



**OPTIMIZING MULTI-DOMAIN
SYSTEM-OF-SYSTEMS USING
MODEL-BASED SYSTEM ENGINEERING**

THESIS

Cody G. Palmer, 2d Lt, USAF
AFIT-ENV-MS-16-M-174

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENV-MS-16-M-174

OPTIMIZING MULTI-DOMAIN SYSTEM-OF-SYSTEMS USING
MODEL-BASED SYSTEM ENGINEERING

THESIS

Presented to the Faculty
Department of Systems Engineering and Management
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Systems Engineering

Cody G. Palmer, B.S.E.E.

2d Lt, USAF

March 9, 2016

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

OPTIMIZING MULTI-DOMAIN SYSTEM-OF-SYSTEMS USING
MODEL-BASED SYSTEM ENGINEERING

THESIS

Cody G. Palmer, B.S.E.E.
2d Lt, USAF

Committee Membership:

John M. Colombi, Ph.D.
Chair

Lt Col Tom C. Ford, Ph.D.
Member

Brad J. Ayres, Ph.D.
Member

Abstract

System-of-systems (SoS) that exist across multiple domains can offer new solutions to complex problems. Model-based systems engineering (MBSE) is a popular approach used in Systems Engineering (SE) for architecting systems or SoS, but it is often difficult to identify and assess many different instantiated architectures. This research looks at applying the Object-Oriented System Engineering Method (OOSEM) to multi-domain SoS problems. The Systems Modeling Language (SysML) is used to create a multi-domain SoS composed of satellites and aircraft for surveillance missions such as search and rescue. An executable architecture is developed to assess and find a near optimal SoS. The executable architecture consists of five components; architecture, optimization, simulation, performance, and value. The architecture component contains attributes describing the SoS which are passed to the optimization component. The optimization component evaluates candidate solutions using simulations, performance metric calculations, and a value model. Once the optimizer has finished, the optimal parameters are passed back to the architecture component and updated in the SysML model. This research found, for a permissive environment, the optimal multi-domain SoS had a 7.4% and 28.7% increase in average value over the aircraft only and satellite only systems respectively.

Table of Contents

	Page
Abstract	iv
List of Figures	viii
List of Tables	x
I. Problem Statement	1
1.1 Background	1
1.2 Research Objective and Questions	2
1.3 Justification	3
1.4 Scope, Assumptions, and Limitations	4
1.5 Methodology and Resources	5
1.6 Overview	6
II. Literature	7
2.1 Background	7
2.2 Terminology	8
System	8
System-of-Systems	9
Model Based Systems Engineering (MBSE)	9
Systems Architecture	10
Domain	10
2.3 MBSE Methodologies	11
INCOSE Object-Oriented Systems Engineering Method (OOSEM)	11
Functions-Based Systems Engineering Method (FBSEM)	14
2.4 MBSE Applications	14
2.5 Summary	16
III. Methodology	17
3.1 Object-Oriented System Engineering Method	17
Define Mission Requirements	18
Define Operational Architecture	20
Synthesize SoS Architectures	24
Analyze System Requirements	25
Optimize and Evaluate Alternatives	25
Validate and Verify	26
3.2 Executable Architecture	27
Architecture Model	28

	Page
Simulation Model	28
Performance Model	29
Value Model	33
Optimization Model	36
3.3 Tool Architecture	39
Magic Draw	41
MATLAB	41
Python	42
Systems Tool Kit (STK)	42
3.4 Methodology Summary	43
IV. Application	44
4.1 Model Verification	44
Parameters	44
Results	48
4.2 Space Solution	54
Space Domain Architecture	54
Results	60
4.3 Air Solution	62
Air Domain Architecture	62
Results	66
4.4 Multi-Domain Solution	67
Multi-Domain Architecture	67
Results	71
V. Results	73
5.1 Analysis of Results	73
5.2 Conclusions	77
5.3 Future Work	80
Computation Time	80
Threat Modeling	81
Stochastic Modeling	81
Cost Model	82
Optimization Methods	83
5.4 Summary	83
Appendix A. Python code	85
A.1 myClasses.py	85
Scenario	85
Area of interest	87
Ground Vehicle	88
rndSensor and sqrSensor	89

	Page
Constellation	91
Aircraft	93
A.2 myFuncs.py	98
conceptEval	98
calcCovAOI	101
calcSat2GVAccess and calcAC2GVAccess	102
calcGSD	103
readCov	109
GSD and acGSD	111
calcVal	112
connect	113
Appendix B. MATLAB code	114
B.1 PythonCall.m	114
B.2 myConstraints.m	116
B.3 myout.m	116
B.4 scenEval.m	117
B.5 Parse.m	117
B.6 mat2py.py	117
References	119

List of Figures

Figure		Page
1.	System development process	13
2.	OOSEM Activities	13
3.	Operational Domain BDD	19
4.	Surveillance SoS bdd	21
5.	True anomaly offset example	22
6.	Example Aircraft Formation	23
7.	Five-model architecture	27
8.	Earth Geometry	32
9.	GSD Single Attribute Value Function	35
10.	Coverage Single Attribute Value Function	35
11.	Optimization Process	40
12.	Tool Architecture	40
13.	Constellation Class	46
14.	Optimal Inclination Parametric Diagram	46
15.	Instance of Space Constellation	47
16.	Solving for inclination using ParaMagic	48
17.	Value of candidate constellations at varying inclinations	49
18.	Optimal value of candidate constellation inclination	50
19.	Raw GSD at optimal inclination over 24 hours	51
20.	Optimal inclination orbit track relative to area of interest	51
21.	Post-value curve GSD at optimal inclination over 24 hours	52
22.	Coverage at optimal inclination over 24 hours	53

Figure	Page
23.	Value of system at optimal inclination over 24 hours 54
24.	Space only operational domain 55
25.	Space domain imaging SoS decomposition 57
26.	Space only domain parametric diagram 59
27.	Space only instance definition 60
28.	Results of optimal space domain architectures 61
29.	Optimum Space Results 62
30.	Air domain imaging SoS decomposition 63
31.	Air only domain parametric diagram 64
32.	Air only instance definition 65
33.	Results of optimal air domain architectures 66
34.	Imaging SoS decomposition 68
35.	Multi-domain parametric diagram 69
36.	Multi-domain instance definition 70
37.	Optimum Multi-Domain Results Break Down 71
38.	Comparison Between Solutions 74
39.	Air-Domain With Potential Threats 75
40.	Optimal Aircraft Formation 76
41.	Potential EA with cost model 83

List of Tables

Table	Page
1. External factor domain effects	26
2. Maximum off nadir look angles for at least 1 m GSD	33
3. Constants used in verification problem.	45
4. Space Domain Architecture Results	61
5. Air Domain Architecture Results	67
6. Multi-Domain Architecture Test Results	72
7. Average Altitude Comparison	77

OPTIMIZING MULTI-DOMAIN SYSTEM-OF-SYSTEMS USING MODEL-BASED SYSTEM ENGINEERING

I. Problem Statement

1.1 Background

The United States Air Force (USAF) core mission of intelligence, surveillance, and reconnaissance has existed since President Truman first assigned the five core missions to the Air Force in 1947 (James & Welsh, 2014). In more recent years the Middle East has provided a permissive combat environment that has driven the development of ISR capabilities. Now the USAF is being directed to focus on enhancing ISR capabilities in future contested and highly contested environments by utilizing cross-domain synergy (United States Air Force, 2013). The domains being referred to in cross-domain synergy include land, air, space, maritime, and cyber. The idea behind cross-domain synergy is to use capabilities that are implemented in different domains together to achieve greater overall effectiveness. Systems that implement a capability in one domain have traditionally been designed and optimized for that specific domain. In order to achieve cross-domain synergy, a system-of-systems (SoS) that has capabilities in multiple-domains needs to be designed and optimized for the overall mission effectiveness. It is not readily apparent if the optimal solution for a single domain system is also the optimal solution for a larger multiple-domain SoS.

1.2 Research Objective and Questions

This primary objective for this thesis is to develop a methodology for architecting optimal multiple domain System-of-Systems (SoS) with the ability to collect intelligence, surveillance, and reconnaissance (ISR) in A2/AD environments. In order to achieve this objective, four research questions have been posed.

The first research question is:

Using SysML, how should a multi-domain SoS be parameterized for assessment?

There are many ways to model both air and space systems using a combination of different parameters. It is not readily apparent if describing individual satellites and aircraft is the best method, or if describing groups of systems is better.

Disaggregated space systems might provide benefit over large single systems, but both should be capable of being parametrized for assessment.

The second research question is:

How should the value of different multi-domain SoS architectures be assessed?

This question is like comparing apples to oranges. Systems that operate in each domain have different benefits and weaknesses that are often hard to compare against each other. Satellites are great at providing global coverage, but aircraft are capable of varying their flight path or loitering over an area. Assessing the value of both satellites and aircraft operating together is not a trivial problem, but this will be required to determine an optimal SoS.

The third research question is:

How should a multiple domain SoS architecture optimization problem be formulated?

Due to the number of possible design parameters for a multiple domain SoS, there are an infinite number of possible candidate design solutions. Constraints will help

bound this problem, but the sample space will still remain too large to perform an exhaustive search of all the possible design solutions. For this reason an optimizer is needed to determine good solutions. The choice of the optimizer will depend on how the problem is formulated, and transforming the design problem into the mathematical optimization standard form is not a simple problem. This will depend a lot on the parameters chosen by the first research question.

The fourth research question is:

What is the added benefit of near-optimal multiple domain solutions?

The central idea behind integrating single domain solutions together into a multiple domain solution is that the systems will complement each other by using their strengths where the other domain has weaknesses. It is unknown what the actual benefit will be by using more than one domain. For some scenarios it might provide great benefit and others it may provide none.

1.3 Justification

This is not a trivial problem because finding an optimal solution in a single domain has proven difficult. The number of solutions for possible space systems far out number what is possible to analyze. There are hundreds of design parameters that can be adjusted, and not all combinations can be tested in a reasonable amount of time. The authors of *Application of Multidisciplinary Design Optimization Techniques to Distributed Satellite Systems* use a constrained example space system problem where only four parameters are allowed to change, orbit altitude, number of collectors, aperture diameter, and geometry of connection between the satellites (Jilla *et al.*, 2000). The orbit altitude has ten possible values while the other three parameters only have four possible values. This combination of parameters give 640 possible solutions in the solution set. The number of variable parameters were kept

small so the entire sample set could be analyzed and an optimal solution could be realized. In a real world situation the problem will have many design variables and the sample set can easily become too large to analyze. For example, a constellation of imaging satellites can vary altitude, inclination, eccentricity, the number of orbit planes, the number of satellites in an orbit plane, a true anomaly phase offset, the argument of perigee if eccentricity is not zero, and payload parameters such as aperture diameter, focal length, and detector properties. The addition of more domains will only increase the number of possible solutions further past what is possible to analyze in a reasonable amount of time.

1.4 Scope, Assumptions, and Limitations

The problem of collecting ISR information in highly contested and denied environments spans a wide range of domains. Both the space and cyberspace domains are identified as increasingly contested domains in the Joint Operational Access Concept and Department of Defense (DoD) 2012 strategic guidance (Dempsey, 2012),(Department of Defense, 2012). Cyberspace is currently a developing domain and is evolving rapidly. However, for the purpose of this thesis only the air and space domains will be considered. The sponsor of this thesis, Air Force Research Laboratory Aerospace Systems Directorate (AFRL/RQ), has defined the air domain systems to include only unmanned aircraft systems (UAS). The space domain systems include spacecraft and or constellations of spacecraft. There are also many different optimization techniques that can be used when attempting to determine optimal solutions. Taguchi's method, single axis exploration, and other search algorithms, such as simulated annealing or a genetic algorithm, are identified as optimization techniques for large complex trade spaces. Determining which method is the best will require more investigation and falls outside the scope of this

thesis. The purpose of this thesis is to develop a framework for evaluating multiple domain SoS and model integration that can be expanded on in future research.

1.5 Methodology and Resources

The methodology for this research will start by architecting, modeling, simulating, and optimizing single domain systems. There has been extensive research done in the space domain involving MBSE with integrated tool sets. Thompson details different types of space systems and optimizing disaggregated space systems (Thompson, 2015). After evaluating single domain systems the next step will be to propose an approach suitable for evaluating and finding an optimal multiple domain systems. The Object-Oriented System Engineering Method (OOSEM) is proposed as the methodology for designing multiple-domain systems. An executable architecture is then developed to support OOSEM. This executable architecture consists of five interrelated models that are architecture, optimization, simulation, performance, and value.

The resources used for this research are modeling, simulation, and analysis software tools. The modeling software tool used is MagicDraw 18.1 by No Magic Inc with the ParaMagic 18.0 plug-in from Intercax LLC. The simulation tool will be Systems Tool Kit (STK) 10.0 from Analytic Graphics Inc. The programming language Python 2.7 will be used for analysis and data handling. MATLAB R2015b (32-bit) is used for optimization and integration. Professional licenses will be needed for STK, MATLAB, MagicDraw, and ParaMagic. It will also be necessary to have a computer with at least 8 GB of RAM to be capable of running all of these programs concurrently. Future work should consider using a supercomputer. The reasoning for this recommendation will be expanded upon in Chapter V.

1.6 Overview

This thesis will follow the traditional five chapter format. Chapter I provides the background to the topic area and introduces the problem. Chapter II will discuss the current research, approaches, and methodologies used to solve problems in similar areas or domains. The methodology and approach used to answer the questions posed in Chapter I are proposed in Chapter III. An application of the proposed methodology to a multiple-domain SoS surveillance mission is presented in Chapter IV. Chapter V will analyze the results of the data presented in Chapter IV and the significance to the research questions. Chapter V will also offer future research paths that were identified during this research.

II. Literature

This chapter describes the background for this thesis and the current research in the area of MBSE. The first section gives a brief background of ISR and the reason behind investigating a multiple domain SoS. The terminology used within Systems Engineering (SE) and for this thesis are given in the second section. Model based systems engineering (MBSE) methodologies are discussed third, followed by a discussion of current MBSE research. At the end of this chapter a summary gives the highlights of each topic.

2.1 Background

The USAF core mission of intelligence, surveillance, and reconnaissance (ISR) has existed since President Truman first assigned the five core missions to the Air Force in 1947 (James & Welsh, 2014). Today the USAF defines ISR as “cross-domain synchronization and integration of the planning and operation of ISR assets; sensors; processing, exploitation and dissemination systems; and, analysis and production capabilities across the globe to enable current and future operations” (Curtis E. Lemay Center, 2015).

Permissive combat environments have driven the development of ISR capabilities in recent years. Now the USAF has been directed to focus on enhancing ISR capabilities in future contested and highly contested environments (United States Air Force, 2013). This direction is derived from the Department of Defense (DoD) 2012 strategic guidance which states a primary mission of “projecting power despite anti-access/area-denial (A2/AD) challenges” (Department of Defense, 2012).

Anti-access can be described as actions and capabilities that prevent opposing forces from entering an operational area (Dempsey, 2012). Area-denial can be described as

actions and capabilities that do not prevent access, but actions or capabilities that limit an opposing forces freedom of action in an operational area (Dempsey, 2012).

To accomplish the mission of projecting power despite A2/AD challenges the DoD 2012 strategic guidance includes the implementation of the Joint Operational Access Concept (JOAC). The JOAC describes how future joint forces will operate in A2/AD environments and the central idea of this concept is “Cross-Domain Synergy” (Dempsey, 2012). Cross-domain synergy will utilize capabilities implemented in more than one domain to enhance or mitigate vulnerabilities of capabilities implemented in other domains (Dempsey, 2012).

2.2 Terminology

To clarify terminology, this section will define the main terms used in this thesis: system, system-of-systems, model-based systems engineering, system architecture, and domain.

System.

A system is a construct or collection of different elements that together produce results not obtainable by the elements alone. The elements, or parts, can include people, hardware, software, facilities, policies, and documents; that is, all things required to produce systems-level results. The results include system level qualities, properties, characteristics, functions, behavior and performance. The value added by the system as a whole, beyond that contributed independently by the parts, is primarily created by the relationship among the parts; that is, how they are interconnected. (Maier & Rechtin, 2009)

The definition of a system by Maier and Rechtin is used by the International Council on Systems Engineering (INCOSE), the leading body for SE. It is also used by many others in the field of SE and will serve as the definition used in this thesis.

System-of-Systems.

A system of systems (SoS) is described by INCOSE as a system whose elements are independent systems themselves. Similar to a system, a SoS produces results that are only achievable by the combination of more than one system. Engineering a SoS is more difficult and complex than engineering a system because the systems operate independently and have their own objectives that may or may not align with the SoS objectives. Dahmann identifies seven challenges to engineering an SoS: SoS authorities; leadership; constituent systems' perspectives; capabilities and requirements; autonomy, interdependence, and emergence; testing, validation, and learning; and SoS principles (Dahmann, 2014). INCOSE also identifies security vulnerabilities as an issue of concern in today's environment. Systems that are secure when operating in isolation may not be secure when interfaced with other systems (INCOSE, 2015). For this thesis the individual systems will not be designed with objectives independent of the SoS objectives. All systems will be designed to provide maximum benefit to the SoS. The security concerns with interfacing of systems is outside the scope of this thesis but could be considered in future work.

Model Based Systems Engineering (MBSE).

INCOSE defines MBSE as “the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases (INCOSE, 2015).” Traditional SE captures system information in documents such as specifications, interface control documents, system description documents, analysis reports, trade studies, and procedures (INCOSE, 2015). This information is often difficult to maintain and assess. Relationships among components in the system are not readily visible and system

changes can have unanticipated effects. MBSE is different from traditional SE because it uses models rather than text to capture and relay information about a system. This improved ability to relay information has many benefits. Relationships among components can be seen by looking at the model and the impact of changes can be realized more easily. Assessing the completeness of a complex model is also much easier when looking at a visual model of the system rather than a pile of documents. MBSE also helps improve communication between system developers and stakeholders by providing an unambiguous model that can be referred to. The use of MBSE tools have also been requested by the sponsor of this research.

Systems Architecture.

There are many definitions of architecture, architecting, and architectural framework. Dictionaries, books, professional organizations, and military standards will all have slightly different definitions. Maier and Rechtin define an architecture as “The structure – in terms of components, connections, and constraints – of a product, process, or element” and systems architecting as “creating and building systems.” The INCOSE Systems Architecture Working Group (SAWG) define systems architecture as “The fundamental and unifying system structure defined in terms of system elements, interfaces, processes, constraints, and behaviors.” The INCOSE SAWG systems architecture definition will be the definition used for this research.

Domain.

In terms of military operations, a domain is where the platform that performs the intended capability is located, and a domain is bounded by the physical or virtual limitations where the capability fails to be effective. The DoD 2012 strategic

guidance specifically lists land, air, maritime, space, and cyberspace as all the domains in which the U.S. forces will operate (Department of Defense, 2012). As an initial investigation, this research will only be considering the air and space domains.

2.3 MBSE Methodologies

A methodology is more than just a process or a tool where one can simply give inputs and have the output generated. A methodology is a collection of processes, methods, and tools that are used together (Estefan, 2008). For this thesis a methodology that is extensible to systems operating in more than one domain is necessary. Even though not all of the domains will be modeled in this thesis, the methodology should support future work to build and adapt it for specific needs. A survey of MBSE methodologies was completed in 2008 by the National Aeronautics and Space Administration Jet Propulsion Laboratory (NASA-JPL) at the request of INCOSE (Estefan, 2008). This survey covers six different methodologies and describes the process, tool support, and availability of each. Of the six methodologies discussed by Estefan, the object-oriented systems engineering method (OOSEM) stood out from the rest as a good match for this research because of the applicability to a range of problems and it does not rely on specific tools. The functions-based systems engineering method (FBSEM) is also identified as a possible approach and is described in the INCOSE handbook (INCOSE, 2015). This section will describe the benefit and weaknesses of OOSEM and FBSEM and how they pertain to this project.

INCOSE Object-Oriented Systems Engineering Method (OOSEM).

This is a top-down iterative method that uses object-oriented modeling concepts, including blocks, objects, encapsulation, and inheritance, all of which are supported

by the Systems Modeling Language (SysML). This method is used to help create flexible and extensible systems that accommodate evolving technologies and changing requirements (INCOSE, 2015). OOSEM is meant to be tailored to the application it is being used for, and it is part of a higher level system development process. This process can be seen in Figure 1 and contains several other processes. OOSEM activities are contained within the “specify and design system” process seen in Figure 1. According to INCOSE, there are seven OOSEM activities and they are *analyze stakeholder needs*, *analyze system requirements*, *define logical architecture*, *synthesize candidate physical architectures*, *optimize and evaluate alternatives*, *manage requirements traceability*, and *validate and verify system* (INCOSE, 2015:195). All the activities except managing requirements traceability are described in detail and applied to an example home security problem in Chapter 16 of (Friedenthal *et al.*, 2009). Managing requirements traceability is performed throughout the other six OOSEM activities. It is used to keep requirements synchronized between the model and a requirements management database. It is also used to assess the impact requirements change will have on the system design, analysis, and verification elements. Figure 2 shows the seven OOSEM activities defined in the INCOSE handbook (INCOSE, 2015:195).

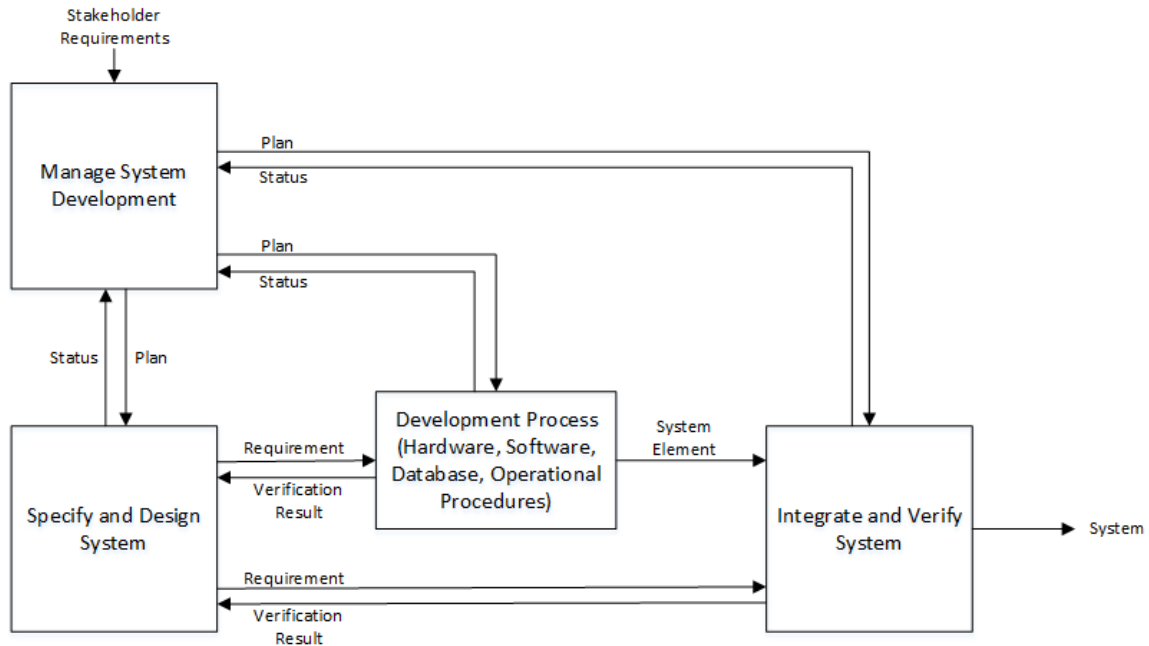


Figure 1. System development process that contains OOSEM activities within the specify and design system process.

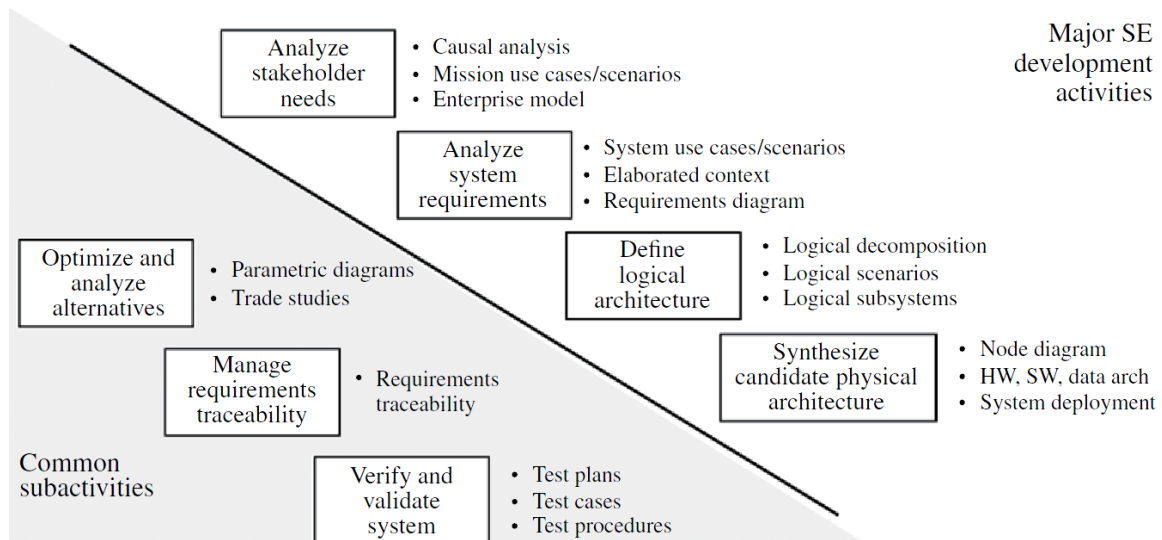


Figure 2. The seven OOSEM activities

Functions-Based Systems Engineering Method (FBSEM) .

The purpose of this method is to define a system using the functions and sub-functions that must be performed. A function is defined as “a characteristic task, action, or activity that must be performed to achieve a desired outcome” (INCOSE, 2015). FBSEM describes what the system will do but not how it will be accomplished (INCOSE, 2015). FBSEM is an iterative process that starts with the functions that must be performed by the system clearly defined. Next the functions are decomposed into sub-functions and system requirements are decomposed and allocated to the sub-functions. If the functions are not clearly defined before starting this process, then they will become more defined as the system requirements evolve (INCOSE, 2015). After sub-functions have been defined, the process is repeated and alternatives are explored. When this process is completed for each function a single functional decomposition is selected and all the internal and external interfaces are defined. The objective of FBSEM is to create a hierarchy of functional flow block diagrams that meet all of the system functional requirements (INCOSE, 2015). FBSEM is not always exclusively model based, however, there are functions-based methods, such as the Vitech MBSE Methodology, that are entirely model-based.

2.4 MBSE Applications

MBSE has been used in a wide variety of applications. This review will focus on a small number of applications relevant to optimizing aircraft or satellite design, multi-domain systems, system-of-systems, and executable architectures.

Friedenthal proposed a modified version of OOSEM for the application to an enterprise level, or SoS level, design (Friedenthal *et al.*, 2009). OOSEM is traditionally focused on the system level design, but it was shown to be adaptable

to the SoS level design as well. This work also supported the use of an executable architecture in the verify and validate development activity.

Bartolomei used a multi-domain matrix to model large-scale complex systems (Bartolomei *et al.*, 2012). The domains included environmental, social, functional, technical, process, and temporal. Five domains are described by six nodes which are the system drivers, stakeholders, objectives, functions, objects, and activities. Temporal is the sixth domain which helps show the relationship between these nodes over time.

Jilla investigated four multidisciplinary design optimization techniques applied to disaggregated space systems (Jilla *et al.*, 2000). The design optimization techniques were Taguchi’s method, simulated annealing, pseudo-gradient search, and single axis exploration. This research helped show the benefits and weaknesses of each method to a constrained space design problem.

Thompson applies the disaggregated integral system concept optimization (DISCO) methodology to a next-generation disaggregated weather system follow-on conceptual architecture (Thompson *et al.*, 2015). The DISCO methodology is intended to assist in optimizing conceptual architectures using simulations, performance assessment, and an optimization routine.

Colombi developed a method for assessing candidate architectures using executable architectures and five interrelated models (Colombi *et al.*, 2014). The five models included architecture, simulation, performance, cost, and value. This method was applied to a conceptual design for a manned mission to Mars.

Haywas investigated the unmanned aircraft systems (UAS) architecture and

identified the problem of resource constraints and excessive coupling in the architecture (Haywas, 2016). Using the results of Monte Carlo simulations, a new architecture is proposed to provide additional benefit. However, Haywas identifies this new architecture may only be applicable to a limited mission trade space.

2.5 Summary

This chapter described the terminology used in this thesis, two MBSE methodologies, and ISR in the A2/AD environment. The OOSEM methodology is ideal for this thesis due to its flexibility and extensibility to accommodate evolving technologies and requirements. Future ISR capabilities will be implemented in multiple-domains and the OOSEM is just one possible way of designing systems to implement those capabilities and adapt to changing requirements.

III. Methodology

This chapter describes a method of applying the Object-Oriented System Engineering Method (OOSEM) to a multi-domain SoS design problem. The detailed application of this method to ISR collection will be shown in Chapter IV. The seven OOSEM activities are discussed first in this chapter. An executable architecture framework, consisting of five interrelated models, was developed for use within one of the OOSEM activities and is discussed second. The tool architecture used to implement the executable architecture is described last.

3.1 Object-Oriented System Engineering Method

Using the review of MBSE methods discussed in Chapter II, OOSEM has been chosen for its flexibility and extensibility. There are seven OOSEM activities and they are *analyze stakeholder needs*, *define system requirements*, *define logical architecture*, *synthesize candidate physical architectures*, *optimize and evaluate alternatives*, *manage requirements traceability*, and *validate and verify system*. Friedenthal proposed a modified version of OOSEM for an application to a larger System of Systems (SoS) problem, Joint Force Protection (JFP) (Friedenthal *et al.*, 2007). This approach applies more emphasis on a SoS level architecture rather than a system level architecture. The application of the modified OOSEM activities to ISR operations in A2/AD environments is described next. However, the manage requirements traceability activity will not be used in this research. This activity takes place concurrently with the system design and can be used to assess the impact requirements change will have. Analyzing requirements change is not necessary and it falls outside the scope of this research.

Define Mission Requirements.

This activity is used to specify mission requirements and define the system boundary. First a conceptual design problem will be proposed to serve as the mission requirements. This conceptual design will be used throughout this chapter to demonstrate the proposed method and then refined in Chapter IV where it will be applied to an example problem.

The design problem is to determine an optimal design of a SoS for collecting imagery of an area of interest. The purpose of the imagery is to detect and identify an object the size of an automobile. The size of the area of interest is approximately 400 x 400 km. The ability to view anywhere within the area of interest at anytime is desired. The area of interest may also reside in a permissive or A2/AD environment. Only solutions consisting of aircraft and satellites will be considered. Optimality will be based on performance and the number of satellites or aircraft used to achieve that level of performance. It is understood that cost increases with the number of aircraft or satellites, but for this research, cost will not be used to determine the value of a solution.

