ONTOLOGY BASED REUSE INFRASTRUCTURE FOR TRAJECTORY SIMULATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

UMUT DURAK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
MECHANICAL ENGINEERING

JUNE 2007

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. S. Kemal İDER
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

Assoc. Prof. Dr. Halit OĞUZTÜZÜN	Prof. Dr. S. Kemal İDER
Co-Supervisor	Supervisor

Examining Committee Members

Prof. Dr. M. Kemal ÖZGÖREN	(METU, ME)	_____________
Prof. Dr. S. Kemal İDER	(METU, ME)	_____________
Prof. Dr. Faruk ELALDI	(Başkent Unv., ME)	_____________
Assoc.Prof. Dr. Göktürk ÜÇOLUK	(METU, CENG)	_____________
Asst. Prof. Dr. A. Buğra KOKU	(METU, ME)	_____________

Name, Last Name	:	Umut DURAK
Signature	:	

ABSTRACT

ONTOLOGY BASED REUSE INFRASTRUCTURE FOR TRAJECTORY SIMULATION

DURAK, Umut
Ph. D., Department of Mechanical Engineering
Supervisor: Prof. Dr. Kemal İDER
Co-Supervisor: Assoc. Dr. Halit OĞUZTÜZÜN
June 2007, 241 pages

In this research, we developed an ontology based reuse infrastructure for trajectory simulation and investigated the use of ontologies and domain engineering practices to develop a formalized methodology to make use of the experience and knowledge leveraged from the past trajectory simulation projects. Trajectory simulation in this context is a computational tool to calculate the flight path and other parameters of munition such as its orientation or angular rates during its operation

In this thesis, engineering knowledge to simulate the trajectory of a munition is captured in an ontology called Trajectory Simulation ONTology (TSONT). Concepts of trajectory simulation and the relation among these concepts are captured by using Web Ontology Language and presented as a domain model that is available for reuse.

Using the formalized domain knowledge, reuse infrastructure specifications are constructed to enable the reuse of software artifacts for two main programming paradigms, which are object oriented programming and function oriented programming. UML and application frameworks are used when constructing for object oriented paradigm. And data flow diagrams are used to formalize the design of the function oriented simulations to compute the trajectory of munition. Object oriented and function oriented platform independent designs are constructed to specify the infrastructure using the knowledge captured in TSONT and made available for reuse. With constructing two different designs for two different paradigms by using the same domain model, evidence of knowledge reuse were produced.

Three different case studies were carried out as infrastructure implementation. In the first case study, an object oriented application framework was developed in MATLAB for six degrees of freedom trajectory simulation using platform independent object oriented design. This framework is reused to develop two different simulations. Using the developed framework for two applications produced evidence of code reuse. In the second case, a point mass trajectory simulation framework is designed to be implemented in C# reusing the same platform independent object oriented design. This case produced the evidence of design reuse. In the last case study, a MATLAB Simulink Blockset is developed for point mass unguided trajectory simulations and two different simulations are built using the Blockset. By this case, we collected the evidence of code reuse also in function oriented paradigm.

Keywords: Trajectory Simulation, Engineering Ontologies, Ontology Driven Simulation, Simulation Reuse.

ÖZ

YÖRÜNGE BENZETİMİ İÇİN ONTOLOJİ TEMELLİ YENİDEN KULLANIM ALTYAPISI

DURAK, Umut
Doktora, Makina Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Kemal İDER
Ortak Tez Yöneticisi: Doç.Dr. Halit OĞUZTÜZÜN
Haziran 2007, 241 sayfa

Bu çalışmada, yörünge benzetimi için ontoloji tabanlı bir yeniden kullanım altyapısı geliştirilmiş, ontolojilerin ve alan mühendisliği yaklaşımlarının başarı ile tamamlanmış yörünge benzetimi projelerinde elde edilen tecrübelerin aktarılması için geliştirilmiş bir yöntem için kullanılması incelenmiştir. Bu bağlamda, yörünge benzetimi mühimmatın uçuşu boyunca konumu, yönelimi ve açısal hızları gibi uçuş parametrelerinin hesaplanması için kullanılan bir araç olarak tanımlanabilir.

Bu tez kapsamında, bir mühimmatın uçuş benzetiminin yapılabilmesi için gerekli olan mühendislik bilgisi kullanılarak TSONT isimli bir ontoloji geliştirilmiştir. Ağ Ontoloji Dili (Web Ontology Language) kullanılarak yörünge benzetimi kavramları ve bu kavramlar arasındaki ilişkiler modellenerek, yeniden kullanılabilecek bir alan modeli olarak kullanıcıya sunulmuştur.

Ontoloji biçiminde resmileştirilmiş alan bilgi birikimi kullanılarak, işlev yönelimli programlama veya nesne yönelimli programlama paradigmaları kullanılarak hazırlanan yazılım ürünlerinin yeniden kullanımına olanak sağlayacak bir yeniden kullanım altyapısı tanımlanmıştır. Nesne yönelimli programlama paradigması için geliştirilen yeniden kullanıp altyapısı için UML ve uygulama çerçeveleri pratiklerinden yararlanılırken, işlev yönelimli programlama paradigması için oluşturulan yeniden kullanım altyapısı için veri akış şemalarından yararlanılmıştır. Bu iki paradigmanın yeniden kullanım altyapılarını tanımlamak için TSONT kullanılarak platformdan bağımsız yazılım tasarımları geliştirilmiş ve yeniden kullanıma sunulmuştur. Aynı alan bilgisi kullanarak iki farklı tasarım geliştirilebilmesi, TSONT'ta modellenen bilgi birikiminin yeniden kullanılabildiği konusunda elimize kanıtlar sunmuştur.

Yeniden kullanım altyapısının uygulaması için üç farklı çalışma yapılmıştır. İlk çalışmada, platformdan bağımsız nesne yönelimli yazılım tasarımı temel alınarak MATLAB ortamında altı serbestlik dereceli yörünge bezetimleri için bir uygulama çerçevesi geliştirilmiştir. Daha sonra da bu çerçeve kullanılarak iki farklı benzetim geliştirilmiştir. Yeniden kullanım altyapısının bir parçası olarak geliştirilen bu uygulama çerçevesinin iki farklı benzetim geliştirmesinde kullanılması, altyapının kod yeniden kullanımını desteklediğine dair bir kanıt olarak değerlendirilmiştir. İkinci çalışmada gene aynı platformdan bağımsız nesne yönelimli yazılım tasarımı kullanılara bu sefer nokta kütle yörünge benzetimi için ve farklı bir platformda, C# dilinde geliştirilecek bir çerçeve tasarlanmıştır. Bu sayede de yeniden kullanım altyapısının tasarım yeniden kullanımını desteklediğine dair kanıtlara ulaşılmıştır. Son çalışmada işlev yönelimli yazılım tasarımı kullanılarak güdümsüz nokta kütle yörünge bezetimi için bir MATLAB Simulink Blockset'i geliştirilmiştir. Daha sonra da bu Blockset kullanılmak vasıtası ile iki farklı yörünge benzetimi geliştirilmiştir. Bu sayede de geliştirilen yeniden kullanım altyapısının, işlev yönelimli programlama paradigmasında da kod yeniden kullanımını desteklediği sonucuna ulaşılmıştır.

Anahtar Kelimeler: Yörünge Benzetimi, Mühendislik Ontolojileri, Ontoloji Tabanlı Benzetim, Simülasyon Yeniden Kullanımı.

To all who spent the best years of their lives on their doctoral studies…

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Assoc. Dr. Halit OĞUZTÜZÜN and Prof Dr. S.Kemal İDER for their guidance and insight through out this study. Also thanks to my previous supervisors Prof.Dr. Ömer A.ANLAĞAN and Prof.Dr.S. Engin KILIÇ for their support and encouragement on this study.

This work has been supported by TÜBİTAK-SAGE. I would also like to convey my thanks to the TÜBİTAK-SAGE Flight Dynamics Group members for their assistance and understanding.

I would like to thank my colleagues Dr. Gökmen MAHMUTYAZICIOĞLU, Koray DAYANÇ, Kadriye TİRYAKİ, Şamil KORKMAZ, Fatih GERİDÖNMEZ and Koray KÜÇÜK for their support and for helpful discussions we have made throughout the study.

My special thanks go to Emrah TUFAN for his endless support and friendship. I would like to thank Rabia ÖZBAŞ for her valuable support on writing this dissertation. I am grateful to Anıl BAŞARAN who reviewed each and every word of this dissertation with a great patience. Thanks also go to all of my friends who kept me motivated.

I send my special thanks to Özer ÖZDİKİŞ and Serdar GÜLER for their cooperation and collaboration.

Finally, my deepest thanks go to my family who gave me the endless support and love, which made this thesis possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

A	Maximum cross sectional area of the missile
$\hat{C}^{(i,j)}$	Orthonormal transformation matrix from frame j to i
C_x, C_y, C_z	Aerodynamic force coefficients
C_x, C_y, C_z	Aerodynamic moment coefficients
d	Diameter of the missile
$\vec{D}$	Drag force
$\Im_i$	Frame “i”
$\vec{F}$	Force vector
F_a	Aerodynamics force
F_t	Thrust force
g	Gravitational acceleration
$\vec{G}$	Gravitational force
I_x	Axial moment of inertia
I_y	Transverse moment of inertia
L, M, N	Components of moment acting on the missile in body fixed reference frame
L_a, M_a, N_a	Components of aerodynamics moment acting on the missile in body fixed reference frame
L_t, M_t, N_t	Components of thrust moment acting on the missile in body fixed reference frame
m	Mass of the missile

M	Mach number
p, q, r	Components of angular velocity in the body fixed reference frame with respect to the earth fixed reference frame
Q_d	Dynamic pressure
u, v, w	Body frame velocity components
$\vec{u}_v$	Total velocity unit vector
V	Magnitude of total velocity vector.
V_s	Speed of sound.
$\vec{V}$	Total velocity vector of the munition
x, y, z	Axes of body fixed reference frame
X, Y, Z	Axes of earth fixed reference frame

Greek Letters

α	Angle of attack
β	Side slip angle
λ	Line of site angle in vertical plane
ϕ	Euler angle in roll plane
θ	Euler angle in pitch plane
ψ	Euler angle in yaw plane
δ_a	Effective aileron deflection
δ_e	Effective elevator deflection
δ_r	Effective rudder deflection
δ_1 ,δ_2 ,δ_3 ,δ_4	Actual four control surface deflections
δ_{1c} ,δ_{2c} ,δ_{3c} ,δ_{4c}	Commanded four control surface deflections

γ	Vertical plane flight path angle
η	Horizontal plane flight path angle
$\dot{\gamma}^*$	Commanded flight path angle in vertical plane
$\dot{\eta}^*$	Commanded flight path angle in horizontal plane
ρ	Density
$\overline{\omega}$	Column vector representation of angular velocity of the munition in body frame with respect to earth fixed frame
ω_n	natural frequency of pitch and yaw autopilots
ω_{nr}	Natural frequency of the roll autopilot
ω_{ncas}	Natural frequency of control actuation system dynamics
ζ	Damping ratio of the pitch and yaw autopilots
ζ_r	Damping ratio of the roll autopilot
ζ_{cas}	Damping ratio of the control actuation system dynamics

CHAPTER 1

INTRODUCTION

1.1 Trajectory Simulation Domain: An Overview

In this chapter, first an overview of trajectory simulation domain is presented. Then the motivation of this research is discussed before introducing the related literature. Chapter concludes with the sections on scope of the research and the organization of this thesis. One will also find the list of publications that presents the results of this research at the end of this chapter.

1.1.1 What is Trajectory Simulation?

Computer simulation is defined as studying various models of real world systems by numerical evaluation using software designed to imitate the systems operations. Computerized models of real or proposed systems are constructed to conduct numerical experiments to obtain a better understanding of the behavior of that system for a given set of conditions [1]. System then can be defined as a combination of elements or components interrelated to each other and to the whole which act together to achieve a certain goal [2]. Model on the other hand is a simplified representation of a system intended to enhance our ability to understand, explain, change, preserve, predict and control the behavior of a system [3].

Trajectory simulation in this context is a computational tool to calculate the flight path and other parameters of munition like its orientation or angular rates during its operation. It is such a tool that implements models of various components of a munition and their interfaces with each other and the environment. A time sequence of the dynamic events describing the operation and the flight of a munition is the result of any trajectory simulation run [4].

Trajectory simulation is based on mathematical model of munition, and environment which consists of equations that describe physical laws and logical sequences. The physical laws in the trajectory simulation govern the motion of munition and the effects of its subsystems. Basically equations of motion determine the acceleration, velocity and position resulting from forces and moments due to gravity, thrust and aerodynamics. There may also be other equations existing to simulate subsystems such as control system.

Zipfel [5] defines hierarchy of modeling and simulation in military simulation at four different levels: engineering, engagement, mission and campaign. Engineering level provides the tools for design tradeoff at the subsystem and system level to support the design, test and performance evaluations. In engagement level, simulations are for determining the effectiveness of the systems as they interact in terms of reliability, survivability, vulnerability and lethality. Mission level simulations are to investigate how operational goals are achieved by incorporating large number of cooperative and diverse players to the simulation. Lastly campaign level simulations engage decision makers in broad scale conflicts like war games.

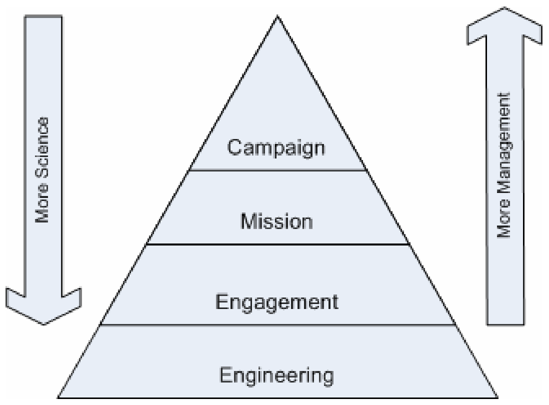


Figure 1 Hierarchy of Modeling and Simulation

Through out this hierarchy depicted above in Figure 1 from bottom to up, trajectory simulations are used for variety of purposes in a variety of ways. While different applications require different simulation approaches, the level of sophistication of simulations varies greatly depending on the application. These levels of sophistication range from simple point mass models to a very detailed six-seven degree of freedom models.

1.1.2 Purpose of Trajectory Simulations

The objectives of a trajectory simulation are greatly determined by the objectives of the intended user. Intended user aims at obtaining an understanding of various aspects of the performance of the munition for any of many different purposes encountered in analysis, development, procurement and operation of munition using trajectory simulations [4]. The U.S. Department of Defense (DoD), as an example defines its aim for using simulations as evaluating weapon system requirements and course of action to reduce the time line and the cost of the complex weapon systems; conducting training; and for realistic mission rehearsal [6]. The objectives of trajectory simulations are summarized in this section referring MIL-HDBK-1211 [4].

For the procurement of new weapon systems or the improvements in the current weapon systems, firstly the requirements are established. In order to establish the requirements for new weapon systems or the improvements of the current ones, analysis are carried out to determine the number of each kind of weapons that will be needed in the national arsenal. These analyses are done using models that cover a spectrum from one-to-one engagements between a weapon and a target to many-to-many engagements between multitudes of weapons of different kinds against a multitude of targets of different kinds. By the operation of these models, particulars of the battle are made visible so that the factors that drive the outcome can be analyzed. Some of these factors are the quantities and locations of fire units, target search and detection system characteristics, weapon launch doctrines, fire unit reaction times, number of munitions per unit fire, reload times, kill assessment times, defended area coverage, munition fly out times, countermeasure capabilities, and kill probabilities. All these efforts are for a better understanding of the improvements needed in the existing systems and the requirements for new systems.

These analyses models, mostly named as war games, rely on munition trajectory simulations for data on performance capabilities of various munitions under the conditions and environments being analyzed. Again the level of detail of trajectory simulation to be used to establish requirements varies depending on main interest underlying the application of the simulation. For example, if the aim of the analysis is to determine the defended area coverage, a simple trajectory simulation is adequate; however, if the reaction of the missile seeker to specific countermeasures is worked on, more detailed seeker simulations may be required. The Extended Air Defense Simulation (EADSIM) of Teledyne Brown Engineering can be a good example of this type. It is a many-on-many simulation of air, missile and space warfare which is extensively used around the world in many agencies. Trajectory simulations are carried out by its weapon model which is one of its physical models. EADSIM physical models are given below in Figure 2 [7].

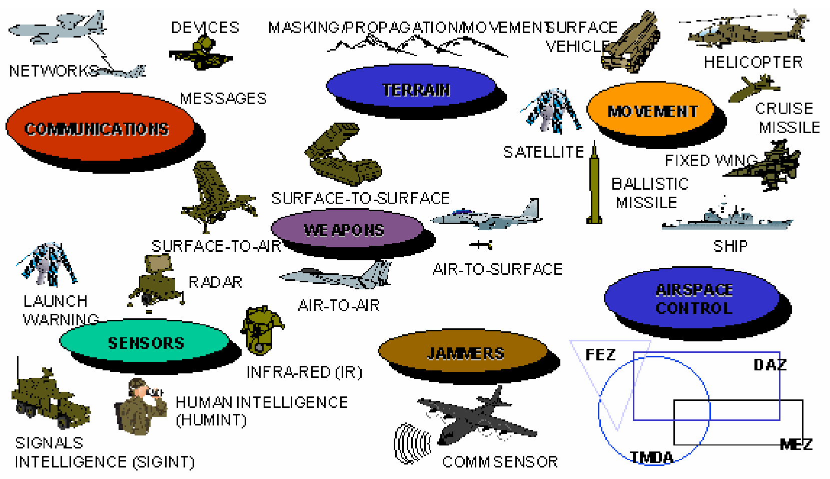
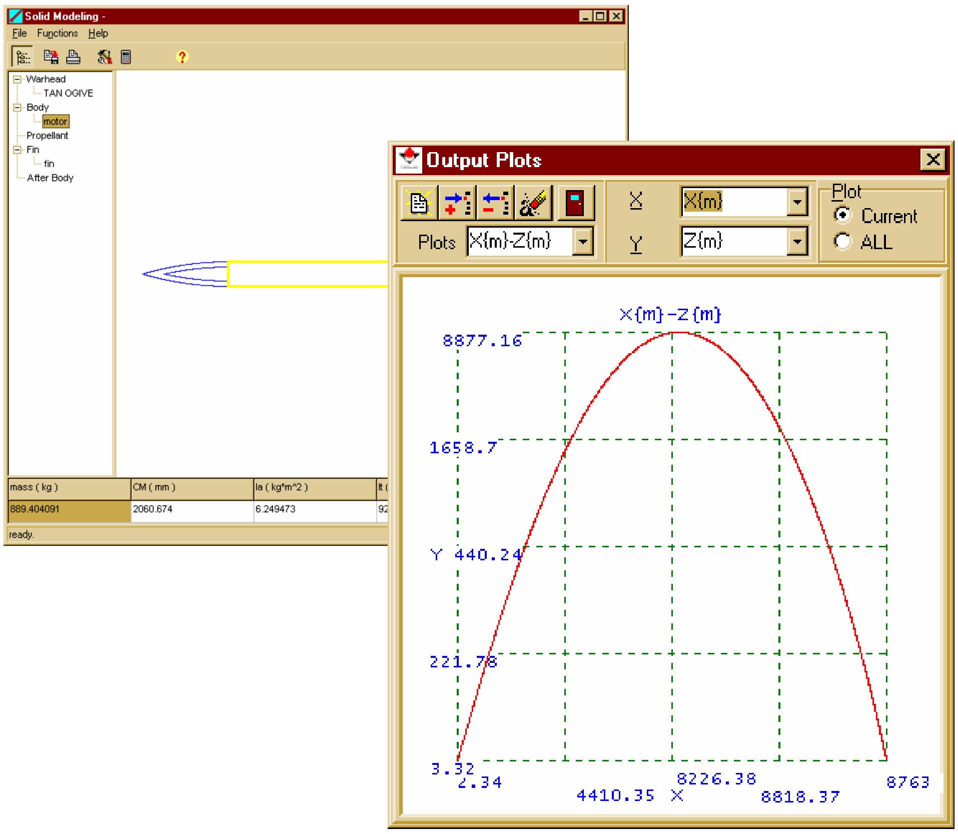


Figure 2 EADSIM Physical Models

Trajectory simulations are extensively used starting from conceptual design to flight tests throughout the development of weapon systems. Designers make use of trajectory simulations starting from optimizing the external configuration to the testing of a subsystem design or to the forecasting of a flight test results. Defense Industry Research and Development Institute of Scientific and Technological Research Council of Turkey (TÜBİTAK-SAGE) Flight Mechanics Computer Aided Design Software, (FMCAD) which is given in Figure 3, is one of the examples of this type [8].



mass (kg)	CM (mm)	Ia (kg*m^2)	It (
889.404091	2060.674	6.249473	92

X: X{m}
Y: Z{m}
Plots: X{m}-Z{m}
Plot: [✓ Current] ALL

Figure 3 FMCAD – Flight Mechanics Computer Aided Design Software of TUBİTAK - SAGE

Each new weapon system has unique characteristics that place different requirements on simulations. Simulation capabilities also evolve within conjunction with the development process of the system. The requirements for the simulation realism through the lifecycle of the weapon system are not the same. During the early development stage, for example in conceptual design, proof of concept and source selection are dominant issues. In this phase relatively simple simulations are mostly appropriate. During full-scale development, for example, when the system performance under adverse combat conditions is analyzed, much more complex simulated environments and ammunition response characteristics are needed. Fleeman, on the other hand, in his book named “Tactical Missile Design” advices the reader to use one to four degrees of freedom trajectory simulations through conceptual design and six degrees of freedom trajectory simulations in the preliminary design phase of weapon system development projects [10].

