣input␣scenario␣parameters␣" + \
        "including␣waypoints␣and␣units.␣Press␣<enter>␣to␣
           begin␣collecting␣data.")

    # Collect DIS details from DISInput
    try:
        dis = disinput.DISInput()
        dis.collect()
    except Exception:
        print("Timeout␣on␣DIS␣collection.")
    self.units = dis.units
    self.waypoints = dis.waypoints

    # Collect excel details from ExcelInput
    answer = input("\nEnter␣Excel␣filename␣to␣parse␣phases␣
       and␣events␣for␣this␣scenario␣(default:␣\"
       past_mission_data.xlsx\"):␣")
    if answer == "":
        answer = "past_mission_data.xlsx"
    ei = excelinput.ExcelInput()
    self.excel_parsed = ei.parse(answer)

    print("Missions␣start␣from␣a␣<Unit>␣location␣and␣
       progress␣through␣a␣series␣of␣<Phases>␣marked␣by␣<
       Waypoints>.")
    # Grab a unit
    answer = ""
    while answer == "":
```

```
    print(self.units)
    answer = input("Name of <Unit> to simulate (default:
       <first element>)? ")
    if answer == "" and (len(self.units) > 0):
        answer = list(self.units.keys())[0]
    if answer in self.units.keys():
        break
    else:
        print("Unit not in list of units.")
        answer = ""
unit = answer
start_wp = self.units[unit]


# pop unit and put in loop for multiple units
#self.units.pop(unit)

sorted_phases = []
while (len(self.excel_parsed.keys()) > 0):
    answer = ""
    # Get the phase to add
    while answer == "":
        print("Phases:",self.excel_parsed.keys())
        answer = input("Name of next phase (default: <
           first element>)? ")
        if answer == "" and (len(self.excel_parsed) > 0)
           :
            answer = list(self.excel_parsed.keys())[0]
        if answer in self.excel_parsed.keys():
            break
        else:
            print("Phase not in list of phases.")
            answer = ""
    phase_name = answer
    answer = ""
    # Get the waypoints for the phase
    while answer == "":
        print("Waypoints:",self.waypoints)
        print("Start waypoint:",start_wp)
        answer = input("Enter phase ending waypoint name
```

```
                    :␣")
                if answer in self.waypoints.keys():
                    break
                else:
                    print("Waypoint␣not␣in␣list␣of␣waypoints.")
                    answer = ""
            end_wp = answer
            end_wp = self.waypoints[end_wp]

            # Create a phase with the known information
            p = phase.Phase(phase_name,start_wp,end_wp,self.
                excel_parsed[phase_name])
            #print("=== Created phase:",p)
            start_wp = end_wp

            sorted_phases.append(p)
            self.excel_parsed.pop(phase_name)

        s = scenario.Scenario(name = scenario_name, phases =
            sorted_phases, units = self.units)
        c.add_scenario(s)
    return c


def test(self):
    """ Test method for InputCollector """
    self.units["ScanEagle"] = (36.58758646824263,
        -121.84557312083683, 56.130328943021595)
    self.waypoints["Waypoint␣1"] = (36.58951149068446,
        -121.8537085367434, 46.495832765474916)
    self.waypoints["Waypoint␣2"] = (36.59157983372531,
        -121.84426439988438, 29.925970377400517)
    self.waypoints["Waypoint␣3"] = (36.584077636235314,
        -121.83478564306455, 74.84547435864806)
    self.waypoints["Waypoint␣4"] = (36.580911876028495,
        -121.84686393534858, 77.91676191240549)
    e1 = event.Event("Airfield",0.75,False)
    e2 = event.Event("Boat",0.25,False)
    e3 = event.Event("Hand",0.1,False)
    e4 = event.Event("Crash",0.05,True)
```