The SoS under design is modeled as a black box that interacts with external systems and users. The users, external systems, natural environment, and any threats are all considered outside the SoS boundary. Threats can consist of enemy capabilities that are attributed to A2/AD environments, the area of interest, or targets. Modeling a permissive environment would simply exclude threats. The natural environment consists of terrain for the area of interest and possible weather effects. External systems are systems that the SoS might utilize but are not directly under its control. Examples of this would be existing communication satellites or existing ground stations that are supporting other missions or users. Existing

systems that will be controlled by the SoS would be considered inside the system boundary. This is done because if the SoS has control of the existing system then a parameter or function is capable of being varied. This means the parameter or function performed by the existing system can be treated as a design variable to the SoS and it should be inside the boundary. The acquisition of new aircraft or satellites directly in support of the ISR system are also considered inside the system boundary by the same logic. A domain logical block definition diagram (bdd) was defined in this first development activity for the ISR SoS and is shown in Figure 3. The reasoning for choosing the two time-variant MOEs, coverage and ground sample distance (GSD), will be described later. Many factors such as data size or bandwidth can impose limits on the SoS, but these factors are more detailed than necessary and are not considered for this research.

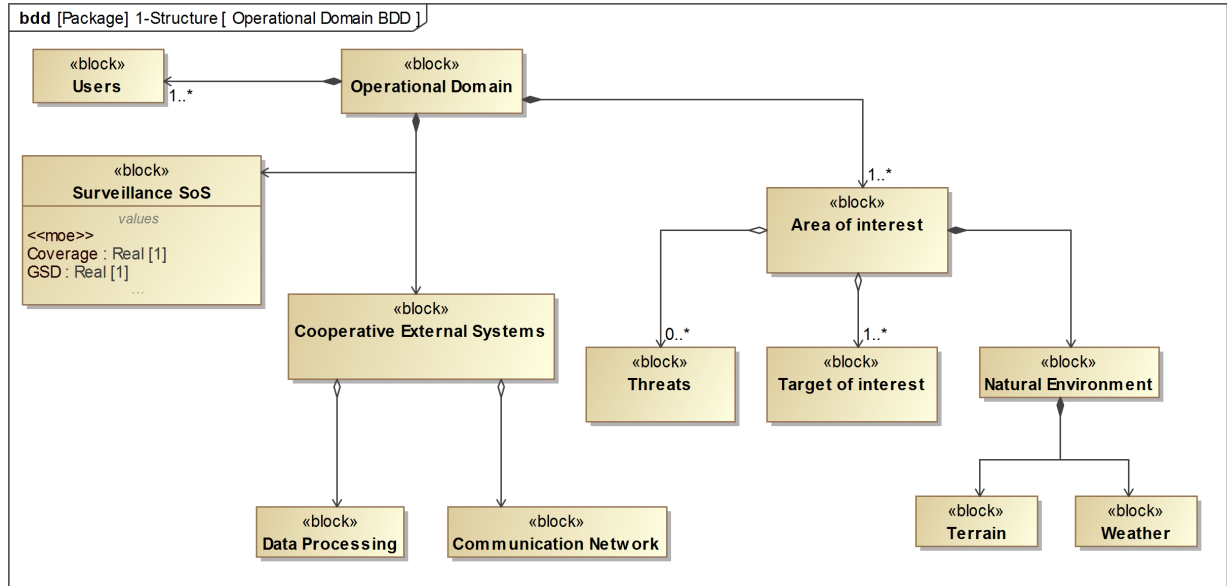


Figure 3. Example operational domain block definition diagram.

Define Operational Architecture.

The define operational architecture activity decomposes and partitions the SoS into logical components that interact to satisfy the requirements. The Surveillance SoS defined in the first development activity has been further decomposed and is shown in Figure 4. Only systems that collect imagery in the air and space domain are considered in this architecture. Other forms of data collection, communication, data processing, and command and control systems could also be considered here but fall outside the scope of this research. The logical components shown in Figure 4 are based around satellites and aircraft using imaging sensors. The space domain component of the SoS is composed of zero, one, or many constellations. A constellation is composed of one or many sensors. Satellites have not been modeled here because the main priority is where the sensors will be placed. If a satellite block were added, then value properties such as a slew rate could be defined. Multiple constellations can be used to address the concept of disaggregated space systems. Disaggregated space systems are not considered in this research, but they could be considered in future research.

The air domain component of the SoS is modeled similar to the space domain. There can be either zero, one, or many aircraft formations composed of one or many sensors. Each aircraft in a formation has the same sensor or sensors and follows a similar flight path as the other aircraft. A more detailed explanation of the aircraft formation will be given in Chapter IV. It should also be noted that the SoS can have zero satellite constellations or aircraft formations. This is to allow the possibility of using a single domain solution and will also be discussed later. An internal block diagram could also be used in this OOSEM activity to show the relationship and information exchanged between the air and space systems, however, one is not

created because the communication and integration of the data from both systems falls outside the scope of this thesis. The value properties chosen to describe a satellite constellation, aircraft formation, and sensors will be described next.

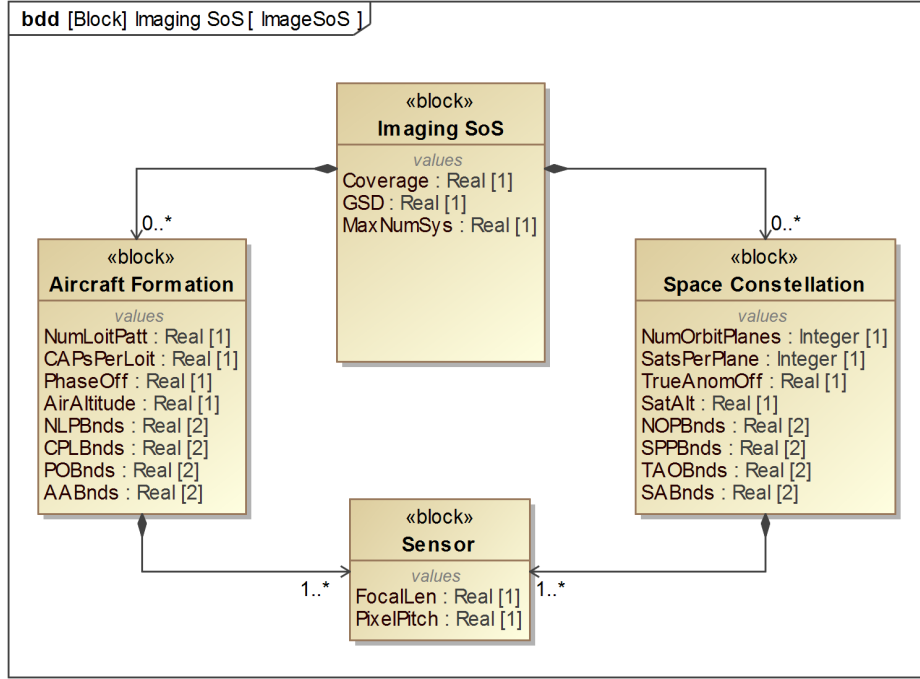


Figure 4. Surveillance SoS bdd

Satellite Constellation Parameters.

Figure 4 shows that a satellite constellation is described by the number of orbit planes, number of satellites per orbit plane, satellite altitude, and the true anomaly offset between orbit planes. Inclination, eccentricity, right ascension of the ascending node (RAAN), and argument of perigee could also have been defined but were not to limit the complexity of the problems described in Chapter IV. Only circular orbits are used for this research which make the eccentricity and the argument of perigee both zero. The inclination is defined as the angle between the equatorial plane and the orbit plane. The semi-major axis corresponds to the radius

of a the orbit plane for circular orbits. This can also be described as the altitude above the earth if the radius of the earth is accounted for elsewhere. The orbit planes are distributed equally around the earth based on the number of orbit planes in the constellation. The satellites are also distributed equally throughout an orbit plane based on the number of satellites per orbit plane. The true anomaly offset between orbit planes is best shown in Figure 5. For illustration purposes, the orbital planes are spaced close together and a true anomaly offset of eight degrees was applied between each orbit plane.

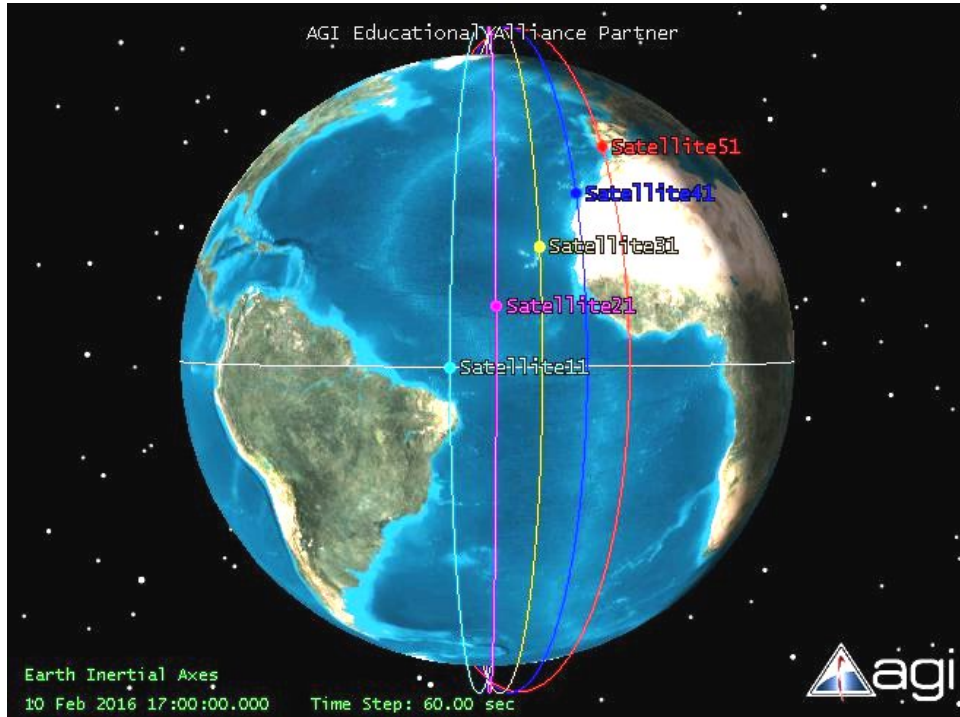


Figure 5. True anomaly offset example

Aircraft Formation Parameters.

An aircraft formation is defined by four parameters which are the number of sub-boxes an area of interest has been divided into, the number of combat air patrols (CAPs) per sub-box, the loiter phase offset between sub-boxes, and the

aircraft operating altitude. A CAP will consist of three aircraft that work together to provide a constant capability over a twenty-four hour period. Each aircraft will operate for eight hours before being replaced by another aircraft from the same CAP. The area of interest was defined as a square that could be divided into either one, four, or sixteen sub-boxes. A circular loiter was defined for each sub-box and the diameter of the loiter will vary with the size of the sub-box. Figure 6 shows an area of interest divided into four sub-boxes and four CAPs per sub-box. Figure 6 also shows an example of a 45 degree loiter phase offset between sub-boxes. The aircraft are also evenly spaced around the loiter pattern depending on the number of CAPs per loiter pattern.

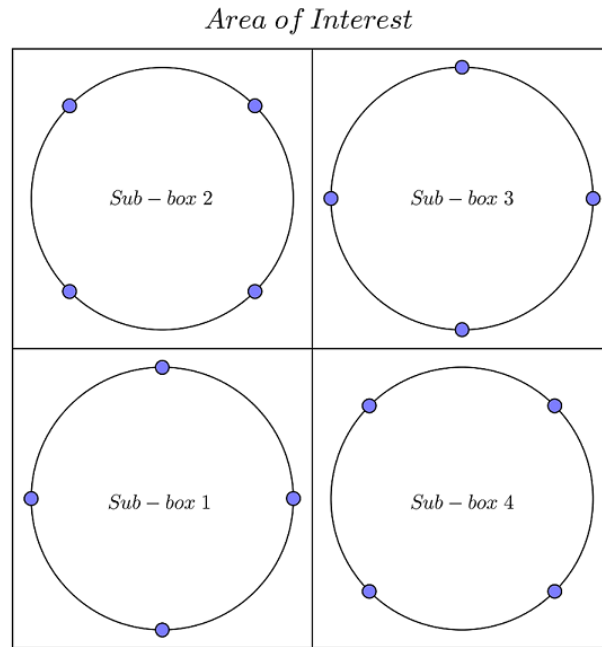


Figure 6. Example Aircraft Formation

Sensor Parameters.

The satellite constellation and aircraft formation use a visible imaging sensor that can be defined by many parameters. Only the sensor focal length and the sensor

detector pixel size are needed because the performance metrics chosen for this research do not require any other parameters. The actual values used for the aircraft sensor and satellite sensor are described in detail later. It should be noted that the architecture shown in Figure 4 shows that one or more sensors can be on a constellation or formation. More than one constellation or formation would need to be defined to create a heterogeneous system. This topic will be discussed as part of the recommendation for future work.

Synthesize SoS Architectures.

The synthesize SoS architectures development activity is where the physical implementation of logical components is specified. This is difficult though, because optimal SoS architectures will vary depending on external factors. A single physical implementation cannot be used to represent all scenarios or combination of external factors. Several external factors were identified in this OOSEM activity and grouped into three categories: environmental, degradation, and kinetic.

Environmental factors are caused by nature and limit the performance of a system, such as clouds or mountains that limit the performance of a satellite or aircraft imaging system. Degradation factors are man-made factors that limit or delay system performance such as a communication jamming that may delay data transfer, but it does not deny data collection. Kinetic factors are man-made factors that deny a system from performing its mission, such as missiles. External factors are considered as either present or not present, which results in eight possible scenarios that the SoS could be expected to operate in. Each scenario will have an optimal physical implementation that may or may not be the same for another scenario. For this research a single SoS architecture was chosen to start with and can be expanded upon in future research. The architecture chosen simply consists of

a single constellation of satellites and a single aircraft formation. The details of the systems will be described in depth in Chapter IV.

Analyze System Requirements.

The analyze system requirements activity captures the as-is systems capabilities, limitations, and areas of improvement. The external factors identified in the previous activity were used to help determine the capabilities and limitations of the SoS architecture. This information is captured and shown in Table 1. This activity highlights the differences between the two domains, but it does not suggest that either one is more capable than the other for all of the scenarios. This also supports the notion that a multi-domain solution will perform better across all scenarios than either single domain solution would alone. An example where this activity might favor a different architecture would be if the environmental factors were the only external factors considered and the area of interest was relatively small. Aircraft would then appear favored over satellites because aircraft are able to mitigate the environmental effects by varying their flight path and altitude compared to satellites that are fixed in their orbits.

Optimize and Evaluate Alternatives.

The optimize and evaluate alternatives activity uses parametric diagrams and external tools to evaluate SoS architectures. An executable architecture was developed for this OOSEM activity and is discussed in Section 3.2. Attributes of the SoS architecture and parameters with bounds are passed to an optimization model. The objective function used by the optimization model to evaluate candidate SoS architectures creates simulations and calculates performance metrics based on data from the simulation and the architecture. The performance metrics are then passed

Table 1. External factor domain effects

	Air Domain		Space Domain	
	Capabilities	Limitations	Capabilities	Limitations
Environmental Factors	Capable of varying altitude or flight path to accomplish mission	Altitude may be restricted; Access to area of interest may be limited or denied by factors like storms, temperature, or terrain	Capable of global coverage and access	Imagery limited or denied
Degradation Factors	Capable of varying altitude of flight path to avoid factors	Possible loss of precision navigation, delayed communication	More difficult for degradation factors to have an effect	Possible loss of precision timing, delayed communication
Kinetic Factors	Attributable solutions may still be capable of performing mission	Highly susceptible to kinetic effects	Less susceptible to kinetic effects	Kinetic factors exist that are capable of effecting the space domain

to a value model which returns an overall SoS architectural value to the optimizer.

The process of creating simulations, evaluating performance, and determining a value is repeated multiple times depending on the optimizer used. Once the optimizer has found an optimal solution, the parameters associated with that solution are passed back to the SoS architecture. The optimization method used must be selected carefully to avoid either becoming trapped in local minimums or not converging in a feasible amount of time. The optimizer chosen and the reasoning for its selection will be described in the optimization model subsection of Section 3.2.

Validate and Verify.

The last development activity is to Validate and Verify the SoS meets stakeholder needs and satisfies the requirements. For this research, this activity consists of reviewing the simulation and calculations performed by the executable architecture and ensuring that constraints were not violated. It is difficult to know if a global

optimum solution has been found because of the large number of candidate solutions and the nature of the sample space. The confidence in the solution can be increased by modifying the optimization parameters, but at the expense of longer calculation times.

3.2 Executable Architecture

An executable architecture (EA) was created to support OOSEM by evaluating SoS concepts and returning an optimal solution. The EA is composed of five interrelated models which are architectural, simulation, performance, value, and optimization. The model packages and relationships between them can be seen in Figure 7. It should be noted that this EA could be tool independent, but currently there are only a limited number of tools that can integrate together and have the required functionality for this research. The tool architecture used for this thesis will be described in detail in Section 3.3. The conceptual design problem proposed in the previous section will be used for examples in this section as well.

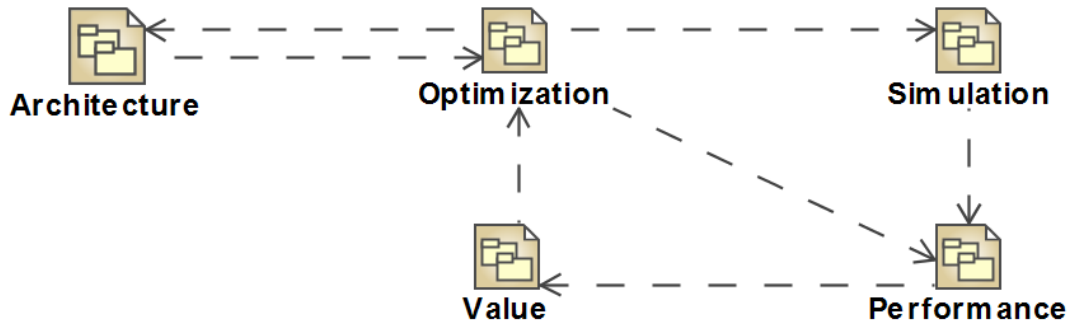


Figure 7. Five-model architecture

Architecture Model.

The architecture model package contains the documents and diagrams describing a physical and logical design of the SoS. This model is the start and end point of the EA. It starts by invoking the optimization model and passing parameters and bounds like variables into a function. After the optimization model has finished, it will return optimal parameters to the architecture model with which to update the SysML diagrams. The contents of the architecture model have been described in the previous section and include block definition diagrams like in Figures 3 and 4 as well as parametric diagrams. Parametric diagrams will be shown in Chapter IV and simply use value properties from blocks in the block definition diagrams as inputs and outputs to parametric equations.

Simulation Model.

The simulation model is used to simulate the physical SoS architecture and the complex dynamics that occur between objects. The simulation model is invoked by the optimization model but outputs results to the performance model. The objects included in the simulation are satellites, aircraft, sensors, an area of interest, and a ground vehicle (target). Parameters defining each object are passed to the simulation model from the optimizer. The outputs are in the form of text files that are read in by the performance model. The outputs are also computed in one-second time steps throughout the entire simulation time period. For twenty-four hours, this results in 86,400 discrete time steps.

For this research the area of interest and ground vehicle do not vary, so their parameters have actually been stored in the simulation model for simplicity and are not passed with the other parameters. The simulation is used to compute the

location of objects relative to one another. Specifically the azimuth, elevation, and range from the satellites or aircraft to the ground vehicle. This is done only when there is direct line of sight between the objects. Depending on the area of interest, terrain can be factored into this calculation to make the model more accurate.

Coverage of the area of interest is also calculated by the simulation. This calculation is slightly more complicated than it first appears. A satellite with a given sensor will have some maximum off nadir pointing angle that it can view before the GSD grows and becomes unusable. This angle will vary depending on the satellite altitude but will maintain the shape of a simple cone. For this research, the area a satellite can view while maintaining the required GSD will be defined as the coverage area. This also assumes that the satellites will be capable of slewing at a certain rate. The slew rate could be factored into the simulations to provide a more accurate model of the system, but this will be left for future research. The coverage provided by the aircraft is calculated a similar way and will be described in the next section.

Performance Model.

The performance model package contains the functions used to calculate applicable measures of performance, which are being treated as the measures of effectiveness for this research. Data for these calculations are provided by the simulation and architecture model packages and the results are passed to the value model for interpretation. For this thesis only two measures of performance are considered, but more could be incorporated in future research. The two measures of performance are the ground sampling distance (GSD) and the area of interest coverage calculation, which are both discrete functions of time with one second time steps. Other functions that could be considered might include how long data transmission takes or the quality of a RF signal received over time. The formulation for image

quality and coverage are defined next.

Image Quality.

The National Imagery Interpretation Rating Scale (NIIRS) is used to express the quality of aerial imagery and the General Image Quality Equation (GIQE) is one method used to predict the NIIRS ratings of any visible sensor (Leachtenauer *et al.*, 1997). The GIQE is a function of ground sample distance (GSD), relative edge response (RER), signal-to-noise ratio (SNR), convolver gain (G), and edge overshoot (H). For this thesis, the GIQE is more complicated than necessary and will not be used. The same filtering can be done to imagery taken from any domain. GSD will change drastically however, depending on where the image is taken from. For this reason GSD alone will be used as the image quality metric. The calculation for GSD is shown in equation 1 and comes from the book *Space Mission Engineering: The New SMAD* (Wertz *et al.*, 2011:173,500-503).

$$GSD = \frac{R_s d}{f \sin(\epsilon)} \quad (1)$$

Where:

R_s = Slant Range

d = Detector Pixel Size

f = Focal Length

ϵ = Satellite Elevation Angle

The GSD calculation receives data from both the simulation model and the optimization model. Because the sensor parameters do not change over time, the

focal length and detector pixels size can be taken directly from the architecture into the GSD calculation. The simulation provides the distance between the satellite and target and the satellite off nadir look angle, η . The satellite elevation angle is then easily calculated using equation 2.

$$\cos(\epsilon) = \frac{\sin(\eta)}{\sin(\rho)} \quad (2)$$

where:

$$\sin(\rho) = \frac{R_e}{(R_e + H)} \quad (3)$$

η = Off Nadir Look Angle

R_e = Earth Radius

H = Satellite Altitude

Figure 8 shows the geometry between a satellite and a target. It should be noted that this calculation works for both satellites and aircraft. Aircraft GSD calculations often use a flat earth model which does not account for the curvature of the earth. Since aircraft are closer to earth this assumption will work, but for satellites it will not.

Coverage.

The coverage calculation will be the area a satellite or aircraft can view with the required GSD. This should not be confused with the sensor ground footprint. If a satellite or aircraft has full coverage of the area of interest it means the sensor can view any part of the area of interest and produce useful images. The amount of

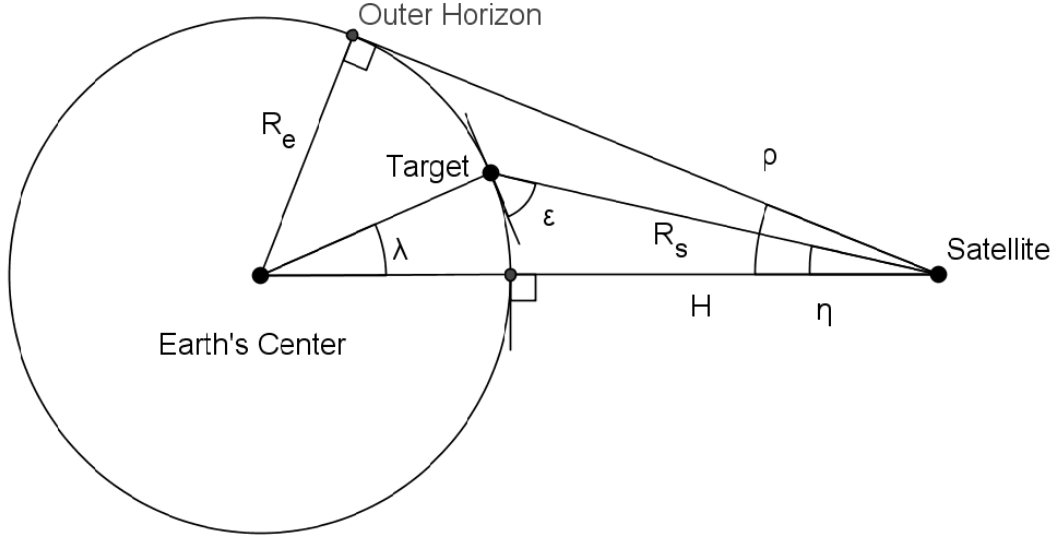


Figure 8. Earth Geometry

coverage a sensor ground footprint can produce will be left for future research along with the transfer and processing of the data. The simplest method for determining the maximum off nadir look angle is by calculating the GSD at nadir, and then incrementing the look angle and recalculating GSD until the GSD exceeds the requirement. The slant range in equation 1 is a function of the look angle and can be calculated using equation 4. Using the equation for slant range the GSD function in equation 1 can be rewritten as a function the look angle and is shown in equation 5.

$$R_s = \frac{R_e \sin(\lambda)}{\sin(\eta)} \quad (4)$$

$$GSD = \frac{R_e d \sin(\lambda)}{f \sin(\epsilon) \sin(\eta)} \quad (5)$$

Where:

$$\lambda = 90 - \epsilon - \eta$$

Table 2 shows the maximum off nadir look angle for satellites and aircraft at various altitudes using equation 5. The satellite sensor parameters were calculated using the published specifications of GeoEye-2, which is a 0.34 meter GSD at a 681 km orbit pointing nadir (Satellite Imaging Corp, 2016). Using the GeoEye-2 published values in equation 1 results in a detector pixel size to focal length ratio of $4.99e^{-7}$. It is known that the size of this satellite is very large and a constellation of satellites similar to this may not be feasible. The reason for choosing this sensor was to provide an accurate imaging sensor best case scenario using current proven commercial technology. The aircraft sensor parameters were taken from a real sensor designed for use on a cubesat. The reason this was chosen for the aircraft is because it gives a conservative estimate for current capabilities (Haywas, 2016:22). The pixel pitch is $2.2 \mu m$ and the focal length is $65 cm$.

Table 2. Maximum off nadir look angles for at least 1 m GSD

Satellites		Aircraft	
Altitude (km)	Max. Look Angle (deg)	Altitude (km)	Max. Look Angle (deg)
200	67	12	78
300	62	13	77
400	58	14	77
500	54	15	77
600	51	16	76
700	47	17	76
800	44	18	75
900	41	19	75
1000	39	20	74

Value Model.

The value model evaluates the results of the performance model to determine an overall concept score. Both GSD and coverage are calculated as discrete functions of time by the performance model and the results are written to text files for the value model to read from. There is one text file generated for the coverage of the area of interest, and there is one text file generated per satellite and aircraft for GSD. The

multiple GSD files are combined into one file by evaluating each time step and using the best GSD value of all aircraft and satellites. A simple linear value curve is applied to the GSD value at each time step, which is described by equation 6 and is shown in Figure 9. The value curve assigns full value anytime the GSD is below twenty centimeters and no value anytime the GSD is above one meter. The maximum and minimum GSD value requirements came from a report describing the required GSD to detect and identify an automobile (Knowlan & Phillips, 2007:12). Coverage is output from the performance model as a percent of the area of interest covered at each time step. The value of the percent covered is described by equation 7 and shown in Figure 10. An overall concept score is calculated by summing the equal weighted ($\alpha = 0.5$), time averaged ($T = 24$ hours) value functions of GSD and coverage and is shown in equation 8. This model provides a value for all concepts between zero and one. A concept that is capable of scoring a perfect score would have to have constant coverage and less than or equal to a twenty centimeter GSD on the target for the entire time period.

$$v_1(GSD(\bar{\mathbf{x}}, t)) = \begin{cases} 1, & GSD \leq 0.2 \\ -1.25 * GSD(\bar{\mathbf{x}}, t) + 1.25, & 0.2 \leq GSD \leq 1 \\ 0, & 1 \leq GSD \end{cases} \quad (6)$$

$$v_2(Cov(\bar{\mathbf{x}}, t)) = \frac{PercentCoverage}{100} \quad (7)$$

$$V(\bar{\mathbf{x}}, t) = \frac{1}{T} \sum_{t=1}^T \alpha * v_1(GSD(\bar{\mathbf{x}}, t)) + (1 - \alpha) * v_2(Cov(\bar{\mathbf{x}}, t)), \quad \alpha = 0.5 \quad (8)$$

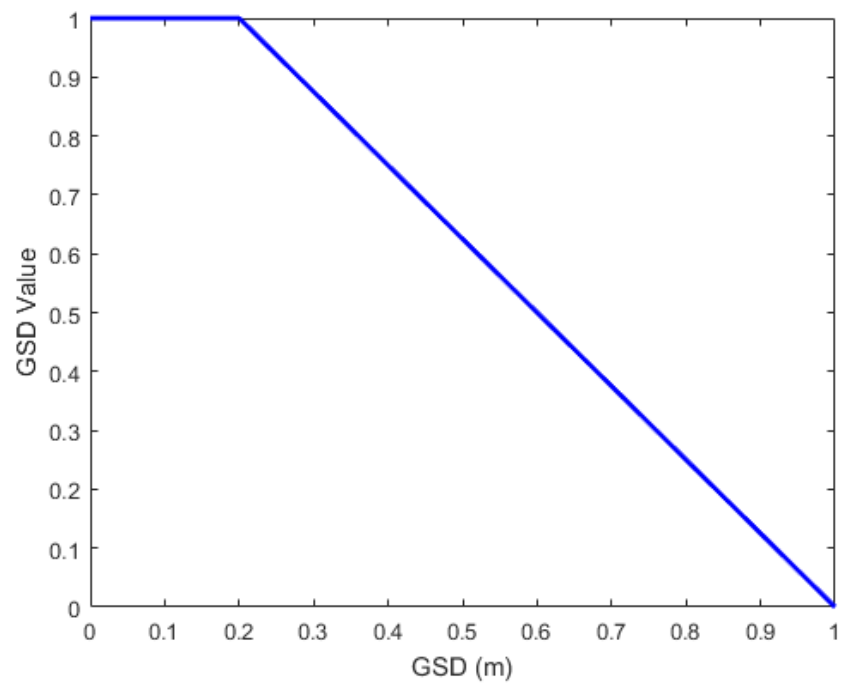


Figure 9. GSD Single Attribute Value Function

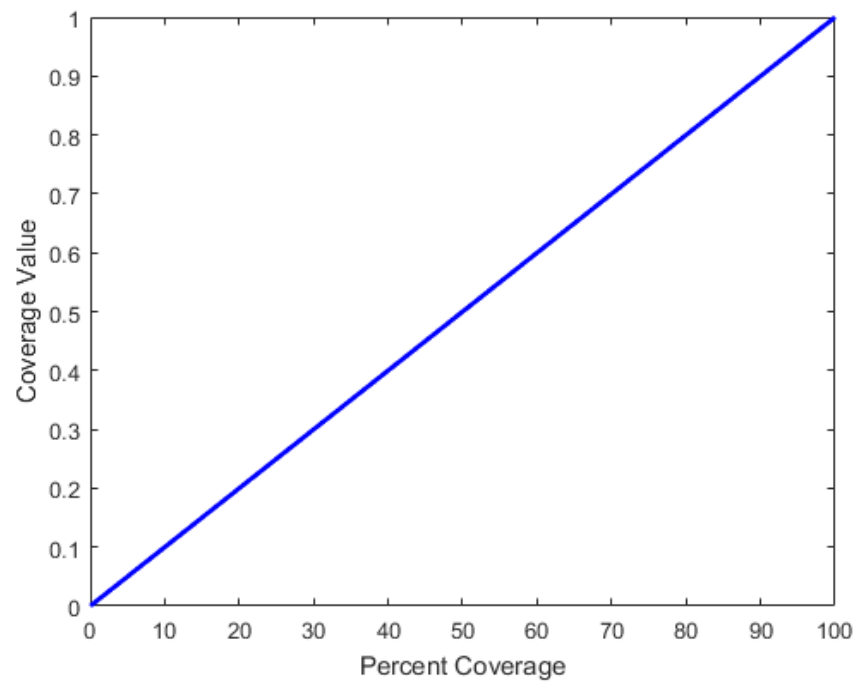


Figure 10. Coverage Single Attribute Value Function

Optimization Model.