Military training is another important area where trajectory simulations are used. Warriors of every rank make use of modeling and simulation to challenge their skills at the tactical, operational, or strategic level through the use of realistic synthetic environments. It is usually hard and costly to conduct exercises to engage warriors without risking the injury, environmental damage or equipment damage. Simulations usually enable conducting trainings in any arena, using weapons that would be unsafe on conventional live ranges [6]. Trajectory simulations as a part of training simulator systems, enables realistic practice and better assessment of crew performance.

Trajectory simulations for training simulators are developed for munitions that are fully developed and where all performance data is available. Furthermore, in most of the cases real time operation requirements apply to trajectory simulations. Due to these two reasons, trajectory simulations are built focusing on representing the overall weapon performance rather than a detailed representation of subsystems.

Fundamentally, all fire control problems are variations of the same basic situation: launching munition from a weapon station to hit a selected target [11]. In an engagement scenario, the target or the weapon station or both may be moving. For all cases, fire control is the science of offsetting the direction of weapon fire from the line of sight of the target in order to hit the target. Fire control systems make use of trajectory simulations to estimate the trajectory of the munition at the specific conditions of fire. Trajectory simulations may be used to generate tables or curve fits that are used by fire control systems or they may be an integral part of fire control systems and work online. NATO Armaments Ballistic Kernel (NABK) can be pronounced as the contemporary example of the use of trajectory simulation for fire control [12]. NABK is used as a part of software's for generating firing tables to be used manually or in fire control systems as look up tables, as well as onboard fire control systems [13,14]. ASELSAN BAIKS2000 Fire Control System is one of the examples that use NABK.

As in the case of training, the simulation used in fire control simulate the trajectory of munitions that are fully developed and all the performance characteristics are well known. Fast calculation, on the other hand, is one of the very important requirements for the trajectory simulations that are used for fire control. So, the models used for the fire control are extremely optimized for minimization of the computation time.

1.1.3 Essentials of Trajectory Simulation

Trajectory simulations consist of number of models and numerical methods. The mathematical model of the motion of the munition constructs the base of any trajectory simulation. Subsystem and environment models aim at computing the effect of subsystem behavior and environment on the motion of the munition. These models are solved by making use of number of numerical methods, like numerical solvers or interpolation algorithms. This section introduces the essentials of trajectory simulations by basically using [4].

Mathematical models that simulate the motion of the ammunition are based on Newton’s and Euler’s laws. While Newton’s second law governs the translational degrees of freedom, Euler’s law controls the attitude dynamics. Munition, considered as a rigid body in space, is a dynamic system that experiences six degrees of freedom [5]. Its motion in space is defined by six components of velocity, three translational, three rotational. Three basic types of forces act on a munition and are included on in almost all trajectory simulations; the forces of gravity, propulsion and aerodynamics. In addition, the gyroscopic moments of internal rotors are sometimes included in simulations. Due to different fidelity and performance requirements, simplifications are made in trajectory simulations by approximating or neglecting the degree of freedom. Some of common simplifications are neglecting munition roll which results in a five degrees of freedom model and approximating all three rotational degrees of freedom that retains the three translational degrees of freedom, which is three degrees of freedom models.

The environment interacts with ammunition in two basic ways. First, the flow of air over the surface of the ammunition produces aerodynamic forces and moments. Second, the ability of atmosphere or the sea to transmit electromagnetic, sonar etc. signals impacts on the performance of the seeker. Trajectory simulations employ tables or models of atmosphere to provide values of atmospheric properties at the instantaneous altitude of the munition for each computational cycle.

Subsystem models incorporate subsystem behaviors to the munition motion. The guidance models in trajectory simulation contain algorithms that model the guidance functions; these include tracking the target and application of guidance law. Propulsion models contain algorithms that model the burning of propellant in terms of its effect on munitions flight by means of thrust force and inertial properties. Weapon models incorporate the effects of weapon behavior with flights initial conditions. Using fuze models, the characteristics of munitions fuze are considered to terminate the trajectory.

The vectors used in trajectory simulations represent factors such as forces, moments, accelerations, velocities and positions. For the direction of a vector to have a meaning, it must be described relative to some frame of reference. A vector is described by its three components on axes of a coordinate system. Number of coordinate systems may be used in a trajectory simulation. Coordinate systems are characterized by the position of their origins, their angular orientations, and their motions relative to inertial space. Common coordinate systems used in trajectory simulations are earth coordinate system, body coordinate system, wind coordinate system, guidance coordinate system, tracker coordinate system and target coordinate system.

Differential equations that are used to compute the trajectory of a munition mostly do not have closed form solutions [15]. These equations in trajectory simulations are solved by making use of a number of numerical methods. Many numerical integration methods are available to solve differential equations [16]. Selection of appropriate numerical method is basically affected by the accuracy and performance requirements of specific simulation. McCoy [17] in his book on exterior ballistics states that first and second order methods are optimum solutions of the point mass models where as for higher degree of freedom models which require higher computational accuracy, higher order methods are suggested.

1.2 Trajectory Simulation Reuse: Motivation

Trajectory simulations, as they are being developed for any purpose discussed before, are subject to different sets of requirements. While accuracy and the performance requirements affect the model to be used, the platform and programming language requirements affects the way they are developed.

Results of a flight simulation software methods survey that was carried out by NASA among the number of facilities that are developing flight simulations showed us that, there is wide variety of practices used in flight simulation development [18].

It won’t be a mistake to extrapolate these results to trajectory simulation facilities. Besides different practices among different facilities; single facility usually uses number of different practices at the same time for different projects.

From our observations, it is a common practice in the industry that developments of these simulations are carried out as isolated projects although they rely on the same body of knowledge [19]. When the complexity of the modeled systems and characteristics of the simulation domain is considered, the risk of failure in trajectory simulation projects is considered to be high. Besides the risk of failure, expenditure of intellectual labor to solve the similar problems of the same domain is a waste. Another aspect is the quality of the products of each development. The verification in trajectory simulation project requires great effort because the verification of the mathematical models and developed software really demands expert reviews and flight data which are expensive.

As other groups that develop trajectory simulations, Modeling and Simulation Team of TÜBİTAK-SAGE suffers from the lack of any formal methodology and tools to make use of past successful implementations of trajectory simulations.

Trajectory simulations are software systems. Then software reuse which actually depends on a very simple idea, using the previously developed software assets in developing new artifacts should also apply to trajectory simulation domain [20]. System development based on reusable software artifacts, in principle, should cost less which, most of the times, means shorter schedules and contain fewer defects because of the “tried and true” parts of which it is composed [21]. Applying software reuse practice in trajectory simulation domain will then lead us to less risky projects which results in high quality trajectory simulations.

The main motivation in this study is developing a reuse infrastructure that will enable trajectory simulation developers to make use of the past successful trajectory simulations in a structured way. Target reuse group of this study is the Modeling and Simulation Team of TÜBİTAK-SAGE.

1.3 Trajectory Simulation Reuse Studies in Literature

There have been number of efforts to make reuse work in the trajectory simulation domain. In the early days of trajectory simulations, developed programs are tried to be fit for multiple purposes. These general purpose trajectory simulations were capable of simulating the flight of a wide range of munitions. These codes were used by different end users rather than then being used as reusable assets in new trajectory simulation projects. GTRAJ is one of the trajectory simulation examples of this type. It is a general purpose trajectory simulation that supports point mass and modified point mass trajectories. It is developed by Firing Tables and Ballistic Branch of US ARDEC [22].

Besides, there are some studies in the literature for generic mathematical models to be used in different trajectory simulations. These studies do not point a specific implementation but introduce mathematical models to be used in variety of implementations [23, 24, 25 and 26].

There are contemporary studies for developing reusable trajectory simulations. These studies aim at developing reusable trajectory simulation software. NABK, genSim, JSBSim and Aerospace Blockset are the major examples of this type.

NABK has been developed till mid 90's as the shareable and reusable ballistic kernel for fire control of cannon artillery, mortars and unguided rockets by various NATO nations. It is implemented as an ADA95 class library which enables library reuse of its operational processors. It supports point mass, modified point mass and five degrees of freedom trajectories [12].

genSim is a generic six degrees of freedom simulation developed at Raytheon Missile Systems. It is a library that includes all of the first level components necessary to build missile simulations for everything from guided projectiles to long range missiles. Program specific algorithm and hardware models can be attached plug-and-play to the genSim architecture. It also supports some interfaces to legacy simulation code developed previously in FORTRAN or ADA [27].

JSBSim is an open source flight dynamics model in C++. It described as a batch simulation application aimed at modeling flight dynamics and control of aircraft. But the framework it provides is said to be handling modeling craft ranging from a simple ball, to a missile, an aircraft, rocket, hybrid vehicle, a rotorcraft, and so on. These crafts can feature different propulsion systems, ground reaction mechanisms, aerodynamic characteristics, and control systems if there exists any [28].

The Aerospace Blockset enables its user to work on aerospace system design, integration, and simulation by providing key aerospace subsystems and components in the adaptable MATLAB Simulink block format. It has number of reusable blocks from environmental models to equations of motion, from gain scheduling to animation. Blockset supports it users by the core components to assemble a broad range of large aerospace system simulations rapidly and efficiently. Trajectory simulation is one of the application areas that Aerospace Blockset is used for [29].

1.4 Scope of the Research

In all trajectory reuse studies in literature, the aim has been either to develop a code that can be reused by a number of projects or to develop a program that can be used by users of different agendas. Due to the diverse requirements different trajectory simulation projects that was discussed in the previous sections, there happens to be no single trajectory simulation that will fit all the requirements of different users who need a product which will facilitate one of engineering, engagement , mission or campaign level modeling and simulation. Each study mentioned in the previous section has its intended user group with a specific problem set and implementation platform. As an example, while NABK has been a strong reuse candidate for fire control systems development projects, Aerospace Blockset targets aerospace system designers. While it is hard to use Aerospace Blockset for a distributed aircraft simulator projects, it won’t be a good choice to use NABK for a weapon system design.

This research aims far more than code or library reuse that will inherently be platform and problem family specific. An infrastructure that will enable knowledge, design, code and library reuse is targeted. To develop a trajectory simulation, domain knowledge is transformed to a software product by using the methods and tools of software engineering. During this transformation it is aimed at enabling reuse in different abstraction levels starting from domain knowledge through platform independent design, platform specific design, and code.

In this research, we investigated the use of ontologies and domain engineering practices to develop a formalized methodology to make use of experience and knowledge leveraged from the past trajectory simulation projects. Formal specification of trajectory simulation domain is developed as a domain model in the form of ontology called Trajectory Simulation ONTology (TSONT) [30, 31]. This ontology of trajectory simulation domain, TSONT, made domain knowledge available for either automatic or manual transformation to a software design. TSONT is then used to develop object oriented and function oriented platform independent software designs. Other than domain knowledge that was made available to reuse in the form of ontology, these designs are developed to be the parts of reuse infrastructure. Every simulation built by transforming these designs is regarded as indispensable parts of reuse infrastructure. An object oriented framework for six degrees of freedom guided missile simulation is developed by transforming the object oriented platform independent design. A guided surface to surface rocket and a guided bomb simulation were built by framework completion [19]. A point mass MATLAB Simulink Blockset was developed using the function oriented platform independent design and number of simulations were built using this Blockset to propose a methodology and a set of reuse artifacts for function oriented paradigm.

Collaborative research has been carried out on automatic transformation of domain ontology to software specifications. In two different studies, two different programming paradigms were targeted. For function oriented software development, we succeeded automatic generation of MATLAB Simulink block definitions from TSONT [32]. For object oriented paradigm, we have been able to produce an abstract software design in the form of a UML class diagram from TSONT using automatic means [33].

1.5 Organization of Thesis

The thesis comprises six chapters. In Chapter 1, a brief overview of the trajectory simulation domain is presented with the scope and the motivation of this thesis.

In Chapter 2, ontology based reuse methodology developed is discussed. First, the basics of software reuse are presented with some historical perspective. Then, domain engineering is explained as the practice of software reuse. Ontology based approach to domain engineering is given as the contemporary approach to domain engineering. And the chapter is concluded with the section which discusses how the ontology driven domain engineering is structured in this thesis.

In Chapter 3, ontology concept as a means of knowledge sharing is explained. After presenting the definition of ontology in computer science, components of ontology and the merits of ontologies are given. Applications of ontologies in general and in engineering domain are discussed by referring the related literature. After presenting the guidelines of building ontologies, we presented the way TSONT is built. In the last section of this chapter, we explain the way we make benefit of Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) effort of National Aeronautics and Space Administration (NASA) for capturing the mathematical models of the domain.

In Chapter 4, TSONT is presented as the domain model of ontology based reuse infrastructure that was developed in this research. After briefing the ontology, we introduce the top level entities of the ontology followed by the hierarchies of the domain captured in TSONT. Then the way we captured the domain entities of trajectory simulation as OWL (Web Ontology Language) classes of TSONT is presented. This chapter is concluded with the discussion about the individuals of TSONT.

In Chapter 5, the specification and the implementation ontology based reuse infrastructure is presented. This section is structured considering two programming paradigms namely object oriented programming and function oriented programming. Both the specification and the implementation of the reuse infrastructure are discussed for these two different paradigms. Case studies are presented.

In Chapter 6, the conclusions emerging from the present work are discussed. We first discussed TSONT effort as one of the first attempts on formalizing the mechanical engineering knowledge with the importance of ontologies in knowledge sharing in engineering domain. Then we evaluated the ontology based trajectory simulation reuse infrastructure. The advantages of this ontology based approach over the past trajectory simulation reuse attempts are discussed. Finally, some future work recommendations are made.

1.6 Publications

The material in this thesis has been previously presented in the following publications.

- Durak, U., Anlağan, Ö., Oğuztüzün, H., *Yeniden Kullanılabilir Uçuş Benzetimi Mimarisi İçin Yol Haritası*, SAVTEK 2004, Ankara, 2004.

- Durak, U., Oğuztüzün, H., Mahmutyazıcıoğlu, G., *Domain Analysis for Reusable Trajectory Simulation*, Euro-SIW 2005, Toulouse, FRANCE, 2005.

- Durak, U., Oğuztüzün, H., İder, K., *An Ontology for Trajectory Simulation*, WinterSim06, Monterey, CA, USA, 2006.

- Durak, U., Oğuztüzün, H., İder, S.K., *Ontology Based Trajectory Simulation Framework*, Journal of Computing and Information Science in Engineering -ACCEPTED-

- Durak, U., Güler, S., Oğuztüzün, H., and İder, K., *An Exercise In Ontology Driven Trajectory Simulation with MATLAB Simulink*, 21st EUROPEAN Conference on Modelling and Simulation, Prague, Czech Republic, 2007.

CHAPTER 2

REUSE METHODOLOGY

In this chapter, ontology based reuse methodology developed is discussed. First, the basics of software reuse are presented with some historical perspective. Then, domain engineering is explained as the practice of software reuse. Ontology based approach to domain engineering is given as the contemporary approach to domain engineering. Eventually, chapter is concluded with a section that discusses how the ontology driven domain engineering is structured in this thesis.

2.1 Software Reuse: Overview

Among many other definitions, software reuse is defined in Reuse Based Software Engineering book of Mili et al. [34] as the process whereby an organization defines a set of systematic operating procedures to specify, produce, classify, retrieve, and adopt software artifacts for the purpose of using them in its development activities.

As explained by Arango [35], although the concept of software reusability can be traced back to the beginning of computer programming, it is pronounced as a software engineering problem at the 1969 NATO Conference [36]. Little progress was reported until mid to late 1970's, when some organizations put to a test the promise of productivity through reusability. The Workshop on Reusability in Programming, in 1983, was a milestone in the process. Early research focused on methods and mechanisms to perform reuse, on representation of reusable components, and on organization of repositories of components.

Three main motivations of software reuse are gains in productivity, quality and development schedules [34]. By reusing existing assets, we save the manpower required to develop them again. When an asset is developed for reuse, larger investment is allocated to its quality and the quality increases by the feed backs of a larger user base. And lastly, using reusable assets not only results in a decrease in manpower but also it shortens time to market.

In the past, reuse was understood as using generalized repositories of "components" and "parts" which can be accessed by many kinds of applications. It took several years of failures characterized by low levels of reuse to make it clear that this approach could not succeed. Then, the domain concept is defined and the success of reuse is related to the use of artifacts in the context of a domain. Domain is defined as the area in which an organization does business [37].

As Mili et al. presented, in recent years, however, it was also recognized that the mere creation of repositories of domain oriented assets was not enough to ensure reuse success. For a domain, it is also necessary to design a generic architecture, known as the domain architecture, of systems in that domain. With the identification of domain architecture, it becomes possible to develop systematically reusable assets that fit the architecture via a suitable interconnection mechanism. Reusable assets can be listed as compiled libraries, source code, requirements specifications, designs, test data, documentation, and software architectures [34].

2.2 Domain Engineering

Arango says that there exists a gap between the kinds and form of the knowledge available about problem domains and the content and form of the items of information that can be reused in software construction. Knowledge about the problem domain is often implicit and informal. While reusable information is made available to the software developer, it must be represented explicitly and formally. The term reuse infrastructure refers to the information that is made available to the software developer, together with auxiliary information needed to use and manipulate it. The process of developing a reuse infrastructure from problem domain is called domain engineering [35].

Domain engineering is defined as a process for building reusable assets, which includes activities for analyzing the domain, identifying common reusable assets, and populating them in the repository [38]. It is presented as an activity of a synthesis process that creates and supports a standardized application engineering process and products in a business area [39]. In this context, application engineering is the process that organizes and directs resources for producing and supporting a system by applying the reuse paradigm. The process includes activities for employing reusable assets from a repository.

It is stated that, domain engineering is carried out to addresses knowledge and asset development, capture, and evolution for a family of systems. It is defined as the process of identifying and recording commonalities and variables in a domain. It aims to create reusable assets and new systems using that information. Domain engineering activities create a "space" of solutions from which application engineers will later draw point solutions. A domain, in this context, is an application area containing systems that share design decisions. Domains can be classified depending on functional capabilities, such as navigation or stores management, or on cross functional areas; e.g., user interfaces, reliability, and security [40].

Arango and Prieto-Diaz explain domain engineering practice in [41] as follows. They state that domain engineering is fundamentally composed of three activities: domain analysis, infrastructure specification and infrastructure implementation. Domain analysis is the identification, acquisition and evolution of reusable information on a problem domain to be reused in software specification and construction. The purpose of domain analysis is to construct the model of the problem domain. Then domain model will then serve as:

- Unified resource of reference to solve ambiguities that may arise during the analysis of the problems or implementation of reusable components
- Repository of shared knowledge for communication and orientation

- Specification of reusable components to the application developer.