```
        p1 = phase.Phase("Launch",self.units["ScanEagle"],self.
            waypoints["Waypoint 1"],[e1,e2,e3,e4])
        e1 = event.Event("Loss Link",0.2,False)
        e2 = event.Event("Enemy Discovery",0.3,False)
        p2 = phase.Phase("Ingress",self.waypoints["Waypoint 1"],self
            .waypoints["Waypoint 2"],[e1,e2])
        e1 = event.Event("HVT Appears",0.1,False)
        e2 = event.Event("Loss Link",0.1,False)
        e3 = event.Event("Enemy AAA",0.8,True
            ,36.5824399797592,-121.92000326858647,42.07877039699254)
        e4 = event.Event("Retasking",0.3,False)
        p3 = phase.Phase("Actions",self.waypoints["Waypoint 2"],self
            .waypoints["Waypoint 3"],[e1,e2,e3,e4])
        e1 = event.Event("Airfield Landing",0.1,False)
        e2 = event.Event("Hard Landing",0.3,False)
        e3 = event.Event("Crash",0.05,True)
        e4 = event.Event("Friendly Aircraft Delay",0.1,False)
        e5 = event.Event("Loss Link",0.05,False)
        p4 = phase.Phase("Recovery",self.waypoints["Waypoint 3"],
            self.waypoints["Waypoint 4"],[e1,e2,e3,e4,e5])
        s = scenario.Scenario("Scenario 1",[p1,p2,p3,p4],self.units)
        s2 = scenario.Scenario("Scenario 2",[p1,p2,p3,p4],self.units
            )
        c = campaign.Campaign("Campaign")
        c.add_scenario(s,1)
        c.add_scenario(s2,2)
        return c

if __name__ == "__main__":
    ic = InputCollector()
    print(ic.test())
```

B.6 Mission.py

```
class Mission:
    """
    Description:
    Represents a single mission, its waypoints, and any events that
        took place.

    """

    def __init__(self, name = "Mission␣#", waypoints = [], events =
        [], scenario = ""):
        self._name = name
        self.waypoints = waypoints
        self._events = events
        self.scenario = scenario

    @property
    def name(self):
        return self._name

    @property
    def events(self):
        return self._events

    def __str__(self):
        return "\nMission␣-␣Name:␣" + self._name + "\nWaypoints:␣" +
            str(self.waypoints) + \
            "\nEvents:␣" + str(self._events) + "\nScenario:␣" + self
                .scenario

    def __repr__(self):
        return self.__str__()

    def test(self):
        """ Test method for Mission """
        print("Testing␣Mission␣class")
```

```
if __name__ == "__main__":
    m = Mission()
    m.test()
    print(m)
```

B.7 Phase.py

```
import Event as event
from scipy.stats import bernoulli
import WaypointFuzzer as wf

class Phase:
    """
    Description:
    Represents a single phase, its start and end waypoint, and any
       events in that phase. generate_events() generates a list of
       events that occurred during this phase, for a single mission.

    Parameters:

    """


    def __init__(self, name = "Phase", start = (0,0,0), end =
       (0,0,0), events = []):
        self.name = name
        self.start_wp = start
        self.end_wp = end
        self.events = events


    def __str__(self):
        return "Phase - Name: " + self.name + "\nStart: " + str(self
           .start_wp) + \
            "\nEnd: " + str(self.end_wp) + "\n" + str(self.events)

    def __repr__(self):
        return self.__str__()

    def generate_events(self, fuzz_std_dev):
        # Grab all the event probabilities
        #print("Generating events from:",self.events)
        # Determine if each event will happen through Bernoulli
           distribution
        fuzzer = wf.WaypointFuzzer(fuzz_std_dev)
        events = []
```

```
        for e in self.events:
            rv = bernoulli.rvs(e.probability,size=1)
            #print("rv:",rv)
            if (rv):
                #print("Event added: ",e)
                #events.append(e)
                new_event = event.Event(e.name, e.probability, e.
                    ending)
                if (e.location):
                    new_event.location = fuzzer.fuzz_single(e.
                        location)
                events.append(new_event)
        return events

    def test(self):
        """ Test method for Phase """
        print("Testing␣Phase␣class")


if __name__ == "__main__":
    e1 = event.Event()
    e2 = event.Event()
    events = [e1,e2]

    p = Phase("steve",events = events)
    p.test()
    print(p)
```

B.8 PrototypeGenerator.py

```
import InputCollector as inputcollector
import numpy as np
import STKConnectGenerator as stkgen
import sys
import getopt

class PrototypeGenerator:
    """
    Description:
    Main top-level class for generating mission profiles. Use '-t'
       for testing mode and '-s <seed>'
    to use a specific seed. run() calls InputCollector to gather
       input, Campaign.generate()
    to generate mission profiles, then STKConnectGenerator to create
        STK output. Writes mission strings
    to console and Connect strings to a file.