The optimization model is the central component for the EA and can implement various optimization methods. The objective of the optimizer is to minimize the objective function by varying architecture parameters subject to constraints. The architecture parameters and constraints are passed to the optimization model as either constants or bounded parameters. The general optimization problem is shown in equation 9.

Minimize

$$f(\bar{\mathbf{x}}) \tag{9}$$

subject to:

$$h_i(\bar{\mathbf{x}}) = \mathbf{0}, i = 1, \dots, p$$

$$g_j(\bar{\mathbf{x}}) \leq \mathbf{0}, j = 1, \dots, m$$

$$x_k \in D_k, k = 1, \dots, n_d$$

Where:

$f(\bar{\mathbf{x}})$ is the objective function

$h_i(\bar{\mathbf{x}})$ is the p equality constraints

$g_j(\bar{\mathbf{x}})$ is the m inequality constraints

$\bar{\mathbf{x}}$ is the vector of n_d design variables

D_k is the discrete set of the k^{th} design variable

The design variables are defined as a discrete set rather than a continuous set to keep the design space bounded and countable. Eight design variables were chosen for this research and are shown in equation 10. The bounds and steps for each will be described in Chapter IV. Since the design variables were bounded and discrete, the only constraint applied was an inequality constraint on the maximum number of systems allowed. This constraint was varied and is described in Chapter IV as well. For this research the objective is to maximize the value returned from the value model. The equivalent objective, in the standard form of the general optimization problem, is to minimizing the negative of the value returned from the value model. The general optimization problem standard form used for this research is given in equation 11.

The optimization techniques can vary from using mathematical methods to heuristic and stochastic methods. It should be noted that a fundamental assumption of gradient based optimization methods is that the design variables are continuous within their bounds. Since the design variables x_1 , x_2 , x_5 , and x_6 can only be integers a non-gradient based optimization method must be used. A Genetic Algorithm (GA) was selected for this research based upon the tool availability. Chapter V will discuss future work using different optimization method and combinations of methods.

$$\bar{\mathbf{x}} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} \text{Number of satellite orbit planes} \\ \text{Number of satellites per orbit plane} \\ \text{True anomaly offset} \\ \text{Satellite orbit altitude} \\ \text{Number of CAP loiter patterns} \\ \text{Number of CAPs per loiter pattern} \\ \text{Phase offset between loiter patterns} \\ \text{Aircraft operating altitude} \end{bmatrix} \quad (10)$$

Minimize

$$f(\bar{\mathbf{x}}) = -V(\bar{\mathbf{x}}, t) \quad (11)$$

subject to:

$$g_1(\bar{\mathbf{x}}) = x_1 * x_2 + x_5 * x_6 * 3 - \text{Max. Number of Systems} \leq 0$$

Figure 11 shows the optimization process used by the model. The GA starts by creating an initial population of individuals. The number of individuals in a population is one of the many optimization parameters that can vary and can be investigated in future work. Each individual in the population is described by random values of the variables from Equation 10. For the multi-domain problem that will be shown in Chapter IV there are over five million possible combinations of these variables. After the population has been created the fitness of each individual is evaluated. For a population of 50 this means 50 simulations are created and

assessed. After all the individual fitnesss have been calculated, the constraints are checked to determine if any individual violates a constraint. If an individual does violate a constraint, a penalty function is applied to the individual's fitness value to make it larger than the largest feasible individual fitness value. After the population fitness has been evaluated, the stopping criteria is checked to determine if another generation will be created. If the criteria was not met a new generation is created. The first step in creating a new generation is selecting the best five percent to be individuals in the next generation population. The remainder of the individuals for the next generation are chosen by generating a mating pool from the old generation and then using special crossover and mutation functions which are described by Deep (Deep *et al.*, 2009). The stopping criteria for the GA was after 10 generations had been created. To improve confidence the GA is performed more than once with a new random population. This was considered the global stopping criteria and is referred to in Figure 11 as the iteration loop. The best individual of all the iterations is used as the global best for the architecture parameters provided. It will be shown in Chapter IV that multiple architecture parameters are tested by changing the constraints on the maximum number of systems that can be used.

3.3 Tool Architecture

This section describes the tool architecture used to implement the EA described in Section 3.2. The tool architecture is composed of MagicDraw 18.1 from No Magic Inc., ParaMagic 18.0 from Intercax LLC, MATLAB R2015b from MathWorks Inc., Python 2.7, and Systems Tool Kit (STK) 10.0 from Analytic Graphics Inc. The relationship between the tools can be seen in Figure 12 and is described in the remainder of this section. The ability of the tool to communicate with external programs was a major factor for selection.

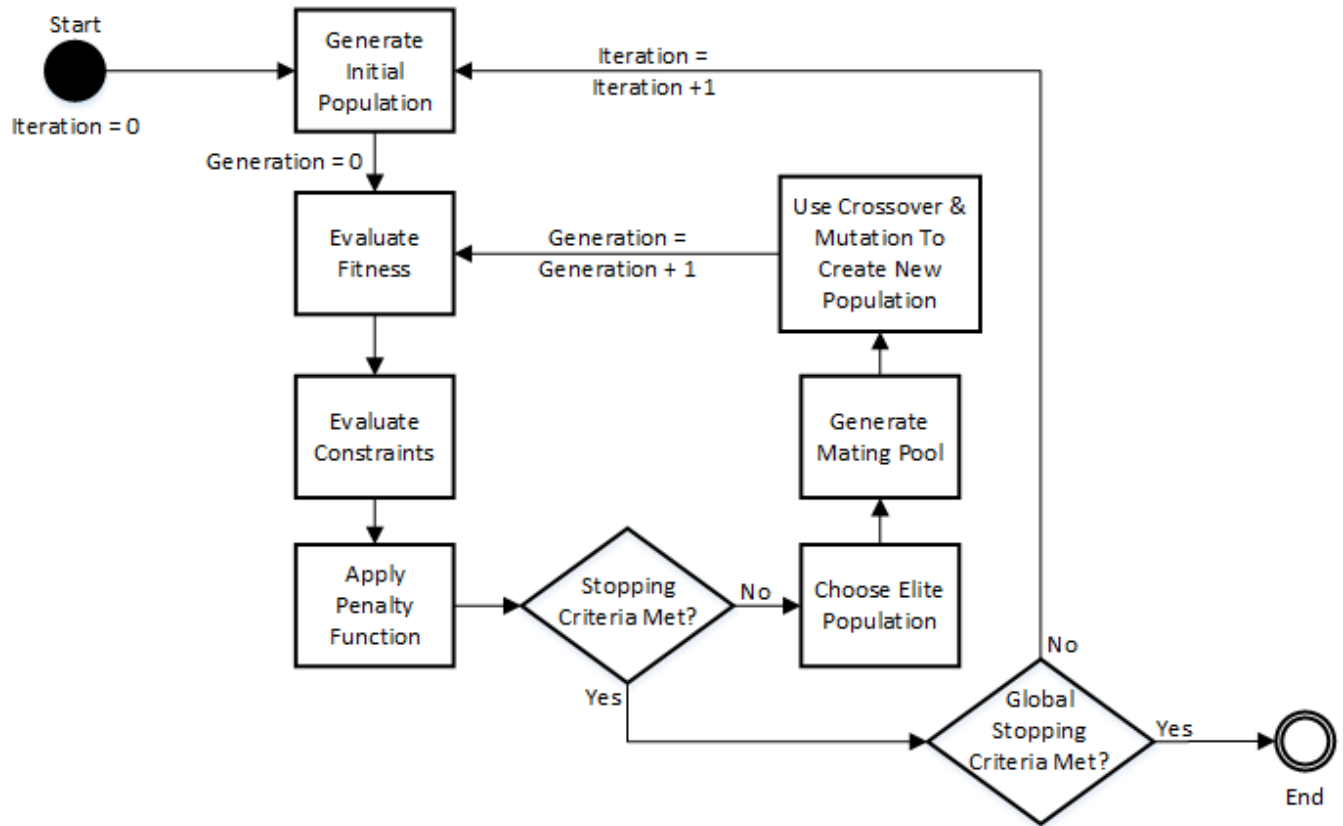


Figure 11. Optimization Process

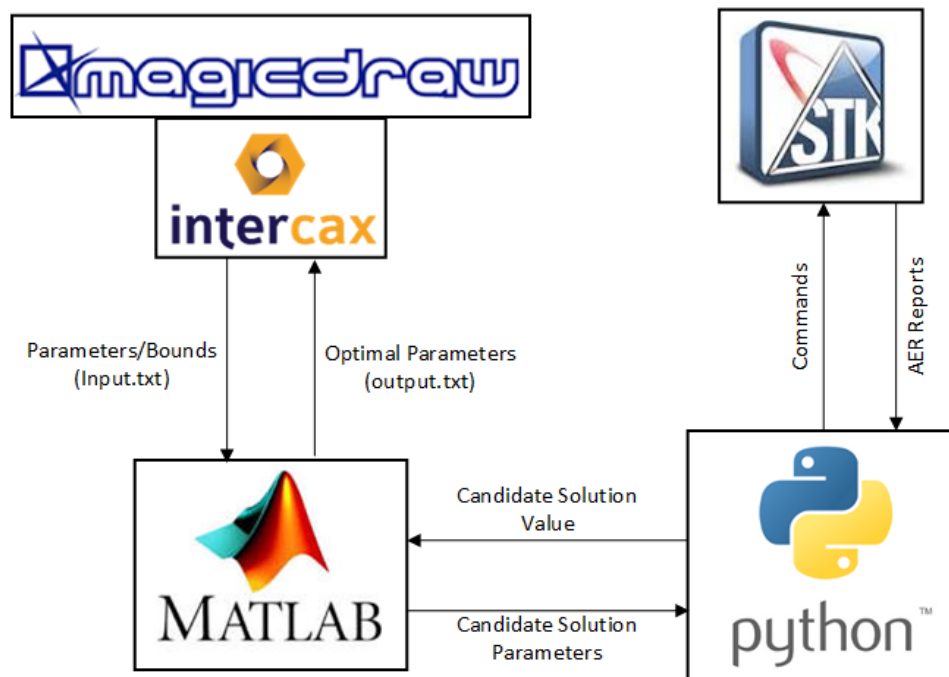


Figure 12. Tool Architecture

Magic Draw.

MagicDraw is a UML modeling tool that supports the Systems Modeling Language (SysML) as a plug-in module. The architecture model of the EA was created using this tool. MagicDraw supports many third party plug-ins, like ParaMagic, to extend its capabilities. The ParaMagic plug-in provides the capability to evaluate parametric equations using specific external solvers. One of the solvers ParaMagic is capable of using is MATLAB. ParaMagic passes variables to either a MATLAB script or function and is capable of receiving a value or an array of values in return. ParaMagic can then update the SysML diagram using the returned values. ParaMagic passes values to MATLAB by writing the values to a text file named *input.txt* and receives returned values through a text file named *output.txt*. ParaMagic provides a critical capability for the EA, however, it is currently limited by the choice of external solvers and the format used for passing variables.

MATLAB.

The EA uses MATLAB to link the architecture and optimization models as described in the previous section as well as actually implementing the optimization model. MATLAB supports many add-ons called toolboxes which contain functions that have been written and tested by MathWorks Inc. The Global Optimization Toolbox contains a GA function which is used for this research. One of the benefits the GA function provides is that it has the capability to use integer values as well as non-integer values. For this research only integers will be used, but future work can investigate using non-integer values. The values MATLAB receives from MagicDraw are the bounds and constraints used by the GA. The objective function used by the GA is a Python script and will be discussed next. MATLAB executes the GA by evaluating the objective function at different candidate solutions. The MATLAB

code used for this research can be seen in Appendix B.

Python.

Python is an Object-Oriented programming language that is used to implement the performance and value model of the EA as well as control the simulation model. It was originally thought that Python could also implement the optimization model, but a suitable optimizer for discrete sample spaces could not be found in the time required. Python is also responsible for controlling where the data from the simulations are saved. This is important because the sample space is on the order of millions and the time required to complete one candidate point is on the order of minutes. Saving the results for access later saves a lot of computational time. The Python code is described and shown in Appendix A.

Systems Tool Kit (STK).

STK is used to implement the simulation component of the EA and is controlled using the STK Connect module. Connect uses a TCP/IP connection to send commands to STK from an external application. The external application for this research is the Python script invoked by the GA. Commands sent using Connect are capable of performing almost all of the same functions the graphical user interface can perform. The benefit of using Connect is that creating scenarios can now be automated and performed much faster. This is necessary since the GA will evaluate hundreds of simulations per iteration. STK generates outputs in the form of text files that are read in by the Python script. This is how STK communicates back to Python as shown in Figure 12.

3.4 Methodology Summary

This Chapter has proposed a method of using OOSEM to develop an optimal SoS comprised of aircraft and satellites. An EA is developed to find optimal candidate solutions using a combination of simulations, performance metrics, value models, and optimization algorithms. The EA is used to perform the OOSEM activity optimize and evaluate alternatives. The tool architecture capable of supporting the EA is described and an example is given. The specific code is attached in appendices B, and A.

IV. Application

This chapter covers the application of the methodology proposed in Chapter III to a multi-domain SoS design problem with aircraft and satellites. First, the model will be verified by using a simple example. Next, the architecture created will be presented and then applied to three scenarios. A space only solution is presented first, followed by an aircraft only solution and then a combined solution consisting of both aircraft and satellites. The application and the results of each solution will be presented in this chapter. The analysis of the results and their significance will be discussed in Chapter V.

4.1 Model Verification

Emery et al. found that low earth orbiting satellites performing imagery collection should be placed in an orbit inclination that is equivalent to the northern latitude of the target theater (Emery *et al.*, 2005). This conclusion will be used to validate the EA is working as expected. The results of the GSD and coverage calculations will be used to verify the model is performing the correct calculations and the sensor parameters are realistic.

Parameters.

The architecture is kept as simple as possible for this verification problem. The area of interest, ground vehicle location, and sensors are all constants and are defined in the simulation code. The values for the ground vehicle and the bounds for the area of interest can be seen in Table 3. The sensor parameters were derived from the GeoEye-2 satellite specifications. The GSD calculation given by equation 1 has only three inputs if the satellite is looking nadir, range, focal length, and detector pixel

size. The range and GSD for GeoEye-2 are published at 681 kilometers and 0.34 meters respectively (Satellite Imaging Corp, 2016). This gives a ratio between the focal length and detector pixel size of $4.99e^{-7}$ that can then be used to calculate GSD at different altitudes.

Table 3. Constants used in verification problem.

Object	Latitude (deg N)	Longitude (deg E)
Ground Vehicle	34.05	41.2
Area of Interest	32-36	40-44

A single constellation with inclination bounds is all that is needed to model this problem. Figure 13 shows the constellation class that was created using MagicDraw. There are eight values defined in the *Space Constellation* class and only *InclinationBnds* is capable of containing more than one value. There is also one constraint property called *Opt_Inc*. A parametric diagram is used to relate the attributes used in the constraint function and is shown in Figure 14. The parametric equation defined inside the constraint block specifies what solver to use, and the input and output variables. The square boxes at the edge of the constraint block represent constraint parameters, of which value properties are assigned to. The output block has a binding connector to the inclination value property of the space constellation. The same type of connector is also used to connect the other seven value properties of the space constellation to the constraint block. The constraint parameters are treated as the variables to the parametric equation. The function *xfwExternal()* calls the external solver, MATLAB in this case. The second argument specifies if a MATLAB script or function is being used, and the third argument is the name of the MATLAB script or function. It should be noted that the default location of the MATLAB script or function must be set in the ParaMagic settings. The remaining arguments will be passed to MATLAB in a text file and they will be arranged in the order specified in the equation. This is

important because no other indicators are used to specify what the variables are.

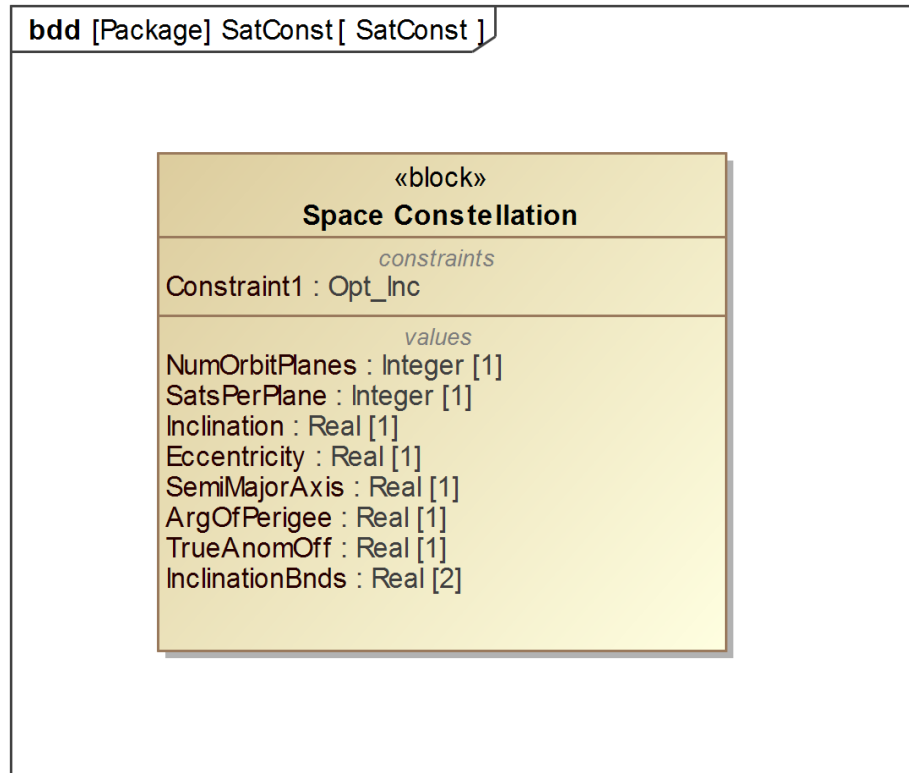


Figure 13. Constellation Class

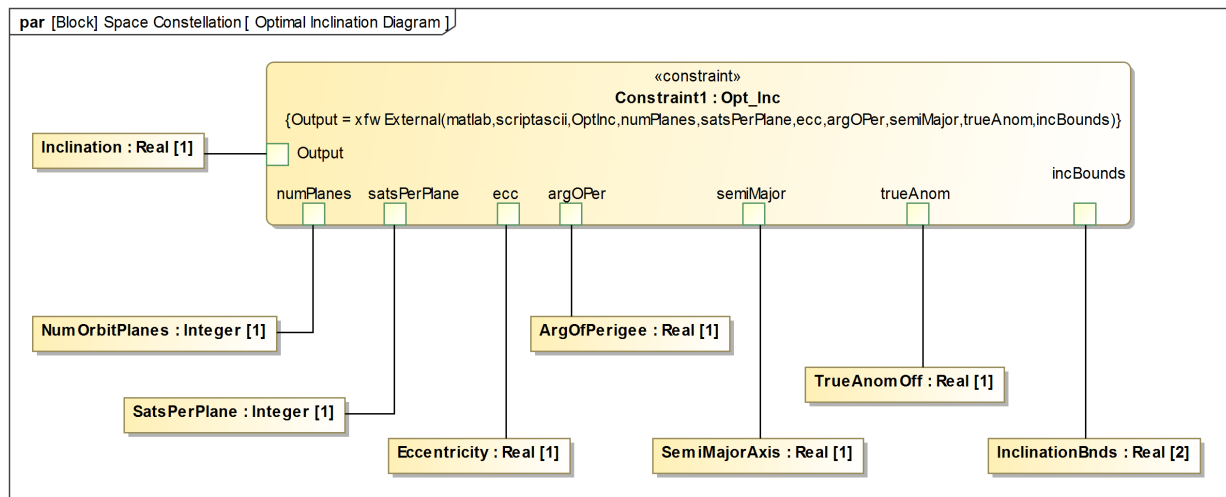


Figure 14. Optimal Inclination Parametric Diagram

The next architecture component created is called an *instance*. This is used to specify the Space Constellation values that will be used to solve the parametric equation. Figure 15 shows the values specified for this problem. The inclination value is left blank since the value will be solved for by the parametric equation. The inclination bounds are bounded between twenty and fifty degrees to limit the time required to find the optimal solution.

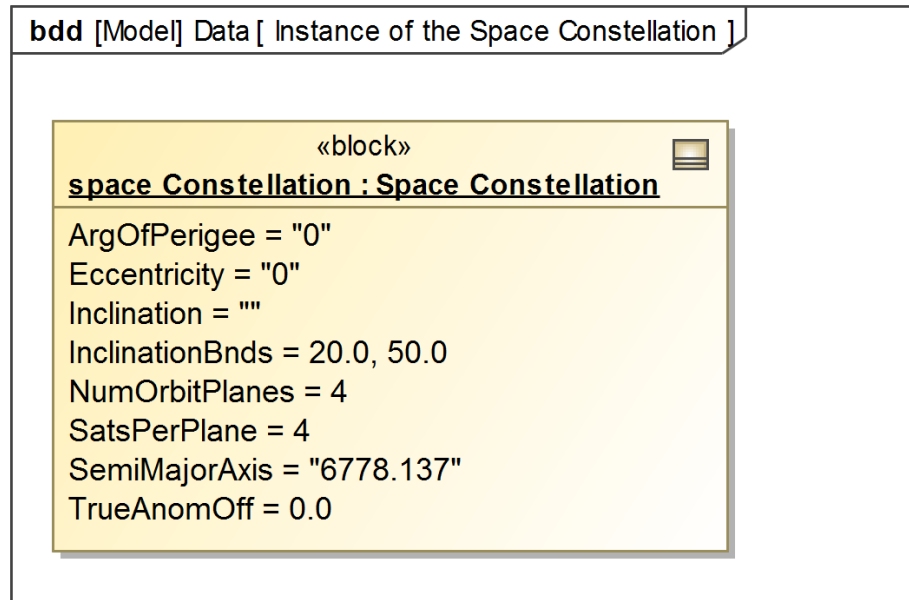


Figure 15. Instance of Space Constellation

The last step performed by the architecture is to invoke the optimizer to solve for the inclination. Figure 16 shows the ParaMagic window that is opened to solve for inclination. The inclination is set as the target and all other values are given. Pushing the solve button starts the EA and returns the optimal inclination.

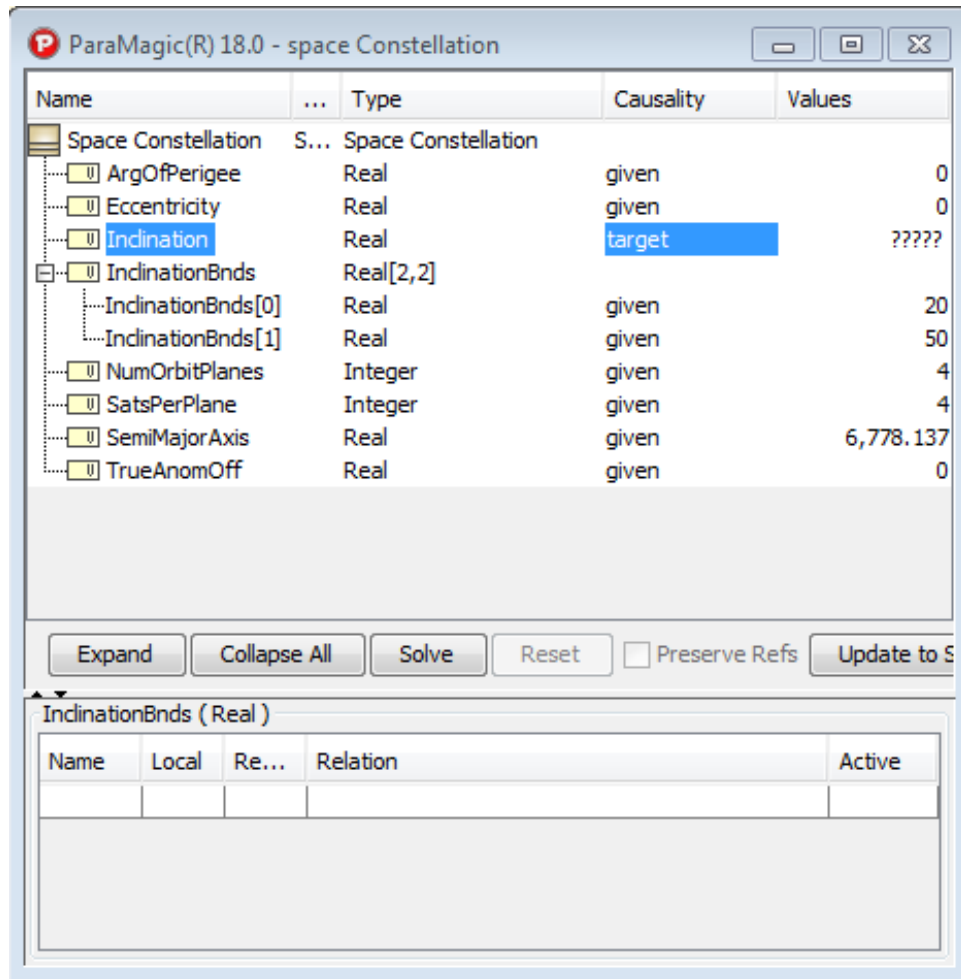


Figure 16. Solving for inclination using ParaMagic

Results.

OptInc.m uses the GA discussed in Chapter III to search for the optimal inclination. Discrete values with a step size of a tenth of a degree were used for the inclination set. The GA was set to use a population size of 20 and perform 15 generations. The GA was then repeated ten times to improve confidence of an optimal solution and verify the method would work for a larger sample space. Figure 17 shows the sample space for the problem and the candidate solutions that were evaluated by the GA. The optimal point is shown clearly in Figure 18 as 37

degrees. The optimizer converged on the optimal inclination of 37 degrees for all ten iterations. For a larger sample space this may not always be the case. The result was successfully returned to MagicDraw and updated in the architecture. The optimal inclination was expected to be equal to the northern latitude of the area of interest, 36 degrees north. The slight difference in results is attributed to the difference in altitude. The satellites Emery was working with operated at 250 kilometers. 400 kilometers was chosen for this example problem to remain closer to the operational altitude of GeoEye-2 for verification of the sensor performance.

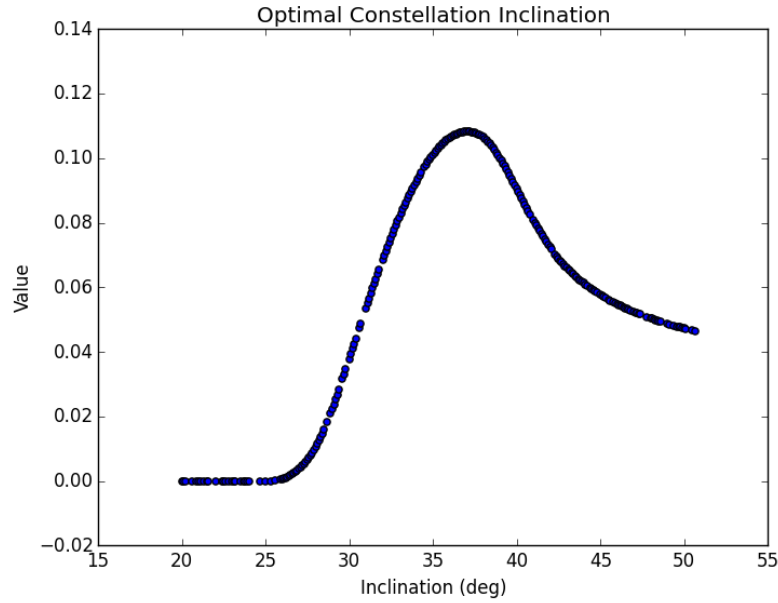


Figure 17. Value of candidate constellations at varying inclinations

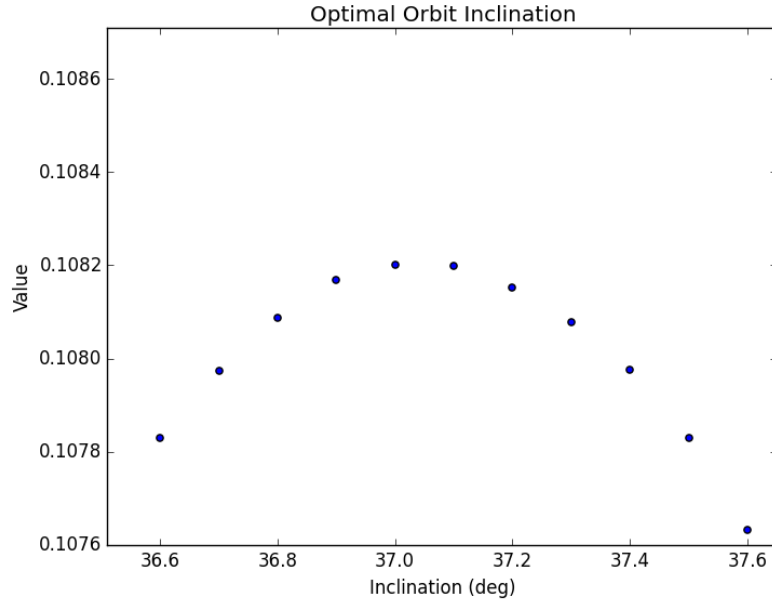


Figure 18. A zoomed in view around the optimal candidate constellation inclination

Figure 19 shows the GSD for the constellation at 37 degrees for a twenty-four hour period. The individual spikes are the times at least one satellite has access to the ground vehicle. The saddle shape formed by the peaks over a couple hours is attributed to the ground vehicle crossing directly under the orbit planes as the earth rotates. Since the ground vehicle latitude is below the satellite inclination, the ground vehicle will cross under the orbit twice; once as the satellite is ascending and once as it descends. Figure 20 shows an example of the low point of a saddle. As time moves forward the ground vehicle will move east and directly under the descending part of the orbit plane.