It is claimed that a domain model is not directly useful for operational reuse. There exists a gap between the kinds and the forms of domain knowledge in a domain model and the content and form of software assets that can be reused in software construction. To bring this gap, a reuse infrastructure is built.

Infrastructure specification is then defined as the selection and organization of reusable information in the model to fit the patterns of reuse in the environment of reuser. As a result, an architecture for reusable information is specified. For example, a library of programs, a database scheme. The infrastructure specification, together with the semantics captured by the domain model, is input to the infrastructure implementation step that actually produces and tests the components.

It is said that infrastructure implementation is the design and encoding of the pieces resulting from the specification process using particular representations required by the technology or reused: for example encoding the specified programs using programming languages.

Among many research activities on domain engineering [42, 43], CAMP (Common ADA Missile Packages) was the first and most famous one [44]. CAMP Project was the first explicitly reported domain engineering experience. In this project eleven tactical missile systems were analyzed, several common components were identified, and grouped by their functionality. A set of general design templates was derived in the form of Ada generics and later integrated in a design support system, the Ada Missile Parts Engineering Expert (AMPEE). AMPEE aimed to support component identification, component selection, and component construction [45].

There are several early efforts described in literature (see [46, 34] for a review) to define domain engineering methods from 90's, such as Feature-Oriented Domain Engineering (FODA), Domain Analysis and Reuse Environment (DARE), Reuse Library Process Model (RPLM), Organisation Domain Modeling (ODM) and

Domain Specific Software Architecture (DSSA). Among these, we focused on ontology based domain engineering approaches [47, 48, 49 and 50]. In this research we defined a derivative of ontology driven domain engineering methodology which we used to develop trajectory simulation reuse infrastructure [31, 32].

2.3 Ontology Based Domain Engineering

Neighbors defines the domain analysis as “the activity of identifying the objects and operations of a class or similar systems in a particular problem domain” [51]. From Webster, domain is “field or sphere of activity or influence” [52]. From the software engineering point of view, domain is defined as the application area of the field for which the software systems are developed [45]. Examples include traffic management systems, management information systems or command and control systems. Domains can be broad like manufacturing or narrow like arithmetic operations. Domains on the other hand are limited by their boundaries which define their scope. The borders of a domain define what objects, operation and relations belong to the domain.

Diaz defines domain analysis as a process where information used in developing software systems is identified, captured, structured, and organized for further reuse [45]. More specifically, domain analysis is said to be dealing with the development and evolution of an information infrastructure to support reuse. The inputs of this process are a domain analysis methodology, custom-built for each specific domain. And output of domain analysis is a domain model. Domain models range in level of complexity and expressive power, from a simple domain taxonomy to functional models to domain languages [41].

Diaz says that as the knowledge about the domain is collected during domain analysis, the problem is representing this knowledge for ease of human understanding and machine processability [45]. Ontology approach to knowledge representation is utilized in this research to solve this problem.

According to Uschold [53], “An ontology may take a variety of forms, but necessarily it will include a vocabulary of terms, and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms”. Thus, ontology consists of concepts and relations, and their definitions, properties and constraints expressed as axioms. An ontology is not only a hierarchy of terms, but a fully axiomatized theory about the domain [54].

In the domain engineering, it is said that ontologies can act both as a domain model and a component in the repository [48]. Ontology based domain engineering is interested in the use of an ontology as a domain model and how to derive components from it.

The advantages of an ontology based approach to domain engineering are discussed in detail by Falbo et al. in his paper “An Ontological Approach to Domain Engineering” [47]. Briefly, ontology enables us to build a domain model independent from the software technology and it gives a strong tool to capture the domain conceptualization.

2.4 Methodology Explained

In our approach to trajectory simulation development with reuse, we defined an original domain engineering methodology. We focused on two basic programming paradigms, namely object oriented programming and function oriented programming. For both, we envisioned to make use of model driven technologies. Trajectory Simulation ONTology (TSONT) is treated as the domain model. It is being developed to be a reusable knowledge library on trajectory simulations. The basic idea behind developing an ontology as the domain model of the trajectory simulation domain is, first, to establish a common vocabulary that is agreed among the people working on trajectory simulations. Another main consideration is to create a backbone for systematization of knowledge on how to build a trajectory simulation [55].

In the last decade ontologies have been used for variety of engineering applications [56, 57, 58, 59, 60, 61, and 62]. In this research, we aim to use the ontology as a basis for constructing trajectory simulation applications. Potential benefits of this approach include documentation, maintenance, reliability, knowledge reuse and interoperability of the developed applications.

For object oriented programming, we turned to software frameworks to realize the notion of infrastructure in our domain engineering practice. Johnson and Foote state that a framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes [63]. As noted by Fayad, frameworks enhance modularity by encapsulating implementation details behind their interfaces and these interfaces enhance the reusability by defining generic components that can be reapplied to create new applications [64]. They will be discussed in detail in proceeding sections.

We find it favorable to construct new simulations by framework completion, provided, of course, a suitable framework is available. Otherwise one needs first to develop a framework, and then complete it for the particular application. This approach is expected to create a collection of related frameworks addressing different platforms and problem families.

We firstly build a platform independent framework architecture, which can be transformed to some platform and problem family specific framework architectures. We propose to design the platform independent framework architecture on the domain model, so that it is traceable to the domain knowledge represented in the ontology. We use the ontology as a guide for the specification of static structure of the framework, behavior model and definition of the interfaces of the framework. The taxonomy of classes of TSONT is reflected in the inheritance hierarchy of the abstract software design. Abstract behavior model is based on the dependency relations of the functions of TSONT, and finally the framework interfaces are designed based on the function specifications of TSONT. This is a testimony to knowledge reuse.

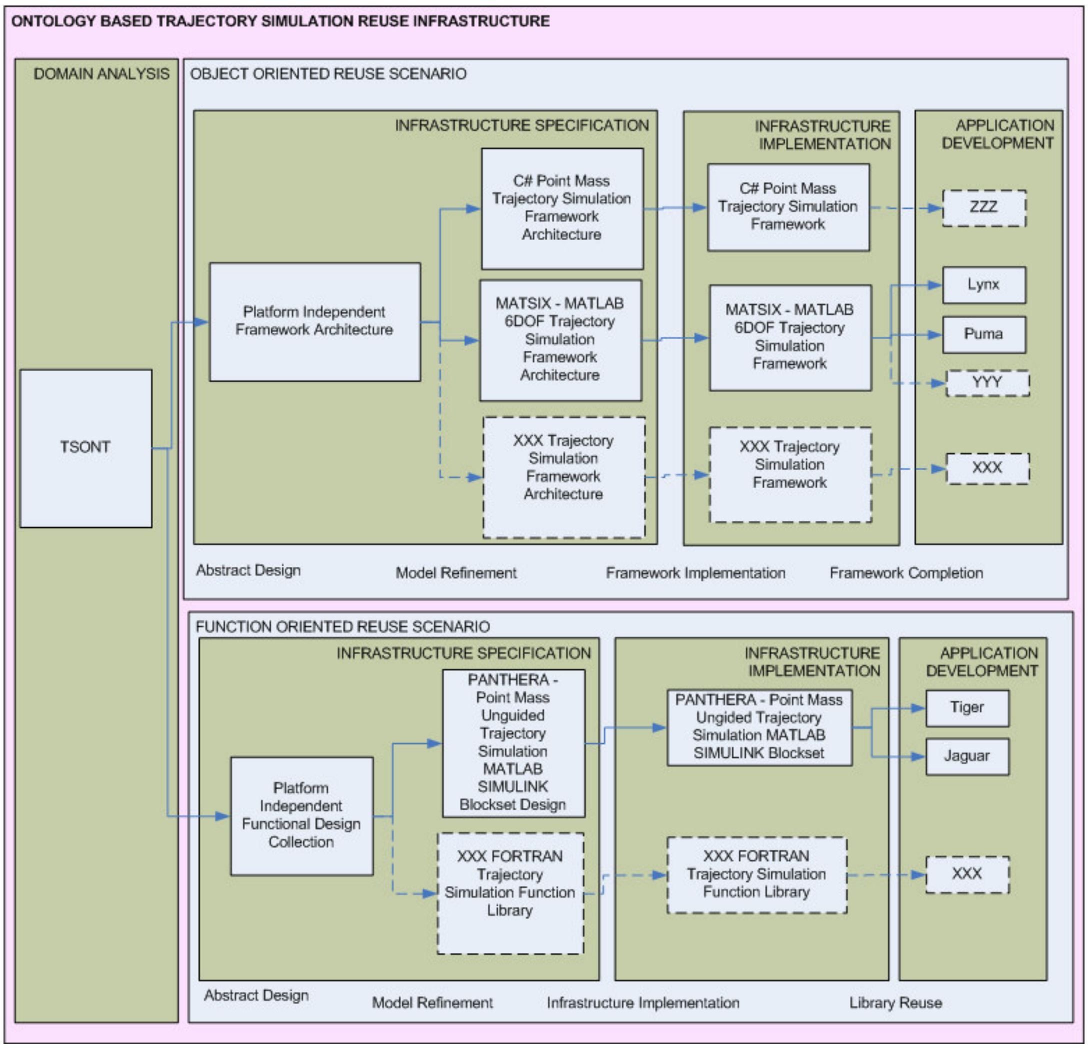

ONTOLOGY BASED TRAJECTORY SIMULATION REUSE INFRASTRUCTURE
DOMAIN ANALYSIS
TSONT
OBJECT ORIENTED REUSE SCENARIO
INFRASTRUCTURE SPECIFICATION
Platform Independent Framework Architecture
C# Point Mass Trajectory Simulation Framework Architecture
MATSIX - MATLAB 6DOF Trajectory Simulation Framework Architecture
XXX Trajectory Simulation Framework Architecture
INFRASTRUCTURE IMPLEMENTATION
C# Point Mass Trajectory Simulation Framework
MATSIX - MATLAB 6DOF Trajectory Simulation Framework
XXX Trajectory Simulation Framework
APPLICATION DEVELOPMENT
ZZZ
Lynx
Puma
YYY
XXX
Abstract Design
Model Refinement
Framework Implementation
Framework Completion
FUNCTION ORIENTED REUSE SCENARIO
INFRASTRUCTURE SPECIFICATION
Platform Independent Functional Design Collection
PANTHERA - Point Mass Unguided Trajectory Simulation MATLAB SIMULINK Blockset Design
XXX FORTRAN Trajectory Simulation Function Library
INFRASTRUCTURE IMPLEMENTATION
PANTHERA - Point Mass Ungided Trajectory Simulation MATLAB SIMULINK Blockset
XXX FORTRAN Trajectory Simulation Function Library
APPLICATION DEVELOPMENT
Tiger
Jaguar
XXX
Abstract Design
Model Refinement
Infrastructure Implementation
Library Reuse

Figure 4 Domain Engineering Methodology

Together with the framework architecture that targets a specific platform and problem family, this platform independent framework architecture is regarded as the outcome of the infrastructure specification activity of domain engineering. The platform dependent and problem family specific frameworks are then the outcome of infrastructure implementation. This is a testimony to design reuse. Domain engineering methodology developed is depicted below in Figure 4.

As presented above in Figure 4, for function oriented programming, reuse infrastructure specification is built again using the knowledge captured in TSONT. Data flow diagrams are treated as the tools for abstract function oriented design. As presented in the famous software engineering book of Sommerville, data flow diagrams are concerned with designing a sequence of functional transformations that convert system inputs into the required outputs. These diagrams illustrate how data flows through a system and how the output is derived from the input through a sequence of functional transformations [65].

Different from our object oriented scenario, we do not propose a single abstract design that covers whole domain. Rather we propose a collection of data flow diagrams for different problem sets, like, point mass data flow diagrams that we will present in the following sections or a modified point mass projectile simulation data flow diagrams. This collection of abstract designs will be the reuse assets for the future projects. Platform specific design will be the refinement of these abstract designs. Reuse infrastructure is implemented in the form of function libraries or blocksets using the platform specific designs for specific trajectory simulation applications. Applications are suggested to be developed using the function libraries or reusable blocks developed as the infrastructure.

In this chapter, after introducing the literature on software reuse, domain engineering and ontology based domain engineering, ontology based reuse methodology developed is presented. Next chapter will introduce some background on ontologies and knowledge sharing.

CHAPTER 3

ONTOLOGIES AND KNOWLEDGE SHARING

In this chapter, ontology concept as a means of knowledge sharing is explained. After briefing the definition of the ontology, components, merits and applications of ontologies are explained. Engineering applications of ontologies are referred to before discussing the principles that are taken into account when constructing TSONT. In the last section, we explain way we make benefit of Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) effort of National Aeronautics and Space Administration (NASA) for capturing the mathematical models of the domain.

3.1 What is Ontology?

The term ontology is borrowed from philosophy, where it has the meaning of a systematic explanation of Existence. In the Artificial Intelligence field, first Neches defined ontology as “An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary” [66]. Later in 1993, Gruber’s definition “Ontology is explicit specification of conceptualization” [68] became famous. Struder and colleagues explained Gruber’s definition. They claimed that conceptualization refers to an abstract model of some phenomenon in the world which identifies the relevant concepts of that phenomenon and they explained the word “explicit” as type of concepts used, and the constraints on their use are explicitly defined [69].

3.2 Components of Ontology

According to Gruber [70], knowledge in ontologies can be formalized using five kinds of components: concepts, relations, functions, axioms and instances.

Concepts can be anything about which something is said, and therefore, can be a description of a task, function, action, strategy etc. Taxonomies are widely used to organize the ontological knowledge in domain using generalization/specialization relationship through simple/multiple inheritance. Relationships represent a type of interaction between the concepts of the domain and functions can be regarded as a special kind of relation. Axioms on the other hand are used to model sentences that are always true. They are added to ontology for several purposes, such as constraining the information contained in the ontology, verifying its correctness or deducting new information. Instances are the terms that are used to represent the elements of the domain. They actually represent the elements of the concepts [71].

3.3 Merits of Ontologies

Mizoguchi in his paper "Ontological Engineering: Foundations of next generation knowledge processing" [55] lists the merits of the ontology as follows:

1. *A common vocabulary*: Ontology creates a vocabulary agreed among the people involved to describe of the target world.
2. *Explication of what has been often left implicit*: Knowledge bases are usually built based on an implicit conceptualization possessed by the builder. This implicitness is one of the main causes of preventing knowledge sharing and reuse. So the explicit representation of assumptions and conceptualization in an ontology is a contribution to knowledge reuse and sharing.

3. *Systematization of knowledge:* An ontology constructs a backbone for the systematization of knowledge by providing a well established vocabulary/concepts that people use to describe phenomena, theories and target things under consideration.

4. *Standardization:* Ontology constructs a standardization of shared terms/concepts that enables a communication among human and computer agents.

5. *Meta-model functionality:* To construct an abstraction of the target in a model, ontology provides us concepts and relations among them to be used as building blocks of the model. This building blocks can be regarded as a meta-model.

3.4 Applications of Ontologies

In "An Ontological Approach to Domain Engineering" paper [47], applications of ontologies are classified in four main categories: Neutral authoring, ontology as specification, common access to information and ontology-based search.

Falbo et al. in the same paper explain each application group as follows [47]: "An ontology is developed in a single language and it is translated into different formats and used in multiple target applications." This enables neutral authoring. "An ontology of a given domain is created and it provides a vocabulary for specifying requirements for one or more target applications. In this case ontology can be viewed as domain model. The ontology is used as a basis for specification and development for domain applications, allowing knowledge reuse." This can be classified as the use of ontology as a specification. For common access to information: "Ontology is used to enable multiple target applications (or human) to have access to heterogeneous sources of information that are expressed using diverse vocabulary or inaccessible format ". Ontology-based search is explained as:

"An ontology is used for searching an information repository for desired resources, improving precision and reducing the overall amount of time spent searching."

Here among the applications of ontologies, ontology as specification is the way that this research is focused on. The basic idea behind ontology as specification is to author an ontology which models the application domain, and provides a vocabulary for specifying the requirement for one or more target applications. The richer the ontology is in expressing the meaning, the less it has the potential for ambiguity in creating requirements. The software is based on the ontology, which thus plays an important role in the development of the software. The benefits of this approach include documentation, maintenance, reliability and knowledge reuse.

3.5 Engineering Ontologies

In this research, a large scale engineering ontology was developed. The first efforts on developing engineering ontologies were in 90's. Ontologies in engineering domain have been developed for various purposes including specifying engineering information systems, integration of engineering applications, supporting engineering design and forming a conceptual foundation for engineering ontologies.

The PhysSys was one of the first engineering ontologies. It is based upon system dynamics theory that is practiced in engineering modeling, simulation and design. The PhysSys was developed to formally define how design engineers or the end users of Computer Aided Engineering (CAE) systems understand their domain and to provide a foundation for the conceptual schema for data structuring in engineering databases, libraries and other CAE information systems [56, 57]. The ideas formalized in PhysSys provided a base for the development of a library of reusable models for engineering and design. This library was developed in the European Union ESPRIT-II program Open Library for Models of mEchatronic Components (OLMECO). The aim of the OLMECO project was to develop a modeling and simulation environment for industrial applications [58].

The KACTUS project targeted at the development of methods and tools for the reuse of knowledge about technical systems during their life-cycle. The project was application-driven: systems were being developed in the domains of preliminary-ship design, oil-production processes, and electrical networks [59].

Mihai Ciocoiu and his colleagues attacked the growing complexity of manufacturing information and the increasing need to exchange this information among various software applications like CAD, performance analysis, manufacturability analysis, product data management system, process planner, production management system, scheduler, and a simulation system. As a solution to this problem, they made use of taxonomies or ontologies of manufacturing concepts and terms, because ontologies provide a way to make explicit the semantics (i.e., the meaning) for the concepts used, rather than relying just on the syntax used to encode those concepts [60].

In MIT Artificial Intelligence Laboratory, a research was carried out aiming to develop a large scale ontology for the mechanical engineering world to support a wide range of tasks including analysis and design. Common patterns of behavior are tried to be identified and labeled with the terms that mechanical engineers use to talk about mechanical devices [61].

In one of the early efforts of ontology development for engineering domain, Gruber and Olsen described an ontology namely EngMath for mathematical modeling in engineering. This ontology builds a conceptualization on abstract algebra and measurement theory. It includes scalar, vector, and tensor quantities, physical dimensions, units of measure, functions of quantities, and dimensionless quantities. EngMath is designed for knowledge sharing purposes. It was aimed to be used as a communication language among cooperating engineering agents, and as a foundation for other engineering ontologies [62].

3.6 Ontology Development

Currently, ontology development is a craft rather than a science. It is still a research area. Falbo et al. defines the ontology development process as set of activities consisting purpose identification and requirements specification, evaluation and documentation, integration existing ontologies, ontology capture, ontology formalization as in Figure 5 [47].

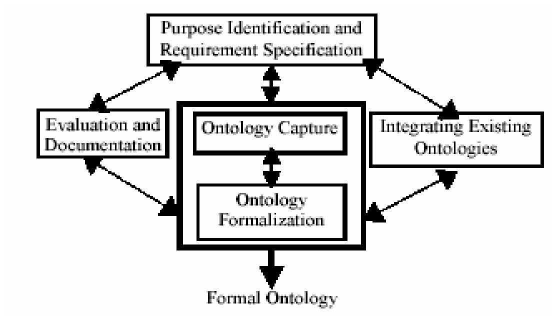

Purpose Identification and Requirement Specification
Evaluation and Documentation
Ontology Capture
Integrating Existing Ontologies
Ontology Formalization
Formal Ontology

Figure 5 Ontology Development Process [47]

During purpose identification and requirements specification, the purpose of the ontology and its intended use is identified.

Ontology capture is to capture the domain conceptualization. The relevant domain entities (e.g. concepts, relations, properties) are identified and organized in this step. Mostly a model represented in a graphical language is used to facilitate the communication with the domain experts.