    """

    def __init__(self, seed = 1775, num_missions = 30, filename = "
       stk.txt", string_filename = "strings.txt"):
        """
        Description:

        Arguments:

        """
        self.filename = filename
        self.num_missions = num_missions
        self.string_filename = string_filename
        np.random.seed(seed)

    def run(self, test = False):
        ic = inputcollector.InputCollector()
        campaign = None
        if (test):
            campaign = ic.test()
        else:
            campaign = ic.collect()
```

```
        # Generate required number of missions per scenario
        missions = campaign.generate(self.num_missions)
        print("Output written to",self.string_filename)
        with open(self.string_filename, "w") as mf:
            for mission in missions:
                mf.write(str(mission))
        #print(missions)

        stk = stkgen.STKConnectGenerator()
        mission_strings = []
        for profile in missions:
            name = profile.name
            string = stk.add_aircraft(name)
            string += stk.add_multiple_waypoints(name,profile.
               waypoints)
            for e in profile.events:
                if e.location:
                    # Given a threat radius, it would be passed here
                        to add_facility, min_range
                    # and max_range in the 3rd and 4th parameters
                    string += stk.add_facility("AAA"+name,e.location
                       )
                    string += stk.compute_access(name)
            mission_strings.append(string)

        print("Output written to",self.filename)
        with open(self.filename, "w") as f:
            for string in mission_strings:
                f.write(string)

if __name__ == "__main__":
    # Grab the command line options
    argv = sys.argv[1:]

    options = []
    args = []
    test = False
    seed = 1775
    num = 30
```

```
filename = "stk.txt"
try:
    options, args = getopt.getopt(argv, 'ts:f:n:')
except getopt.GetoptError:
    print("Invalid option provided.")

for opt, arg in options:
    if opt == "-s":
        seed = int(arg)
    if opt == "-t":
        test = True
    if opt == "-f":
        filename = str(arg)
    if opt == "-n":
        num = int(arg)

pg = PrototypeGenerator(seed,num,filename)
if (test):
    pg.run(test = True)
else:
    pg.run()
```

B.9 Scenario.py

```
import Mission as mission
import WaypointFuzzer as wf

class Scenario:
    """
    Description:
    Holds the specific phases and units for a given scenario.
       generate() generates the requested
    number of mission profiles for this particular scenario and
       returns the mission strings and a
    counter for the mission number.

    """

    def __init__(self, name = "Scenario", phases = {}, units = {}):
        self.name = name
        self.phases = phases
        self.units = units

    def __str__(self):
        return "Scenario - Name: " + self.name + " ----------\n" +
           str(self.phases) + "\nUnits: " + str(self.units) + "\n"

    def __repr__(self):
        return self.__str__()


    def generate(self, mission_iterations = 1, fuzz_std_dev = 0.001,
        mission_no = 1):
        """
        generate


        Parameters:
        scenario_iterations:
        """
        missions = []
        fuzzer = wf.WaypointFuzzer(fuzz_std_dev)
```

```
        # Generate a mission profile by phase
        for _ in range(mission_iterations):
            name = str(mission_no)
            mission_no += 1
            waypoints = []
            first = True
            events = []
            for phase in self.phases:
                if (first):
                    waypoints.append(fuzzer.fuzz_single(phase.
                        start_wp))
                    first = False
                waypoints.append(fuzzer.fuzz_single(phase.end_wp))