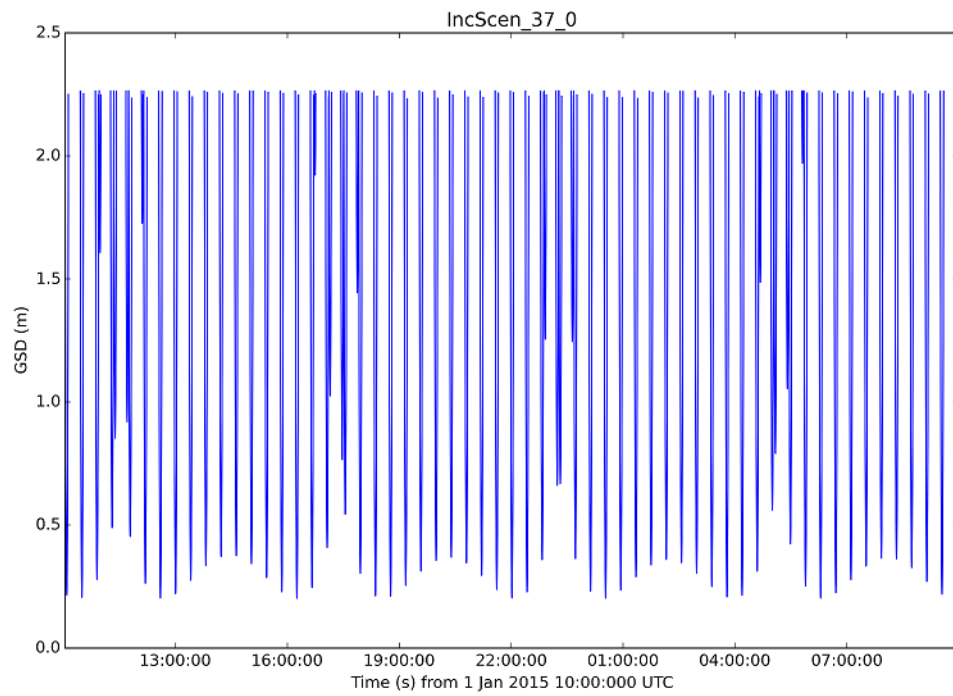


Figure 19. Raw GSD at optimal inclination over 24 hours



Figure 20. Optimal inclination orbit track relative to area of interest

The results also showed that no value was given when the GSD was greater than 1 meter. The minimum GSD achieved was 0.206 meters and resulted in a value of 0.9925. This GSD is realistic given the GeoEye-2 sensor parameters and the change in altitude. If a satellite passed directly over the ground vehicle and was exactly 400 kilometers away, the best achievable GSD would be 0.199 meters. Figure 21 shows the GSD value over the entire time period after the value function given in equation 6 has been applied.

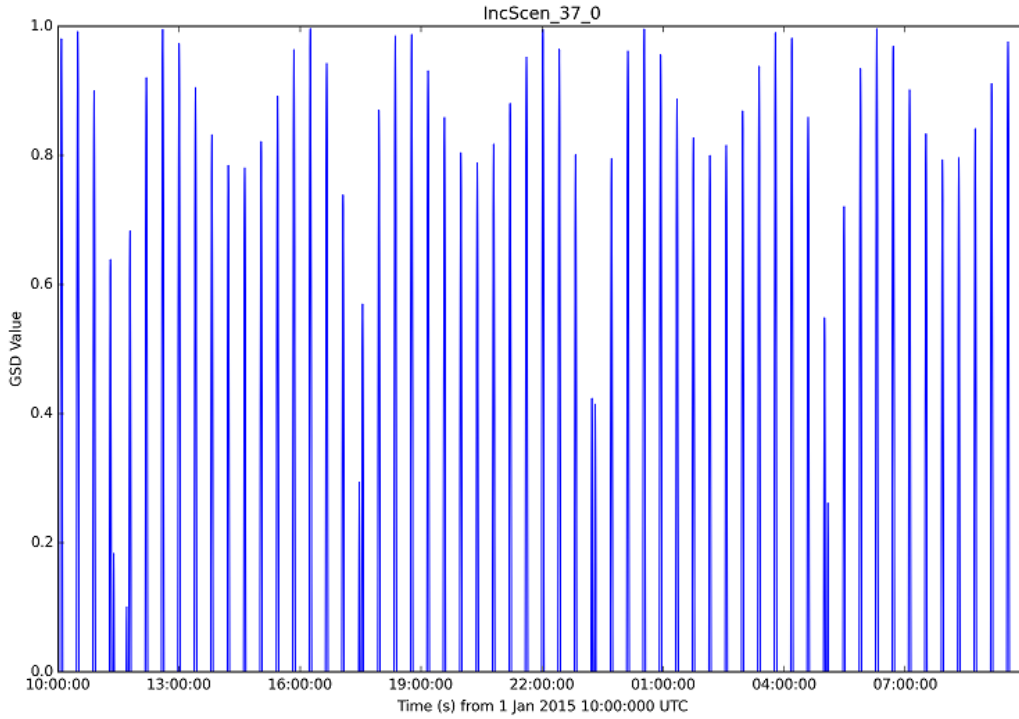


Figure 21. Post-value curve GSD at optimal inclination over 24 hours

The coverage calculations are also verified as realistic based on GeoEye-2. The coverage is determined by calculating the maximum off-nadir look angle that can be used and still achieve a GSD under one meter. As the off-nadir look angle is increased the curvature of the earth increases the GSD. GeoEye-2 has published a

maximum off-nadir look angle of 60 degrees. At the operating altitude of GeoEye-2 , the angle between nadir and the edge of the earth is 65 degrees. The maximum look angle should increase as the altitude is lowered because the angle between nadir and the horizon is also increasing and the curvature of the earth is less prominent. From a 400 km orbit the angle between nadir and the edge of the earth is approximately 70 degrees. For this example the satellites are at 400 kilometers and have a maximum off-nadir look angle of 58 degrees. This verifies the one meter GSD requirement is constraining the coverage model calculation. Figure 22 shows the value for the coverage of the scenario given by equation 7. Figure 23 shows the weighted combination ($\alpha = 0.5$) of GSD and coverage over the twenty-four hour period.

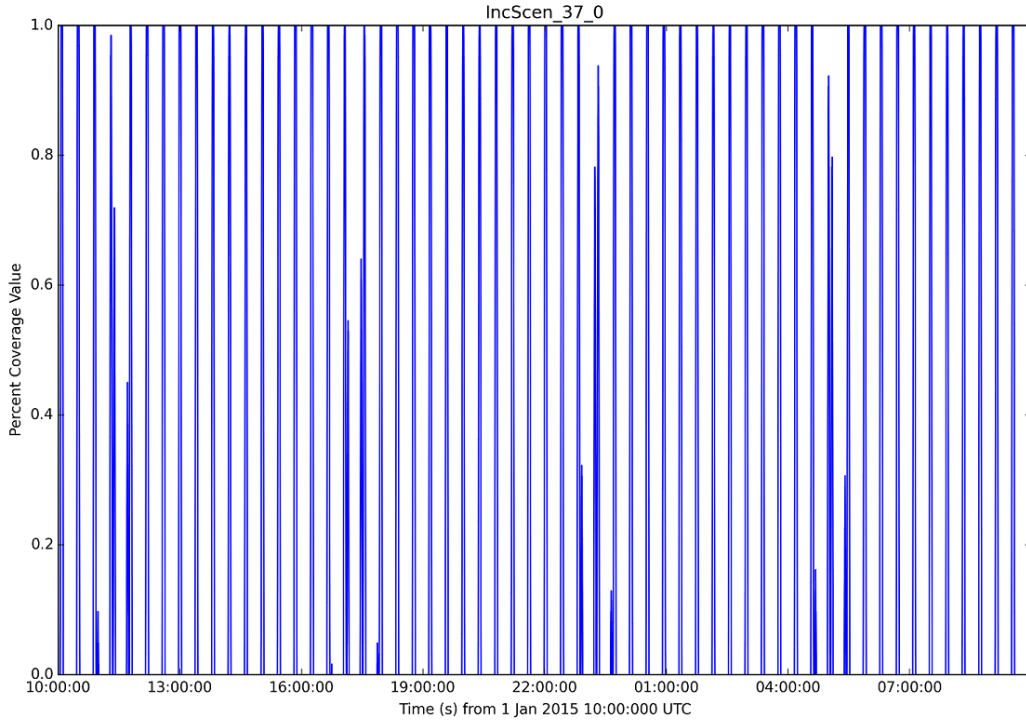


Figure 22. Coverage at optimal inclination over 24 hours

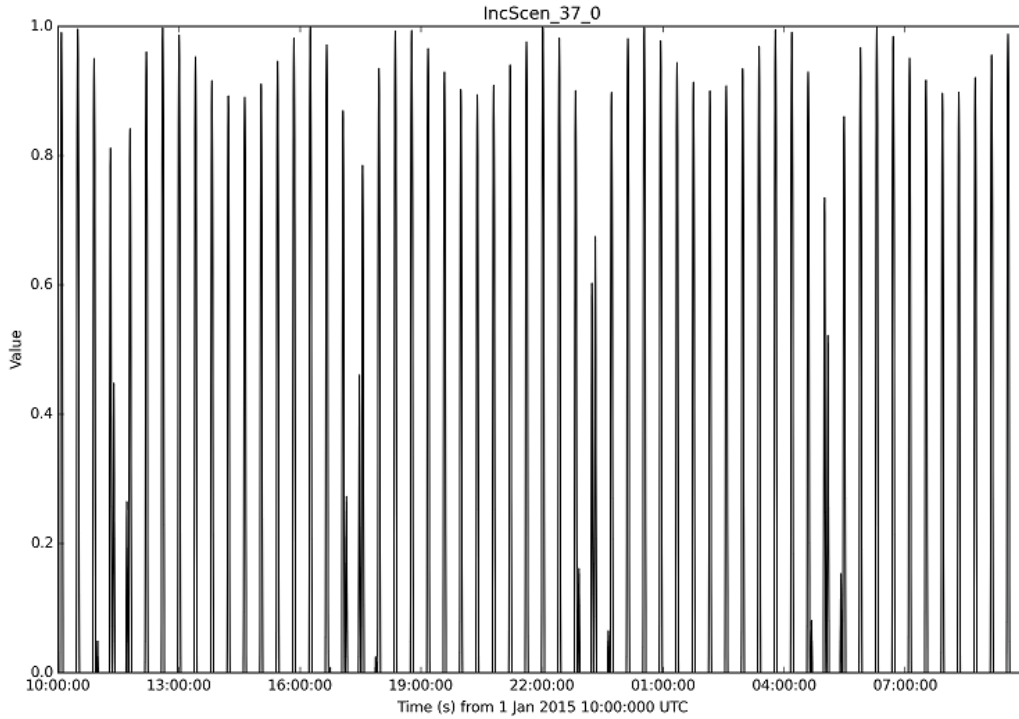


Figure 23. Value of system at optimal inclination over 24 hours

4.2 Space Solution

In this section, the methodology discussed in Chapter III will be applied to a space domain only design problem. The architecture created and parameters used will be presented first followed by the results.

Space Domain Architecture.

The operational domain was defined for the first OOSEM activity. A BDD was created to model the operational domain is shown in Figure 24. The operational domain consists of an Imaging SoS and an area of interest with a single ground vehicle. The area of interest is defined by latitude and longitude boundary points.

The ground vehicle has a location inside the area of interest defined by a latitude and longitude. Terrain and obstructions are modeled with a uniform elevation angle mask that is applied to each object inside the area of interest. The constraint function to run the EA is associated with the operational domain block. This is done because the constraint function can use attributes from both the imaging SoS and the area of interest.

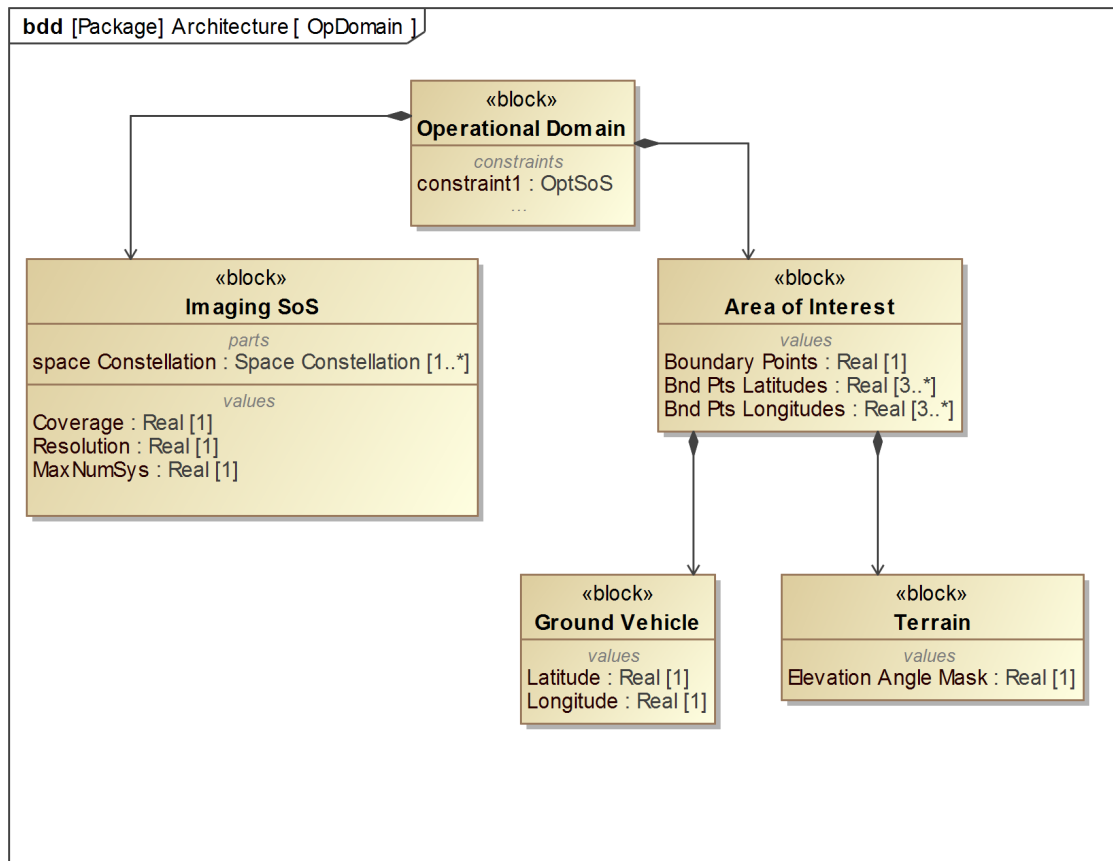


Figure 24. Space only operational domain

The operational architecture was defined for the second OOSEM activity. For this the imaging SoS was defined as being composed of one or more space constellations composed of one or more sensors. Constellations are traditionally composed of satellites which are composed of components like sensors, but they have not been

modeled here because the main focus is where the sensors will be placed. Multiple constellations would be used to model different orbits or different sensors within the same orbit. If a constellation were defined as containing multiple sensors, then it would be equivalent to saying multiple sensors are on each satellite in the constellation. Multiple constellations used to define different sensors in the same orbit would be equivalent to saying there are different types of satellites in the same orbit. Figure 25 shows the BDD for the space domain only imaging SoS. For this problem the value property *MaxNumSys* is the constraint that determines the maximum number of systems that can be used and *Coverage* and *GSD* are the best results found under the constraint. However, this problem could easily be reversed and the *Coverage* or *GSD* value properties could be the constraints and the *MaxNumSys* could be the resulting minimum number of systems needed to achieve the constraint values. The space constellation block has eight value properties. Four value properties are the bounds for the optimization problem and the other four properties are the optimal values that will be returned by the EA. The bounds properties are capable of holding a lower and upper value while the optimal value properties are capable of only holding single values.

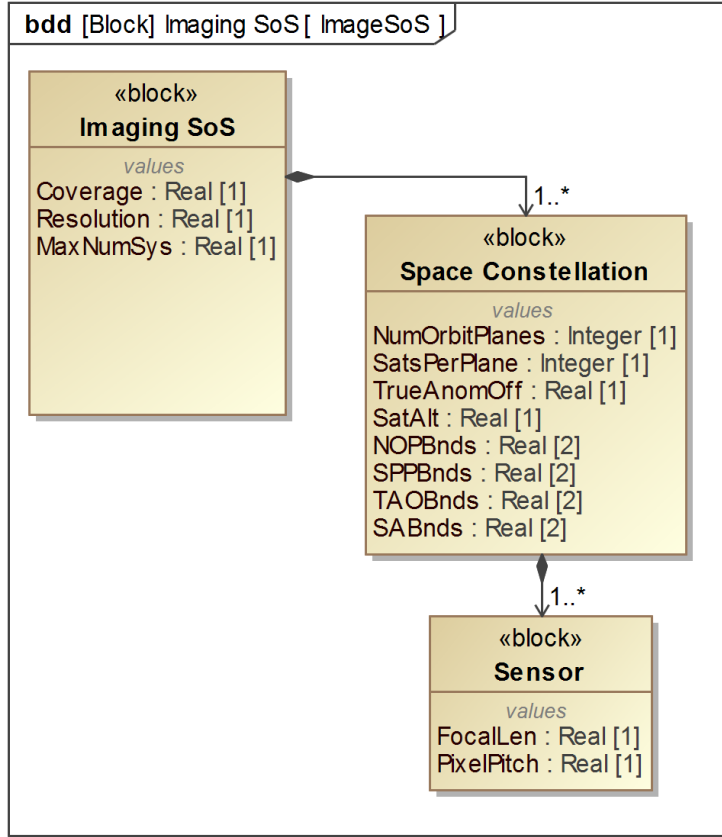


Figure 25. Space domain imaging SoS decomposition

The *Space Constellation* is parameterized by *NumOrbitPlanes*, *SatsPerPlane*, *TrueAnomOff*, and *SatAlt*. *NumOrbitPlanes* and *SatsPerPlane* are the number of equally spaced orbit planes around the earth and the number of satellites in each orbit plane respectively. *SatAlt* is the satellite orbit altitude in kilometers and *TrueAnomOff* is the true anomaly offset of satellites between orbit planes. An example of true anomaly offset is shown in Figure 5 where, for illustration purposes, the orbital planes are spaced close together and a true anomaly offset of eight degrees is applied between each orbit plane. The other four parameters, *NOPBnds*, *SPPBnds*, *TAOBnds*, and *SABnds* are simply the upper and lower bounds of each parameter.

The parametric diagram is shown in Figure 26 and shows the bounds and maximum number of systems constraint being passed into the constraint property *OptSoS*. The constraint property then uses the external solver MATLAB to run a script that uses the constraint parameters as inputs. The script returns an array of values that are assigned to the constraint parameter *Output*. Another constraint property is needed to assign the individual constraint parameters in the array to value properties outside the constraint property. This is the purpose of the *Parse* constraint properties shown in Figure 26. These constraint properties use the external solver MATLAB to evaluate a function with two inputs, the array and the index of the desired value in the array. The area of interest and sensor parameters were placed directly in the EA code because they do not vary for this research and would add unnecessary complexity.

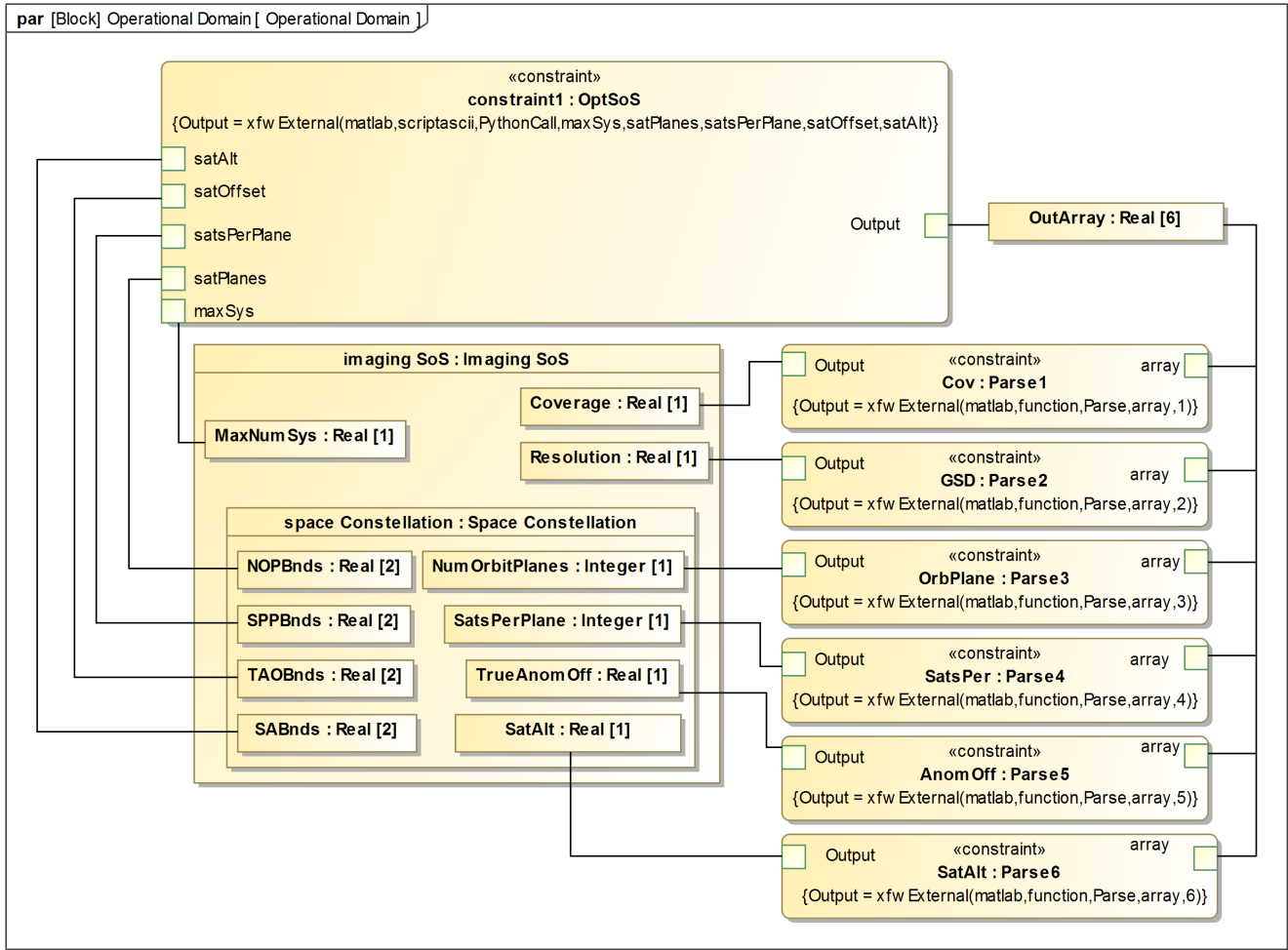


Figure 26. Space only domain parametric diagram

The next OOSEM activity performed was to synthesize a candidate SoS architecture. Figure 27 shows the candidate SoS architecture that was created for the space only design problem. This architecture consists of a single satellite constellation with the same sensor on each satellite. The number of orbit planes and satellites per orbit plane is bounded from one to nine. The altitude is allowed to vary from 200 to 1000 kilometers and the phase offset between planes is from zero to 45 degrees. The maximum number of systems shown in Figure 27 is set to 81. This constraint will be varied however to show how value changes with respect to the number of systems used.

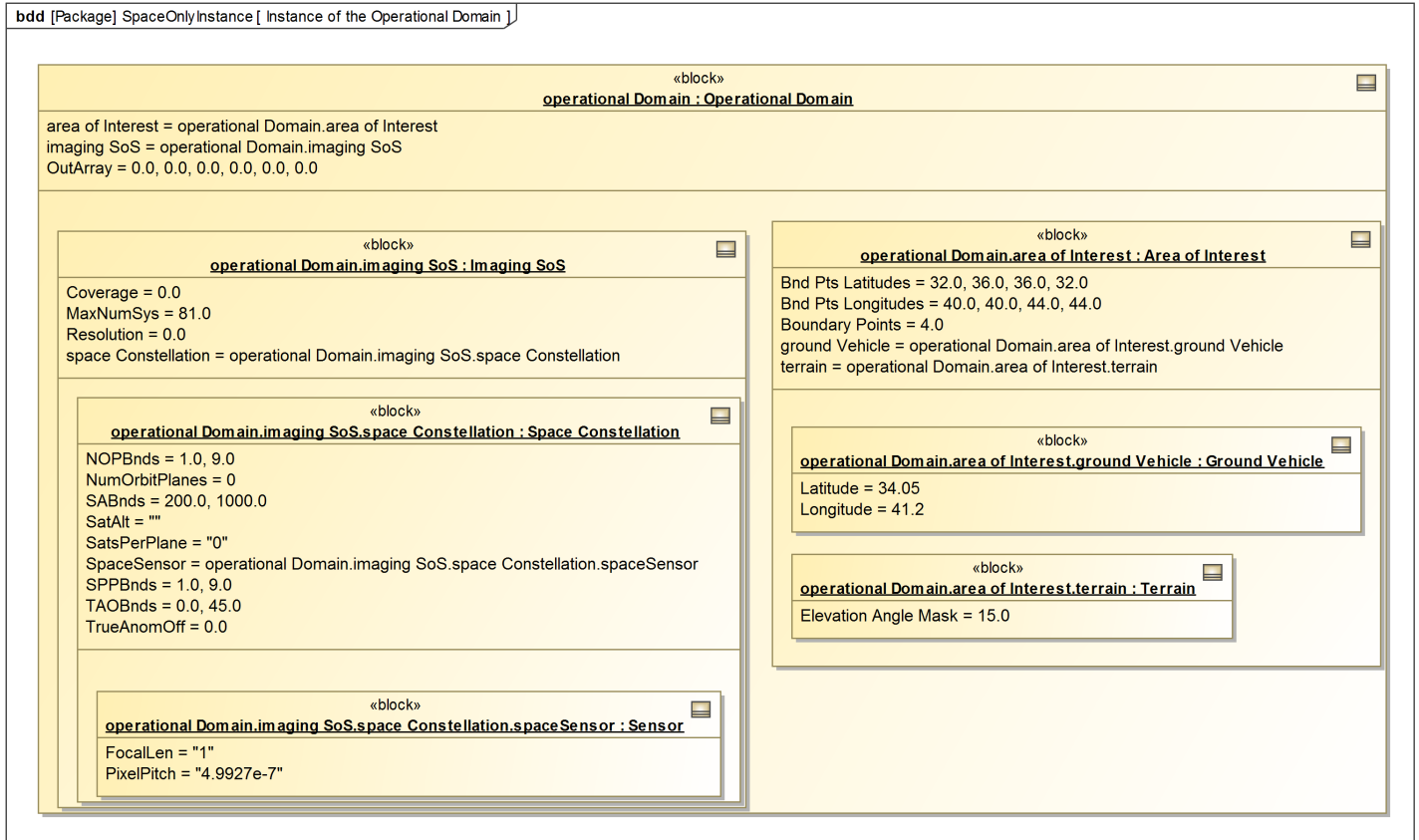


Figure 27. Space only instance definition

Results.

The GA used in the EA was configured to use a population size of 40 and perform ten iterations with ten generations per iteration. Seven architectures were defined by setting the maximum number of satellites constraint to 81, 72, 64, 55, 45, 36, and 30. Figure 28 shows the optimal results of each iteration for the seven different architectures. An interesting result that should be noted was with the architecture that was constrained to 64 satellites. The optimal result that was returned only used 63 satellites composed of nine orbit planes with seven satellites per plane. All of the optimal points and corresponding architecture parameters associated with each point are listed in Table 4. Looking at only the optimal points shows a linear

growth with a slope of 7.4×10^{-3} . The optimal points and their composition of GSD and coverage are shown in Figure 29

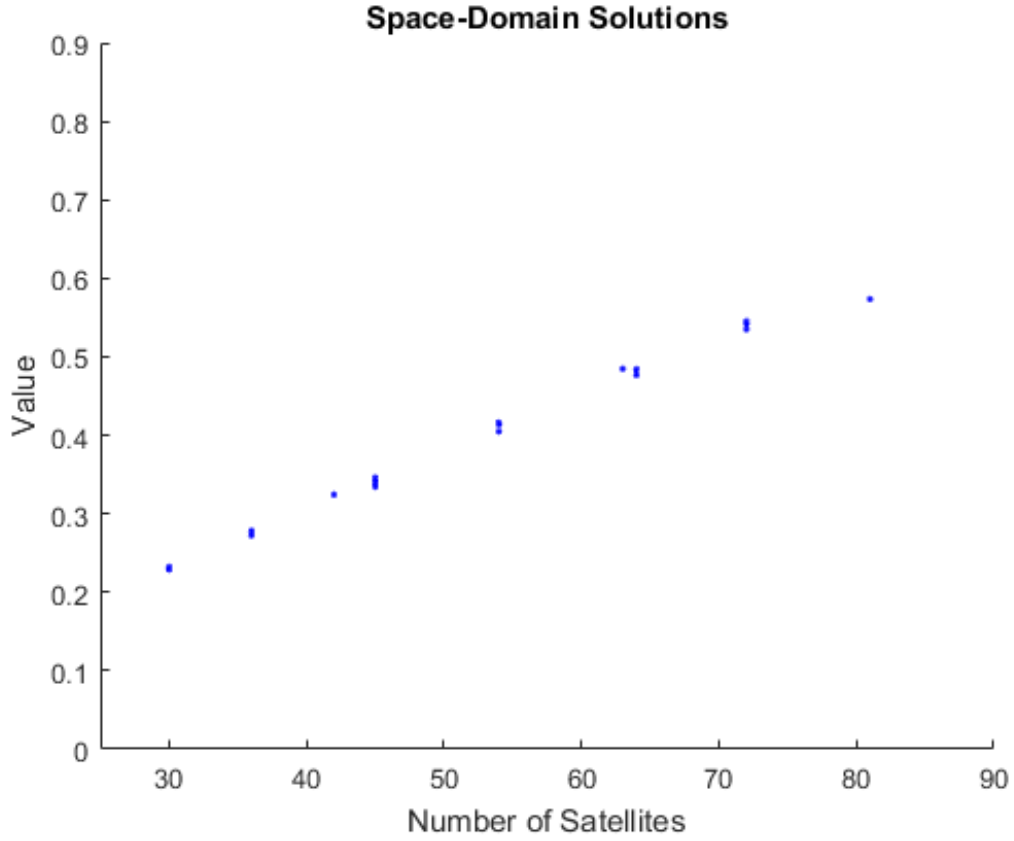


Figure 28. Results of optimal space domain architectures

Table 4. Space Domain Architecture Results

Constraint	Value			Orbit Parameters			
	Mean Value	GSD	Coverage	Orbit Planes	Satellites per Plane	Offset	Altitude
81	.574	.389	.758	9	9	35	700
72	.545	.341	.750	8	9	20	800
64	.485	.298	.672	9	7	0	800
55	.416	.256	.576	6	9	15	800
45	.346	.213	.479	9	5	15	800
36	.278	.171	.385	6	6	30	800
30	.232	.143	.322	6	5	5	800

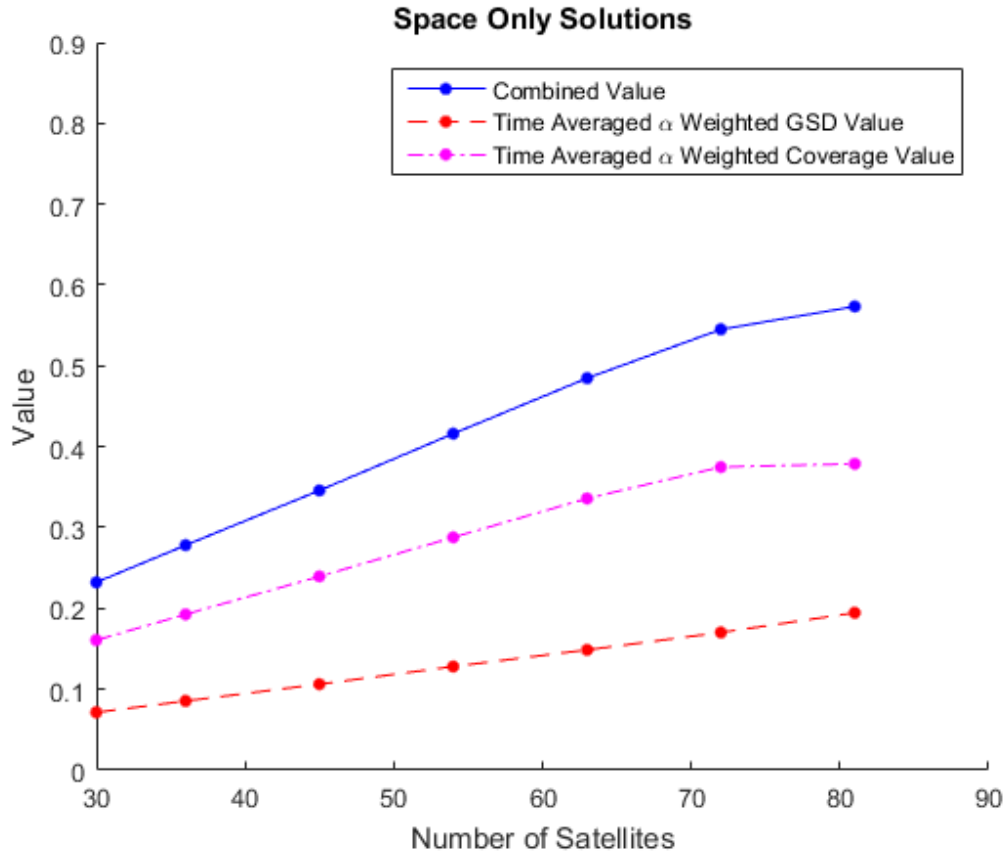


Figure 29. Optimum Space Results

4.3 Air Solution

In this section, the methodology discussed in Chapter III will be applied to an air domain only design problem. The architecture created and parameters used will be presented first followed by the results.