Ontology formalization aims to explicitly represent the conceptualization in a formal language. This language is used to represent the elements that model the existing domain entities in a precise and unambiguous way.

It is common practice to integrate the developed ontology with existing ones to use previously established conceptualization during ontology capture and/or formalization.

Ontologies are checked whether the ontology satisfies the specification requirements or not in the evaluation step. Ontologies are evaluated against the ontology competence and some design quality criteria.

Purpose, requirements, textual description of conceptualization, and the formal ontology must be documented, including. This activity is done in the documentation step.

TSONT is being developed considering the guidelines Fablo defined. The purpose of the TSONT was identified in the proposal of this research. Protégé is used as the ontology development environment. It is a tool developed by Stanford University. It enables a graphical environment to facilitate the communication with the domain experts besides enabling an integrated formalization of the captured conceptualization while constructing graphical representation of ontology [67]. TSONT is formalized using Web Ontology Language which will be presented in the proceeding sections. We did not integrate TSONT with other ontologies but we aligned TSONT using Suggested Upper Merged Ontology (SUMO) of IEEE in order to enable painless integration with other mid level ontologies like TSONT. This thesis is being regarded as the documentation of ontology developed. We do not regard the ontology development process to be completed. TSONT is planned to be continuously maintained and enhanced as the reuse infrastructure is used. With the experience gained by new projects, it will become more mature and more complete.

3.7 Principles of Building Ontology

Ontologies are actually designed. One chooses how to represent something in an ontology by making design decisions. Following guidelines defined by Gruber [68] was taken into account when developing TSONT.

Clarity: To make TSONT effectively communicate the intended meaning of defined terms, definitions are stated objectively and independent of social or computational context.

Coherence: To make TSONT coherent, the definitions are checked against logical consistency.

Extendibility: TSONT is designed to encourage the use of the shared vocabulary. One can either expand TSONT or add individuals to define new terms for special uses based on the existing vocabulary.

Minimal encoding bias: The conceptualization in TSONT is specified at the knowledge level. We do not use any particular symbol-level encoding.

Minimal ontological commitment: TSONT has ontological commitment on trajectory simulation developments that is sufficient to support trajectory simulation development knowledge sharing activities.

3.8 How to Represent an Ontology?

Early attempts on representation systems resulted to several languages. Some examples are Ontolingua, OKBC, OCML, Loom, and FLogic. Contemporary studies in representation systems resulted to web languages like OIL, DAML, DAML+OIL and OWL for building ontologies [71].

Web Ontology Language (OWL) was developed to be the standardized and broadly acceptable ontology language of the Semantic Web by World Wide Web Consortium (W3C) Web Ontology Working Group [72, 73, 74]. Considering the current support to this language in terms of tools and publications, OWL is selected as the language to represent TSONT.

The requirements of OWL were well-defined syntax, well-defined semantics, efficient reasoning support, sufficient expressive power and convenience of expressions. The requirement of a well defined syntax is necessary condition for machine processing of information. Formal semantics describes precisely the meaning of knowledge. "Precisely" here means that the semantics is not subjective and it is open to different interpretations by different people or machines. Reasoning support on the other hand is necessary to check the consistency of the ontology and knowledge. These requirements leaded W3C's Web Ontology Working Group to define a language as powerful as a combination of Resource Description Framework (RDF) Schema with a full logic [75]. They then defined OWL as three sub languages, each of which is geared towards fulfilling different of these requirements:

OWL Full: The entire language is called OWL Full. It uses all the OWL languages primitives.

OWL DL: For computational efficiency, OWL DL (short for: Description Logic) is a sublanguage of OWL Full is defined. It restricts the way in which the constructors from OWL and RDF can be used.

OWL Lite: With further restrictions, OWL DL is limited to a subset of the language constructors. For example, OWL Lite excludes enumerated classes, disjointness statements and arbitrary cardinality (among others). The advantage of this is a language that, it is easier to grasp for users and easier to implement for tool builders. The disadvantage is, as one would expect, its restricted expressivity.

OWL is built on RDF and RDF Schema (RDFS) and uses RDF's Extensible Markup Language (XML) syntax. OWL documents are usually called OWL Ontologies. They are also RDF documents.

An OWL ontology starts with a collection of assertion for housekeeping purposes. The assertions are grouped under *owl:Ontology* element which contains comments, version control and inclusions of other ontologies. For Example:

```
<owl:Ontology rdf:about="">
        <rdfs:comment>An example OWL ontology</rdfs:comment>
        <owl:priorVersion rdf:resource="http://www.mydomain.org/spacecraft"/>
        <owl:imports rdf:resource="http://www.mydomain.org/aircraft"/>
        <rdfs:label>Spacecraft Ontology</rdfs:label>
</owl:Ontology>
```

Classes are defined by using *owl:Class* element. For example we can define a ramjet as:

```
<owl:Class rdf:ID="Ramjet">
        <rdfs:subClassOf rdf:resource="#Thruster"/>
</owl:Class>
```

OWL has the definitions for disjoint classes and equivalent classes as *owl:disjointWith* and *owl:equivalanetClass*. There are two predefined classes, *owl:Thing* and *owl:Nothing*. Thing is the most general class. Nothing on the other hand is the empty class.

OWL has two kinds of properties. Object properties relate objects to objects and datatype properties relate the objects to datatype values. *Rdfs:subClassOf* is used to define inheritance restriction. *owl:allValuesFrom* is used to specify the class of possible values of the property specified by *owl:onProperty*. *owl:hasValue* states a specific value that the property, specified by *owl:onProperty* must have. Cardinality relations can be given using *owl:cardinality*, *owl:minCardinality* and *owl:maxCardinality*. Some properties of the elements can be defined directly:

- *owl:TransitiveProperty* defines a transitive property, such as “has better sound than”.

- *owl:SymmetricProperty* defines a symmetric property, such as “has same height as”.

- *owl:FunctionalProperty* defines a property that has at most one unique value for each object, such as “weight”.

- *owl:InverseFunctionalProperty* defines a property for which two different objects cannot have the same value

Boolean combinations e.g. union, intersection of classes can also be defined by using owl. *owl:oneOf* element, on the other hand, is used for enumerations and is used to define a class by listing all its elements.

Instances of classes are declared as in RDF. Unique names assumption is not adopted by OWL. Thus, just because two instances have different name, does not imply they are different individuals.

OWL does not allow derived data types, although XML Schema provides mechanism for derived data types. OWL document just consists of data types that are most frequent used ones like strings, integer, boolean, time and date.

When the layered structure of the language is considered, in OWL Full, one can use all the language constructors as long as the result is legal RDF. When one needs to use OWL DL, the constraints to be obeyed are as following; Any resource in OWL DL is allowed to be either class, a datatype, a datatype property, an object property, an individual, a data value or a part of built in vocabulary. All resources must be partitioned, and this partitioning must be stated explicitly. Furthermore, no cardinality restriction can be applied on transitive properties. And lastly anonymous classes are only allowed in the domain and range of o*wl:equivalentClass* and *owl:disJointWith* and *rdfs:subClassOf*. Each OWL Lite ontology must be and OWL

DL ontology. For OWL Lite, it must further obey the following constraints. *owl:OneOf*, *owl:disjointWith*, *owl:unionOf*, *owl:ComplementOf* and *owl:hasValue* are not allowed. Furthermore, cardinality statements can only be made on values 0 and 1. And lastly *owl:equivalentClass* statements cannot be made between anonymous classes, but only between class identifiers.

One should decide upon the sub-language to use before starting working on an ontology. There are simple rules of thumb when deciding upon a sub language, formulated as follows by Horridge et al. [76].

- The choice between OWL-Lite and OWL-DL is better to be based upon whether the simple constructs of OWL-Lite are sufficient or not.
- The choice between OWL-DL and OWL-Full is better to be based upon whether to carry out automated reasoning on the ontology or to be able to use highly expressive and powerful modeling facilities is important.

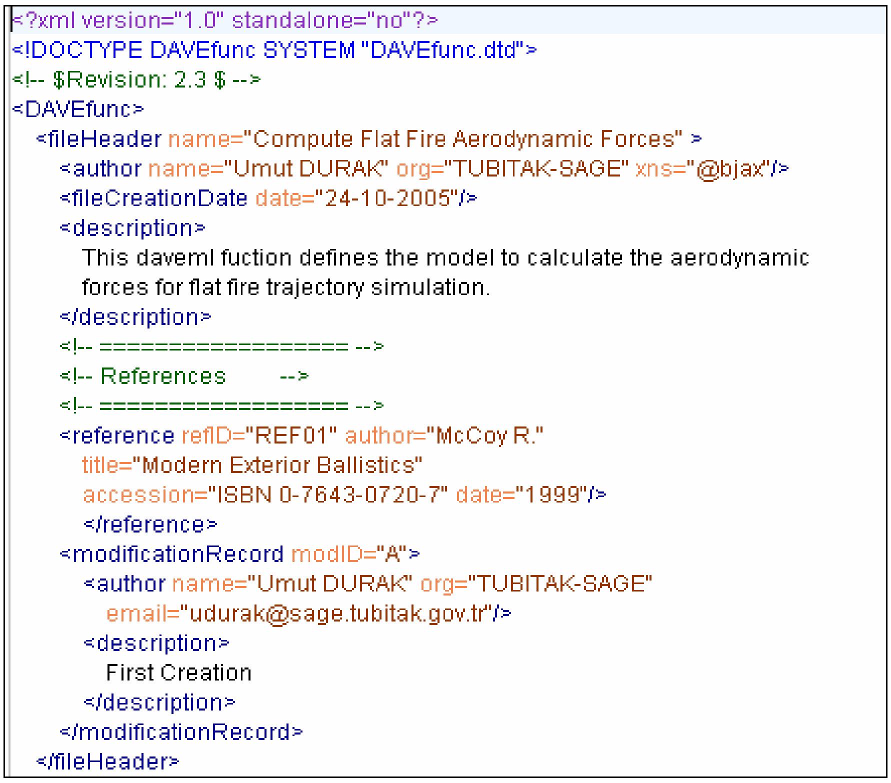
In this study, what we want to do is to capture as much knowledge from the domain as possible to lead us to some software architecture. So we selected to use the most expressive one, OWL-Full, in order not to be constrained by the language.

3.9 DAVE-ML

Trajectory simulation domain involves mathematical models that account for some kind of behavior or some law. Capturing these models in a systematic way and representing them as an integrated part of the ontology is an important concern. At this juncture, the Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) effort of National Aeronautics and Space Administration (NASA) for the benefit of flight modeling and simulation community has been leveraged [77].

DAVE-ML is a proposed standard to interchange of aerospace dynamic models. It is aimed to provide a programming language independent representation of aerodynamics, mass/inertia, propulsion and guidance, navigation and control laws of a vehicle. DAVE-ML is XML-based. It uses MathML to describe mathematical relations. MathML is an XML-based language for describing mathematics for machine to machine communication. We take advantage of DAVE-ML to incorporate mathematical models into our ontology TSONT.

DAVE-ML is being regarded as the way to document the mathematical model implementations in TSONT. DAVE-ML's intentions is defined as to allow a programming language independent representation of the aerodynamic, mass/inertia, propulsion, guidance navigation and control laws for trajectory simulations [78].

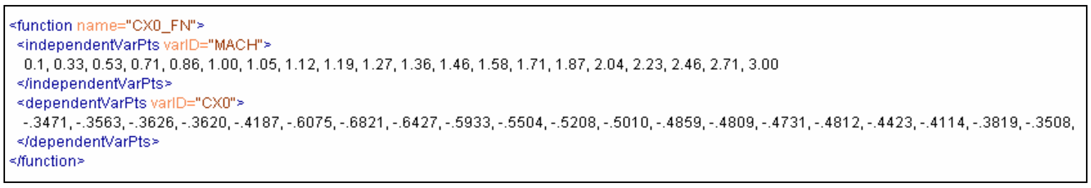


```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DAVEfunc SYSTEM "DAVEfunc.dtd">
<!-- $Revision: 2.3 $ -->
<DAVEfunc>
  <fileHeader name="Compute Flat Fire Aerodynamic Forces" >
    <author name="Umut DURAK" org="TUBITAK-SAGE" xns="@bjax"/>
    <fileCreationDate date="24-10-2005"/>
    <description>
      This daveml fuction defines the model to calculate the aerodynamic
      forces for flat fire trajectory simulation.
    </description>
    <!-- =================== -->
    <!-- References        -->
    <!-- =================== -->
    <reference refID="REF01" author="McCoy R."
      title="Modern Exterior Ballistics"
      accession="ISBN 0-7643-0720-7" date="1999"/>
      </reference>
    <modificationRecord modID="A">
      <author name="Umut DURAK" org="TUBITAK-SAGE"
        email="udurak@sage.tubitak.gov.tr"/>
      <description>
        First Creation
      </description>
    </modificationRecord>
  </fileHeader>
```

Figure 6 An Example DAVE-ML File Header

There is only one basic element in DAVE-ML. It is DAVEFunc. It is used to describe static models such as aerodynamic and inertia/mass models. It is composed of data tables and equations for a particular model. It has five components: a file header, variable definitions, breakpoint definitions, table definition and a function definition. File header is used to give a background and reference data for the represented model. A file header example is given above in Figure 6.

Breakpoints define a list of monotonically increasing floating point values. Function table definitions generally contain the data points for aerodynamic coefficients as a function of one or more parameter like mach, angle of attack, control surface deflections. Function definitions as given below in Figure 7; connect the data tables to breakpoints to define how an output should vary with one or more input.



```
<function name="CX0_FN">
 <independentVarPts varID="MACH">
  0.1, 0.33, 0.53, 0.71, 0.86, 1.00, 1.05, 1.12, 1.19, 1.27, 1.36, 1.46, 1.58, 1.71, 1.87, 2.04, 2.23, 2.46, 2.71, 3.00
 </independentVarPts>
 <dependentVarPts varID="CX0">
  -.3471, -.3563, -.3626, -.3620, -.4187, -.6075, -.6821, -.6427, -.5933, -.5504, -.5208, -.5010, -.4859, -.4809, -.4731, -.4812, -.4423, -.4114, -.3819, -.3508,
 </dependentVarPts>
</function>
```

Figure 7 An Example DAVE-ML Function Definition

Variables are used to capture inputs, calculations and outputs for a model. Variables can be regarded as the signal routes in a block diagram or the parameters in the computer program. They can be either inputs of the models, constants used in the models, intermediate results or outputs of the models. MathML is used to represent the mathematical relation of input variables and outputs. An example is presented above in Figure 8.

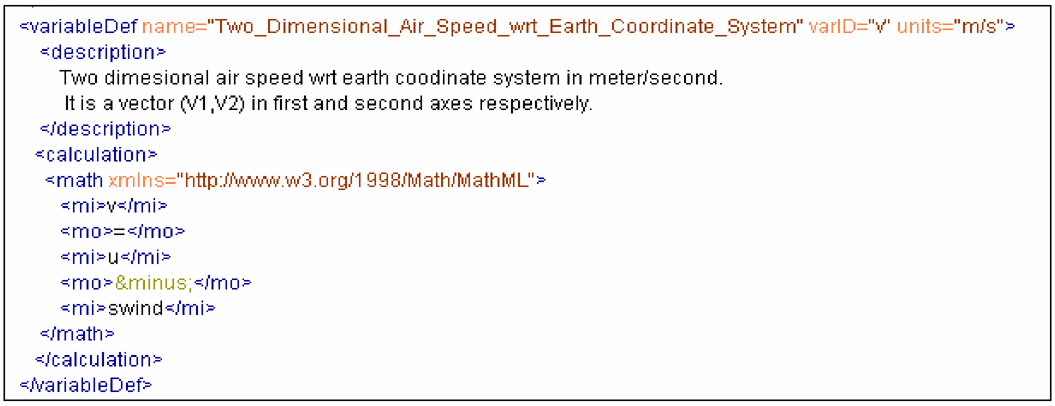

```
<variableDef name="Two_Dimensional_Air_Speed_wrt_Earth_Coordinate_System" varID="v" units="m/s">
  <description>
    Two dimesional air speed wrt earth coodinate system in meter/second.
     It is a vector (V1,V2) in first and second axes respectively.
  </description>
 <calculation>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <mi>v</mi>
    <mo>=</mo>
    <mi>u</mi>
    <mo>&minus;</mo>
    <mi>swind</mi>
  </math>
 </calculation>
</variableDef>
```

Figure 8 An Example DAVE-ML Variable Definition

This chapter introduced the basic concepts of ontologies and knowledge sharing. First the definition of ontology is given. Then the components, merit and applications of ontologies are explained. Ontology development efforts in engineering domain in the literature are reviewed. After presenting the basic practices of ontology construction, DAVE-ML effort of NASA is presented. In the next chapter, we will introduce Trajectory Simulation Ontology that was built as the domain model of trajectory simulation reuse infrastructure.

CHAPTER 4

TRAJECTORY SIMULATION ONTOLOGY

In this chapter, Trajectory Simulation ONTology (TSONT) is being presented. After an overview, top level entities, the hierarchies, classes and individuals of TSONT are discussed.

4.1 TSONT: An Overview

Trajectory Simulation Ontology, abbreviated as TSONT, is being developed as the domain model of Trajectory Simulation Reuse Infrastructure. It is being developed as a reusable knowledge library on trajectory simulations for trajectory simulation developers.

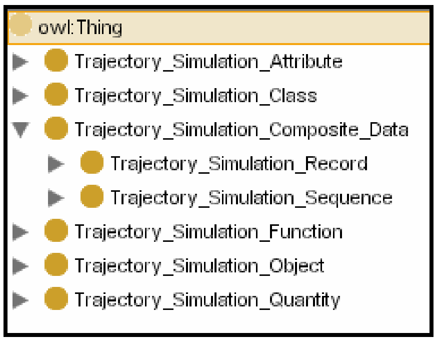
As mentioned earlier, ontologies are designed. For each artifact, the goal of its design is to conform to its requirements. The aim of developing TSONT as the domain model of the Trajectory Simulation Reuse Infrastructure is first to establish a common vocabulary that is agreed among people working on trajectory simulations and to create a backbone for systematization of knowledge on how to build a trajectory simulation. Considering these two requirements, TSONT design tried to capture common vocabulary of trajectory simulation and to present the entities and the relation among the entities in a trajectory simulation in a way to drive the design and development of simulation software.

There can be many other ways to capture and systemize or formalize the knowledge about trajectory simulation development. While there is no barrier for any team that is developing trajectory simulation to use TSONT as a domain model, TSONT is being developed for Modeling and Simulation Team of TUBITAK-SAGE. So, although the literature about trajectory simulation development is widely used to construct TSONT, it is being peer reviewed by the target reuse group in order to make it capture the shared vocabulary and the conceptualization of the people participating in that group.

Once TSONT is presented in the proceeding sections, one will see that some of the hierarchies are not complete and some knowledge about some classes is missing. The current state tries to capture the shared vocabulary and experienced conceptualization as related to ongoing projects rather than all available in the literature. As TSONT will be used as the domain model of the reuse infrastructure, it will be enhanced with the new experiences of the group. In this manner TSONT can be regarded as the knowledge base that is serving the gained experience in a formal way in order to be used in the future projects.