                # Determine if an event occurs in this phase
                e = phase.generate_events(fuzz_std_dev)
                if len(e) > 0:
                    events.extend(e)
            m = mission.Mission(name, waypoints, events, self.name)
            missions.append(m)
        return missions,mission_no


    def test(self):
        """ Test method for Scenario """
        print("Testing␣Scenario␣class")


if __name__ == "__main__":
    s = Scenario()
    s.test()
    print(s)
```

B.10 STKConnectGenerator.py

```
class STKConnectGenerator:
    """
    Description:
    Creates STK Connect strings for simulation. add_aircraft(),
       add_facility(), add_waypoint(), and compute_access() generate
        the appropriate strings needed to perform those respective
       actions in STK.
    """

    def __init__(self, port = 5001):

        self.port = port

    def add_aircraft(self, name = "Aircraft", model_file = "C:\\
       Program␣Files\\AGI\\STK␣12\\STKData\\VO\\Models\\Air\\uav.mdl
       "):
        string = []
        if (len(name) > 0):
            # Add new aircraft string, propagator, altitude ref, and
                object model
            string = "New␣/␣*/Aircraft␣" + name + "\n" + \
                "SetPropagator␣*/Aircraft/" + name + "␣GreatArc\n" +
                    \
                "AltitudeRef␣*/Aircraft/" + name + "␣Ref␣MSL\n" + \
                "VO␣*/Aircraft/" + name + "␣Model␣File␣\"" +
                   model_file + "\"\n"
        else:
            raise Exception("Cannot␣add␣empty␣aircraft␣name.")
        return string

    def add_facility(self, name = "Facility", loc = (0,0,0),
       min_range = 50, max_range = 4000, model_file = "C:\\Program␣
       Files\\AGI\\STK␣12\\STKData\\VO\\Models\\Land\\sa10-mobile-a.
       mdl"):
        string = []
        if (len(name) > 0):
            # Add new facility string, model
```

```
            sensor_name = name+"Sensor"
            string = "New␣/␣*/Facility␣" + name + "\n" + \
                "SetPosition␣*/Facility/" + name + "␣Geodetic␣" +
                    str(loc[0]) + "␣" + str(loc[1]) + "␣" + str(loc
                    [2]) + "\n" + \
                "VO␣*/Facility/" + name + "␣Model␣File␣\"" +
                    model_file + "\"\n" + \
                "New␣/␣*/Facility/" + name + "/Sensor␣" +
                    sensor_name + "\n" + \
                "Define␣*/Facility/" + name + "/Sensor/" +
                    sensor_name + "␣SimpleCone␣90\n" + \
                "SetConstraint␣*/Facility/" + name + "/Sensor/" +
                    sensor_name + "␣Range␣Min␣" + str(min_range) + \
                "␣Max␣" + str(max_range) + "\n"
        else:
            raise Exception("Cannot␣add␣empty␣facility␣name.")
        return string

    def compute_access(self, mission_no = "-1"):
        string = []
        if mission_no != "-1":
            string = "Access␣*/Aircraft/" + mission_no + "␣*/
                Facility/AAA" + mission_no + "/Sensor/AAA" +
                mission_no + "Sensor\n"
        else:
            raise Exception("Invalid␣mission␣number␣to␣compute␣
                access.")
        return string

    def add_waypoint(self, name = "Aircraft", waypoint = (0,0,0),
       vel = 50):
        x = str(waypoint[0])
        y = str(waypoint[1])
        z = str(waypoint[2])
        string = "AddWaypoint␣*/Aircraft/" + name + "␣
            DetTimeAccFromVel␣" + x + \
                    "␣" + y + "␣" + z + "␣" + str(vel) + "\n"
        return string

    def add_multiple_waypoints(self, name = "Aircraft", waypoints =
```

```
        [], vel = 50):
        string = ""
        for waypoint in waypoints:
            string += self.add_waypoint(name,waypoint,vel)
        return string

    def test(self):
        """ Test method for STKConnectGenerator """
        print("Testing␣STKConnectGenerator␣class")
        string = self.add_aircraft("ScanEagle")
        string += (self.add_waypoint("ScanEagle",(36.58754402066111,
            -121.84687600156725, 56.13280162216775)))
        string += (self.add_waypoint("ScanEagle",(36.58864976294971,
            -121.85225056965419, 46.49720940472794)))
        print(string)


if __name__ == "__main__":
    stk = STKConnectGenerator()
    stk.test()
```

B.11 WaypointFuzzer.py