Air Domain Architecture.

The operational domain was not altered from the space domain only problem shown in Figure 24. The operational architecture was changed to use aircraft instead of satellites however. The imaging SoS was defined as being composed of one or more

aircraft formations composed of one or more sensors. Figure 25 shows the BDD for the air domain only imaging SoS. The imaging SoS parameters remain the same as in the previous problem. *MaxNumSys* is the constraint that determines the maximum number of systems, aircraft now, that can be used and *Coverage* and *GSD* are the best results found under the constraint. The aircraft formation block has eight parameters, similar to the space constellation. Four parameters are the bounds for the optimization problem and the other four parameters are the optimal values that will be returned by the EA.

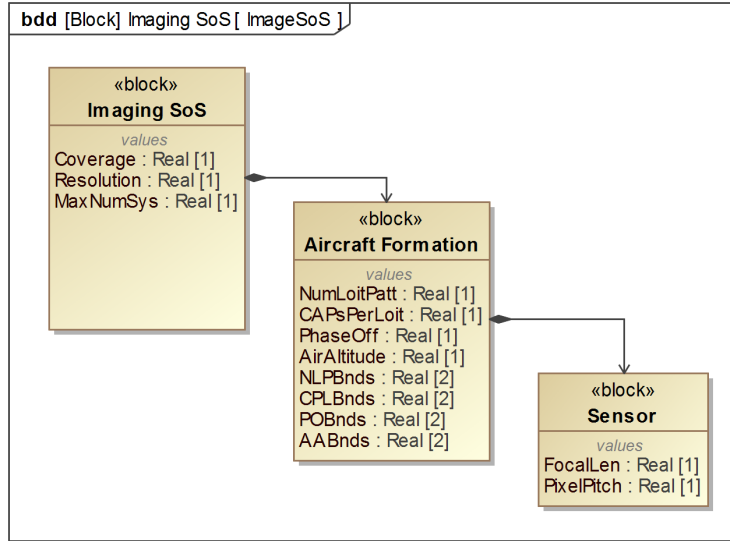


Figure 30. Air domain imaging SoS decomposition

The *Aircraft Formation* is parameterized by *NumLoitPatt*, *CAPsPerLoit*, *PhaseOff*, and *AirAlt*. *NumLoitPatt* and *CAPsPerLoit* are the number of loiter patterns in the area of interest and the number of combat air patrols (CAPs) in each loiter pattern respectively. *AirAlt* is the aircraft operating altitude in kilometers and *PhaseOff* is the phase offset of CAPs between neighboring loiter patterns. A 45 degree phase offset was shown previously in Figure 6. The other four parameters are simply the upper and lower bounds of each aircraft formation parameter. The parametric

diagram is shown in Figure 31 and shows the bounds and maximum number of systems constraint being passed into the constraint function. The result is passed back to the architecture parameters describing the aircraft formation. Like with the space domain, the area of interest and sensor parameters were placed directly in the EA code because they do not vary for this research and would add unnecessary complexity.

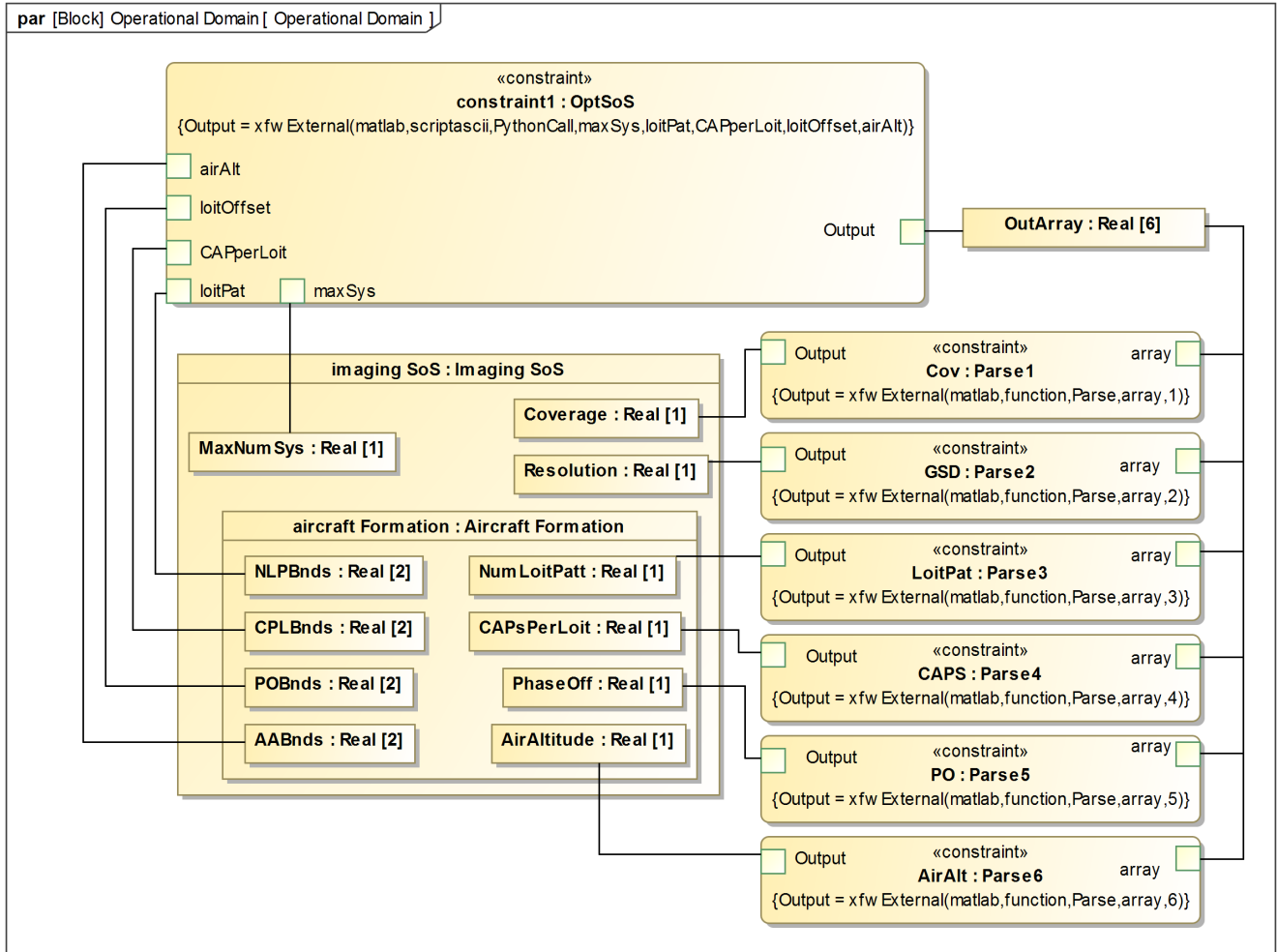


Figure 31. Air only domain parametric diagram

Figure 32 shows the candidate SoS architecture that was created for the air only design problem. This architecture consists of a single aircraft formation with the

same sensor on each aircraft. The number of loiter patterns and CAPs per loiter pattern is bounded between one and 16, but this is not as straight forward as with the space domain. To keep the loiter patterns circular and uniform, there are only three options for the number of loiter patterns, one, four, or 16. As shown in Figure 32 the maximum number of systems is set to 48. This means the number of possible CAPs per loiter pattern will change depending on the number of loiter patterns present. The altitude is allowed to vary from six to 20 kilometers and the phase offset between loiter patterns is from zero to 90 degrees in increments of 15 degrees.

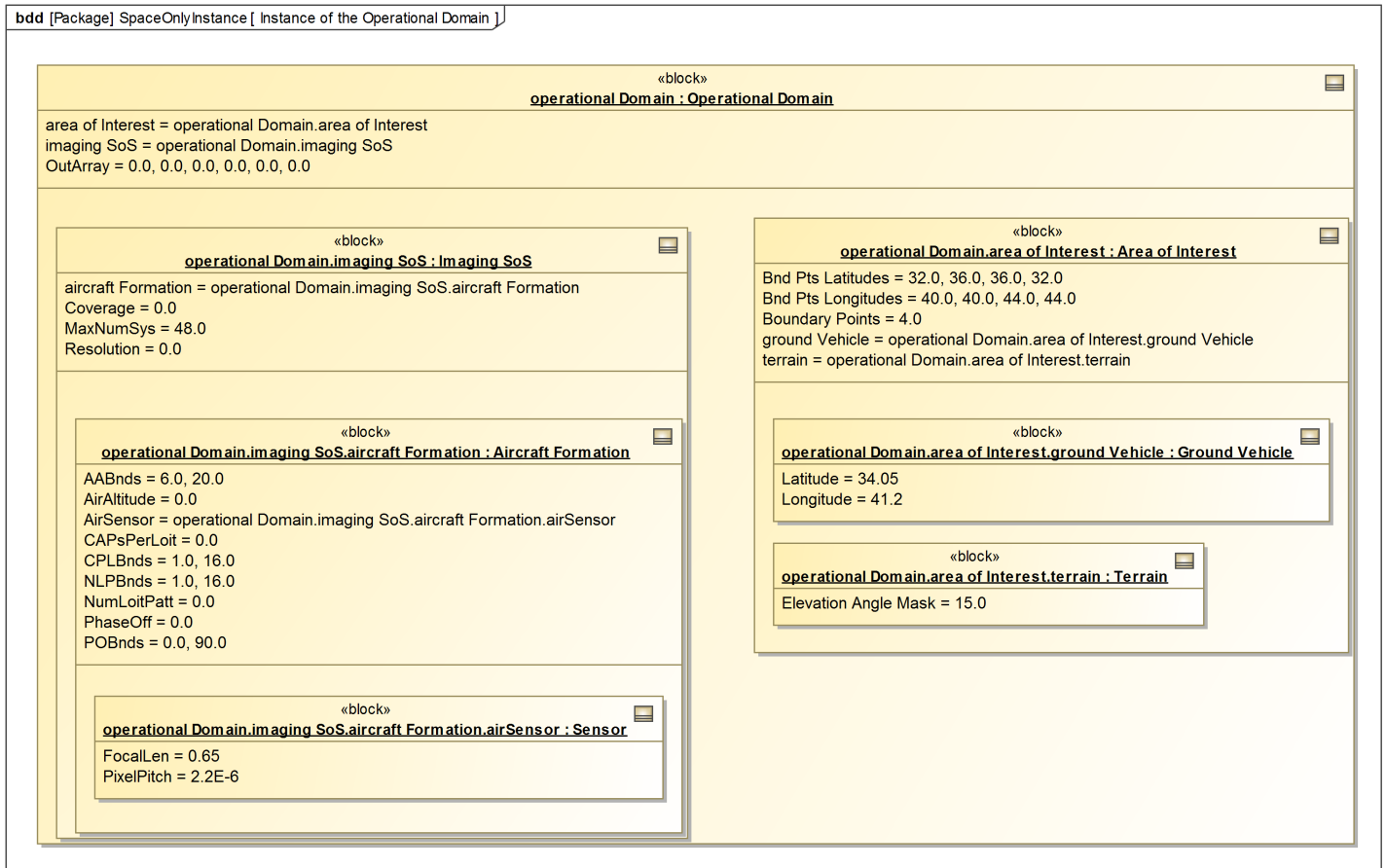


Figure 32. Air only instance definition

Results.

The GA used in the EA was configured to use a population size of 40 and perform ten iterations with ten generations per iteration. Four architectures were defined by setting the maximum number of aircraft to 48, 36, 24, and 12. Figure 33 shows the optimal results of each iteration for the four different architectures. All of the optimal points and corresponding architecture parameters associated with each point are listed in Table 5.

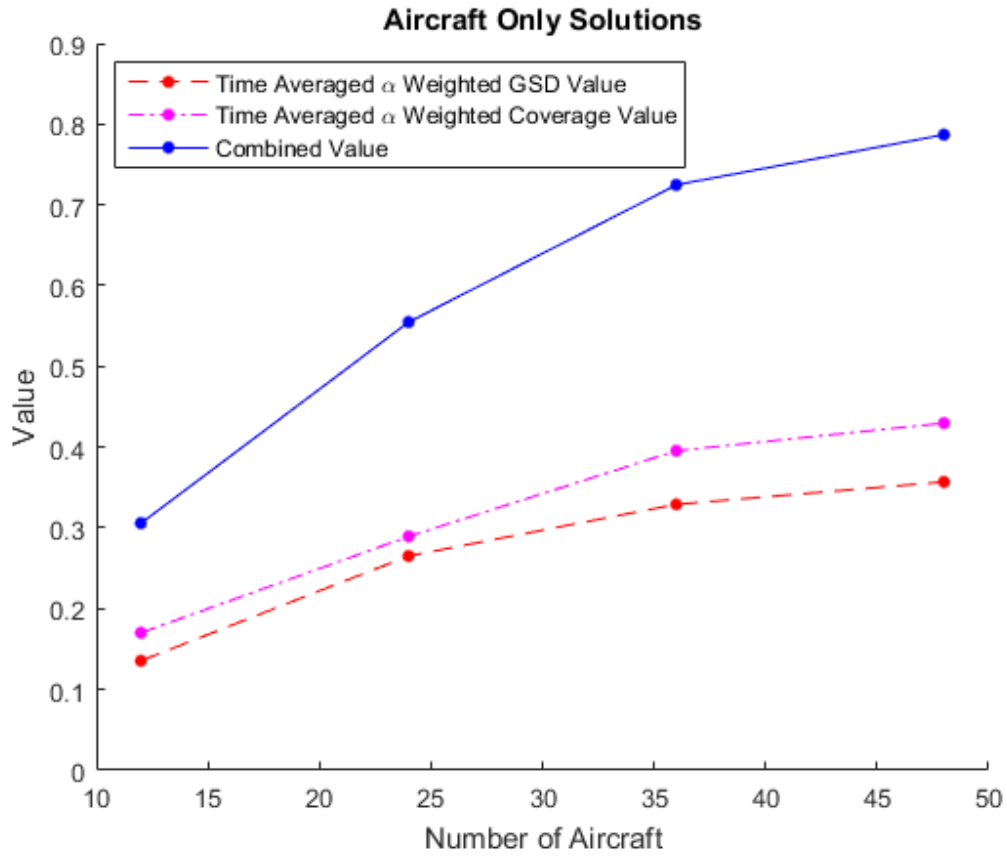


Figure 33. Results of optimal air domain architectures

Table 5. Air Domain Architecture Results

Constraint	Value			Aircraft Formation Parameters			
Max Aircraft	Mean Value	GSD	Coverage	Loiter Patterns	CAPs Per Loiter	Offset	Altitude
48	.787	.714	.860	4	4	15	16
36	.725	.659	.791	4	3	90	20
24	.555	.531	.579	4	2	60	20
12	.306	.271	.341	4	1	30	20

4.4 Multi-Domain Solution

In this section, the methodology discussed in Chapter III will be finally be applied to a multi-domain design problem. The architecture created and parameters used will be presented first followed by the results.

Multi-Domain Architecture.

The operational domain was not altered from the air and space domain problems, but the operational architecture was changed to use both aircraft and satellites. The imaging SoS was created by combining the systems used in the air and space domain problems together. Figure 34 shows the BDD for the multi-domain imaging SoS. The imaging SoS parameters remain the same as in the previous problems. *MaxNumSys* is the constraint that determines the maximum number of aircraft and satellites that can be used and *Coverage* and *GSD* are the best results found under the constraint.

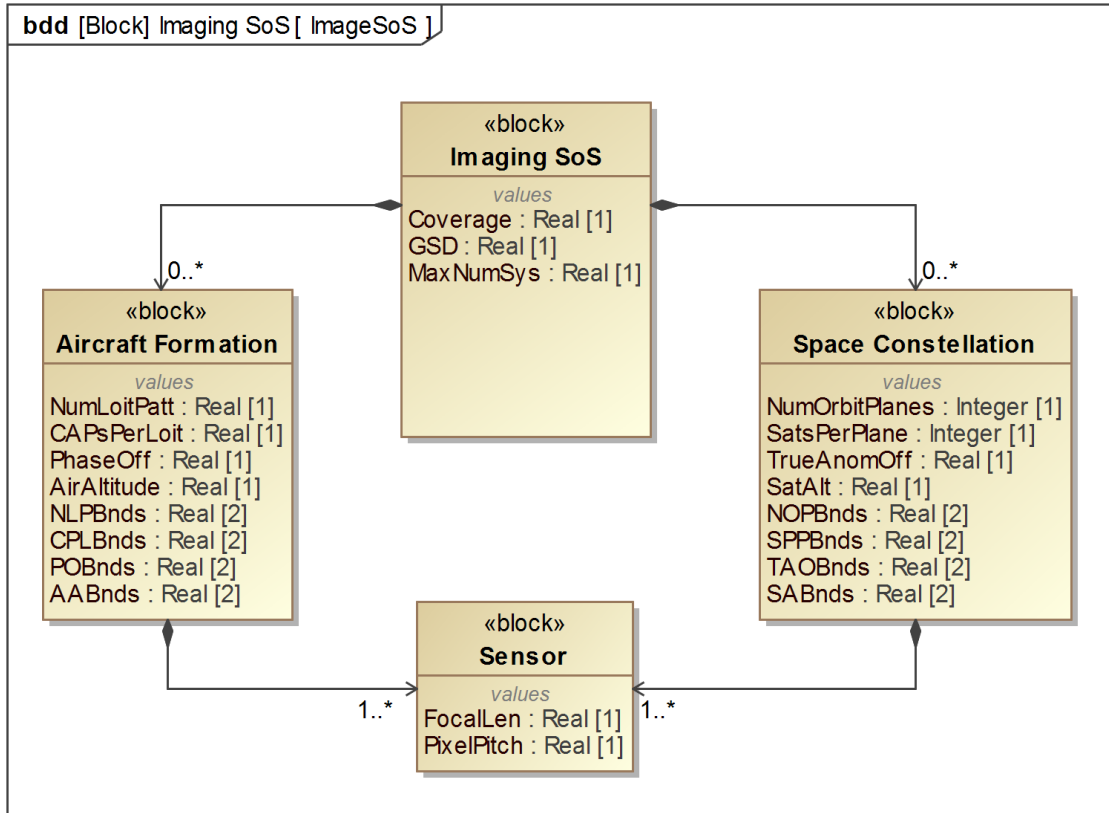


Figure 34. Imaging SoS decomposition

The parametric diagram is shown in Figure 35 and shows the bounds and maximum number of systems constraint being passed into the constraint function. This diagram shows how quickly the design can grow.

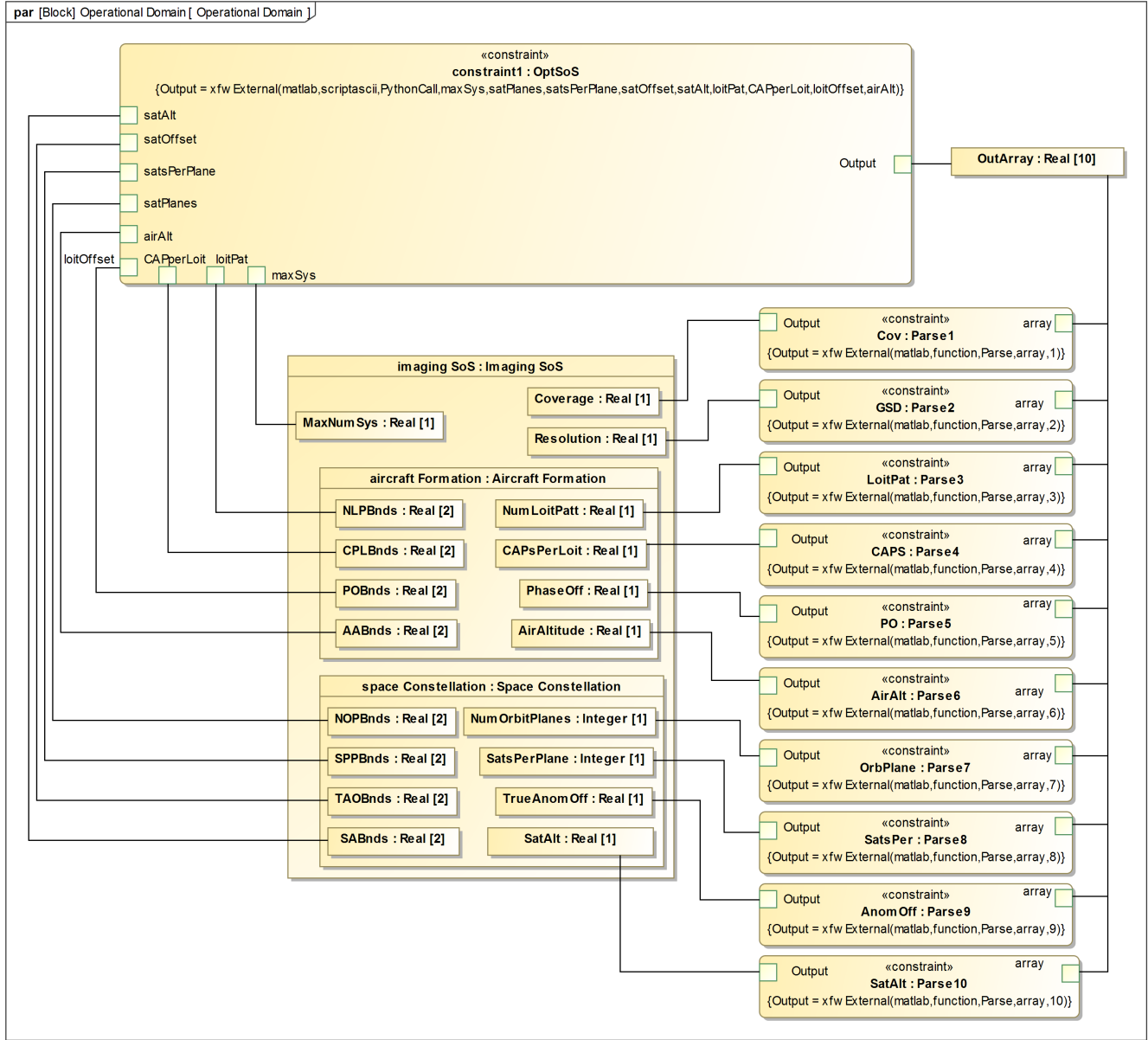


Figure 35. Multi-domain parametric diagram

The next OOSEM activity performed was to synthesize a candidate SoS architecture. Figure 36 shows the candidate SoS architecture that was created for the multi-domain design problem. The bounds and parameters are the same from each single domain problem.

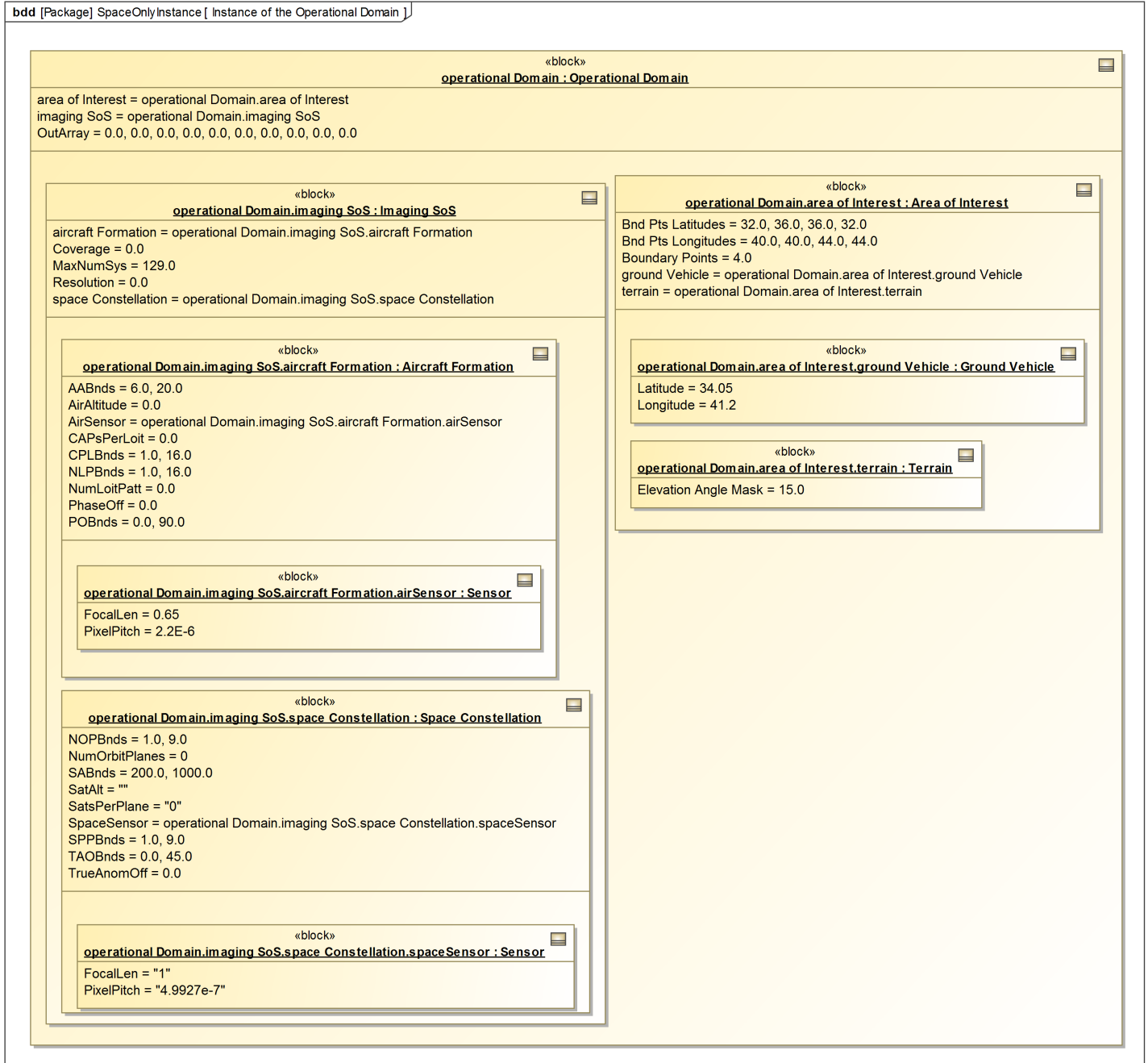


Figure 36. Multi-domain instance definition

Results.

For the multi-domain problem the GA used in the EA was configured to use a population size of 50 with ten generations and one iteration. The primary reason for changing the iterations from ten to one was the computation time for a single iteration exceeded 24 hours. Runs were conducted by varying the constraint parameter between 75 and 129 systems. The 129 systems is achievable by using the maximum number of satellites and aircraft, 81 and 48 respectively. The results from each run are captured in Table 6. Figure 37 shows the GSD and coverage components of the optimal solutions. Comparisons of the multi-domain solution and single domain solutions will be given in Chapter V.