4.2 Top Level TSONT

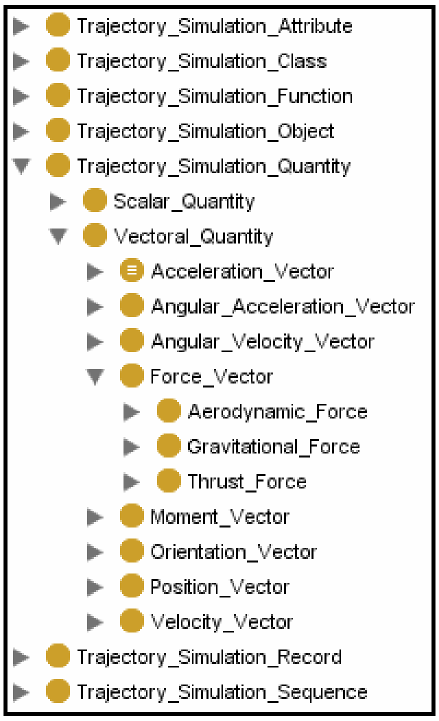
Top level entities of TSONT are Trajectory Simulation Attribute, Trajectory Simulation Class, Trajectory Simulation Function, Trajectory Simulation Object, Trajectory Simulation Quantity, Trajectory Simulation Record and Trajectory Simulation Sequence, as shown in Figure 9.



owl:Thing
- Trajectory_Simulation_Attribute
- Trajectory_Simulation_Class
- Trajectory_Simulation_Composite_Data
  - Trajectory_Simulation_Record
  - Trajectory_Simulation_Sequence
- Trajectory_Simulation_Function
- Trajectory_Simulation_Object
- Trajectory_Simulation_Quantity

Figure 9 TSONT Top Level Entities

These top level entities of TSONT agree with those of SUMO (Suggested Upper Merged Ontology). By reusing SUMO, we promote interoperability with other domain ontologies. SUMO is an upper level ontology proposed by the Standard Upper Ontology Working Group, an IEEE-sanctioned working group of collaborators from the fields of engineering, philosophy, and information science. The SUMO provides definitions for general-purpose terms and acts as a foundation for more specific domain ontologies [79].



```
▶ Trajectory_Simulation_Attribute
▶ Trajectory_Simulation_Class
▶ Trajectory_Simulation_Function
▶ Trajectory_Simulation_Object
▼ Trajectory_Simulation_Quantity
    ▶ Scalar_Quantity
    ▼ Vectoral_Quantity
        ▶ Acceleration_Vector
        ▶ Angular_Acceleration_Vector
        ▶ Angular_Velocity_Vector
        ▼ Force_Vector
            ▶ Aerodynamic_Force
            ▶ Gravitational_Force
            ▶ Thrust_Force
        ▶ Moment_Vector
        ▶ Orientation_Vector
        ▶ Position_Vector
        ▶ Velocity_Vector
▶ Trajectory_Simulation_Record
▶ Trajectory_Simulation_Sequence
```

Figure 10 Excerpt from TSONT Top Level

Trajectory Simulation Attribute can be regarded as the subclass of SUMO Attribute. It is defined as qualities in trajectory simulation domain which we cannot or prefer not to reify into subclasses of an object. Similarly, Trajectory Simulation Class is regarded as a subclass of SUMO Class and Trajectory Simulation Function as a subclass of SUMO Function. Trajectory Simulation Object, again a subclass of

SUMO Object, corresponds roughly to the class of ordinary physical objects in Trajectory Simulation domain. Trajectory Simulation Quantity is defined as any specification of how many or how much of something in Trajectory Simulation domain; it is a subclass of SUMO Quantity.

Trajectory Simulation Record and Trajectory Simulation Sequence are Trajectory Simulation Composite Data types that can be used for developing trajectory simulation codes. Although these data types are well established in programming, we refer to Vienna Development Method Specification Language (VDM-SL), an ISO Standard modeling language, for the sake of definiteness [80].

Figure 10 presents an excerpt from TSONT to show how these top level entities are inherited down to concepts of trajectory simulation domain. Trajectory Simulation can be a Scalar Quantity or a Vectoral Quantity. Acceleration Vector, Angular Acceleration Vector, Angular Velocity Vector, Force Vector, Moment Vector, Orientation Vector, Position Vector and Velocity Vector are all types of Vectoral Quantity. Further, Aerodynamic Force, Gravitational Force and Thrust Force are all derived from the Force Vector.

4.3 TSONT Hierarchies

4.3.1 Trajectory Simulation Objects

Trajectory Simulation Objects are the physical entities whose behavior is simulated. SUMO has a parallel definition for objects which corresponds objects roughly to the class of ordinary objects such as normal physical objects.

Munition, munition subsystems and weapon are said to be trajectory simulation objects. Munition is defined as a complete device charged with explosives, propellants, pyrotechnics, initiating composition, or nuclear, biological, or chemical material for use in military operations, including demolitions [81]. Munition subsystems are the parts of munition which affects its simulation, like guidance system, propellant or autopilot. Weapon is defined as the launch platform of munition. The hierarchy of Trajectory Simulation Object is given below in Figure 11.

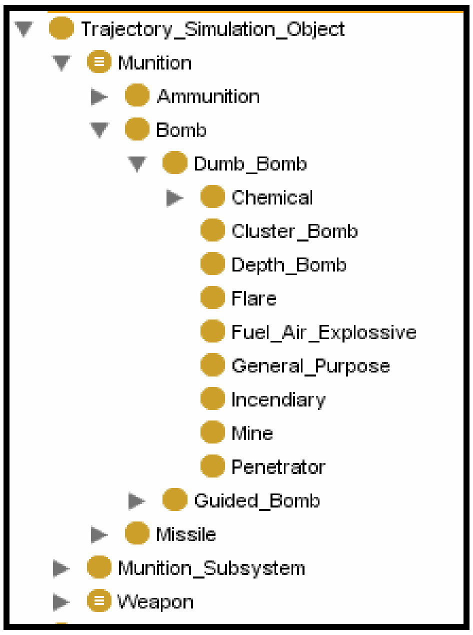


Figure 11 Trajectory Simulation Object Hierarchy

The munition classification is carried out in this domain analysis effort in order to scope the target group of system whose flight will be simulated by the trajectory simulations that will be developed by using the reuse infrastructure. There is no best classification or the correct classification for munitions. There can be number of ways to classify. There is no “one” classification in the literature that classifies all types of munition. Different classifications are unified in TSONT.

Three different sources are used to capture the taxonomy. The first one is AOP 29, “NATO Indirect Fire Ammunition Interchangeability” [82]. It is used for the classification of ammunitions, particularly the projectiles. The second one is DoD

101, “An Introduction to Military” which is published through Federation of American Scientists web site [83]. DoD 101 is used for the classification of bombs. The last one is DoD 4120.14-L, dated May, 12th, 2004 [84]. The Appendix 2 of this document is “Approved Mission Design Series Designators and Symbols for Guided Missiles, Rockets, Probed, Boosters, and Satellites”. This section is used for the classification of the missiles. These three classifications with their examples are presented in APPENDIX A. Besides, TSONT is given in APPENDIX N.

4.3.2 Trajectory Simulation Classes

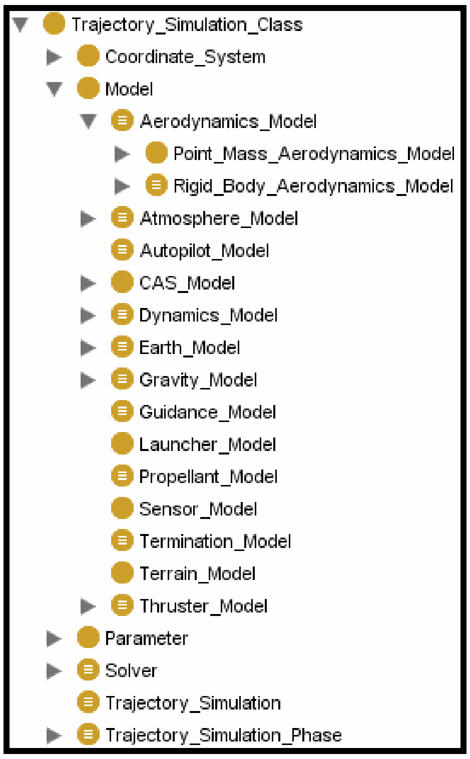


Figure 12 Trajectory Simulation Class Hierarchy

Trajectory Simulation Classes are subsets of the SUMO class. They are the abstract entities of trajectory simulation domain which are used to compute a trajectory.

This abstraction does not rely on any literature rather tried to capture the agreed conceptualization of trajectory simulation problem among the co-workers who will use the trajectory reuse infrastructure. The top level classes are Coordinate System, Model, Parameter, Solver, Trajectory Simulation and Trajectory Simulation Phase as given above in Figure 12.

Trajectory Simulation is defined as a tool to compute the flight path and other parameters of munition as it leaves the launcher and engages to a target based on mathematical model of munition, its subsystems and environment which consist of equations that describe physical laws and logical sequences [4].

Trajectory Simulation Phase is used to define some number of generic trajectory phases. These phases are defined considering the set of models they require to compute the trajectory throughout any instance of them. TSONT phase hierarch is depicted below in Figure 13.

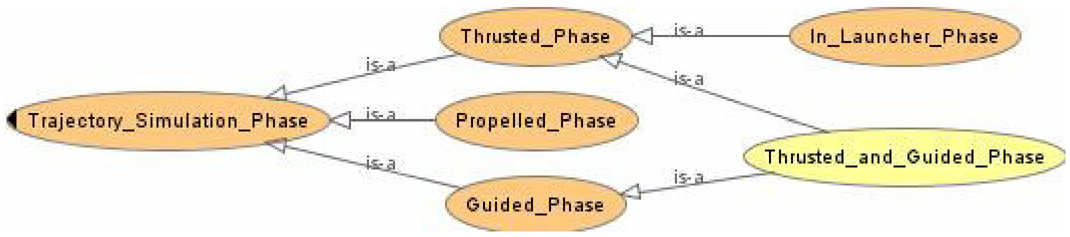


Figure 13 Trajectory Simulation Phase Hierarchy

Phase captures the basic models, such as Aerodynamics Model or Dynamics Model. Thrusted Phase, Guided Phase and Propelled Phase capture the related models.

Trajectory Simulation Phase stands for trajectory simulation phases which are neither guided nor propelled or thrusted. Trajectory Simulation Phase has an Aerodynamics Model (to compute aerodynamic forces and in some cases moments), Dynamics Model (to compute accelerations), Earth Model, Atmosphere

Model and Gravity Model. Propelled Phase stands for the trajectory simulation phases for the munitions which are propelled from a gun by a charge. So this phase has a propellant model in addition to the standard phase definition. Guided Phase stands as a class for guided munition trajectory phases. It extends the standard phase definition by adding Autopilot Model, CAS Model, Guidance Model and Sensor Model to calculate any guided trajectory segments. Thrusted Phase is added to represent the trajectory segments in which the thruster is working. It adds thruster model to standard phase definition. In Launcher Phase is a kind of Thrusted Phase where launcher model is used to consider the affects of launcher on trajectory.

Some phases have hybrid characteristics. In a trajectory phase both guidance and thruster might be active. E.g. Air-to-air missile simulation. In this case, that phase is derived both from guided phase and thrusted phase definitions so it has all the characteristics of both.

Model refers to logical or mathematical models of the actors that affect the flight of the munition. They encapsulate the approximations and assumptions, both structural and quantitative, about the affects of these actors to trajectory [1]. Aerodynamics Model, Atmosphere Model, Autopilot Model, CAS Model, Dynamics Model, Earth Model, Gravity Model, Guidance Model, Launcher Model, Propellant Model, Sensor Model, Termination Model, Terrain Model and the Thruster Model are the ones currently captured by TSONT.

Aerodynamics Model incorporates the effects of aerodynamic flow over munition body on its flight. This model is for computing the aerodynamics forces and moments acting on the munition. The classification is presented belove in Figure 14. The first level taxonomy divides the model into two as Point Mass Aerodynamics Model and Rigid Body Aerodynamics Model. Point Mass Aerodynamics Models only deal with the force but the Rigid Body Models also compute moments. Then the taxonomy is detailed to capture different aerodynamic models for different dynamic representations of munitions which affect the number of forces and moments computed. The last level in the taxonomy captures the reference frames that these forces and moments are computed.

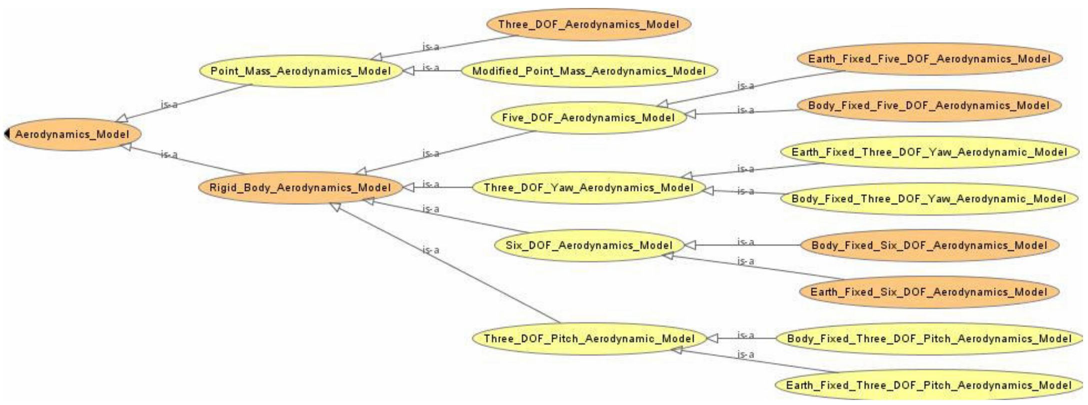

Aerodynamics_Model
Point_Mass_Aerodynamics_Model
Rigid_Body_Aerodynamics_Model
Three_DOF_Aerodynamics_Model
Modified_Point_Mass_Aerodynamics_Model
Five_DOF_Aerodynamics_Model
Three_DOF_Yaw_Aerodynamics_Model
Six_DOF_Aerodynamics_Model
Three_DOF_Pitch_Aerodynamic_Model
Earth_Fixed_Five_DOF_Aerodynamics_Model
Body_Fixed_Five_DOF_Aerodynamics_Model
Earth_Fixed_Three_DOF_Yaw_Aerodynamic_Model
Body_Fixed_Three_DOF_Yaw_Aerodynamic_Model
Body_Fixed_Six_DOF_Aerodynamics_Model
Earth_Fixed_Six_DOF_Aerodynamics_Model
Body_Fixed_Three_DOF_Pitch_Aerodynamics_Model
Earth_Fixed_Three_DOF_Pitch_Aerodynamics_Model
is-a

Figure 14 Aerodynamics Model Hierarchy

Atmosphere Model is assigned to provide the required meteorological conditions to the models which require them in order to incorporate the effects of atmospheric conditions to the munitions flight.

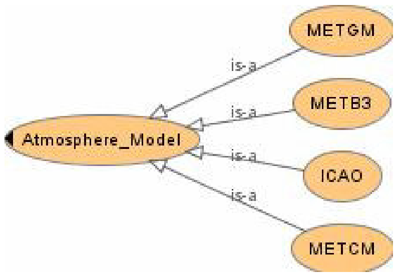


Figure 15 Atmosphere Model Hierarchy

TSONT captures four different representations of atmospheric conditions for trajectory simulations. These include Grided Met Message (METGM) [85], Computer Met Message (METCM) [86], Ballistic Met Message (METB3) [87] and standard atmosphere (ICAO) [88].

Although there are few different standard atmosphere definitions, the definition of International Civil Aviation Organisation which is widely used, is captured in TSONT. Grided Met Message is a pretty new concept. Technology development and validation efforts are still in progress. It provides atmosphere state at points in three dimensional space at a time. Below is a figure presenting use of METGM in a trajectory simulation.

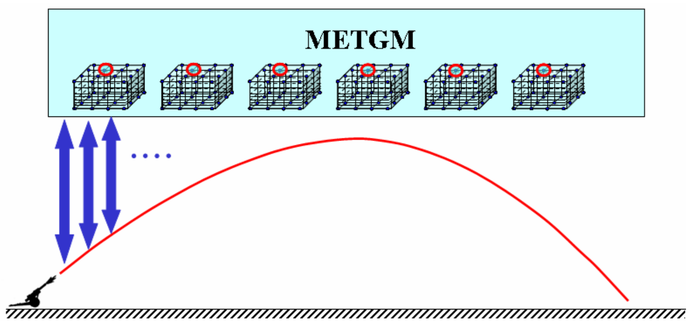
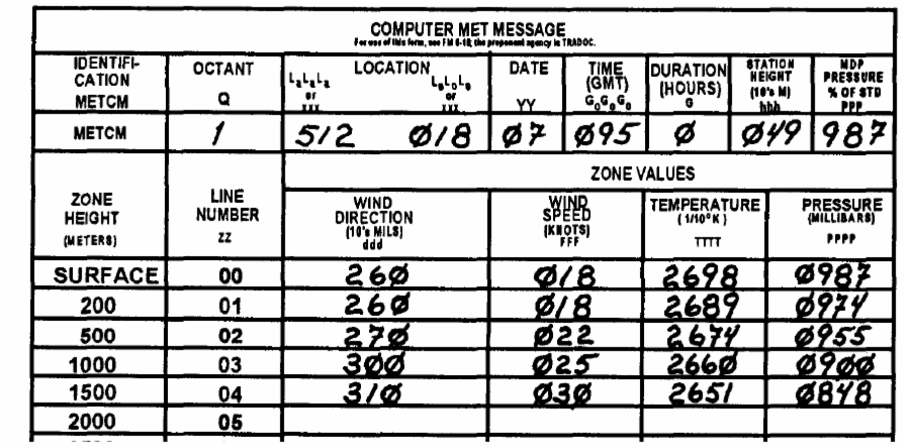


Figure 16 Grided Met Message in a Trajectory Simulation [89]

Ballistic met messages and computer met message are coded messages that report the atmospheric conditions in selected layers starting at the surface and extending to an altitude that will normally include the maximum ordinate of trajectory. Ballistic met message used in manual computations in which the weather conditions existing in one layer or zone are weighted against the conditions in lower layers and reported as percentages of standard. Computer met message on the other hand reports actual average wind direction, wind speed, air temperature, and pressure in each layer. The computer met message is designed to be used by the computer system in the computation of the equations of motion. An example of computer met message from Field Manual 6-40, is given below in Figure 17 [90].



COMPUTER MET MESSAGE

For use of this form, see FM 6-16; the proponent agency is TRADOC.

IDENTIFICATION METCM	OCTANT Q	LOCATION $L_aL_aL_a$ or XXX	LOCATION $L_oL_oL_o$ or XXX	DATE YY	TIME (GMT) $G_oG_oG_o$	DURATION (HOURS) G	STATION HEIGHT (10's M) hhh	MDP PRESSURE % OF STD PPP
METCM	1	512	Ø18	Ø7	Ø95	Ø	Ø49	987

ZONE HEIGHT (METERS)	LINE NUMBER ZZ	ZONE VALUES			
		WIND DIRECTION (10's MILS) ddd	WIND SPEED (KNOTS) FFF	TEMPERATURE (1/10°K) TTTT	PRESSURE (MILLIBARS) PPPP
SURFACE	00	26Ø	Ø18	2698	Ø987
200	01	26Ø	Ø18	2689	Ø974
500	02	27Ø	Ø22	2674	Ø955
1000	03	3ØØ	Ø25	266Ø	Ø9ØØ
1500	04	31Ø	Ø3Ø	2651	Ø848
2000	05				

Figure 17 Computer Met Message [90]

Autopilot Model stands for mathematical models that transform the guidance commands to control commands. Autopilots are actually control systems, which produce control action commands for the missile to track the commands coming from the guidance subsystem. They work as a translator between the guidance system and the control actuation system. There are a number of different autopilot implementations in literature [91]. Autopilot itself, receives instructions from the guidance subsystem about the strategy for how to steer the munition to intercept, and it translates these instructions into appropriate control of the munition [92]. Autopilot Model class of TSONT captures the basic functionality of mentioned above. As different autopilot models will be simulated by using TSONT, the Autopilot Model taxonomy of TSONT will be enhanced.

Control Actuation System Model represents the behavior of control actuation system of munition. It models how the commanded fin deflections are converted to actual fin deflections. Currently on a second order system model is captured in TSONT as given below in Figure 18 [92].

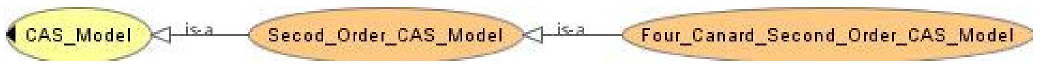


Figure 18 CAS Model Hierarchy

Dynamics Model employ the equations of motion, which describe the relationships between the forces and moments acting on the munition and the resulting motion [4]. Dynamics Models uses forces and moments to compute the dynamic model's state derivatives, namely velocity and acceleration of the munition.