```
from numpy.random import normal as norm

class WaypointFuzzer:
    """
    Description:
    Generates a random x,y,z waypoint within a normal distribution
       given horizontal and vertical standard deviation. fuzz_single
       () takes a single waypoint and fuzz_group() takes a list of
       waypoints.
    """
    def __init__(self, std_dev = 0.001, vert_std_dev = 0.0001):
        # Standard deviation of 0.01 in the x and y equates to about
            1-2 km difference in distance from original point, most
           of the time
        # Std_dev of 0.005 results in roughly 250m-750m distance
           between original and fuzzed waypoint
        # Std_dev of 0.001 results in roughly 70-300m separation
        self.std_dev = std_dev
        self.vert_std_dev = vert_std_dev

    def fuzz_single(self, waypoint=(0,0,0)):
        x = waypoint[0]
        y = waypoint[1]
        z = waypoint[2]
        new_x = norm(x, self.std_dev, None)
        new_y = norm(y, self.std_dev, None)
        new_z = norm(z, self.vert_std_dev, None)
        return (new_x,new_y,new_z)

    def fuzz_group(self, waypoints = []):
        fuzzed_group = []
        for waypoint in waypoints:
            new_wp = self.fuzz_single(waypoint)
            fuzzed_group.append(new_wp)
        return fuzzed_group


    def test(self):
        group = []
```

```
        waypoint =
            (36.58758646824263,-121.84557312083683,56.130328943021595)

        print("Original: ",waypoint)
        new_wp = self.fuzz_single(waypoint)
        print("New: ",new_wp)
        group.append(new_wp)

        print("Original: ",new_wp)
        new_wp = self.fuzz_single(new_wp)
        print("New: ",new_wp)
        group.append(new_wp)

        print("Original: ",new_wp)
        new_wp = self.fuzz_single(new_wp)
        print("New: ",new_wp)
        group.append(new_wp)

        print("Original Group: ",group)
        new_group = self.fuzz_group(group)
        print("New Group: ",new_group)