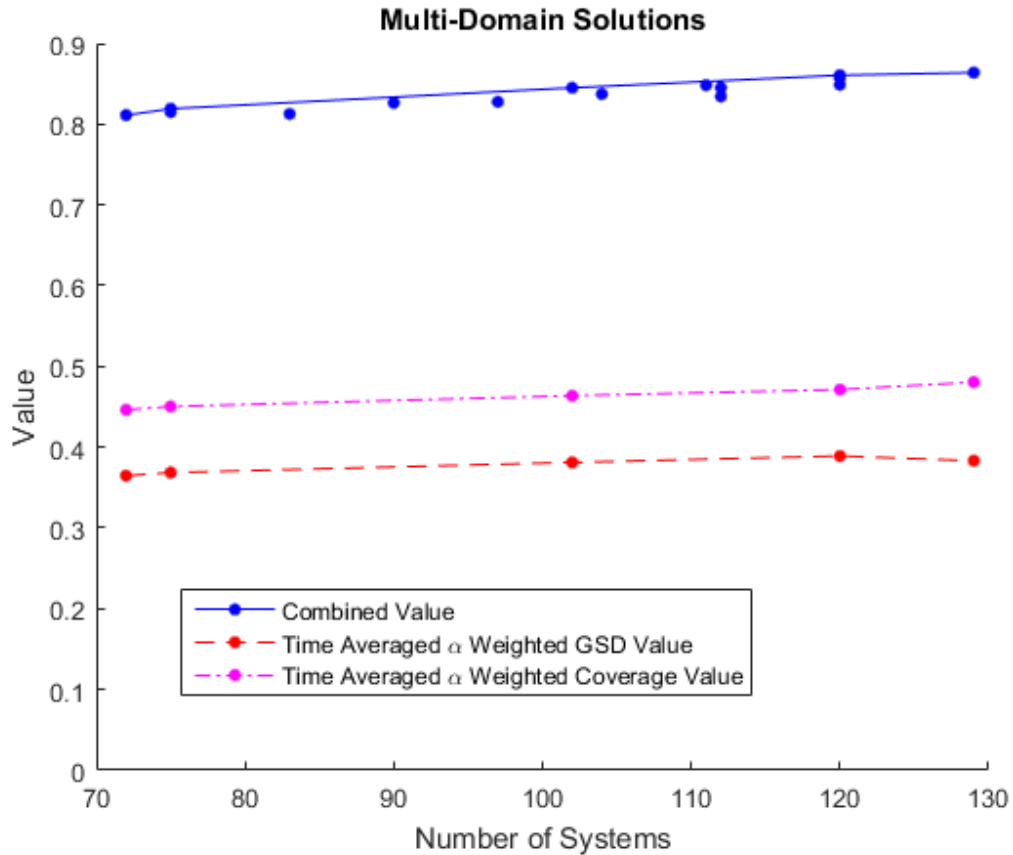


Figure 37. Optimum Multi-Domain Results Break Down

Table 6. Multi-Domain Architecture Test Results

Constraint	Optimal Values				Orbit Parameters				Aircraft Parameters			
Max Systems	Systems	Value	GSD	Cov	Orbits	Sats/Plane	Offset	Alt	Loiters	CAPs/Loiter	Offset	Alt
75	72	.811	.730	.893	8	3	15	600	4	4	15	16
75	75	.815	.731	.899	9	3	15	700	4	4	15	16
75	75	.819	.737	.901	3	9	5	600	4	4	15	16
85	83	.813	.737	.888	7	5	10	600	16	1	0	16
95	90	.827	.735	.918	6	7	15	600	16	1	0	18
100	97	.828	.730	.926	7	7	35	700	16	1	0	18
110	104	.837	.743	.932	8	7	40	600	16	1	0	18
115	112	.834	.725	.944	8	8	35	600	16	1	15	18
115	112	.845	.754	.937	8	8	35	700	4	4	15	15
120	102	.845	.762	.928	6	9	15	400	4	4	15	16
120	120	.849	.744	.954	8	9	25	800	16	1	0	17
120	120	.859	.768	.949	9	8	25	500	4	4	15	16
125	120	.857	.760	.954	8	9	15	600	4	4	15	16
129	111	.848	.762	.935	7	9	15	500	4	4	15	16
129	120	.861	.778	.943	8	9	15	400	16	1	0	18
129	129	.864	.766	.961	9	9	30	600	4	4	15	16

V. Results

This chapter will first analyze the results presented in Chapter IV and then answer the research questions posed in Chapter I. Recommendations for future work will be discussed next and then a summary of this research will be given last.

5.1 Analysis of Results

The results for the single domain and multi-domain solutions were presented separately in Chapter IV. Figure 38 shows the values combined on the same plot. When comparing the multi-domain solutions to the single domain solutions, the multi-domain does not appear provide much additional value over the air domain only solutions. The multi-domain results shown in Table 6 show that the maximum number of aircraft, 48, were used for every multi-domain solution. This could be expected since the aircraft performed much better than the satellites in the single domain solutions.

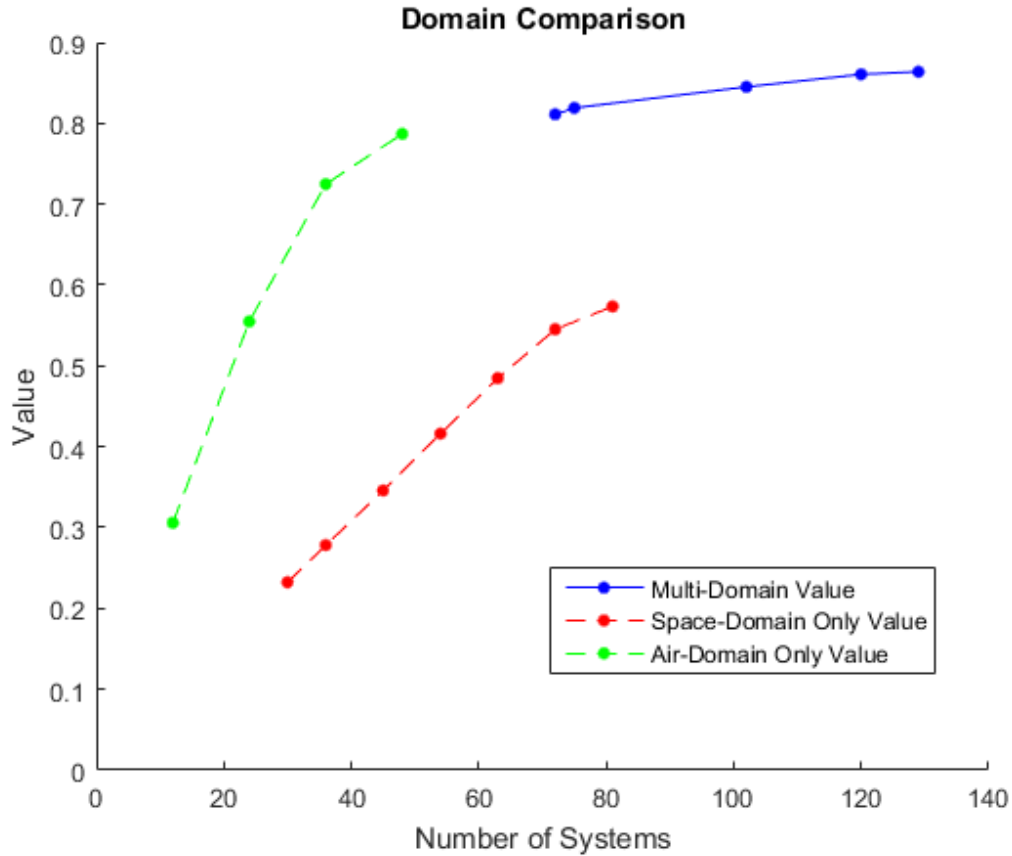


Figure 38. Comparison Between Solutions

There are several reasons why the aircraft performed better than the satellites. The first reason is that the area of interest was modeled as a permissive environment. In a permissive environment the aircraft had a significant advantage by being able to loiter in the area of interest. If threats were modeled it could be expected that the performance would decrease or the number of aircraft needed to achieve the same level of performance would increase. Figure 39 shows an estimate of what the aircraft only solutions would be if threats were present. Threat 1 shows the scenario if 1.5 times as many aircraft are needed to obtain the same value and threat 2 shows the case where the number of aircraft are doubled to obtain the same value. This is interesting because there is not only a shift in the curve, but a flattening as well.

Assuming that aircraft would be more affected by threats, the flattening rate would be greater for aircraft than satellites. This suggests that satellites might eventually outperform aircraft in contested or denied environments, which would mean the multi-domain solutions would be composed of more satellites than aircraft. Since both domains contribute to the multi-domain value, it is difficult to estimate how the curve would change.

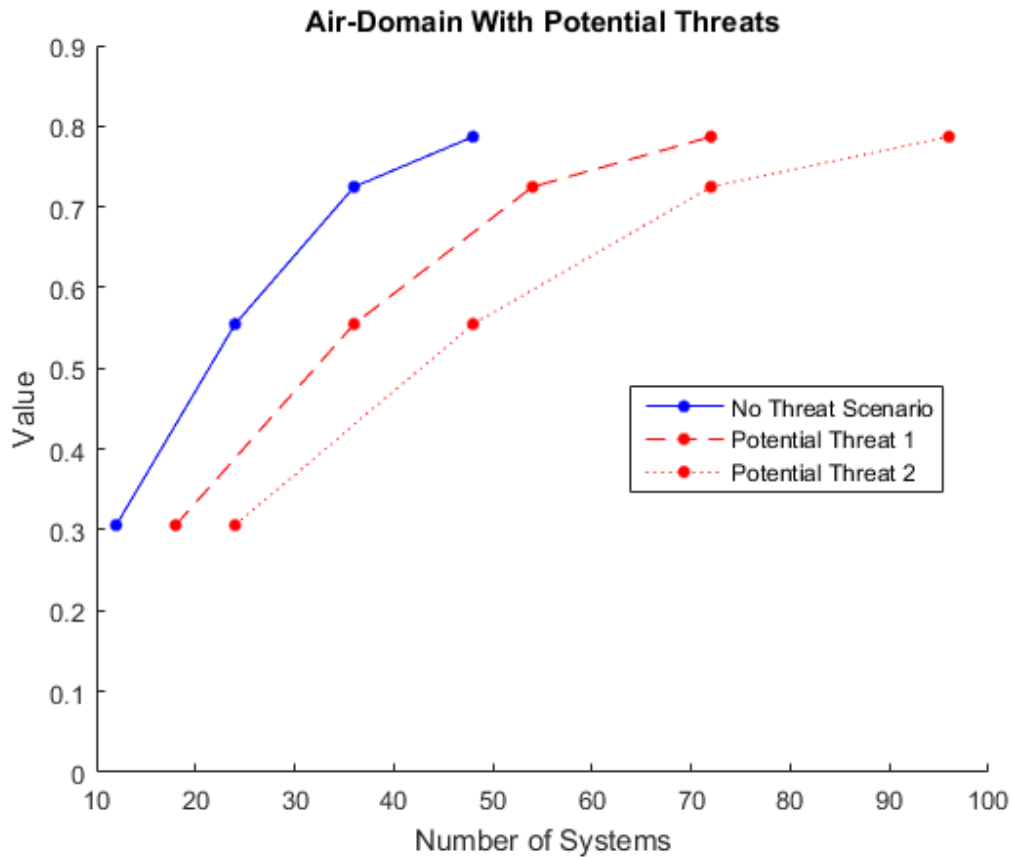


Figure 39. Air-Domain With Potential Threats

Another reason the aircraft performed well is due to the size of the area of interest and the maximum number of CAPs allowed for almost complete coverage of the area of interest and a better GSD value. Only very small gaps were left uncovered and are shown in Figure 40. The satellites would cover these gaps for short periods

of time and provide small amounts of coverage value while the aircraft remained the primary GSD value contributor. The simulations and calculations were reviewed and no errors were found suggesting the code was working properly and the results are accurate. However, the results are only as accurate as the model. As described previously, the sensor parameters were taken from actual sensors that have been built and tested which provide model validity. Assumptions were made about the number of CAPs available, the aircraft operating altitude and time, and that the formation could keep the specified spacing.

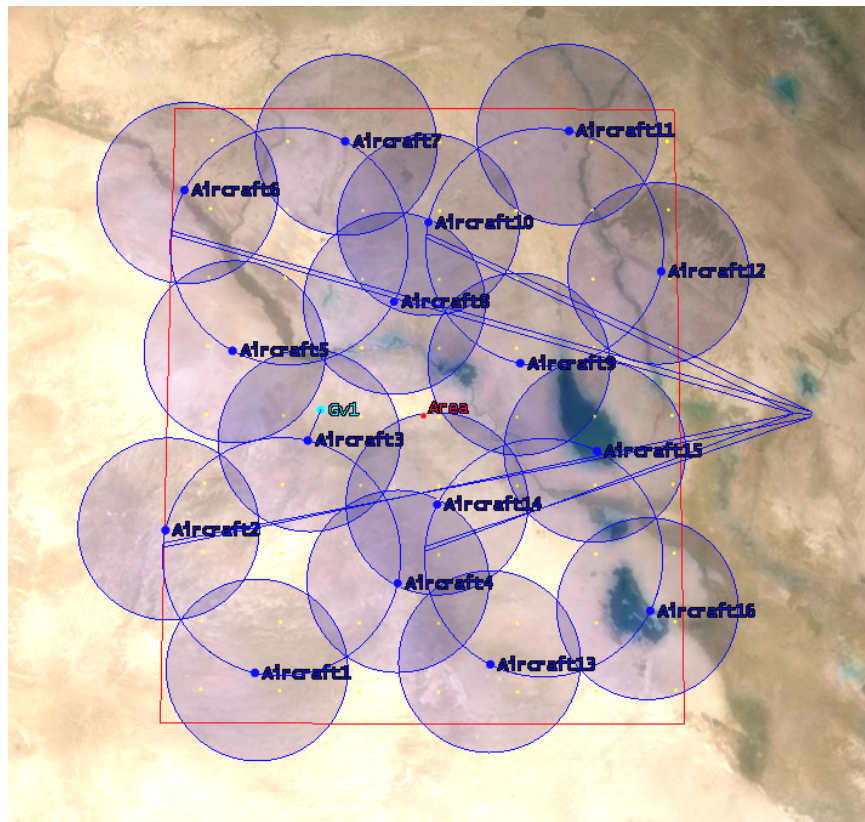


Figure 40. Optimal Aircraft Formation

Another interesting comparison is the altitude relationship between the single domain and multi-domain solutions. The average altitude of the satellites and aircraft in the multi-domain solution was lower than the single domain solution

averages and can be seen in Table 7. This is significant because it suggests that the optimal solution in a single domain is not always the optimal for a multi-domain solution. However, more iterations would need to be run to improve the confidence in the multi-domain solutions. Several solutions were composed of both optimal single domain solutions, and several other solutions were composed of significantly different architecture parameters. The multi-domain solution that used 102 systems is an example where the optimal air domain solution was used but the satellite parameters were changed significantly. The constellation altitude was lowered to 400 km instead of the optimal single domain solution altitude of 800 km.

Table 7. Average Altitude Comparison

Domain	Mean Satellite Altitude (km)	Mean Aircraft Altitude (km)
Space	785.7	N/A
Air	N/A	19
Multi	593.75	16.625

5.2 Conclusions

This primary objective of this thesis was to develop a methodology for architecting optimal multiple domain SoS with the ability to collect ISR in A2/AD environments. This research goal was achieved by developing an EA that was used within OOSEM to evaluate and optimize multi-domain architectures. Even though threats associated with A2/AD environments were not modeled, the method was shown to be capable of including threats. Including threats would require more computational power to achieve results in a feasible amount of time. This will be discussed in the next section.

The first research question posed was *Using SysML, how do you parametrize a multi-domain SoS for assessment?* The best method found was by using blocks, composition, block definition diagrams, and parametric diagrams. Systems with

similar parameters were grouped together and depicted as a SysML block with the similar parameters as value properties associated with that type of block. This is commonly done with satellites in constellations but was extended to the air domain by creating aircraft formations. Only four parameters were used in this research for each grouping, but more could easily be added. A benefit of this method is that it allows for creating single systems by having only one system in a constellation or formation. Heterogeneous systems, like a disaggregated space system, can also be implemented by creating more than one constellation with either different orbit parameters or different payloads depending on the disaggregation strategy. Multiplicity allows for more than one threat, ground vehicle, sensors, constellations and formations.

The second research question posed was *How do you assess the value of different multi-domain SoS architectures?* This question was answered by carefully choosing the performance measures by which to evaluate a candidate SoS. Traditional metrics used in evaluating single domain problems tend to not extend well to other domains. Revisit rate is a common metric used for comparing space systems against each other, but aircraft systems are not naturally assessed this way. Coverage and GSD are two metrics common among both domains and can be described as functions of time. Only using one metric would favor one domain over another though. To increase the GSD, the air domain would dominate because aircraft can get closer and achieve a better GSD. To increase coverage, the satellites would be favored because they are further away and can cover more area. For collecting imagery, the optimal solution would cover the entire area of interest with some maximum GSD. Normalizing the GSD eliminates the periods where the GSD is not useful and puts it on the same scale as coverage. Averaging both GSD and coverage over time give the value for the individual components and then averaging them

together gives an overall value. A perfectly optimal system would have a value of one relating to both continuous coverage and maximum GSD.

The third research question posed was *How do you formulate a multiple domain SoS architecture optimization problem?* This was accomplished by maximizing the SoS value described in the previous question subject to constraints. Without constraints the problem would grow unbounded and create unfeasible solutions. Cost is typically the constraint on performance but was not being considered in this research. The number of systems used was set as the constraint recognizing the fact that as more systems are added the cost will tend to rise. The difference in cost of satellites and aircraft has been acknowledged and is left for future research to incorporate. It was also discussed that traditional gradient based optimization methods would not work because the set of feasible solutions is not continuous. A genetic algorithm was used to search the sample space and find optimal solutions.

The last research question posed was *What is the added benefit of near-optimal multiple domain solutions?* The added benefit of a multi-domain solution depends on the scenario. If resources are unlimited and the environment is permissive, then there is little to no benefit of a multi-domain solution over an aircraft only solution. However, resources are always constrained and having an unlimited number of aircraft may not be an option. Multi-domain solutions provide benefit by adding value to resource constrained systems. If only 24 aircraft, or 8 CAPs, were available for the design problem used in this research, then the maximum value that could be achieved by aircraft would be 55.5% of the full value. Adding a constellation of 81 satellites would increase the score to 77.7% of the full value, a better score than either solution could achieve alone. Another reason resources might be constrained would be if there were more than one area of interest. This would be an interesting

design problem because the aircraft would need to double in numbers while the satellites could remain the same.

5.3 Future Work

This research generated many questions and topics that can be investigated in future work. This section will describe each topic and how the author would approach the research.

Computation Time.

The first question comes from the amount of computation time it took to run the EA. *How can the EA run faster?* This research exhausted every known method for speeding up the EA. Two computers were used for this research and were using Intel i7 processors with 8 and 16 GB of RAM. It should be noted that 8 GB of RAM is the smallest amount of RAM recommended based on experience. STK consumed the most time and was configured to run the fastest by closing all the graphics windows, turning off the acknowledgment messages, writing commands to a text file that STK reads in, turning off the graphical user interface buttons, and computing access in parallel with four processor cores. This still resulted in over 24 hour run times on the multi-domain design problem.

Future work should consider using supercomputers to run the simulations and calculations. STK Engine is used on these computers which is essentially STK without any graphical interface. This requires knowledge of STK Connect commands and how to send them to the program which was demonstrated in this research. Currently, Wright-Patterson Air Force Base has the high performance

computing (HPC) facility with supercomputers that can be used by AFIT students and faculty. Permissions and procedures for using the HPC would need to be investigated though.

Threat Modeling.

The second question comes from the need to model threats for an A2/AD environment. *How do you parametrize threats in a multi-domain model?* The best method for accomplishing this is unclear. If more time was available, this research would model a threat with a probability of effectiveness (P_{eff}) and/or a probability of detection (P_{det}) functions. These probabilities would vary over time depending on the system it is trying to degrade or destroy. The range, altitude, velocity, and heading of a system might greatly effect the probability the threat will detect the system or not. The range between the threat and systems would be calculated and passed to a function that uses the PK to determine if the system is effected or not. The type of threat would determine the interaction it would have with the systems and the type of function used to determine if the system is effected.

Stochastic Modeling.

Stochastic modeling can be used to account for variation or probability of variables. This research made a fundamental assumption that the satellites and aircraft knew where to look in the area of interest and that the ground vehicle was stationary. This is not always the case for real world applications. Allowing the ground vehicle to randomly move about the area of interest would increase the accuracy of the model. The Disaggregated Integral System Concept Optimization (DISCO) methodology is one method of applying stochastic parameters and has a process

similar to the one used for this research. The DISCO method updates the stochastic parameters after the global stopping criteria has been satisfied. Thompson describes this process in more detail in his dissertation (Thompson, 2015). Depending on the stochastic parameters, the DISCO methodology would not be recommended for future work.

The recommended method would be to update the stochastic parameters when evaluating the fitness of a candidate solution. This method would determine each candidate solution's mean value and variance. The DISCO methodology would determine the mean value and variance of the set of optimal results found by the GA. There is also large computational efficiencies to be gained by not using the DISCO methodology. It is much more efficient to create one scenario and move the ground vehicle 100 times, recalculating the measures of performance after every move, rather than creating 100 scenarios that have the ground vehicle in a different location. However, if the stochastic parameters can violate a constraint, then this method should not be used because not all of the variations would be checked against the constraints.

Cost Model.

Cost was not considered in this research but could be added to increase the validity of the results. There are a couple ways the cost model could be incorporated in the EA. One placement is shown in Figure 41. This would determine the estimated cost of a candidate solution and return it to the optimizer. The optimizer could then use cost as a constraint or part of the objective function. Minimizing cost subject to constraints on the level of performance and number of systems will be discussed next.

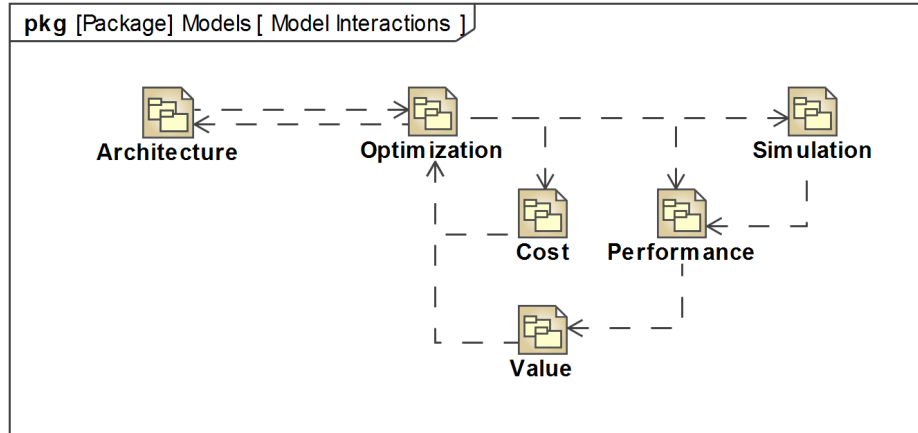


Figure 41. Potential EA with cost model

Optimization Methods.

This research used an optimization algorithm that was capable of handling integer variables. Some variables, like the number of satellites, can only be described as integers on a discrete sample space while others, like altitude and inclination, can take on real values over a continuous set. Future research should look to used mixed integer optimizers that allow for some variables to be integers and others continuous real values. The MATLAB GA function does handle mixed integer problems but was not implemented due to time restrictions and the need to minimize the sample space. Multi-objective optimizers would also be useful if a cost model is implemented. This would allow for the minimization of cost and maximization of performance as the objective functions.

5.4 Summary

This research presented a methodology for evaluating and optimizing multi-domain SoS using MBSE. Chapter I presented the background information for why this is relevant, particularly for the military, and posed four research questions. Chapter II

presented more background details, definitions, and investigated current MBSE methodologies and their benefits and weaknesses. OOSEM was identified for its flexibility and extensibility to a wide range of design problems. Chapter III describes the methodology of using OOSEM with an EA that was developed to evaluate and optimize a design. The specific tool architecture used for this research is presented last in Chapter III. Chapter IV starts by verifying the model with a simple design problem, and then applying it to an air domain only, space domain only, and multi-domain design problems. The results from these problems are analyzed in Chapter V and concluded that the methodology worked and could be extended to include modeling A2/AD environments or other design problems.

Appendix A. Python code

This appendix provides a detailed description of the Python code generated for this research. Each section will describe a Python file and each subsection will describe the functions within that file. The first file presented was used to communicate with STK and generate the scenarios. The second file presented was used to control the simulation, performance, and value components of the EA.

A.1 myClasses.py

myClasses.py contains the classes of objects used by the simulation. This section will describe the different classes and what they require to be added to a simulation. In general the *Scenario* class must be defined before any other class can be added to a simulation. The packages shown in Listing A.1 are needed by the classes in this file.

Listing A.1. myClasses.py included packages

```
1 import os
2 import myFuncs
3 import myClasses
4 import numpy as np
5 import math
```

Scenario.

The *Scenario* class was used to create and hold the values associated with a scenario and the code used is shown in Listing A.2. The first thing that this class does when initialized is establish a TCP/IP connection with STK. The socket opened by the connect function will be discussed later. Lines 5-11 set the save location for all outputs associated with this scenario and creates the folder to save this data in if it has not already been created. Lines 12-33 are values associated with the scenario that can be used by other functions. For example, the value *Secs* on line 32 are the

total number of seconds in the scenario and is used by a function to create a matrix that has rows for every second of the scenario. Lines 34-36 are used to help speed up STK. Acknowledgment messages and verbose commands can be useful for debugging but are not necessary. The high speed command on line 36 turns off the graphical user interface and only responds to commands sent using Connect. Lines 37-39 create the scenario in STK and set the time period. Line 40 creates a text file that Connect commands will be written to and then passed to STK all at once. This is done to improve speed and will be shown in functions later.

Listing A.2. myClasses.Scenario class

```

1 class Scenario:
2     def __init__(self, name, days=None):
3         # Create a new connection to STK
4         self.socket = myFuncs.connect()
5         self.name = name
6         self.saveLoc = str(os.getcwd()) + '\\Scenarios\\' + str(self.name
7             ↪ ) + '\\ '
8         #Check if scenario has been created before; if not, create it
9         if not os.path.exists(self.saveLoc):
10             os.makedirs(self.saveLoc)
11         if not os.path.exists(self.saveLoc + 'ScenarioFiles\\'):
12             os.makedirs(self.saveLoc + 'ScenarioFiles\\')
13         #Default scenario starting values
14         self.startDay = 1
15         self.startMonthYear = ' January 2015 '
16         self.startHour = 10
17         self.startMin = 00
18         self.startSec = 00
19         self.startDate = str(self.startDay) + self.startMonthYear + str(
20             ↪ self.startHour) + ':' + str(self.startMin) + ':' + str(
21             ↪ self.startSec) + '.00'
22         #Days are the scenario length
23         if days is None:
24             self.days = int(1)
25         else:
26             self.days = days
27         #Calculate ending values based on days
28         endDay = self.startDay + int(self.days)
29         endMonthYear = ' January 2015 '
30         endHour = 10
31         endMin = 00
32         endSec = 00
33         self.endDate = str(endDay) + endMonthYear + str(endHour) + ':' +
34             ↪ str(endMin) + ':' + str(endSec) + '.00'

```

```

31     #Secs is the total number of seconds in the scenario
32     self.Secs = (endDay-self.startDay)*24*60*60 + (endHour-self.
        ↪ startHour)*60*60 + (endMin-self.startMin)*60 + endSec-self
        ↪ .startSec
33     self.startTime = self.startDay*24*60*60 + self.startHour*60*60 +
        ↪ self.startMin*60 + self.startSec
34     self.socket.send( 'ConControl / AckOff\n' )
35     self.socket.send( 'ConControl / VerboseOff\n' )
36     self.socket.send( 'ConControl / HighSpeedOn\n' )
37     self.socket.send( 'New / Scenario ' + str(self.name) + ' \n' )
38     setTimePeriodStr = 'SetTimePeriod * ' + self.startDate + ' " " '
        ↪ + self.endDate + '"\n'
39     self.socket.send( setTimePeriodStr )
40     mr = open( self.saveLoc + 'connect.txt', 'w' )
41     mr.close()

```

Area of interest.

This class creates an area of interest defined by boundary points. The default values can be altered or a numpy array of points can be passed in to define the boundary. Lines 1-10 show the initialization and lines 11-26 define a function of the class to add it to a scenario. The socket created with the scenario is assigned to a variable on line 12. Line 13 creates an area target in STK and lines 14-17 convert the numpy array to text and then defines the boundary points for the area target. Lines 18-21 use the area target to create a coverage definition with 0.5 degree latitude/longitude resolution. This is used when computing coverage.

Listing A.3. myClasses.AOI class

```

1 class AOI:
2     def __init__(self, name=None, boundary=None):
3         if name is None: #default name
4             self.name = 'AOI_1'
5         else:
6             self.name = name
7         if boundary is None: #default boundary
8             self.boundary = np.array([[32, 40],[32, 44],[36, 44],[36,
                ↪ 40]])
9         else:
10            self.boundary = boundary
11     def add(self, scen):
12         s = scen.socket

```

```

13     s.send('New / */AreaTarget ' + self.name + '\n')
14     bnd = ' '.join(map(str, self.boundary))
15     bnd = bnd.replace('[', '')
16     bnd = bnd.replace(']', '')
17     s.send('SetBoundary */AreaTarget/' + self.name + ' Pattern
           ↪ LatLon ' + str(len(self.boundary)) + ' ' + bnd + '\n')
18     #Define Coverage Definition for Region Coverage – Currently this
           ↪ coverage returns basic access, but can be copied and
           ↪ altered to track different figures of merit
19     s.send('New / */CoverageDefinition ' + self.name + '\n')
20     s.send('Cov */CoverageDefinition/' + self.name + ' Grid
           ↪ AreaofInterest Custom AreaTarget AreaTarget/' + self.name
           ↪ + '\n')
21     s.send('Cov */CoverageDefinition/' + self.name + ' Grid
           ↪ PointGranularity LatLon .5 \n')

```

Ground Vehicle.

This classes creates a ground vehicle in STK. The initialization and default values are shown lines 1-14 of Listing A.4. The position can be set by simply defining the latitude and longitude when creating a object of this class. Lines 15-23 define the function used to add the ground vehicle to a STK scenario. The socket contained in the scenario is used again to send commands to STK. The ground vehicle could be allowed to move over time but is set stationary by defining two way-points in the same location and is shown on lines 22 and 23.

Listing A.4. myClasses.GroundVehicle class

```

1 class GroundVehicle:
2     def __init__(self, name=None, lat=None, lon=None):
3         if name is None: #Default value
4             self.name = 'GV1'
5         else:
6             self.name = name
7         if lat is None: #Default value
8             self.lat = 34.05
9         else:
10            self.lat = lat
11        if lon is None: #Default value
12            self.lon = 41.2
13        else:
14            self.lon = lon

```

```

15     def add(self, scen):
16         s = scen.socket
17         s.send('New /*/GroundVehicle '+ str(self.name)+' \n' )
18         objPath = '/*/GroundVehicle/' + str(self.name)
19         s.send('SetConstraint ' +str(objPath) + ' ElevationAngle Min 15 \
    ↪ n')
20         s.send('SetConstraint ' +str(objPath) + ' LineOfSight On \n')
21         s.send('SetPropagator ' +str(objPath) + ' GreatArc \n')
22         s.send('AddWaypoint ' +str(objPath) + ' DetVelFromTime LatLon ' +
    ↪ str(self.lat) + ' ' + str(self.lon) + ' Terrain "' +str(
    ↪ scen.startDate) + '"\n')
23         s.send('AddWaypoint ' +str(objPath) + ' DetVelFromTime LatLon ' +
    ↪ str(self.lat) + ' ' + str(self.lon) + ' Terrain "' +str(
    ↪ scen.endDate) + '"\n')

```

rndSensor and sqrSensor.

These two classes are used to create the sensors used by the aircraft and satellites.

The round sensor is used to compute coverage while the square sensor is used for GSD. The main difference between the two sensors is their shape. The square sensor half angles are taken from an actual imaging sensor that will not be discussed here. The round sensor half angle is actually defined based on the aircraft or satellite altitude and is shown later. The other sensor parameters like aperture diameter, focal length, and number of pixels are the same for both sensors, but the square sensor is the only sensor that uses these for calculations and will be discussed later.