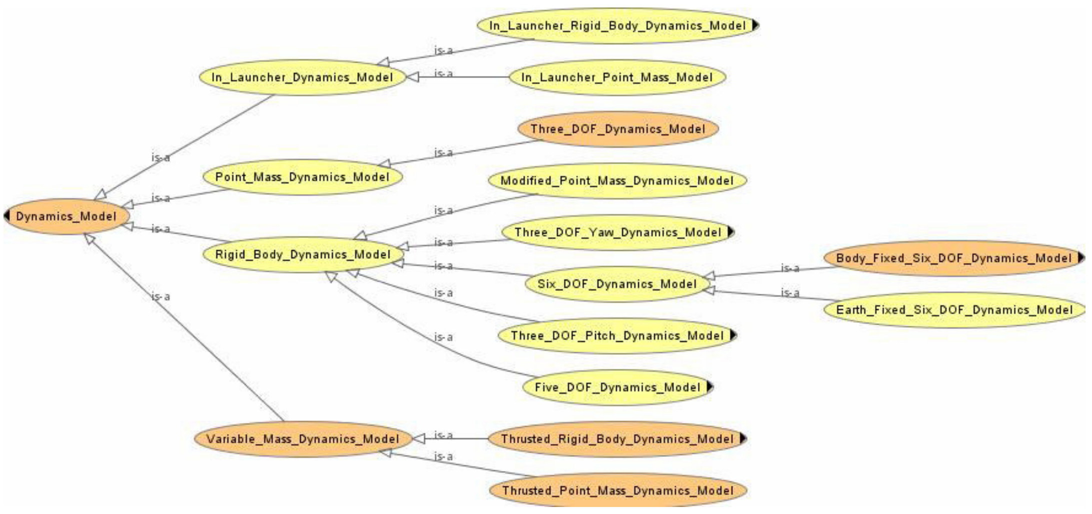

Dynamics_Model
In_Launcher_Dynamics_Model
In_Launcher_Rigid_Body_Dynamics_Model
In_Launcher_Point_Mass_Model
Point_Mass_Dynamics_Model
Three_DOF_Dynamics_Model
Modified_Point_Mass_Dynamics_Model
Rigid_Body_Dynamics_Model
Three_DOF_Yaw_Dynamics_Model
Six_DOF_Dynamics_Model
Body_Fixed_Six_DOF_Dynamics_Model
Earth_Fixed_Six_DOF_Dynamics_Model
Three_DOF_Pitch_Dynamics_Model
Five_DOF_Dynamics_Model
Variable_Mass_Dynamics_Model
Thrusted_Rigid_Body_Dynamics_Model
Thrusted_Point_Mass_Dynamics_Model
is-a

Figure 19 A Portion of Dynamics Model Hierarchy

Dynamics Models can be classified into two, as Point Mass and Rigid Body Dynamics Models in the first place considering the abstraction of the munition in the space. Besides these two, In Launcher Model represents a specific type of Dynamics Model where launcher constraints apply on the munition. Variable Mass Dynamics Models on the other hand stands for the dynamics models that thrust forces are also in consideration. The taxonomy is then detailed further considering the degrees of freedom and the reference frame. Figure 19 depicts a portion of Dynamics Model Hierarchy.

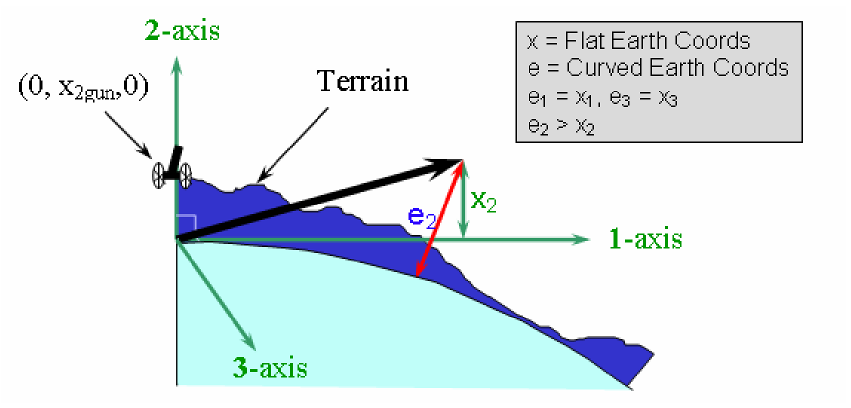


Figure 20 Earth Models [89]

Earth Model represents the model of the Earth on which the munition flies. This effects how the altitude of the munition in its flight is computed as given above in Figure 20 [23]. Two fundamental approaches are captured in TSONT. Those Earth Models are Flat Earth Model and Round Earth Model. Earth Model Hierarchy is presented below in Figure 21.

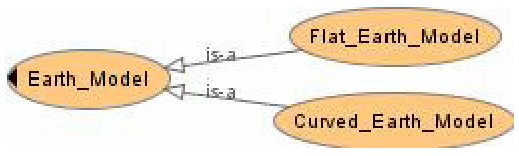


Figure 21 Earth Model Hierarchy

During the engagement process of a guided munition, number sensors measures one or more parameters of the path of the missile relative to the target. The logical process to determine the required flight path corrections based on the sensor measurements, is called a guidance law. The objective of a guidance law is to cause the munition to come as close as possible to the target. Guidance laws usually can be expressed in mathematical terms and are implemented through a combination of electrical circuits and mechanical control functions [4]. Guidance Models model the guidance laws of munitions which compute commanded accelerations using the relative target and munition motion. Rather than all guidance methods in the literature, TSONT captures the Guidance Models that have been experienced by target reuse group. Those are Proportional Navigation Guidance Model, Polynomial Guidance Model and Command Line of Sight Guidance Model.

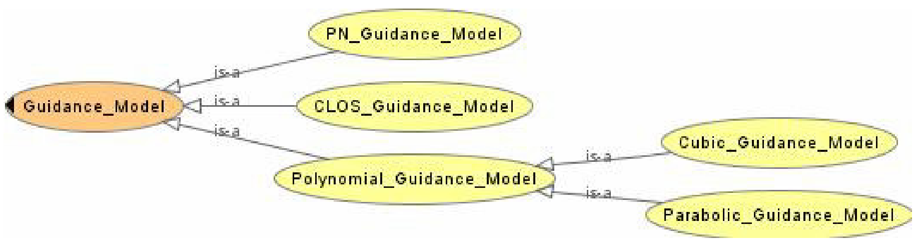


Figure 22 Guidance Model Hierarchy

Figure 22 presents the Guidance Model Hierarchy captured in TSONT. Proportional navigation guidance law computes acceleration commands, perpendicular to the munition and the target line of sight, which are proportional to line of sight rate and closing velocity [15]. Command Line of Sight (CLOS) guidance attempts to keep the missile within a guidance beam transmitted from the ground [4]. Polynomial guidance on the other hand, is based on generating the necessary commands on either the rates of the flight path angles or the normal acceleration components that keep the missile on a polynomial trajectory. The polynomial definition of the trajectory can be second or third order [92].

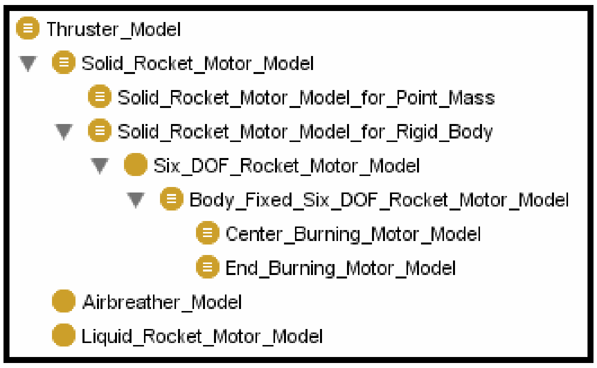
Launcher Model stands to represent the interactions of the munition and the launcher such as tip off rates and friction [23, 92].

For the projectiles that are launched by using a propellant charge, the muzzle velocity depends on factors like propellant type and propellant temperature. Propellant Models computes the muzzle velocity using the propellant properties [89].

In order to guide a munition for a successful intercept a target, it is vital to get the correct information about the motion of the target and munition itself during the flight. That information is provided by various sensors, such as inertial sensors, seekers, radar altimeters and GPS [92, 94]. The Sensor Model of TSONT includes the models of these sensors.

Termination Model is used to identify the end of either a phase or the whole trajectory. This logical model uses the phase termination conditions or the fuze data of the munition to determine the end of a trajectory phase or the trajectory itself.

Terrain Model represents the terrain the munition flies over. This model is responsible to provide the height of the terrain from sea level.



- Thruster_Model
  - Solid_Rocket_Motor_Model
    - Solid_Rocket_Motor_Model_for_Point_Mass
    - Solid_Rocket_Motor_Model_for_Rigid_Body
      - Six_DOF_Rocket_Motor_Model
        - Body_Fixed_Six_DOF_Rocket_Motor_Model
          - Center_Burning_Motor_Model
          - End_Burning_Motor_Model
  - Airbreather_Model
  - Liquid_Rocket_Motor_Model

Figure 23 Thruster Model Hierarchy

Above is the Thruster Model Hierarchy Several types of thrusters are used to propel the munition. Thruster Models are responsible to compute the thrust force and moment, and the mass of the thruster during the flight of the munition. Presently, solid propellant rocket motors, liquid propellant rocket motors and airbreathers are captured in TSONT. Based on the experience gained in the previous trajectory simulation projects, Solid Rocket Motors are further detailed in the hierarchy..

Solid Rocket Motor Models compute the thrust force depending on the design of the propellant which results in a specific impulse and the instantaneous ambient atmospheric pressure acting on an area equivalent to the exit area of the rocket nozzle. Liquid Rocket Motor models are essentially the same as Solid Rocket Motor Models unless the potential for throttle control is exploited in the design of the liquid system. Airbreather Models compute performance as a function of throttle control setting, Mach number, and ambient atmospheric properties [4].

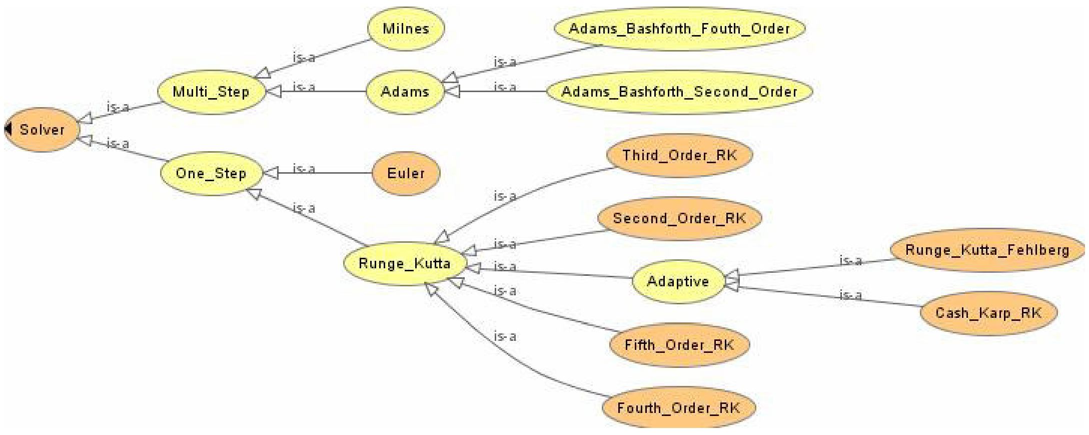


Figure 24 Solver Hierarchy

Above, Figure 24 depicts the solver hierarchy in TSONT. The differential equations frequently encountered in trajectory simulations cannot be solved by classical analytical methods. A large number of numerical integration methods have been developed to solve these equations using computers. Numeric solvers are classified as one-step and multi-step methods in TSONT. A one-step solver uses the value of the dependent variable only at the current integration step to compute the value at the succeeding step. A multi-step solver on the other hand uses values of the dependent variable at the current integration step and also at one or more preceding steps. One-step difference equations are self-starting, and multi step processes depend on a self-starting method to calculate the first few integration intervals. Euler's and the Runge-Kutta solvers are examples of one step solvers Milne's and the Adams solvers are examples of multi-step solvers [4].

Munition and its subsystems like motor, fuze or sensor, are represented by a set of parameters in the trajectory simulations. Parameter classes refer to the group of classes responsible to provide simulation parameters to the Model classes that simulate the behavior.

Aerodynamics and Physicals are the parameters of the munition itself. Subsystem parameters, Autopilot Data, CAS Data, Charge Data, Fuze Data, Guidance Data, Sensor Data, Solid Rocket Motor Data and Weapon Data are also captured in TSONT. This parameter class hierarchy is further detailed considering the set of data provided.

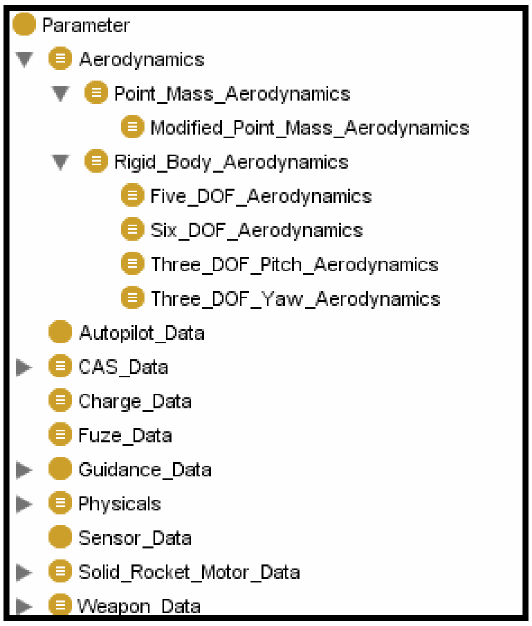


Figure 25 Parameter Hierarchy

The hierarchy of Parameter classes is presented above in Figure 25.Aerodynamics class for example is classified into Point Mass Aerodynamics and Rigid Body Aerodynamics. Then Rigid Body Aerodynamics class is further classified to classes like Five DOF Aerodynamics and Modified Point Mass Aerodynamics.

Vectors in three-dimensional space are widely used in trajectory simulation to represent factors such as forces, accelerations, velocities, positions, moments, angular accelerations, and angular rates. A vector has a meaning when it is described relative to some frame of reference. Right-handed, orthogonal coordinate systems are commonly used as frames of reference. A vector is described by its three components on the axes of a coordinate system. A number of different coordinate systems maybe used in a trajectory simulation. Coordinate systems are characterized by the positions of their origins, their angular orientations, and their motions relative to inertial space or relative to other specified systems. A vector can then be described by its coordinates in any of the coordinate systems [4]. Number coordinate systems are captured in TSONT, such as Body Coordinate System and Earth Coordinate System. The Coordinate System hierarchy is depicted below in Figure 26.

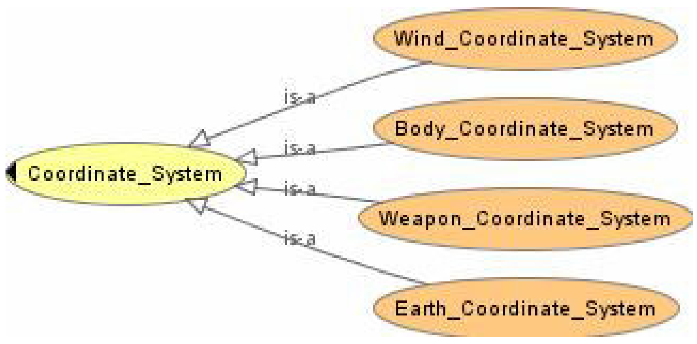


Figure 26 Coordinate System Hierarchy

4.3.3 Trajectory Simulation Functions

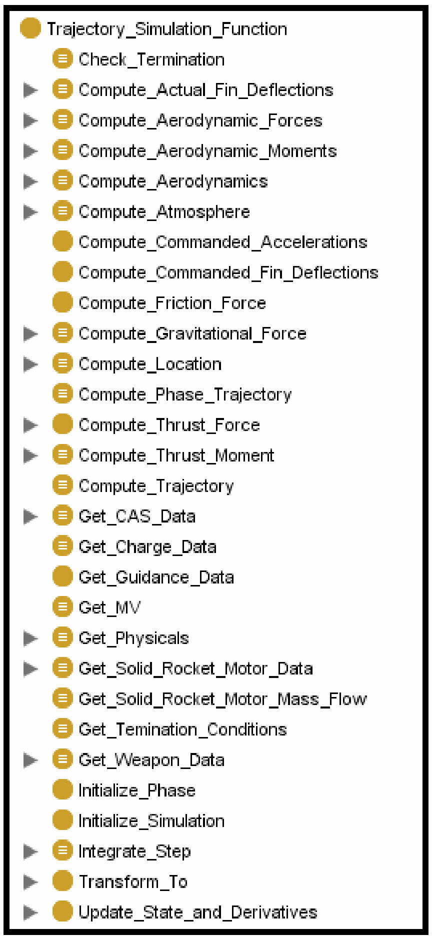
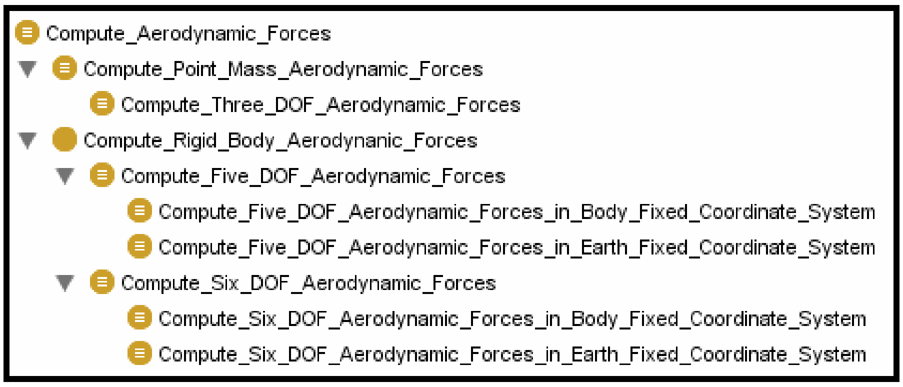

- Trajectory_Simulation_Function
  - Check_Termination
  - ▶ Compute_Actual_Fin_Deflections
  - ▶ Compute_Aerodynamic_Forces
  - ▶ Compute_Aerodynamic_Moments
  - ▶ Compute_Aerodynamics
  - ▶ Compute_Atmosphere
  - Compute_Commanded_Accelerations
  - Compute_Commanded_Fin_Deflections
  - Compute_Friction_Force
  - ▶ Compute_Gravitational_Force
  - ▶ Compute_Location
  - Compute_Phase_Trajectory
  - ▶ Compute_Thrust_Force
  - ▶ Compute_Thrust_Moment
  - Compute_Trajectory
  - ▶ Get_CAS_Data
  - Get_Charge_Data
  - Get_Guidance_Data
  - Get_MV
  - ▶ Get_Physicals
  - ▶ Get_Solid_Rocket_Motor_Data
  - Get_Solid_Rocket_Motor_Mass_Flow
  - Get_Temination_Conditions
  - ▶ Get_Weapon_Data
  - Initialize_Phase
  - Initialize_Simulation
  - ▶ Integrate_Step
  - ▶ Transform_To
  - ▶ Update_State_and_Derivatives

Figure 27 Trajectory Simulation Function Hierarchy

Trajectory Simulation Functions are a subset of SUMO function. Trajectory Simulation Function hierarchy captures the functionalities served by classes underneath the Trajectory Simulation Class hierarchy presented in the previous section. The list of functions captured in TSONT is given above in Figure 27.