if __name__ == "__main__":
    wp = WaypointFuzzer()
    wp.test()
```

List of References

[1] D. Berger, "Commandant's planning guidance," July 2019. Available: https://www.marines.mil/Portals/1/Publications/Commandant's%20Planning%20Guidance_2019.pdf?ver=2019-07-17-090732-937

[2] *The Defense Acquisition System*, DOD Directive 5000.01, Under Secretary of Defense for Acquisition and Sustainment, Washington, DC, USA, 2020. [Online]. Available: https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodd/500001p.pdf?ver=2020-09-09-160307-310

[3] *Interim National Security Strategic Guidance*, The White House, Washington, DC, USA, March 2021. [Online]. Available: https://www.whitehouse.gov/briefing-room/statements-releases/2021/03/03/interim-national-security-strategic-guidance/

[4] D. Cebul, "How is China developing AI technology so much faster than the US?" Mar 2018. [Online]. Available: https://www.c4isrnet.com/home/2018/03/15/how-is-china-developing-ai-technology-so-much-faster-than-the-us/

[5] AcqNotes. "Acquisition process overview," Sep. 15, 2020. [Online]. Available: https://acqnotes.com/acqnote/acquisitions/acquisition-process-overview

[6] R. Rendon and K. Snider, *Management of Defense Acquisition Projects*, 2nd ed. American Institute of Aeronautics and Astronautics Inc., 2019.

[7] J. Heagney, *Fundamentals of Project Management*, 4th ed. AMACOM, 2011.

[8] F. Hartman, "Modeling, simulation and analysis: Enabling early acquisition decisions," Naval Postgraduate School, Monterey, CA, USA, Tech. Rep. NPS-AM-10-035, 2010. [Online]. Available: http://hdl.handle.net/10945/33468

[9] R. Darken, "Early synthetic prototyping," Monterey, CA, USA, 2013. [Online]. Available: http://hdl.handle.net/10945/51401

[10] R. Smith and B. Vogt, "Early synthetic prototyping digital warfighting for systems engineering," *Cybersecurity and Information Systems Information Analysis Center*, vol. 5, no. 4, Winter 2018 [Online]. doi: https://csiac.org/articles/early-synthetic-prototyping-digital-warfighting-for-systems-engineering/.

[11] R. Smith, "War game introduces early synthetic prototyping," United States Army, Jun. 8, 2018. [Online]. Available: https://www.army.mil/article/206543/war_game_introduces_early_synthetic_prototyping

[12] J. Berry and J. Mulski, "Other transaction authority (OTA) application for warfighting development," M.S. thesis, Graduate School of Defense Management, NPS, Monterey, CA, USA, 2020. [Online]. Available: http://hdl.handle.net/10945/66585

[13] *Operation of the Adaptive Acquisition Framework*, DOD Instruction 5000.02, Under Secretary of Defense for Acquisition and Sustainment, Washington, DC, USA, 2020. [Online]. Available: https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/500002p.pdf?ver=2020-01-23-144114-093

[14] J. Demotes-Mainard, "RAH-66 Comanche - the self-inflicted termination: Exploring the dynamics of change in weapons procurement," *Defense Acquisition Research Journal*, vol. 19, no. 2, Apr 2012 [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/a564477.pdf

[15] AcqNotes. "Simulation based acquisition," 2009. [Online]. Available: https://acqnotes.com/acqnote/tasks/simulation-based-acquisition-sba

[16] G. Hunt, "The analysis of simulation based acquisition economic breakpoints in the life cycle of major programs," M.S. thesis, Graduate School of Defense Management, NPS, Monterey, CA, USA, 2002. [Online]. Available: http://hdl.handle.net/10945/3749

[17] S. Soh, "Determining intelligence, surveillance and reconnaissance (ISR) system effectiveness, and integration as part of force protection and system survivability," M.S. thesis, Graduate School of Engineering and Applied Sciences, NPS, Monterey, CA, USA, 2013. [Online]. Available: http://hdl.handle.net/10945/37721

[18] S. Gray, "Agent-based simulation to support the effectiveness, procurement, and employment of non-lethal weapon systems," M.S. thesis, Graduate School of Operational and Information Sciences, NPS, Monterey, CA, USA, 2017. [Online]. Available: http://hdl.handle.net/10945/55604

[19] K. Murray, "Early synthetic prototyping: Exploring designs and concepts within games," M.S. thesis, Graduate School of Operational Information Sciences, NPS, Monterey, CA, USA, 2014. [Online]. Available: http://hdl.handle.net/10945/44627

[20] A. Dobkin, "The video game that could shape the future of war," The Atlantic, Oct. 27, 2017 [Online]. Available: https://www.theatlantic.com/technology/archive/2017/10/operation-overmatch/544062/