Listing A.5. myClasses.rndSensor class

```

1 class rndSensor:
2     def __init__(self, name=None, HalfAng=None, apDiam=None, lambd=None,
    ↪ dWidth=None, focLen=None, numPixels=None):
3         self.senShape = 'round'
4         if name is None:#Default value
5             self.name = 'Narrow'
6         else:
7             self.name = name
8         if apDiam is None:#Default value
9             self.apDiam = .5 #Meters
10        else:
11            self.apDiam = apDiam
12        if lambd is None:#Default value

```

```

13         self.lambd = 0.65E-6 #Short Wave in meters
14     else:
15         self.lambd = lambd
16     if dWidth is None:#Default value
17         self.dWidth = 2.2e-6 #Focal plane detector element width (m)
18     else:
19         self.dWidth = dWidth
20     if focLen is None:#Default value
21         self.focLen = .65 # Meters
22     else:
23         self.focLen = focLen
24     if numPixels is None:#Default value
25         self.numPixels = 4E6
26     else:
27         self.numPixels = numPixels
28     if HalfAng is None:#Default value
29         self.HalfAng = 40
30     else:
31         self.HalfAng = HalfAng

```

Listing A.6. myClasses.sqrSensor class

```

1 class sqrSensor:
2     def __init__(self, name=None, vHalfAng=None, hHalfAng=None, apDiam=
    ↪ None, lambd=None, dWidth=None, focLen=None, numPixels=None):
3         self.senShape = 'square'
4         if name is None:
5             self.name = 'Narrow'
6         else:
7             self.name = name
8         if apDiam is None:
9             self.apDiam = .5 #Meters
10        else:
11            self.apDiam = apDiam
12        if lambd is None:
13            self.lambd = 0.65E-6 #Short Wave in meters
14        else:
15            self.lambd = lambd
16        if dWidth is None:
17            self.dWidth = 2.2e-6 #Focal plane detector element width (m)
18        else:
19            self.dWidth = dWidth
20        if focLen is None:
21            self.focLen = .65 # Meters
22        else:
23            self.focLen = focLen
24        if numPixels is None:
25            self.numPixels = 4E6
26        else:
27            self.numPixels = numPixels
28        if vHalfAng is None:
29            self.vHalfAng = .1885 #Degrees
30        else:

```

```

31         self.vHalfAng = vHalfAng
32     if hHalfAng is None:
33         self.hHalfAng = .2513 #Degrees
34     else:
35         self.hHalfAng = hHalfAng

```

Constellation.

This class is used to create a constellation of satellites and the default initialization values are shown on lines 3-54 of Listing A.7. Lines 31-51 are two default sensor values. The first sensor is quickly defined on line 33 as a square sensor with a vertical half angle, horizontal half angle, detector pixel width, and focal length. The values used are derived from the GeoEye-2 satellite imaging sensor (Satellite Imaging Corp, 2016). The second sensor defined is a simple cone sensor that has a half angle that depends on the satellite altitude. Equations 1 - 5 are used to determine the maximum look angle that will result in a GSD less than one meter and are shown on lines 34-49. Lines 56-76 define the function used to add a constellation to a scenario. A single satellite which the constellation will be based on is added on lines 58-60. Lines 61-66 add the two sensors to the single satellite. Line 67 uses the single satellite with sensors as a template to create a constellation of satellites with unique names. Line 68 removes the single satellite that is not part of the constellation. Because the code to compute access and coverage are not written to use this satellite it can optionally be left in.

Listing A.7. myClasses.Constellation class

```

1 class Constellation:
2     def __init__(self, name=None, numPlanes=None, numSats=None, semiMaj=
    ↪ None, trueAnom=None, inclination=None, ecc=None, sensor=None):
3         if name is None:
4             self.name = 'Sats01'
5         else:
6             self.name = name
7         if numPlanes is None:

```

```

8         self.numPlanes = 5
9     else:
10         self.numPlanes = numPlanes
11     if numSats is None:
12         self.numSats = 5
13     else:
14         self.numSats = numSats
15     if semiMaj is None:
16         self.semiMaj = 6778.137
17     else:
18         self.semiMaj = semiMaj
19     if inclination is None:
20         self.inclination = 36
21     else:
22         self.inclination = inclination
23     if trueAnom is None:
24         self.trueAnom = 0
25     else:
26         self.trueAnom = trueAnom
27     if ecc is None:
28         self.ecc = 0
29     else:
30         self.ecc = ecc
31     if sensor is None:
32         self.sensor = [0 for row in range(2)]
33         self.sensor[0] = myClasses.sqrSensor(None, .61, .61, None, None
34             ↪ ,4.99e-7,1,None)
35         #This code is to determine the look angle for a GSD < 1m
36         rho = math.asin(6378.137/self.semiMaj) # This is angle from
37             ↪ sat to horizon
38         LA = 0 # intial look angle is nadir
39         notComplete = True
40         nadirGSD = (self.semiMaj-6378.137)*1000*self.sensor[0].
41             ↪ dWidth/self.sensor[0].focLen
42         if nadirGSD > 1: # if GSD>1m then stop
43             notComplete = False
44         while notComplete: #Try GSD at next degree look angle
45             eps = math.acos(math.sin(math.radians(LA+1))/math.sin(
46                 ↪ rho))
47             lam = math.radians(90-math.degrees(eps)-(LA+1))
48             newGSD = 6378.137*1000*math.sin(lam)*self.sensor[0].
49                 ↪ dWidth/(math.sin(math.radians(LA+1))*math.sin(eps)
50                 ↪ *self.sensor[0].focLen)
51             if newGSD < 1 and math.radians(LA+1) <= rho:
52                 LA = LA + 1
53             else:
54                 notComplete = False
55         self.sensor[1] = myClasses.rndSensor('Wide',LA)
56     else:
57         self.sensor = sensor
58     self.AoP = 0 #Arugement of perigee
59     self.RAAN = 0 #Right ascension of ascending node

```

```

54         self.M = 0          #Mean anomaly
55
56     def add(self, scen):
57         s = scen.socket
58         s.send('New / */Satellite Sat \n')
59         State = ('SetState */Satellite/Sat Classical J2Perturbation ' +
60                 ↪ 'UseScenarioInterval 5 J2000 ' + str(scen.startDate) + ' '
61                 ↪ ' + str(self.semiMaj*1000) + ' ' + str(self.ecc) + ' ' +
62                 ↪ str(self.inclination) + ' ' + str(self.AoP) + ' ' + str(
63                 ↪ self.RAAN) + ' ' + str(self.M) + ' \n')
64         s.send(State)
65     for x in range(2):
66         s.send('New / */Satellite/Sat/Sensor ' + str(self.sensor[x].
67                 ↪ name) + ' \n')
68         if self.sensor[x].senShape == 'square':
69             s.send('Define */Satellite/Sat/Sensor/' + str(self.sensor[x].
70                 ↪ name) + ' Rectangular ' + str(self.sensor[x].
71                 ↪ vHalfAng) + ' ' + str(self.sensor[x].hHalfAng) + ' '
72                 ↪ '\n')
73         elif self.sensor[x].senShape == 'round':
74             s.send('Define */Satellite/Sat/Sensor/' + str(self.
75                 ↪ sensor[x].name) + ' SimpleCone ' + str(self.sensor
76                 ↪ [x].HalfAng) + ' \n')
77         s.send('Walker */Satellite/Sat ModDelta ' + str(self.numPlanes)
78                 ↪ ' + ' ' + str(self.numSats) + ' ' + str(self.trueAnom) + ' '
79                 ↪ ' + str(360/self.numPlanes) + ' Yes ConstellationName ' +
80                 ↪ str(self.name) + ' SetUniqueNames\n')
81         s.send('Unload / */Satellite/Sat \n')

```

Aircraft.

The aircraft class shown in Listing A.8 is used to create a formation of aircraft, and is slightly more complicated than creating a constellation. Lines 3-29 are the basic initialization values. The sensors are defined using the same method that was used for a constellation. Lines 30-67 define the loiter patterns for an aircraft formation. For a single loiter pattern the loiter diameter is defined on line 31. The time to complete one loiter is defined on line 32 and is in seconds. This is calculated by dividing the circumference of the loiter by the cruising airspeed of the aircraft. The default STK basic UAV cruise true airspeed is 333.36 km/hr. The number of loiter turns is defined on line 33 and is calculated by dividing 8 hours by the time to

complete one loiter. Line 34 defines the way points of an aircraft formation as a numpy array. Since only one loiter is used there is only one way point. Lines 40-43 and 49-67 defined the way points for a formation with four loiter patterns and 16 loiter patterns respectively. The first two elements of each way point are the latitude and longitude. For circular loiter patterns this point corresponds to the eastern most point of the circle. The third argument is the loiter type and the last argument is the take-off delay time used for that circular loiter. The delay time is used to ensure the aircraft are timed to start their loiters at exactly the same time which is important for the phase offset to be correct. This was calculated by manually guessing and checking delay times.

Lines 68-113 define the function to add the aircraft to a scenario. There are three loops in this function and are defined by the variables x, y, and z. The outer loop, x, repeats three times since there are three aircraft per CAP. The middle loop, y, repeats for the number of loiter patterns defined in the area of interest. The inner loop, z, repeats for the number of aircraft in a single loiter pattern. The phase offset defined on line 75 is only applied to every other loiter pattern and is described by the time delay in seconds it would take an aircraft to travel to travel the offset distance. The delay on line 76 is used to space the aircraft evenly around a loiter pattern. Line 78-82 calculates the take-off time of each aircraft. Lines 83-94 create a new aircraft and define the default parameters. Line 85 only needs to be run the first time to create a copy of the UAV which is automatically saved by STK. Lines 95-113 define the route each aircraft will follow based on the way points.

Listing A.8. myClasses.Aircraft class

```

1 class Aircraft:
2     def __init__(self, name, boxes, acPerBox, acPhase, acAlt):
3         if name is None:
4             self.name = 'AC1'
5         else:

```

```

6         self.name = name
7         self.acPhase = acPhase
8         self.bboxes = bboxes
9         self.acPerBox = acPerBox
10        self.numAC = bboxes*acPerBox*3
11        self.acAlt = acAlt
12        self.sensor = [0 for row in range(2)]
13        self.sensor[0] = myClasses.sqrSensor()
14        #This code is to determine the look angle for a GSD < 1m
15        rho = math.asin(6378.137/(6378.137+self.acAlt)) # This is angle
           ↳ from AC to horizon
16        LA = 0 # initial look angle is nadir
17        notComplete = True
18        nadirGSD = (self.acAlt)*1000*self.sensor[0].dWidth/self.sensor
           ↳ [0].focLen
19        if nadirGSD > 1: # if GSD>1m then stop
20            notComplete = False
21        while notComplete: #Try GSD at next degree look angle
22            eps = math.acos(math.sin(math.radians(LA+1))/math.sin(rho))
23            lam = math.radians(90-math.degrees(eps)-(LA+1))
24            newGSD = 6378.137*1000*math.sin(lam)*self.sensor[0].dWidth/(
           ↳ math.sin(math.radians(LA+1))*math.sin(eps)*self.sensor
           ↳ [0].focLen)
25            if newGSD < 1 and math.radians(LA+1) <= rho:
26                LA = LA + 1
27            else:
28                notComplete = False
29        self.sensor[1] = myClasses.rndSensor('Wide',LA)
30        if self.bboxes == 1:
31            self.loitD = '283 km'
32            self.lapTime = 9601
33            self.loitTurns = '3'
34            self.wpts = np.array([[34,40.5, '"Holding-Circular"' ,0]])
35
36        if self.bboxes == 4:
37            self.loitD = '170 km'
38            self.lapTime = 5767
39            self.loitTurns = '5'
40            self.wpts = np.array([[33,40, '"Holding-Circular"' ,43],
41                                  [35,40, '"Holding-Circular"' ,0],
42                                  [35,42, '"Holding-Circular"' ,1863],
43                                  [33,42, '"Holding-Circular"' ,1990]])
44
45        if self.bboxes == 16:
46            self.loitD = '94 km'
47            self.lapTime = 3189
48            self.loitTurns = '9'
49            self.wpts = np.array([[32.5,40, '"Holding-Circular"' ,63],
50                                  [33.5,40, '"Holding-Circular"' ,315],
51                                  [34.5,40, '"Holding-Circular"' ,293],
52                                  [35.5,40, '"Holding-Circular"' ,0],
53

```

```

54         [35.5,41,"Holding-Circular",901],
55         [34.5,41,"Holding-Circular",1272],
56         [33.5,41,"Holding-Circular",1312],
57         [32.5,41,"Holding-Circular",1013],
58
59         [32.5,42,"Holding-Circular",1931],
60         [33.5,42,"Holding-Circular",2307],
61         [34.5,42,"Holding-Circular",2240],
62         [35.5,42,"Holding-Circular",1755],
63
64         [35.5,43,"Holding-Circular",2519],
65         [34.5,43,"Holding-Circular",3185],
66         [33.5,43,"Holding-Circular",3297],
67         [32.5,43,"Holding-Circular",2783]])
68     def add(self, scen):
69         s = scen.socket
70         count = 1
71         for x in range(3):# 3 aircraft per CAP
72             for y in range(len(self.wpts)):
73                 phaseShift = 0
74                 if (y+1) % 2 == 0:#only apply phase offset to every
75                     ↪ other loiter pattern
76                     phaseShift = self.lapTime*self.acPhase/360
77                 delay = self.lapTime/self.acPerBox
78                 for z in range(self.acPerBox):
79                     tot = scen.startTime + int(self.wpts[y,3])+z*delay+
80                     ↪ phaseShift
81                     totM, totS = divmod(tot, 60)
82                     totH, totM = divmod(totM, 60)
83                     totD, totH = divmod(totH, 24)
84                     TOT = (str(totD) + scen.startMonthYear + str(totH +
85                     ↪ x*8) + ':' + str(totM) + ':' + str(totS) + '
86                     ↪ .00')
87                     s.send("New / */Aircraft Aircraft" + str(count) + "\n
88                     ↪ ")
89                     s.send('SetPropagator */Aircraft/Aircraft'+str(count
90                     ↪ )+' MissionModeler \n')
91                     # s.send("MissionModeler */Aircraft/Aircraft" + str(
92                     ↪ count) + " Aircraft Copy \"Basic UAV\"" + '\n')
93                     s.send("MissionModeler */Aircraft/Aircraft" + str(
94                     ↪ count) + " Aircraft Choose \"Copy of Basic UAV
95                     ↪ \"\" + '\n')
96                     s.send("MissionModeler */Aircraft/Aircraft" + str(
97                     ↪ count) + " Aircraft SetValue
98                     ↪ DefaultCruiseAltitude "+str(self.acAlt)+' km\n
99                     ↪ ')
100                    s.send("MissionModeler */Aircraft/Aircraft" + str(
101                    ↪ count) + " Aircraft SetValue Ceiling " + str(
102                    ↪ self.acAlt) + " km \n")
103                    s.send("SetConstraint */Aircraft/Aircraft" + str(
104                    ↪ count) + " AngularRate Max 1.0" + '\n')
105                    s.send("New / */Aircraft/Aircraft" + str(count) + "/

```

```

91         ↪ Sensor "+ str(self.sensor[0].name) + '\n'
s.send("Define */Aircraft/Aircraft" + str(count) + "/
92         ↪ Sensor/" + str(self.sensor[0].name) + '
          ↪ Rectangular ' + str(self.sensor[0].vHalfAng)+
          ↪ ' +str(self.sensor[0].hHalfAng)+' \n')
93 s.send("New */Aircraft/Aircraft" + str(count) + "/
          ↪ Sensor "+str(self.sensor[1].name) + '\n')
s.send("Define */Aircraft/Aircraft" + str(count) + "/
          ↪ Sensor/" +str(self.sensor[1].name) + '
          ↪ SimpleCone ' + str(self.sensor[1].HalfAng) + '
          ↪ \n')
94 s.send("Point */Aircraft/Aircraft" + str(count) + "/
          ↪ Sensor/" +str(self.sensor[1].name) + '
          ↪ FixedInRef AzEl 0 90 '+' "Aircraft/Aircraft" +
          ↪ str(count)+ ' FlightPath" \n')
95 #Add aircraft waypoints
96 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + ' Procedure Add AsFirst SiteType
          ↪ Runway ProcedureType Takeoff \n')
97 s.send('MissionModeler */Aircraft/Aircraft' + str(
          ↪ count) + ' Procedure SetTime 1 "' + TOT + '"
          ↪ UTCG \n')
98 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + " Site 1 SetValue Latitude 34 deg \n")
99 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + " Site 1 SetValue Longitude 45 deg \n"
          ↪ )
100 #Add holding waypoint
101 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + ' Procedure Add After 1 SiteType
          ↪ Waypoint ProcedureType '+self.wpts[y,2] + ' \n
          ↪ ')
102 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + ' Procedure 2 SetValue Turns ' + self
          ↪ .loitTurns + ' \n')
103 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + ' Procedure 2 SetValue Diameter ' +
          ↪ self.loitD + ' \n')
104 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + ' Site 2 SetValue Latitude ' + str(
          ↪ self.wpts[y,0]) + ' deg \n')
105 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + ' Site 2 SetValue Longitude ' + str(
          ↪ self.wpts[y,1]) + ' deg \n')
106 #Add landing procedure
107 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + ' Procedure Add After 2 SiteType
          ↪ Runway ProcedureType Landing \n')
108 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + " Site 3 SetValue Latitude 34 deg \n")
109 s.send("MissionModeler */Aircraft/Aircraft" + str(
          ↪ count) + " Site 3 SetValue Longitude 45 deg \n

```

```

110         ↪ ")
        s.send("MissionModeler */Aircraft/Aircraft" + str(
            ↪ count) + " ConfigureAll" + '\n')
111     s.send("MissionModeler */Aircraft/Aircraft" + str(
            ↪ count) + " CalculateAll" + '\n')
112     s.send("MissionModeler */Aircraft/Aircraft" + str(
            ↪ count) + " SendNtfUpdate" + '\n')
113     count = count + 1

```

A.2 myFuncs.py

myFuncs.py contains the functions used to control the simulation, performance, and value components of the EA. The packages shown in Listing A.9 are needed by the functions in this file.

Listing A.9. myFuncs.py included packages

```

1 import os
2 import time
3 import myFuncs
4 import myClasses
5 import math
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import datetime
9 import gc
10 import cPickle as pickle

```

conceptEval.

This function is the main function that controls the EA components to create a simulation, evaluate the performance of the systems, and calculate and return an overall value. The code is fairly well commented and shown in Listing A.10. To save time the first thing this function does is check to see if the scenario has been run before and the value can be read in. This is shown in lines 9-14. If not a scenario is created and a ground vehicle and area of interest are added to the scenario in lines 15-21. Depending on the parameters passes to the function, either a constellation or aircraft formation or both are created in lines 22-28. Coverage, access, GSD, and value are calculated in lines 29-56 depending on what combination of constellations and aircraft formations are present. The functions used for these calculation will be shown later. The remaining lines save the value to a binary file for use if the scenario is created later and a text file describing the scenario.

Listing A.10. myFuncs.conceptEval function

```

1 def conceptEval(orbPlane, satsPerPlane, satPhase, satAlt, boxes,
    ↪ acPerBox, acPhase, acAlt):
2     gc.enable() # Enable garbage collection
3     start_time = time.time() # Time used to tell how long the
    ↪ calculation took
4     #Check if a valid scenario; if invalid return zero
5     if orbPlane == 0 and boxes == 0:
6         print('Invalid choice')
7         val = 0
8     else:
9         #Check if scen has been run before
10        evalName = ('Scen_' + str(orbPlane) + '_' + str(satsPerPlane) +
    ↪ '_' + str(satPhase) + '_' + str(satAlt) + '_' + str(boxes)
    ↪ + '_' + str(acPerBox) + '_' + str(acPhase) + '_' + str(
    ↪ acAlt))
11        if os.path.exists(str(os.getcwd()) + '\\Scenarios\\' + evalName+
    ↪ '\\Value.p'):
12            #If so read in value from that scenario and return it
13            with open(str(os.getcwd()) + '\\Scenarios\\' + evalName+ '\\
    ↪ Value.p', 'rb') as f:
14                val = pickle.load(f)
15        else:
16            #Else create new scenario with a ground vehicle and area of
    ↪ interest
17            Scen = myClasses.Scenario(evalName)
18            gv1 = myClasses.GroundVehicle('Gv1')
19            gv1.add(Scen)
20            area = myClasses.AOI('Area')
21            area.add(Scen)
22            #Add satellites and aircraft
23            if orbPlane != 0: # If not 0 then create the satellite
    ↪ constellation
24                con1 = myClasses.Constellation('MyConst', orbPlane,
    ↪ satsPerPlane, float(6378.137 + satAlt), satPhase)
25                con1.add(Scen)
26            if boxes != 0: # if not 0 then create the aircraft formation
27                ac1 = myClasses.Aircraft('MyAir', boxes, acPerBox, acPhase
    ↪ , acAlt)
28                ac1.add(Scen)
29            #Calculate coverage, access, and value
30            if orbPlane != 0 and boxes != 0: #if sats and ac do this
31                myFuncs.calcCovAOI(Scen, area, con1, ac1)
32                myFuncs.calcSat2GVAccess(Scen, gv1, con1)
33                myFuncs.calcAC2GVAccess(Scen, gv1, ac1)
34                s = Scen.socket
35                s.send('ConFile / "' + Scen.saveLoc + 'connect.txt"\n')#
    ↪ All previous commands were written to this file
36                s.send('ConControl / HighSpeedOff\n') #Was set in
    ↪ scenario creation
37                s.send('ConControl / Disconnect\n') #Closes connection,
    ↪ starts text file connection

```

```

38         GSD = myFuncs.calcGSD(Scen, gv1, con1, ac1)
39     elif orbPlane == 0: #else if just ac do this
40         myFuncs.calcCovAOI(Scen, area, None, ac1)
41         myFuncs.calcAC2GVAccess(Scen, gv1, ac1)
42         s = Scen.socket
43         s.send('ConFile / "' + Scen.saveLoc + 'connect.txt'\n')#
44             ↪ All previous commands were written to this file
45         s.send('ConControl / HighSpeedOff\n')#Was set in
46             ↪ scenario creation
47         s.send('ConControl / Disconnect\n')#Closes connection,
48             ↪ starts text file connection
49         GSD = myFuncs.calcGSD(Scen, gv1, None, ac1)
50     else:#else satellites so do this
51         myFuncs.calcCovAOI(Scen, area, con1, None)
52         myFuncs.calcSat2GVAccess(Scen, gv1, con1)
53         s = Scen.socket
54         s.send('ConFile / "' + Scen.saveLoc + 'connect.txt'\n')#
55             ↪ All previous commands were written to this file
56         s.send('ConControl / HighSpeedOff\n')#Was set in
57             ↪ scenario creation
58         s.send('ConControl / Disconnect\n')#Closes connection,
59             ↪ starts text file connection
60         GSD = myFuncs.calcGSD(Scen, gv1, con1, None)
61         Cov = myFuncs.readCov(Scen)
62         val = myFuncs.calcVal(Scen, GSD, Cov)
63         #Close socket, stop time, write outputs
64         s.close()
65         totTime = time.time() - start_time
66         with open(str(os.getcwd()) + '\Scenarios\' + evalName+ '\
67             ↪ Value.p', 'wb') as f:
68             pickle.dump(val, f) #This saves the value in a binary
69             ↪ file
70         with open(str(os.getcwd()) + '\Scenarios\' + evalName+ '\
71             ↪ _ScenarioInfo.txt', 'w') as f:
72             f.write('Orbit Planes: ' + str(orbPlane) + "\n")
73             f.write('Satellites Per Plane: ' + str(satsPerPlane) + "\
74             ↪ \n")
75             f.write('Satellite True Anomaly Offset: ' + str(satPhase
76             ↪ ) + "\n")
77             f.write('Satellite Altitude: ' + str(satAlt) + "\n")
78             f.write('AOI Sub-boxes: ' + str(boxes) + "\n")
79             f.write('Aircraft Per Box: ' + str(acPerBox) + "\n")
80             f.write('Aircraft Radial Phase Offset: ' + str(acPhase)
81             ↪ + "\n")
82             f.write('Aircraft Cruise Altitude: ' + str(acAlt) + "\n")
83             ↪ )
84             f.write('Total Run Time: ' + str(totTime)+ '\n')
85             f.write('Overall Value: ' + str(val) + '\n')
86
87     gc.collect()
88     return val

```

calcCovAOI.

This function calculates the percent of the area of interest covered with respect to time. This function appends all of the commands to the text file created by the scenario class for STK to read in. Once the socket is disconnected, all the commands in this file will be executed. Line 2 opens the file to append to it and line 3 sets the output format to be used. Lines 4-10 define the round sensor on the satellites as the sensor used to calculate coverage. This is repeated for every satellite in the constellation. The same process is used for the aircraft in lines 11-14. Lines 15-19 compute the access and save the results in a text file using the custom style *Percent Coverage*. This style outputs only the scenario time and the percent of the area of interest covered at each time step of one second.

Listing A.11. myFuncs.calcAOI function

```
1 def calcCovAOI(scen, area, Sats=None, AC=None):
2     mr = open(scen.saveLoc + 'connect.txt', 'a')
3     mr.write("ExportConfig / Connection Headers None KeepReportLines Off
4         ↪ " + "ShowStartStop Off WriteReportTitle Off WriteObjectNames
5         ↪ Off " + "WriteSectionTitles Off" + '\n')
6     if Sats is not None:
7         count = 1
8         for x in range(1, Sats.numPlanes+1):
9             for y in range(1, Sats.numSats+1):
10                 mr.write('Cov */CoverageDefinition/' + area.name + '
11                     ↪ Asset */Satellite/Sat' + str(x) + str(y) + '/Sensor
12                     ↪ /' + str(Sats.sensor[1].name) + str(count) + '
13                     ↪ Assign \n')
14                 mr.write('Cov */CoverageDefinition/' + area.name + '
15                     ↪ Asset */Satellite/Sat' + str(x) + str(y) + '/Sensor
16                     ↪ /' + str(Sats.sensor[1].name) + str(count) + '
17                     ↪ Activate \n')
18                 count = count + 1
19     if AC is not None:
20         for x in range(1, AC.numAC+1):
21             mr.write('Cov */CoverageDefinition/' + area.name + ' Asset
22                 ↪ */Aircraft/Aircraft' + str(x) + '/Sensor/' + str(AC.
23                 ↪ sensor[1].name) + ' Assign \n')
24             mr.write('Cov */CoverageDefinition/' + area.name + ' Asset
25                 ↪ */Aircraft/Aircraft' + str(x) + '/Sensor/' + str(AC.
26                 ↪ sensor[1].name) + ' Activate \n')
```

```

15     mr.write('Cov */CoverageDefinition/' + area.name + ' Access
        ↪ AutoRecompute On \n')
16     mr.write('Cov */CoverageDefinition/' + area.name + ' Access SaveMode
        ↪ Save \n')
17     mr.write('Cov */CoverageDefinition/' + area.name + ' Access Compute
        ↪ \n')
18     mr.write('ReportCreate */CoverageDefinition/' + area.name + ' Style "
        ↪ My Styles/Percent Coverage" Type Export File ' + scen.saveLoc
        ↪ + 'PercentCoverage.txt TimePeriod UseScenarioInterval ' + '
        ↪ TimeStep 1 \n')
19     mr.close()

```

calcSat2GVAccess and calcAC2GVAccess.

The functions *calcSat2GVAccess* and *calcAC2GVAccess* shown in Listings A.12 and A.13 perform the same functions, but for either satellites or aircraft. The text file created by the scenario class is appended to again for both of these functions. The first command used by both functions sets the square sensor on the satellite or aircraft to point at the ground vehicle. The second command used by both computes the access between each satellite or aircraft and the ground vehicle, and the last command exports this data in text file containing the azimuth-elevation-range between the sensor and ground vehicle for the entire scenario time period in one second time intervals.

Listing A.12. myFuncs.calcSat2GVAccess function

```

1 def calcSat2GVAccess(scen, GV, Sats):
2     mr = open(scen.saveLoc + 'connect.txt', 'a')
3     count = 1
4     for x in range(1, Sats.numPlanes+1):
5         for y in range(1, Sats.numSats+1):
6             mr.write('Point */Satellite/Sat' + str(x) + str(y) + '/Sensor
                ↪ /'+str(Sats.sensor[0].name) + str(count) + ' Targeted
                ↪ Add GroundVehicle/'+str(GV.name)+' Rotate SaveAccesses
                ↪ On\n')
7             mr.write('Access */Satellite/Sat' + str(x) + str(y) + '/
                ↪ Sensor/'+str(Sats.sensor[0].name) + str(count) + ' */
                ↪ GroundVehicle/'+str(GV.name)+' TimePeriod
                ↪ UseScenarioInterval \n')

```

```

8         mr.write('ReportCreate */Satellite/Sat' + str(x) + str(y) +
        ↪ '/Sensor/' + str(Sats.sensor[0].name) + str(count) + '
        ↪ Style AER Type Export File ' + scen.saveLoc + 'Sat' +
        ↪ str(x) + str(y) + 'Access.txt TimePeriod
        ↪ UseAccessTimes TimeStep 1 AccessObject */GroundVehicle
        ↪ /' + str(GV.name) + ' \n')
9         count = count + 1
10    mr.close()

```

Listing A.13. myFuncs.calcAC2GVAccess function

```

1 def calcAC2GVAccess(scen, GV, AC):
2     mr = open(scen.saveLoc + 'connect.txt', 'a')
3     for x in range(1, AC.numAC+1):
4         mr.write('Point */Aircraft/Aircraft' + str(x) + '/Sensor/' +
5                 str(AC.sensor[0].name) + ' Targeted Add GroundVehicle/'
6                 + str(GV.name) + ' Rotate SaveAccesses On\n')
7         mr.write('Access */Aircraft/Aircraft' + str(x) + '/Sensor/' +
8                 str(AC.sensor[0].name) + ' */GroundVehicle/' + str(GV.name)
9                 + ' TimePeriod UseScenarioInterval \n')
10        mr.write('ReportCreate */Aircraft/Aircraft' + str(x) +
11                '/Sensor/' + str(AC.sensor[0].name) +
12                ' Style AER Type Export File ' + scen.saveLoc +
13                ' Aircraft' + str(x) + 'Access.txt TimePeriod UseAccessTimes
        ↪ ' +
14                'TimeStep 1 AccessObject */GroundVehicle/' + str(GV.name) + ' \n
        ↪ ')
15    mr.close()

```

calcGSD.

This function is used to create the GSD function that will be used to determine value. The best achievable GSD by either a satellite or aircraft will be used as the GSD value for that single time step. As shown in Listing A.14, the inputs to this function are a scenario object, ground vehicle, constellation, and aircraft formation. The ground vehicle is actually unused and was never taken out of the code. This function first creates a matrix called *outfile* with six columns and as many rows as there are seconds in the scenario to store values in and is shown in lines 2-23. The first five columns are the scenario time broken into days, month/year, hour, minutes, and seconds. The last column is for the best GSD value. Lines 30-89 are

only executed if a constellation was passed into the function, and lines 90-146 only execute if an aircraft formation was passed into the function as well. Both sets of code perform similar functions for either aircraft or satellites. The first thing both sets of code check are for all the AER text files to be created. Waiting for a single file to be created and then immediately using it to calculate GSD created bugs. Python would read a file that had not been completely written to and only use the little data that had been written. This is why the code waits for all files to be created before reading any of them.

In lines 40-81 for the satellites, each AER file is read into a temporary matrix called *contents* that contains 7 columns and as many rows as there are in the AER file. For each time step (or row) of the AER file, the GSD is calculated using the elevation and range. The GSD is then stored in the last column while the first six mirror the AER file. This file is then saved on lines 82-86.