Forces and moments acting on the munition during its flight are computed by using functions Compute Aerodynamics Force, Compute Aerodynamics Moment, Compute Friction Force, Compute Gravitational Force, Compute Thrust Force and Compute Thrust Moment. Compute Aerodynamics Force, Compute Aerodynamics Moment are served by Aerodynamics Model, Compute Friction Force is served by Launcher Model, Compute Gravitational Force is served by Gravity Model and Compute Thrust Force and Compute Thrust Moment are served by Thruster Model. Some of these functions also have their own hierarchy. Compute Aerodynamic Forces functions which use aerodynamic coefficients, atmosphere data, physical properties of the munition and the dynamic model state to compute the aerodynamic forces are further detailed to capture different types of implementations of these functions as given below in Figure 28. They are classified depending on the degrees of freedom of the dynamics model that will use this forces and the reference frame in which the forces are defined.



```
Compute_Aerodynamic_Forces
▼ Compute_Point_Mass_Aerodynamic_Forces
    Compute_Three_DOF_Aerodynamic_Forces
▼ Compute_Rigid_Body_Aerodynanic_Forces
  ▼ Compute_Five_DOF_Aerodynamic_Forces
      Compute_Five_DOF_Aerodynamic_Forces_in_Body_Fixed_Coordinate_System
      Compute_Five_DOF_Aerodynamic_Forces_in_Earth_Fixed_Coordinate_System
  ▼ Compute_Six_DOF_Aerodynamic_Forces
      Compute_Six_DOF_Aerodynamic_Forces_in_Body_Fixed_Coordinate_System
      Compute_Six_DOF_Aerodynamic_Forces_in_Earth_Fixed_Coordinate_System
```

Figure 28 Compute Aerodynamic Forces Hierarchy

Compute Commanded Acceleration is the functionality provided by Guidance Model. It computes the commanded acceleration of the munition using the guidance law. Then Compute Commanded Fin Deflections functionality that is served by Autopilot Model uses the commanded accelerations to compute the commanded fin deflections. CAS Model serves Compute Actual Fin Deflections functionality. As given below in Figure 29, TSONT has a sole function under this hierarchy for four canard systems. The aim of this function is to compute actual fin deflections using commanded fin deflections.

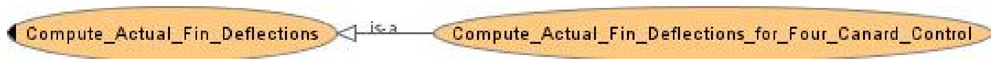

Compute_Actual_Fin_Deflections
is-a
Compute_Actual_Fin_Deflections_for_Four_Canard_Control

Figure 29 Compute Actual Fin Deflections Hierarchy

Compute Atmosphere function is served by Atmosphere Model. It is used to provide atmospheric properties at any instant of flight depending on the height. These functions which are given below in Figure 30, are classified depending on the format that they read the metrological definition.

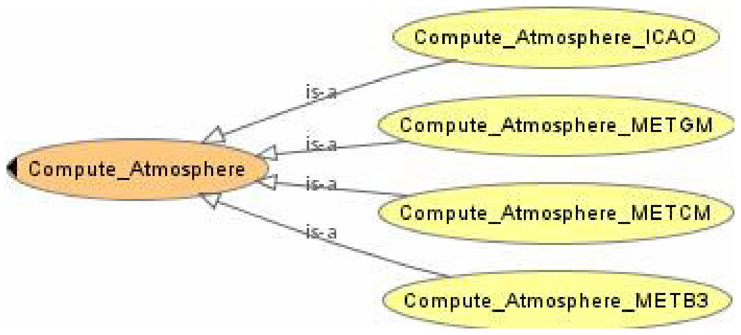


Figure 30 Compute Atmosphere Hierarchy

Check Termination function is served by Termination model. It computes the termination status using State information and a Termination Record which defines the termination conditions.

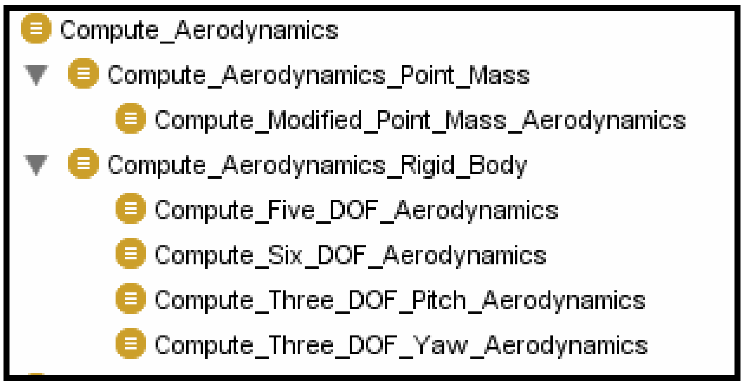

Compute_Aerodynamics
Compute_Aerodynamics_Point_Mass
Compute_Modified_Point_Mass_Aerodynamics
Compute_Aerodynamics_Rigid_Body
Compute_Five_DOF_Aerodynamics
Compute_Six_DOF_Aerodynamics
Compute_Three_DOF_Pitch_Aerodynamics
Compute_Three_DOF_Yaw_Aerodynamics

Figure 31 Compute Aerodynamics Hierarchy

There are number of functions in the hierarchy that are served by Parameter classes to provide the simulation parameters. Compute Aerodynamics, Get Physicals, Get Solid Rocket Motor Data are some of them. Compute Aerodynamics, as an example, refers to the functionality provided by Aerodynamics class. It is responsible for computing the aerodynamic coefficients using the flight conditions. Its hierarchy is depicted above in Figure 31.

Initialize Phase and Initialize Simulation functions refers to the functionalities served by Phase and Simulation classes to accomplish series of tasks to initialize a trajectory simulation or a phase of a trajectory simulation like setting the initial state. Likewise Compute Trajectory functionality of Trajectory Simulation class is responsible to compute the whole trajectory and Compute Phase Trajectory functionality of any Phase class is responsible to compute the trajectory of a particular phase.

Below, Figure 32 is the hierarchy of Integrate Step functions captured in TSONT. Those functions are provided by Solver classes to integrate the differential equation to compute the state of the simulation in the next time step. These functions require state derivatives to be computed by Update State and Derivative functions whose hierarchy is given below in Figure 33.

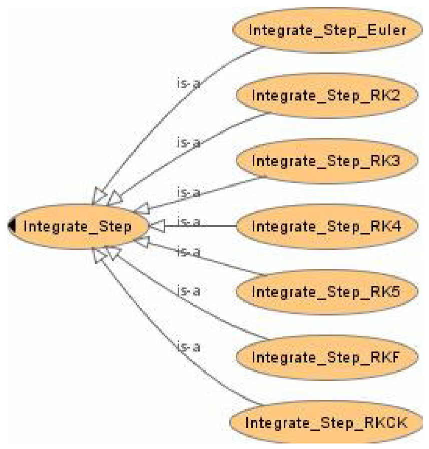


Figure 32 Integrate Step Hierarchy

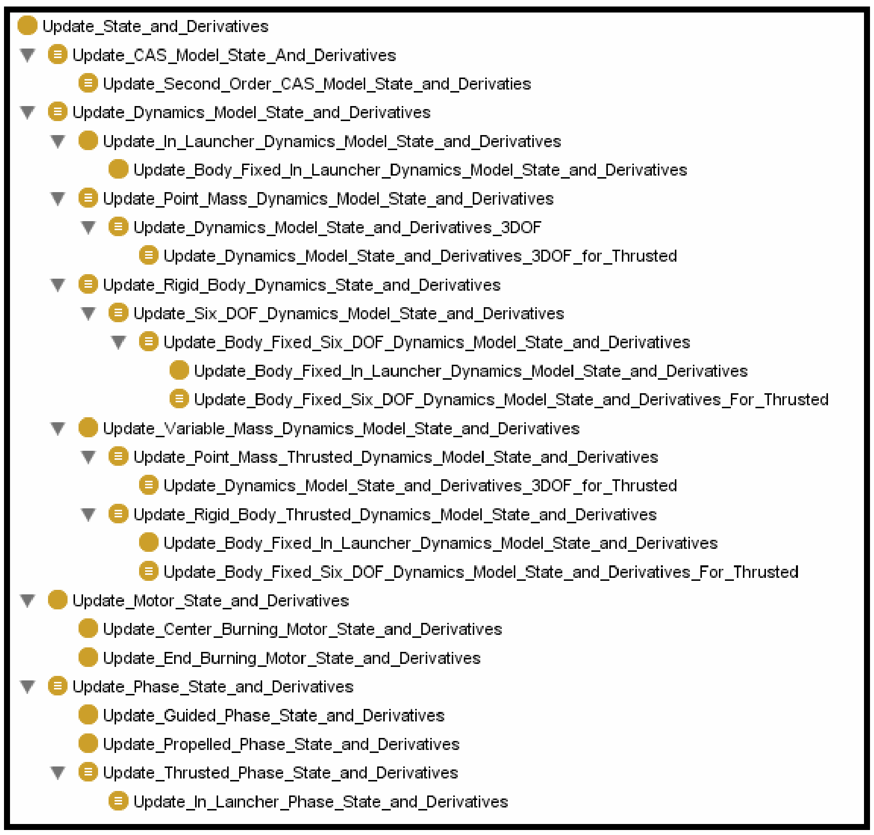

- Update_State_and_Derivatives
- Update_CAS_Model_State_And_Derivatives
  - Update_Second_Order_CAS_Model_State_and_Derivaties
- Update_Dynamics_Model_State_and_Derivatives
  - Update_In_Launcher_Dynamics_Model_State_and_Derivatives
    - Update_Body_Fixed_In_Launcher_Dynamics_Model_State_and_Derivatives
  - Update_Point_Mass_Dynamics_Model_State_and_Derivatives
    - Update_Dynamics_Model_State_and_Derivatives_3DOF
      - Update_Dynamics_Model_State_and_Derivatives_3DOF_for_Thrusted
  - Update_Rigid_Body_Dynamics_State_and_Derivatives
    - Update_Six_DOF_Dynamics_Model_State_and_Derivatives
      - Update_Body_Fixed_Six_DOF_Dynamics_Model_State_and_Derivatives
        - Update_Body_Fixed_In_Launcher_Dynamics_Model_State_and_Derivatives
        - Update_Body_Fixed_Six_DOF_Dynamics_Model_State_and_Derivatives_For_Thrusted
  - Update_Variable_Mass_Dynamics_Model_State_and_Derivatives
    - Update_Point_Mass_Thrusted_Dynamics_Model_State_and_Derivatives
      - Update_Dynamics_Model_State_and_Derivatives_3DOF_for_Thrusted
    - Update_Rigid_Body_Thrusted_Dynamics_Model_State_and_Derivatives
      - Update_Body_Fixed_In_Launcher_Dynamics_Model_State_and_Derivatives
      - Update_Body_Fixed_Six_DOF_Dynamics_Model_State_and_Derivatives_For_Thrusted
- Update_Motor_State_and_Derivatives
  - Update_Center_Burning_Motor_State_and_Derivatives
  - Update_End_Burning_Motor_State_and_Derivatives
- Update_Phase_State_and_Derivatives
  - Update_Guided_Phase_State_and_Derivatives
  - Update_Propelled_Phase_State_and_Derivatives
  - Update_Thrusted_Phase_State_and_Derivatives
    - Update_In_Laincher_Phase_State_and_Derivatives

Figure 33 Update State and Derivatives Hierarchy

4.3.4 Trajectory Simulation Quantities

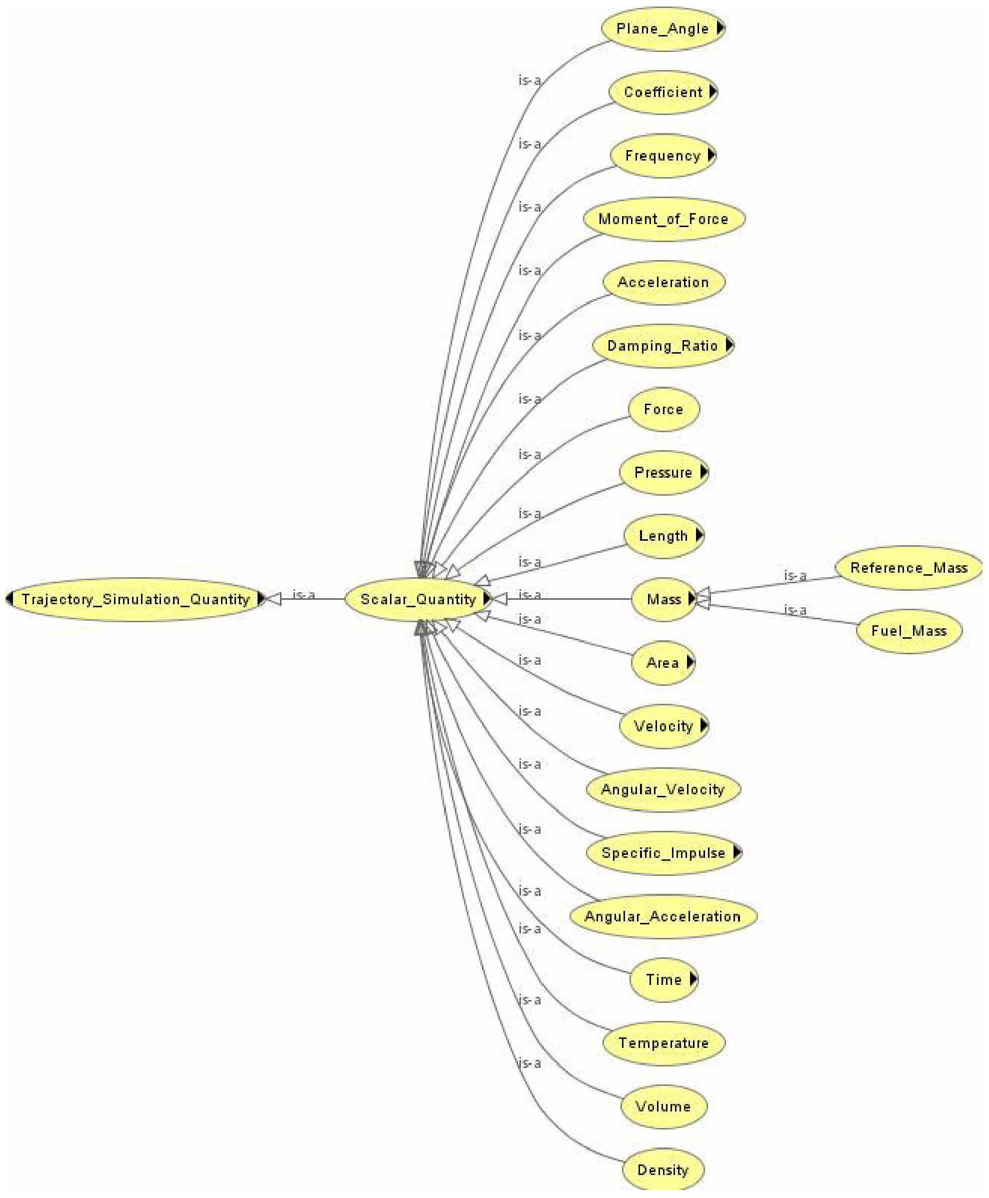
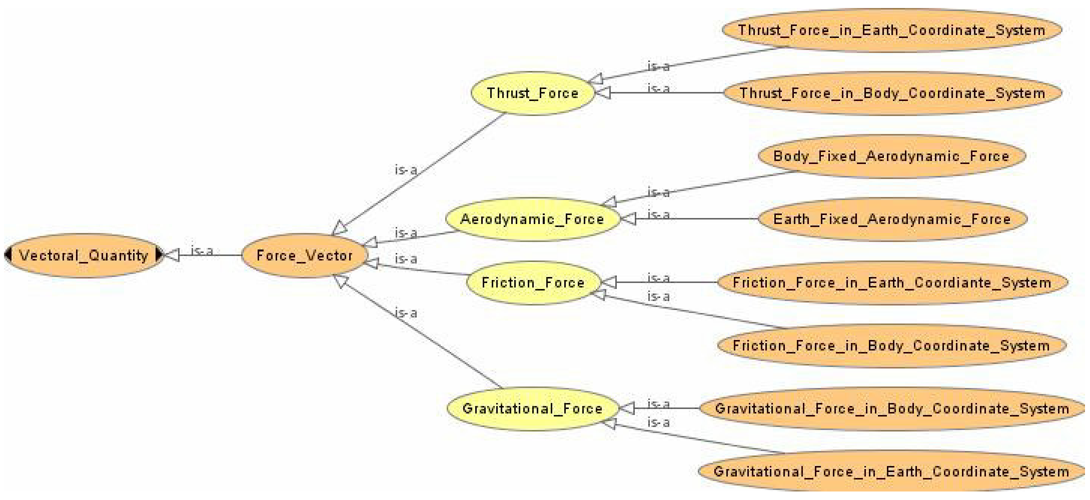


Figure 34 A Portion of Scalar Quantity Hierarchy

OWL classes under the Trajectory Simulation Quantity construct a subset of SUMO quantity. They are used to specify the quantities of trajectory simulation domain.

Trajectory Simulation Quantity is divided into two subsets, namely Scalar Quantities and Vectoral Quantities. Scalar Quantities are then divided to subgroups using the classification given in The International System of Units [95]. Some of scalar quantities captured in TSONT are Density, Mass and Length. A portion of Scalar Quantity hierarchy is depicted above in Figure 34.

Vectoral quantities in trajectory simulation domain are grouped as Acceleration Vector, Angular Acceleration Vector, Velocity Vector, Angular Velocity Vector, Force Vector, Moment Vector, Position Vector and Orientation Vector. Then these groups are detailed to capture the quantities underneath them. The hierarchy of the Force Vector is presented below in Figure 35, as an example.



Vectoral_Quantity
is-a
Force_Vector
Thrust_Force
Aerodynamic_Force
Friction_Force
Gravitational_Force
Thrust_Force_in_Earth_Coordinate_System
Thrust_Force_in_Body_Coordinate_System
Body_Fixed_Aerodynamic_Force
Earth_Fixed_Aerodynamic_Force
Friction_Force_in_Earth_Coordiante_System
Friction_Force_in_Body_Coordinate_System
Gravitational_Force_in_Body_Coordinate_System
Gravitational_Force_in_Earth_Coordinate_System

Figure 35 Force Vector Hierarchy

4.3.5 Trajectory Simulation Attributes

Trajectory Simulation Attributes are a subset of SUMO attribute. Attribute is defined as qualities which we cannot or choose not to reify into subclasses of Object in SUMO [79]. Trajectory Simulation Attribute defines a set of qualities of Trajectory Simulation Classes and Trajectory Simulation Objects like the termination status of a trajectory or the ellipsoid of a location.

4.3.6 Trajectory Simulation Composite Data

Composite types are types whose values are composed or structured from simpler values [96]. They are used to group some data that forms a coherent construct. In developing trajectory simulation software, composite data types are widely used. TSONT tries to capture the composite data types that are used in the target reuse community. Trajectory Simulation Record and Trajectory Simulation Sequence are base Trajectory Simulation Composite Data types. Although these data types are well established in programming, Vienna Development Method Specification Language (VDM-SL), an ISO Standard modeling language, is referred for the sake of definiteness [80].

VDM-SL defines record as a construct, similar to the record or struct in programming languages that is used to model values made up of several components [80]. A portion of Trajectory Simulation Record is depicted below in Figure 36.

Sequence is defined as ordered collection of values in VDM-SL [80]. We present Tuple hierarch captured in TSONT, as a part of Trajectory Simulation Sequence hierarchy, below in Figure 37.