[21] Operation Overmatch. "Operation Overmatch Sept 2018 highlight video." Sep. 27, 2018. [YouTube video]. Available: https://www.youtube.com/watch?v=FKNiEZrAlt8

[22] Defense Industry Daily, "From dolphins to destroyers: The ScanEagle UAV," Jun. 22, 2021 [Online]. Available: https://www.defenseindustrydaily.com/from-dolphins-to-destroyers-the-scaneagle-uav-04933/

[23] Bohemia Interactive Simulations. "VBS4". [Online]. Available: https://vbs4.com/

[24] USMC. Virtual Battlespace Simulation (VBS). [Online]. Available: https://www.29palms.marines.mil/training/magtftcsims/vbs/

[25] N. Edwards, "Three ways VBS4 accelerates UAS sensor operation and intelligence analyst training," Accessed Jul. 11, 2021 [Online]. Available: https://vbs4.com/blogs/Three-Key-Ways-VBS4-Accelerates-UAS-Sensor-Operator-and-Intelligence-Analyst-Training

[26] MAK. VR-Forces. [Online]. Available: https://www.mak.com/products/simulate/vr-forces

[27] AGI, "STK Aviator," Accessed Jul. 30, 2021 [Online]. Available: https://www.agi.com/products/stk-specialized-modules/stk-aviator

[28] AGI, "Getting started with Connect," Accessed Jul. 10, 2021 [Online]. Available: https://help.agi.com/stk/Subsystems/connect/connect.htm

[29] BISim, "New integrated development environment simplifies creation of modular software for video game development, other software applications," Accessed Aug. 1, 2021 [Online]. Available: https://bisimulations.com/company/news/press-releases/gears-studio-new-development-environment-simplifies-modular-software-creation

[30] *UAS Multi-Service Tactics, Techniques, and Procedures for the Tactical Employment of Unmanned Aircraft Systems*, USA ATP 3-04.64, USMC MCRP 3-42.1A, USN NTTP 3-55.14, USAF AFTTP 3-2.64, Air Land Sea Application Center, Hampton, VA, USA, 2015. [Online]. Available: https://armypubs.army.mil/ProductMaps/PubForm/Details.aspx?PUB_ID=104824

[31] Bohemia Interactive Simulations. "Create training scenarios rapidly with the VBS plan mode in VBS4", Aug. 7, 2020. [YouTube video]. Available: https://www.youtube.com/watch?v=enT1a7BYc9M

[32] AGI, "STK Object Model tutorial (C, Java, MATLAB, Python)," Accessed Jul. 1, 2021 [Online]. Available: https://help.agi.com/stkdevkit/index.htm

[33] *IEEE Standard for Distributed Interactive Simulation (DIS) – Communication Services and Profiles*, IEEE 1278.2-2015, 2014. [Online]. Available: https://standards.ieee.org/standard/1278_2-2015.html

[34] S. Rose, “Why is python programming a perfect fit for big data?” Towards Data Science, Dec. 27, 2019 [Online]. Available: https://towardsdatascience.com/why-is-python-programming-a-perfect-fit-for-big-data-5ac54ee8f95e

[35] Open-DIS, “An open source implementation of the distributed interactive simulation protocol,” Jan. 10, 2021 [Online]. Available: http://open-dis.org/

[36] Paessler. “IT explained: Packet sniffing”. [Online]. Available: https://www.paessler.com/it-explained/packet-sniffing

[37] Open-DIS, “dis_receiver.py,” Jun. 25, 2015 [Online]. Available: https://github.com/open-dis/open-dis-python/blob/master/examples/dis_receiver.py

[38] pandas, “pandas.read_excel(),” Accessed Jun. 6, 2021 [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

[39] NIST, “Fuzz testing,” Accessed Jul. 6, 2021 [Online]. Available: https://csrc.nist.gov/glossary/term/Fuzz_Testing

[40] NumPy, “numpy.random.normal,” Accessed Jun. 6, 2021 [Online]. Available: https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html

[41] MeridianOutpost, “Latitude/longitude distance calculator,” Accessed Apr. 9, 2021 [Online]. Available: https://www.meridianoutpost.com/resources/etools/calculators/calculator-latitude-longitude-distance.php?

[42] Wikipedia, “File:multivariate normal sample.svg,” Accessed Jul. 20, 2021 [Online]. Available: https://commons.wikimedia.org/wiki/File:Multivariate_normal_sample.svg

[43] SciPy, “scipy.stats.bernoulli,” Accessed Jun. 6, 2021 [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bernoulli.html

[44] AGI, “STK Integration,” Acccessed Jan. 5, 2021 [Online]. Available: https://www.agi.com/products/stk-systems-bundle/stk-integration

[45] Wikipedia, “9K32 Strela-2,” Accessed Jul. 21, 2021 [Online]. Available: https://en.wikipedia.org/wiki/9K32_Strela-2

[46] MappingSupport, “GISsurfer map with military grid reference system,” Accessed Jul. 20, 2021 [Online]. Available: https://mappingsupport.com/p2/gissurfer.php?center=14SQH05239974&zoom=4&basemap=USA_basemap

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California