The method for determining the best GSD at any given time is simple. The matrix *outfile* contains as many rows as there are seconds in the scenario. Each row of an AER file contains the time, azimuth, elevation, and range. Knowing the time the scenario started at allows one to convert the time in an AER file to an index for the matrix *outfile*. This is shown in lines 74-77 for a satellite. Once the index is known, the GSD between what is currently the best can be compared against the new GSD calculated at the time step in the AER file. If the new GSD is better than the value in the matrix *outfile*, then it replaces the current value in *outfile*.

Once this process has been repeated for all the aircraft and satellite AER files *outfile* will contain only the best GSD values for the scenario. *Outfile* is then saved in a binary form on line 147 for use if needed later. The remaining lines plot the GSD over time and save it to the current working directory. The value returned by

the function is only the GSD values and does not contain the times with each row.

Listing A.14. myFuncs.calcGSD function

```

1 def calcGSD(scen, GV, Sats=None, AC=None):
2     outfile = []
3     outfile = [[0 for col in xrange(6)] for row in xrange(scen.Secs)]
4     #This for loop initializes the entire matrix
5     for x in xrange(0, scen.Secs):
6         if x == 0: #Scenario Start Time
7             outfile[x][0] = scen.startDay
8             outfile[x][1] = scen.startMonthYear
9             outfile[x][2] = scen.startHour
10            outfile[x][3] = scen.startMin
11            outfile[x][4] = scen.startSec
12            outfile[x][5] = None
13        else:
14            currentT = scen.startTime + x
15            curM, curS = divmod(currentT, 60)
16            curH, curM = divmod(curM, 60)
17            curD, curH = divmod(curH, 24)
18            outfile[x][0] = curD
19            outfile[x][1] = scen.startMonthYear
20            outfile[x][2] = curH
21            outfile[x][3] = curM
22            outfile[x][4] = curS
23            outfile[x][5] = None
24    notComplete = True
25    while notComplete:
26        if os.path.isfile(scen.saveLoc+'PercentCoverage.txt'):#Checking
27            ↪ this file is not a mistake. It ensures it has been
28            ↪ created for the readCov function
29            notComplete = False
30        else:
31            time.sleep(0.1)
32    if Sats is not None:
33        #This for loop runs through all the Satellite access files
34        for x in range(1, Sats.numPlanes+1):
35            for y in range(1, Sats.numSats+1):
36                notComplete = True
37                while notComplete:
38                    if os.path.isfile(scen.saveLoc + 'Sat' + str(x) +
39                        ↪ str(y) + 'Access.txt'):
40                        notComplete = False
41                    else:
42                        time.sleep(0.1)
43    for x in range(1, Sats.numPlanes+1):
44        for y in range(1, Sats.numSats+1):
45            #Check if AER file is empty, if not do this else create
46            ↪ empty GSD file
47            if os.stat(scen.saveLoc + 'Sat' + str(x) + str(y) + '
48                ↪ Access.txt').st_size != 0 :

```

```

44         with open(scen.saveLoc + 'Sat' + str(x) + str(y) + ',
45             ↪ Access.txt', 'r') as f:
46             for i, l in enumerate(f):
47                 pass
48             fileEnd = i
49             contents = []
50             contents = [[0 for col in xrange(7)] for row in
51                 ↪ xrange(fileEnd)]
52             with open(scen.saveLoc + 'Sat' + str(x) + str(y) + ',
53                 ↪ Access.txt', 'r') as f:
54                 lineOtext = []
55                 lineOtext = [0 for row in xrange(fileEnd)]
56                 for txt in range(0, fileEnd):
57                     lineOtext[txt] = f.readline()
58                     words = lineOtext[txt].split(" ")
59                     AzElRh = words[3].split(",")
60                     day = words[0]
61                     contents[txt][0] = day
62                     month = words[1]
63                     contents[txt][1] = month
64                     year = words[2]
65                     contents[txt][2] = year
66                     minshrs = AzElRh[0]
67                     contents[txt][3] = minshrs
68                     sensEl = AzElRh[2]
69                     contents[txt][4] = sensEl
70                     sensRange = AzElRh[3]
71                     contents[txt][5] = sensRange.strip('\n')
72                     GSD = myFuncs.GSD(Sats, sensRange, sensEl)
73                     contents[txt][6] = str(GSD)
74                     hrs = int(minshrs[0:2])
75                     mins = int(minshrs[3:5])
76                     secs = int(float(minshrs[6:8]))
77                     #GSDTime is the conversion of the scenario
78                     ↪ time to month-seconds
79                     GSDTime = int(day)*24*60*60 + int(hrs)*60*60
80                     ↪ + int(mins)*60 + int(secs)
81                     #loc is the index for this GSD in the matrix
82                     loc = GSDTime - scen.startTime
83                     if outfile[loc][5] is None:
84                         outfile[loc][5] = GSD
85                     elif GSD < outfile[loc][5]:
86                         outfile[loc][5] = GSD
87                     mr = open(scen.saveLoc + 'Sat' + str(x) + str(y)
88                         ↪ + 'GSD.txt', 'w')
89                     mr.write('Day Month Year Time Elev Range GSD' +
90                         ↪ "\n")
91                     for line in contents:
92                         mr.write(" ".join(str(elem) for elem in line
93                             ↪ ) + "\n") # prints DTG, min Range etc,
94                             ↪ for sat that gives min range data
95                             ↪ altitude

```

```

86         mr.close()
87     else: #Create a blank file
88         mr = open(scen.saveLoc + 'Sat' + str(x) + str(y) + '
            ↪ GSD.txt', 'w')
89         mr.close()
90 if AC is not None:
91     for x in range(1, AC.numAC+1):
92         notComplete = True
93         while notComplete:
94             if os.path.isfile(scen.saveLoc+'Aircraft'+str(x)+'Access
            ↪ .txt'):
95                 notComplete = False
96             else:
97                 time.sleep(0.1)
98 for x in range(1, AC.numAC+1):
99     #Check if AER file is empty, if not do this else create
    ↪ empty GSD file
100 if os.stat(scen.saveLoc+'Aircraft'+str(x)+'Access.txt').
    ↪ st_size != 0 :
101     with open(scen.saveLoc+'Aircraft'+str(x)+'Access.txt', 'r
        ↪ ') as f:
102         for i, l in enumerate(f):
103             pass
104     fileEnd = i
105     contents = []
106     contents = [[0 for col in xrange(7)] for row in xrange(
        ↪ fileEnd)]
107     with open(scen.saveLoc+'Aircraft'+str(x)+'Access.txt', 'r
        ↪ ') as f:
108         lineOtext = []
109         lineOtext = [0 for row in xrange(fileEnd)]
110         for txt in range(0, fileEnd):
111             lineOtext[txt] = f.readline()
112             words = lineOtext[txt].split(" ")
113             AzElRh = words[3].split(",")
114             day = words[0]
115             contents[txt][0] = day
116             month = words[1]
117             contents[txt][1] = month
118             year = words[2]
119             contents[txt][2] = year
120             minshrs = AzElRh[0]
121             contents[txt][3] = minshrs
122             sensEl = AzElRh[2]
123             contents[txt][4] = sensEl
124             sensRange = AzElRh[3]
125             contents[txt][5] = sensRange.strip('\n')
126             GSD = myFuncs.acGSD(AC, sensRange, sensEl)
127             contents[txt][6] = str(GSD)
128             hrs = int(minshrs[0:2])
129             mins = int(minshrs[3:5])
130             secs = int(float(minshrs[6:8]))

```

```

131         #GSDTime is the conversion of the scenario time
           ↪ to month-seconds
132         GSDTime = int(day)*24*60*60 + int(hrs)*60*60 +
           ↪ int(mins)*60 + int(secs)
133         #loc is the index for this GSD in the matrix
134         loc = GSDTime - scen.startTime
135         if outfile[loc][5] is None:
136             outfile[loc][5] = str(GSD)
137         elif GSD < outfile[loc][5]:
138             outfile[loc][5] = str(GSD)
139         mr = open(scen.saveLoc+'Aircraft'+str(x)+'GSD.txt',
           ↪ 'w')
140         mr.write('Day Month Year Time Elev Range GSD' + "\n"
           ↪ )
141         for line in contents:
142             mr.write(" ".join(str(elem) for elem in line) +
           ↪ "\n") # prints DTG, min Range etc, for sat
           ↪ that gives min range data altitude
143         mr.close()
144     else: #Create a blank file
145         mr = open(scen.saveLoc + 'Aircraft' + str(x) + 'GSD.txt',
           ↪ , 'w')
146         mr.close()
147     with open(scen.saveLoc + str(scen.name) + 'OUTPUT.p', 'wb') as mr:
148         pickle.dump(outfile, mr)
149     plotGSD = []
150     plotGSD = [0 for row in xrange(scen.Secs-1)]
151     for x in xrange(1,scen.Secs):
152         plotGSD[x-1] = outfile[x][5]
153     if plotGSD.count(None) < len(plotGSD):
154         DT = datetime.datetime(2015, 1, scen.startDay, scen.startHour,
           ↪ 0, 0)
155         t = [DT + datetime.timedelta(seconds=i) for i in range(len(
           ↪ plotGSD))]
156         plt.figure(figsize=(12,8))
157         plt.xlabel('Time (s) from 1 Jan 2015 10:00:000 UTC')
158         plt.ylabel('GSD (m)')
159         plt.title(scen.name)
160         plt.plot(t,plotGSD)
161         plt.savefig(scen.saveLoc + scen.name + 'GSD.png', dpi=300)
162         plt.clf()
163         plt.close('all')
164     with open(scen.saveLoc + str(scen.name) + 'GSD.p', 'wb') as f:
165         pickle.dump(plotGSD, f)
166     mr = open(scen.saveLoc + '\\Testfile.txt', 'w')
167     for line in outfile:
168         mr.write(" ".join(str(elem) for elem in line) + "\n") # prints
           ↪ DTG, min Range etc, for sat that gives min range data
           ↪ altitude
169     mr.close()
170     return plotGSD

```

readCov.

This function is similar to the myFuncs.calcGSD function shown in Listing A.14.

However, this function is much simpler because only a single file contains the information needed.

Listing A.15. myFuncs.readCov function

```
1 def readCov(scen):
2     outfile = []
3     outfile = [[0 for col in xrange(6)] for row in xrange(scen.Secs)]
4     #This for loop initializes the entire matrix
5     for x in xrange(0,scen.Secs):
6         if x == 0: #Scenario Start Time
7             outfile[x][0] = scen.startDay
8             outfile[x][1] = scen.startMonthYear
9             outfile[x][2] = scen.startHour
10            outfile[x][3] = scen.startMin
11            outfile[x][4] = scen.startSec
12            outfile[x][5] = 0
13        else:
14            currentT = scen.startTime + x
15            curM, curS = divmod(currentT, 60)
16            curH, curM = divmod(curM, 60)
17            curD, curH = divmod(curH, 24)
18
19            outfile[x][0] = curD
20            outfile[x][1] = scen.startMonthYear
21            outfile[x][2] = curH
22            outfile[x][3] = curM
23            outfile[x][4] = curS
24            outfile[x][5] = 0
25    notComplete = True
26    while notComplete:
27        if os.path.isfile(scen.saveLoc+'PercentCoverage.txt'):
28            notComplete = False
29        else:
30            time.sleep(0.1)
31    #Check if AER file is empty, if not do this else create empty GSD
32    ↪ file
33    if os.stat(scen.saveLoc+'PercentCoverage.txt').st_size != 0 :
34        with open(scen.saveLoc+'PercentCoverage.txt','r') as f:
35            for i, l in enumerate(f):
36                pass
37            fileEnd = i
38            contents = []
39            contents = [[0 for col in xrange(7)] for row in xrange(fileEnd)]
40            with open(scen.saveLoc+'PercentCoverage.txt','r') as f:
41                lineOtext = []
```

```

41     lineOtext = [0 for row in xrange(fileEnd)]
42     for txt in range(0, fileEnd):
43         lineOtext[txt] = f.readline()
44         words = lineOtext[txt].split(" ")
45         AzElRh = words[3].split(",")
46         day = words[0]
47         contents[txt][0] = day
48         month = words[1]
49         contents[txt][1] = month
50         year = words[2]
51         contents[txt][2] = year
52         minshrs = AzElRh[0]
53         contents[txt][3] = minshrs
54         Cov = AzElRh[1]
55         Cov = Cov.strip('\n')
56         contents[txt][4] = Cov
57         hrs = int(minshrs[0:2])
58         mins = int(minshrs[3:5])
59         secs = int(float(minshrs[6:8]))
60         #CovTime is the conversion of the scenario time to month
61         ↪ -seconds
62         CovTime = int(day)*24*60*60 + int(hrs)*60*60 + int(mins)
63         ↪ *60 + int(secs)
64         #loc is the index for this Cov in the matrix
65         loc = CovTime - scen.startTime
66         if outfile[loc][5] is 0:
67             outfile[loc][5] = float(Cov)/100
68     plotCov = []
69     plotCov = [0 for row in xrange(scen.Secs-1)]
70     for x in xrange(1,scen.Secs):
71         plotCov[x-1] = outfile[x][5]
72     if plotCov.count(None) < len(plotCov):
73         DT = datetime.datetime(2015, 1, scen.startDay, scen.startHour,
74         ↪ 0, 0)
75         t = [DT + datetime.timedelta(seconds=i) for i in range(len(
76         ↪ plotCov))]
77         plt.figure(figsize=(12,8))
78         plt.xlabel('Time (s) from 1 Jan 2015 10:00:000 UTC')
79         plt.ylabel('Percent Coverage (%)')
80         plt.title(scen.name)
81         plt.plot(t,plotCov)
82         plt.savefig(scen.saveLoc + scen.name + 'Coverage.png', dpi=300)
83         plt.clf()
84         plt.close('all')
85     with open(scen.saveLoc + str(scen.name) + 'Coverage.p', 'wb') as f:
86         pickle.dump(plotCov, f)
87     return plotCov

```

GSD and acGSD.

This function performs the calculations to determine the GSD for satellites. The equations used are directly from the book *Space Mission Engineering: The New SMAD* and include corrections from the errata (Wertz *et al.*, 2011). Some terms could be canceled out to simplify the equation to the one shown in Chapter III, but they were left in to show the steps. The only difference between this function and the aircraft function is the degradation term in the denominator. Aircraft use a flat earth model and the off nadir look angle is also the degradation elevation angle. Satellites account for the curvature of the earth and require extra calculations.

Listing A.16. myFuncs.GSD function

```
1 def GSD(Sats, strRange, strElev):
2     diam = float(Sats.sensor[0].apDiam)
3     minRange = float(strRange)
4     minRange = minRange*1000
5     elev = math.radians(90 + float(strElev))
6     # Sensor parameters
7     d= Sats.sensor[0].dWidth #pixel detector size
8     f= Sats.sensor[0].focLen  #Focal length
9     lambda1= Sats.sensor[0].lambda #detector operating wavelength (lambda)
10    ↪ set at mean wavelength for SWIR
11    Q = lambda1*f/(diam*d)      #SME:SMAD Eq. 17-10
12    rho = math.asin(6378137/(Sats.semiMaj*1000)) #SME:SMAD Eq. 8-26
13    epsil = math.acos(math.sin(elev)/math.sin(rho)) #SME:SMAD Eq. 8-28
14    gndRes = (minRange*lambda1)/(diam*math.sin(epsil)) #SME:SMAD Eq.
15    ↪ 17-6 with degradation 1/sin(epsil)
16    X = gndRes/Q #SME:SMAD Eq. 17-7
17    return X
```

Listing A.17. myFuncs.acGSD function

```
1 def acGSD(AC, strRange, strElev):
2     minRange = float(strRange)
3     minRange = minRange*1000
4     elev = math.radians(90 + float(strElev))
5     # Sensor parameters
6     d= AC.sensor[0].dWidth #pixel detector size
7     f= AC.sensor[0].focLen  #Focal length
8     X = minRange*d/(f*math.cos(elev))
9     return X
```

calcVal.

This function is used to compute the value given a coverage function and GSD function. Lines 2-5 in Listing A.18 are applying the linear value curve to the GSD function which normalizes it between zero and one. The new modified GSD function is plotted and saved in lines 6-15. The coverage and GSD are then scaled in half and added together to create the value function and is shown in lines 16-19. The value function is plotted and saved in lines 20-29. The value function is then averaged and returned as the overall value in lines 30 and 31.

Listing A.18. myFuncs.calcVal function

```
1 def calcVal(scen, GSD, Cov):
2     GSD[:] = [0 if x is None else float(x) for x in GSD]
3     GSD[:] = [0 if x > 1 else float(x) for x in GSD]
4     GSD[:] = [-1.25*x+1.25 if x > 0 else x for x in GSD]
5     GSD[:] = [1 if x > 1 else x for x in GSD]
6     DT = datetime.datetime(2015, 1, scen.startDay, scen.startHour, 0, 0)
7     t = [DT + datetime.timedelta(seconds=i) for i in range(len(GSD))]
8     plt.figure(figsize=(12,8))
9     plt.xlabel('Time (s) from 1 Jan 2015 10:00:000 UTC')
10    plt.ylabel('GSD (m)')
11    plt.title(scen.name)
12    plt.plot(t,GSD)
13    plt.savefig(scen.saveLoc + scen.name + 'modGSD.png', dpi=300)
14    plt.clf()
15    plt.close('all')
16    value = []
17    GSD[:] = [x*.5 for x in GSD]
18    Cov[:] = [x*.5 for x in Cov]
19    value[:] = np.add(GSD,Cov)
20    plt.figure(figsize=(12,8))
21    plt.xlabel('Time (s) from 1 Jan 2015 10:00:000 UTC')
22    plt.ylabel('Value')
23    plt.title(scen.name)
24    plt.plot(t,value, 'k')
25    plt.plot(t,Cov, 'b')
26    plt.plot(t,GSD, 'r')
27    plt.savefig(scen.saveLoc + scen.name + 'value.png', dpi=300)
28    plt.clf()
29    plt.close('all')
30    avgValue = sum(value)/len(value)
31    return avgValue
```

connect.

This function creates a connection to STK using a TCP/IP connection. This function was created entirely by Mr. David Meyer.

Listing A.19. myFuncs.connect function

```
1 def connect():
2     # Establishes the connection with STK, STK must be running first
3     import socket #imports a python class needed to establish TCP/IP
4     import sys
5     HOST = socket.gethostname()
6     PORT = 5001 # This is the default port identified by AGI
7     s = None # s is a socket object that we will use to pass info from
8         ↪ our Python program to STK
9     for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.
10        ↪ SOCK_STREAM):
11         af, socktype, proto, canonname, sa = res
12         try:
13             s = socket.socket(af, socktype, proto)
14         except socket.error:
15             s = None
16             continue
17         try:
18             s.connect(sa)
19         except socket.error:
20             s.close()
21             s = None
22             continue
23         break
24     if s is None:
25         print 'Could not open socket - Please start STK first'
26         sys.exit(1)
27     s.send('Unload / *\n')
```

Appendix B. MATLAB code

This appendix provides a detailed description of the MATLAB code generated for this research. Each section will describe a MATLAB file.

B.1 PythonCall.m

This script is invoked by ParaMagic to begin the EA. The values in the text file *input.txt* are read in on the first line. As described earlier, the input file contains a constraint on the number of systems that can be used and the upper and lower bounds for eight design variables. The number of variables is manually set on line 2 and the values read in from the text file are mapped to the upper and lower bounds arrays on lines 3 and 4. Only integers values for the variables will be used to keep the sample space within a reasonable number. The first two bounds correspond to the number of orbit planes and satellites per orbit plane. Third variable bounds corresponds to the true anomaly offset and the fourth is the satellite altitude. The remaining four variable bounds are the number of loiter patterns, aircraft per loiter pattern, phase offset, and aircraft altitude. Several variables are scaled to limit the sample space even further. This is accounted for and described later. The loiter pattern range was set manually to correspond to having 1, 4, or 16 loiter patterns. This is also accounted for and described later.

Lines 5-7 define other functions that will be used to evaluate a scenario, evaluate constraints, and save specific data. Line 9 define the options for the genetic algorithm (GA) function. Lines 10 and 11 are useful if performing more than one iteration to save data from each iteration. The GA is run on line 13 and uses the function handles, number of variables, upper and lower bounds, and options just defined. An important note is that the output function will write the best solution

found to *GenBest* after each generation. This will be described again with the output function but is worth noting twice. The remainder of this script writes the output to the text file *output.txt* which will be read in by ParaMagic.

Listing B.1. PythonCall.m MATLAB script

```

1 V = importdata('input.txt');
2 nvars = 8;
3 LB = [V(2), V(4), V(6), V(8)/100, 1, 1, V(14)/15, V(16)];
4 UB = [V(3), V(5), V(7)/5, V(9)/100, 3, 4, V(15)/15, V(17)];
5 ObjFunc = @scenEval;
6 ConstrFunc = @(x) myConstraints(x, V(1));
7 OutFunc = @myout;
8 IntCon = [1 2 3 4 5 6 7 8];
9 opts = gaoptimset('PlotFcns', @gaplotbestf, 'Generations', 10, 'OutputFcns',
    ↪ OutFunc, 'PopulationSize', 50);
10 GenBest = [];
11 S = struct('Iter', {struct('Pop', {}, 'GenBest', {})});
12 for i = 1
13     [x, fval, exitFlag, output, population, scores] = ga(ObjFunc, nvars
    ↪ , [], [], [], [], LB, UB, ConstrFunc, IntCon, opts);
14     S.Iter(i).Pop = population;
15     S.Iter(i).GenBest = GenBest;
16     GenBest = [];
17 end
18 if x(5) == 1
19     loit = 1;
20 else if x(5) == 2
21     loit = 4;
22     else if x(5) == 3
23         loit = 16;
24     end
25 end
26 end
27 output = {num2str(-fval, '%.3g'), ...
28     num2str(x(1)), ...
29     num2str(x(2)), ...
30     num2str(x(3)*5), ...
31     num2str(x(4)*100), ...
32     num2str(loit), ...
33     num2str(x(6)), ...
34     num2str(x(7)*15), ...
35     num2str(x(8))};
36 file = fopen('output.txt', 'w');
37 fprintf(file, strjoin(output));
38 exit

```

B.2 myConstraints.m

This is a MATLAB function used by the GA to check if constraints are satisfied or not. X is the array of variables for a particular concept and bnd is the constraint on the maximum number of systems that can be used. The number of loiter patterns is first determined on lines 2-8 and then the only inequality constraint is checked on line 9.

Listing B.2. myConstraints.m MATLAB function

```
1 function [ c, ceq ] = myConstraints( x, bnd )
2     loit = x(5);
3     if loit == 2
4         loit = 4;
5     else if loit == 3
6         loit = 16;
7     end
8 end
9 c = [x(1)*x(2)+3*loit*x(6)-bnd];
10 ceq = [];
11 end
```

B.3 myout.m

This function is used inside the GA to save the best result of each generation. It should be noted that this function uses two MATLAB commands that can potentially be dangerous. *evalin* and *assignin* are dangerous in the fact that they alter variables outside the current working space. Future work should avoid using/altering these without understanding how they fully work first.

Listing B.3. myout.m MATLAB function

```
1 function [state, options, optchanged] = myout(options, state, flag)
2 [Y, I] = min(state.Score);
3 val = state.Population(I,:);
4 optchanged = false;
5 switch flag
6     case 'init'
7         GenBest = evalin('base', 'GenBest');
8         temp = [state.Generation, Y, val];
9         GenBest = [GenBest; temp];
```

```

10         assignin( 'base' , 'GenBest' , GenBest );
11     case { 'iter' }
12         GenBest = evalin( 'base' , 'GenBest' );
13         temp = [ state.Generation , Y , val ];
14         GenBest = [ GenBest ; temp ];
15         assignin( 'base' , 'GenBest' , GenBest );
16     case 'done'
17 end

```

B.4 scenEval.m

This function is used by the GA to evaluate a candidate point. The variables are formatted to a Python list and then passed to an intermediate Python function which invokes the *conceptEval* function shown in Listing A.10. The returned value is multiplied by a negative since GA attempts to minimize and the goal is to maximize the system value.

Listing B.4. scenEval.m MATLAB function

```

1 function [ y ] = scenEval(x)
2 params = py.list([x(1) x(2) x(3) x(4) x(5) x(6) x(7) x(8)]);
3 y = -1*(py.mat2py.testfunc(params));
4 end

```

B.5 Parse.m

This is the function called by ParaMagic to return a single value in an array. The value is returned by writing it to the output file *output.txt*.

Listing B.5. Parse.m MATLAB function

```

1 function out = Parse( vec , loc )
2 out = vec(loc);
3 save( 'output.txt' , 'out' , '-ASCII' );
4 exit

```

B.6 mat2py.py

This is the intermediate function used to convert the MATLAB values and call the python function. The number of loiter patterns are interpreted first and then the

parameters are passed into the Python function to evaluate the concept. This is also where the values are converted from the integers to their respective values.

Listing B.6. mat2py.py Python function

```
1 import os
2 import myFuncs
3 def testfunc(x):
4     if (int(x[4])==1):
5         kbs = 1
6     elif (int(x[4])==2):
7         kbs = 4
8     elif (int(x[4])==3):
9         kbs = 16
10    val = myFuncs.conceptEval(int(x[0]),int(x[1]),int(x[2])*5,int(x[3])
    ↪ *100,kbs,int(x[5]),int(x[6])*15,int(x[7]))
11    return val
```

References

- Bartolomei, J., Hastings, D., de Neufville, R., & Rhodes, D. 2012. Engineering Systems Multiple Domain Matrix: an Organizing Framework for Modeling Large-Scale Complex Systems. *Systems Engineering*, **15**(1), 41–61.
- Colombi, John M, Miller, Michael E, Bohren, James S, & Howard, John K. 2014. Conceptual Design Using Executable Architectures for a Manned Mission to Mars. *IEEE Systems Journal*, 1–13.
- Curtis E. Lemay Center. 2015. *Introduction to Global Integrated ISR*.
- Dahmann, Judith. 2014. 1.4.3 System of Systems Pain Points. *INCOSE International Symposium*, **24**(1), 108–121.
- Deep, Kusum, Singh, Krishna Pratap, Kansal, M.L., & Mohan, C. 2009. A real coded genetic algorithm for solving integer and mixed integer optimization problems. *Applied Mathematics and Computation*, **212**(2), 505–518.
- Dempsey, Martin E. 2012. *Joint Operational Access Concept*.
- Department of Defense. 2012. *Sustaining U.S. Global Leadership: Priorities for 21st Century Defense*.
- Emery, John, Oberg, Elena, Robertson, Kelly, Sanchez, David, & Smuck, David. 2005. *The Utility and Logistics Impact of Small-Satellite Constellations in Matched Inclination Orbits*. Thesis, Air Force Insitute of Technology.
- Estefan, Jeff A. 2008. Survey of Model-Based Systems Engineering (MBSE) Methodologies, Rev. B. *INCOSE MBSE Focus Group*.
- Friedenthal, Sanford, Izumi, Lisa, Hill, Flint, & Street, Rosehaven. 2007. Object-Oriented Systems Engineering Method (OOSEM) applied to Joint Force Projection (JFP), a Lockheed Martin Integrating Concept (LMIC). *INCOSE International Symposium*, **17**(1), 1471–1491.
- Friedenthal, Sanford, Moore, Alan, & Steiner, Rick. 2009. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann.
- Haywas, Luke. 2016. *ARCHITECTING A MORE ROBUST UNMANNED AIRCRAFT SYSTEM OF SYSTEMS*. Thesis, Air Force Institute of Technology.
- INCOSE. 2015. *Systems Engineering Handbook*. 4th edn. John Wiley & Sons.
- James, Deborah Lee, & Welsh, Mark A. 2014. *Fiscal Year 2015 Air Force Posture Statement*.

- Jilla, Cyrus D., Miller, David W., & Sedwick, Raymond J. 2000. Application of Multidisciplinary Design Optimization Techniques to Distributed Satellite Systems. *Journal of Spacecraft and Rockets*, **37**(4), 481–490.
- Knowlan, Garrett, & Phillips, Craig. 2007. *White Paper : Analysis of Risk Associated with Integrating Sensors on Micro UAVs*. Tech. rept. Air Force Institute of Technology.
- Leachtenauer, Jon C., Malila, William, Irvine, John, Colburn, Linda, & Salvaggio, Nanette. 1997. General Image-Quality Equation: GIQE. *Applied Optics*, **36**(32), 8322.
- Maier, Mark W., & Rechtin, Eberhardt. 2009. *The Art of Systems Architecting, Third Edition (Systems Engineering)*. CRC Press.
- Satellite Imaging Corp. 2016. *GeoEye-2 Satellite Imagery and Satellite Sensor Specifications*.
- Thompson, Robert E. 2015. *A Methodology For The Optimization Of Disaggregated Space System Conceptual Designs*. Dissertation, Air Force Institute of Technology.
- Thompson, Robert E, Colombi, John M, & Black, Jonathan. 2015. Model-Based Conceptual Design Optimization Methods: Disaggregated Weather System Follow-On. *Journal Of Spacecraft and Rockets*, 1–17.
- United States Air Force. 2013. *Air Force ISR 2023: Delivering Decision Advantage*.
- Wertz, James R., Everett, David F., & Puschell, Jeffery J. (eds). 2011. *Space Mission Engineering: The New SMAD*. Microcosm Press.

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> <i>OMB No. 0704-0188</i>	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.				
1. REPORT DATE (DD-MM-YYYY) 24-03-2016		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From — To) Sept 2014 – Mar 2016
4. TITLE AND SUBTITLE Optimizing Multi-Domain System-of-Systems Using Model-Based Systems Engineering			5a. CONTRACT NUMBER	
			5b. GRANT NUMBER	
			5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Palmer, Cody G, 2d Lt			5d. PROJECT NUMBER	
			5e. TASK NUMBER	
			5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENV-MS-16-M-174	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RQVI Capt Jason Sutherlin Bldg 65, Rm 305W WPAFB, OH 45433 Jason.sutherlin@wpafb.af.mil			10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RQVI	
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				
13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.				
14. ABSTRACT System-of-systems (SoS) that exist across multiple domains can offer new solutions to complex problems. Model-based systems engineering (MBSE) is a popular approach used in Systems Engineering (SE) for architecting systems or SoS, but it is often difficult or time consuming to assess many different MBSE architectures. This research looks at applying the Object-Oriented System Engineering Method (OOSEM) to multi-domain SoS problems. The Systems Modeling Language (SysML) is used to create a multi-domain SoS composed of satellites and aircraft for surveillance missions such as search and rescue. An executable architecture is developed to assess and find a near optimal SoS. The executable architecture consists of five components; architecture, optimization, simulation, performance, and value. The architecture component contains attributes describing the SoS and bounds which are passed to the optimization component. The optimization component evaluates candidate solutions using simulations, performance metric calculations, and a value model. Once the optimizer has finished, the optimal parameters are passed back to the architecture component and updated in the SysML model. This research found, for a permissive environment, the optimal multi-domain SoS had a 7.4% and 28.7% increase in average value over the aircraft only and satellite only systems respectively.				
15. SUBJECT TERMS Model-Based Systems Engineering, System-of-Systems, Multi-Domain, Executable Architecture				
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U	UU	131
			19a. NAME OF RESPONSIBLE PERSON Dr. John M. Colombi, AFIT/ENV	
			19b. TELEPHONE NUMBER (Include Area Code) (937) 255-3636 x3347	