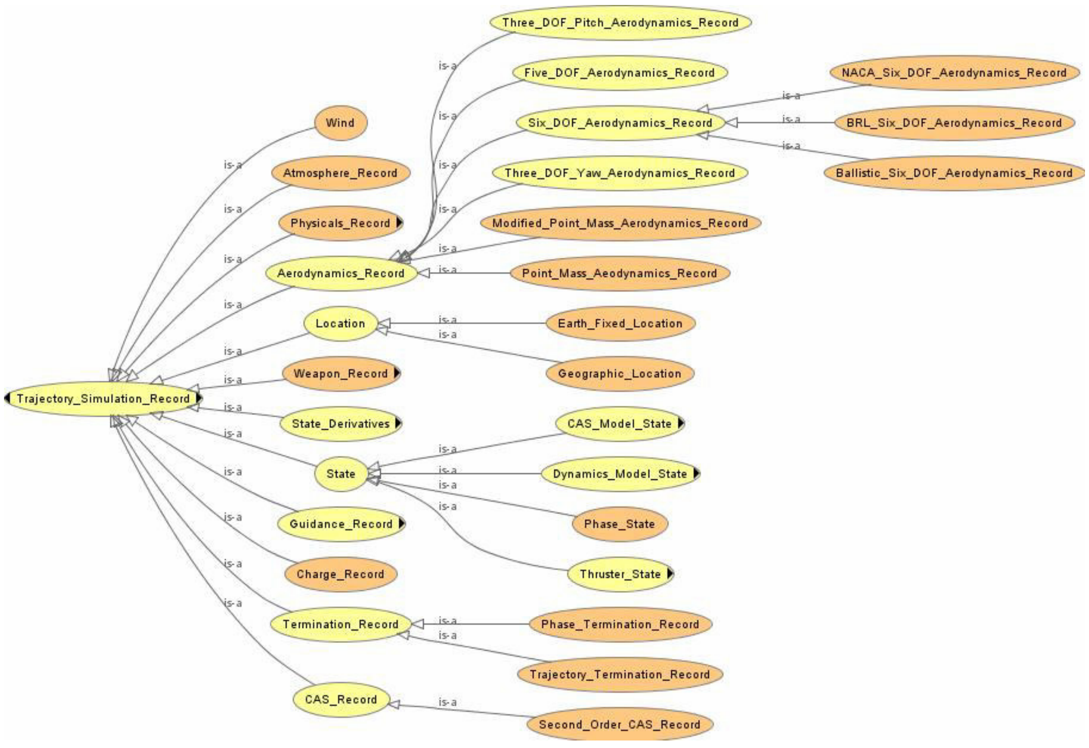

Trajectory_Simulation_Record
Wind
Atmosphere_Record
Physicals_Record
Aerodynamics_Record
Location
Weapon_Record
State_Derivatives
State
Guidance_Record
Charge_Record
Termination_Record
CAS_Record
Three_DOF_Pitch_Aerodynamics_Record
Five_DOF_Aerodynamics_Record
Six_DOF_Aerodynamics_Record
Three_DOF_Yaw_Aerodynamics_Record
Modified_Point_Mass_Aerodynamics_Record
Point_Mass_Aeodynamics_Record
NACA_Six_DOF_Aerodynamics_Record
BRL_Six_DOF_Aerodynamics_Record
Ballistic_Six_DOF_Aerodynamics_Record
Earth_Fixed_Location
Geographic_Location
CAS_Model_State
Dynamics_Model_State
Phase_State
Thruster_State
Phase_Termination_Record
Trajectory_Termination_Record
Second_Order_CAS_Record
is-a

Figure 36 A Portion of Trajectory Simulation Record Hierarchy

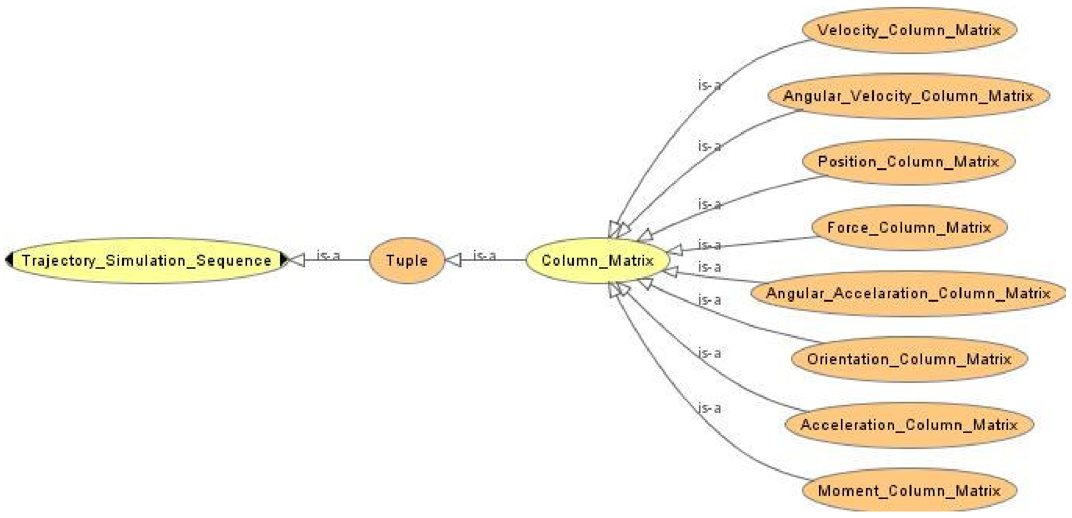
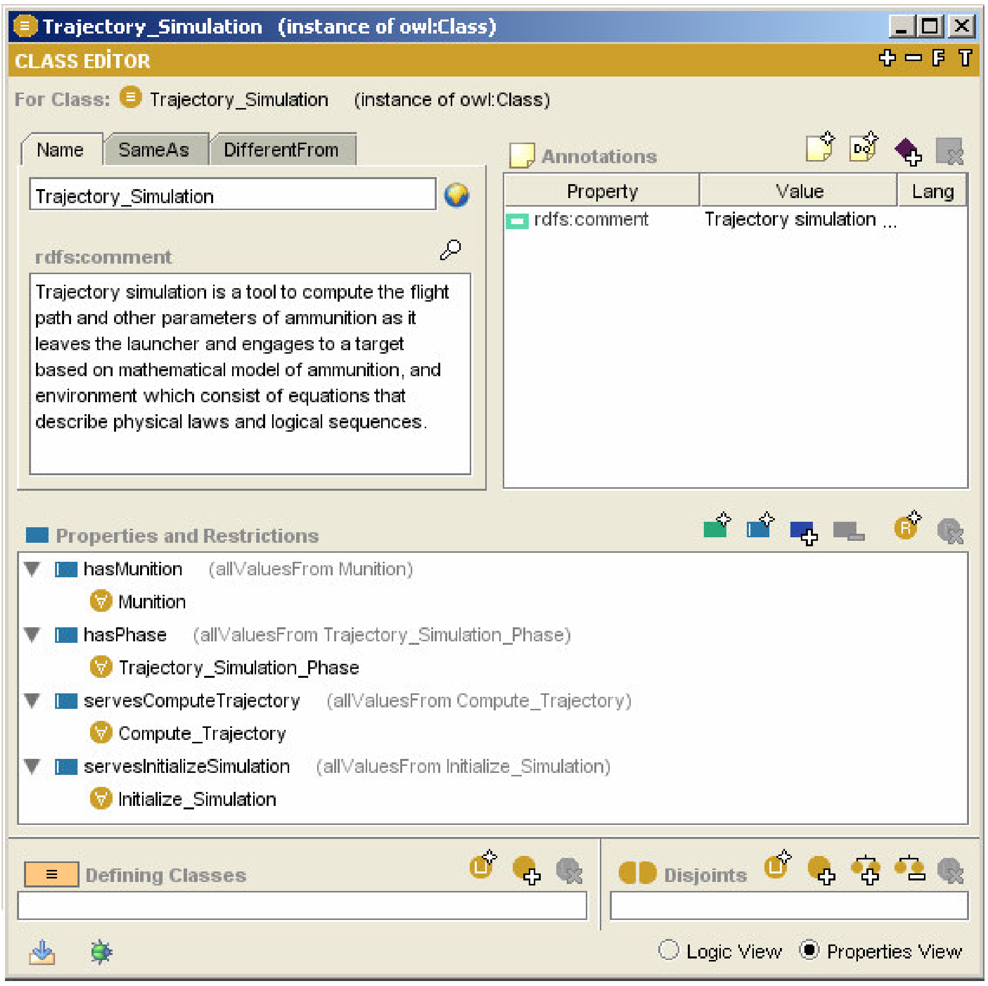

Trajectory_Simulation_Sequence
Tuple
Column_Matrix
Velocity_Column_Matrix
Angular_Velocity_Column_Matrix
Position_Column_Matrix
Force_Column_Matrix
Angular_Accelaration_Column_Matrix
Orientation_Column_Matrix
Acceleration_Column_Matrix
Moment_Column_Matrix
is-a

Figure 37 Tuple Hierarchy

4.4 TSONT Classes

After presenting the taxonomy of trajectory simulation concepts in the previous section, this section will discuss how these concepts are defined in TSONT. The relations among these concepts will also be given. This section will start with the definition of a trajectory simulation, continue with classes, services, and conclude with quantities and composite data. The relations of the concepts captured in TSONT and the structure of them will be discussed in this sequence.


Trajectory_Simulation (instance of owl:Class)
CLASS EDİTOR
For Class: Trajectory_Simulation (instance of owl:Class)
Name
SameAs
DifferentFrom
Annotations
Properties and Restrictions
Defining Classes
Disjoints
Logic View
Properties View

Name: Trajectory_Simulation

rdfs:comment: Trajectory simulation is a tool to compute the flight path and other parameters of ammunition as it leaves the launcher and engages to a target based on mathematical model of ammunition, and environment which consist of equations that describe physical laws and logical sequences.

Property	Value	Lang
rdfs:comment	Trajectory simulation ...	

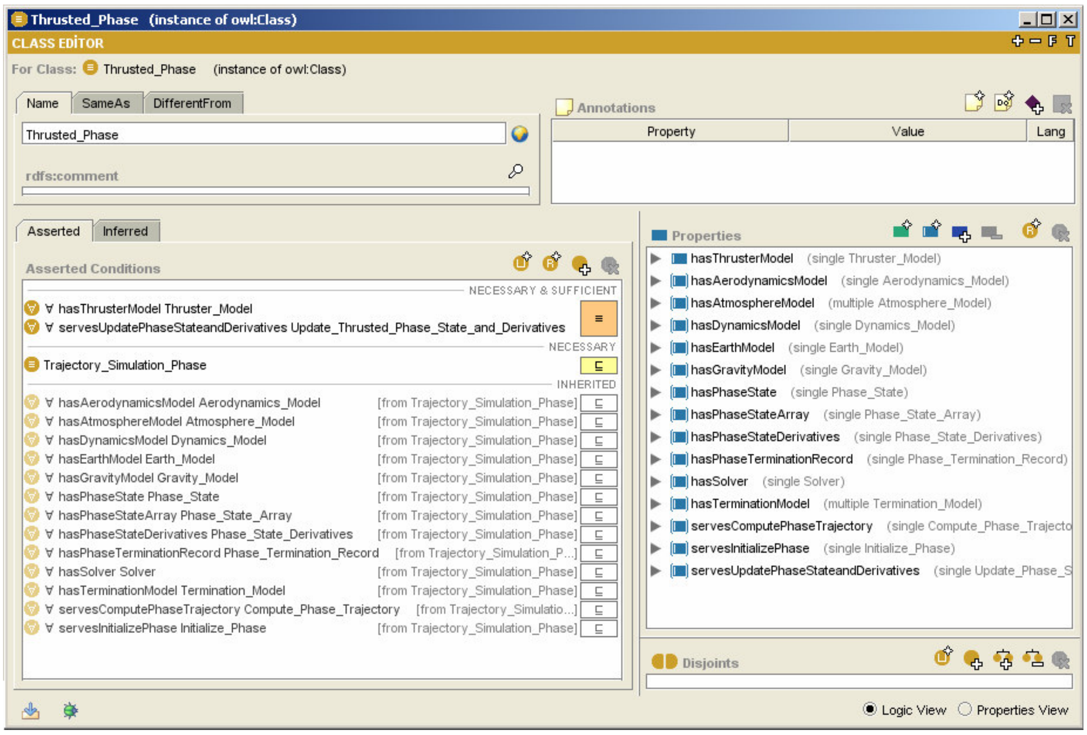
Properties and Restrictions:

- hasMunition (allValuesFrom Munition)
  - Munition
- hasPhase (allValuesFrom Trajectory_Simulation_Phase)
  - Trajectory_Simulation_Phase
- servesComputeTrajectory (allValuesFrom Compute_Trajectory)
  - Compute_Trajectory
- servesInitializeSimulation (allValuesFrom Initialize_Simulation)
  - Initialize_Simulation

Logic View [✓ Properties View]

Figure 38 Trajectory Simulation Class

The structure of TSONT is devised to render concept to implementation mapping amenable to reuse by trajectory simulation developers. Trajectory simulations, which can be composed of multiple phases, are to be executed to calculate the trajectories of munitions. One may need to initialize a trajectory simulation by setting the initial conditions before running it. These facts are reflected in TSONT as depicted in Figure 38. Trajectory Simulation is defined by hasMunition, hasPhase, servesInitializeSimulation and servesComputeTrajectory properties. These properties formalize the definition of the trajectory simulation.


Thrusted_Phase (instance of owl:Class)

CLASS EDITOR

For Class: Thrusted_Phase (instance of owl:Class)

Name | SameAs | DifferentFrom

Thrusted_Phase

rdfs:comment

Annotations

Property	Value	Lang

Asserted | Inferred

Asserted Conditions

NECESSARY & SUFFICIENT
- ∀ hasThrusterModel Thruster_Model
- ∀ servesUpdatePhaseStateandDerivatives Update_Thrusted_Phase_State_and_Derivatives

NECESSARY
- Trajectory_Simulation_Phase

INHERITED
- ∀ hasAerodynamicsModel Aerodynamics_Model [from Trajectory_Simulation_Phase]
- ∀ hasAtmosphereModel Atmosphere_Model [from Trajectory_Simulation_Phase]
- ∀ hasDynamicsModel Dynamics_Model [from Trajectory_Simulation_Phase]
- ∀ hasEarthModel Earth_Model [from Trajectory_Simulation_Phase]
- ∀ hasGravityModel Gravity_Model [from Trajectory_Simulation_Phase]
- ∀ hasPhaseState Phase_State [from Trajectory_Simulation_Phase]
- ∀ hasPhaseStateArray Phase_State_Array [from Trajectory_Simulation_Phase]
- ∀ hasPhaseStateDerivatives Phase_State_Derivatives [from Trajectory_Simulation_Phase]
- ∀ hasPhaseTerminationRecord Phase_Termination_Record [from Trajectory_Simulation_P...]
- ∀ hasSolver Solver [from Trajectory_Simulation_Phase]
- ∀ hasTerminationModel Termination_Model [from Trajectory_Simulation_Phase]
- ∀ servesComputePhaseTrajectory Compute_Phase_Trajectory [from Trajectory_Simulatio...]
- ∀ servesInitializePhase Initialize_Phase [from Trajectory_Simulation_Phase]

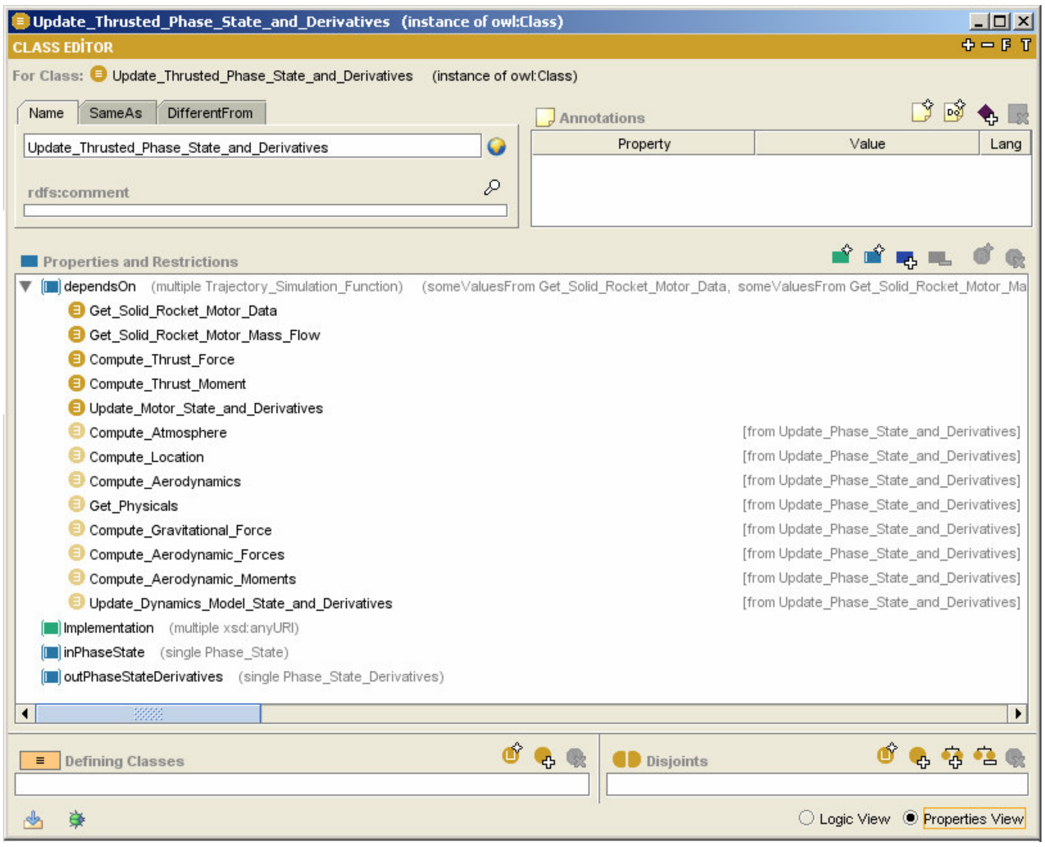
Properties
- hasThrusterModel (single Thruster_Model)
- hasAerodynamicsModel (single Aerodynamics_Model)
- hasAtmosphereModel (multiple Atmosphere_Model)
- hasDynamicsModel (single Dynamics_Model)
- hasEarthModel (single Earth_Model)
- hasGravityModel (single Gravity_Model)
- hasPhaseState (single Phase_State)
- hasPhaseStateArray (single Phase_State_Array)
- hasPhaseStateDerivatives (single Phase_State_Derivatives)
- hasPhaseTerminationRecord (single Phase_Termination_Record)
- hasSolver (single Solver)
- hasTerminationModel (multiple Termination_Model)
- servesComputePhaseTrajectory (single Compute_Phase_Trajecto
- servesInitializePhase (single Initialize_Phase)
- servesUpdatePhaseStateandDerivatives (single Update_Phase_S

Disjoints

Logic View [✓] | Properties View

Figure 39 Thrusted Phase

Trajectory simulation phases are defined as the segments of a munition flight whose simulation can be performed by using a distinct set of models solved by a numeric solver. For example, computing the trajectory during boost phase and after motor is off, which is called free flight, requires a particular sets of models. Figure 39 presents the definition of Thrusted Phase in TSONT. This definition specifies the models that will be used to compute a segment of a trajectory where a type of a thruster is producing thrust. It also says that, one may need to initialize a phase before computing the phase trajectory. In addition, it states that each phase will require some kind of a solver to compute the numerical solutions of differential equations. The definitions of some of the other phases will be given in APPENDIX B with some other TSONT class examples. For a complete TSONT, refer to APPENDIX N.


Update_Thrusted_Phase_State_and_Derivatives (instance of owl:Class)

CLASS EDITOR

For Class: Update_Thrusted_Phase_State_and_Derivatives (instance of owl:Class)

Name | SameAs | DifferentFrom

Update_Thrusted_Phase_State_and_Derivatives

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

- dependsOn (multiple Trajectory_Simulation_Function) (someValuesFrom Get_Solid_Rocket_Motor_Data, someValuesFrom Get_Solid_Rocket_Motor_Ma
  - Get_Solid_Rocket_Motor_Data
  - Get_Solid_Rocket_Motor_Mass_Flow
  - Compute_Thrust_Force
  - Compute_Thrust_Moment
  - Update_Motor_State_and_Derivatives
  - Compute_Atmosphere [from Update_Phase_State_and_Derivatives]
  - Compute_Location [from Update_Phase_State_and_Derivatives]
  - Compute_Aerodynamics [from Update_Phase_State_and_Derivatives]
  - Get_Physicals [from Update_Phase_State_and_Derivatives]
  - Compute_Gravitational_Force [from Update_Phase_State_and_Derivatives]
  - Compute_Aerodynamic_Forces [from Update_Phase_State_and_Derivatives]
  - Compute_Aerodynamic_Moments [from Update_Phase_State_and_Derivatives]
  - Update_Dynamics_Model_State_and_Derivatives [from Update_Phase_State_and_Derivatives]
- Implementation (multiple xsd:anyURI)
- inPhaseState (single Phase_State)
- outPhaseStateDerivatives (single Phase_State_Derivatives)

Defining Classes

Disjoints

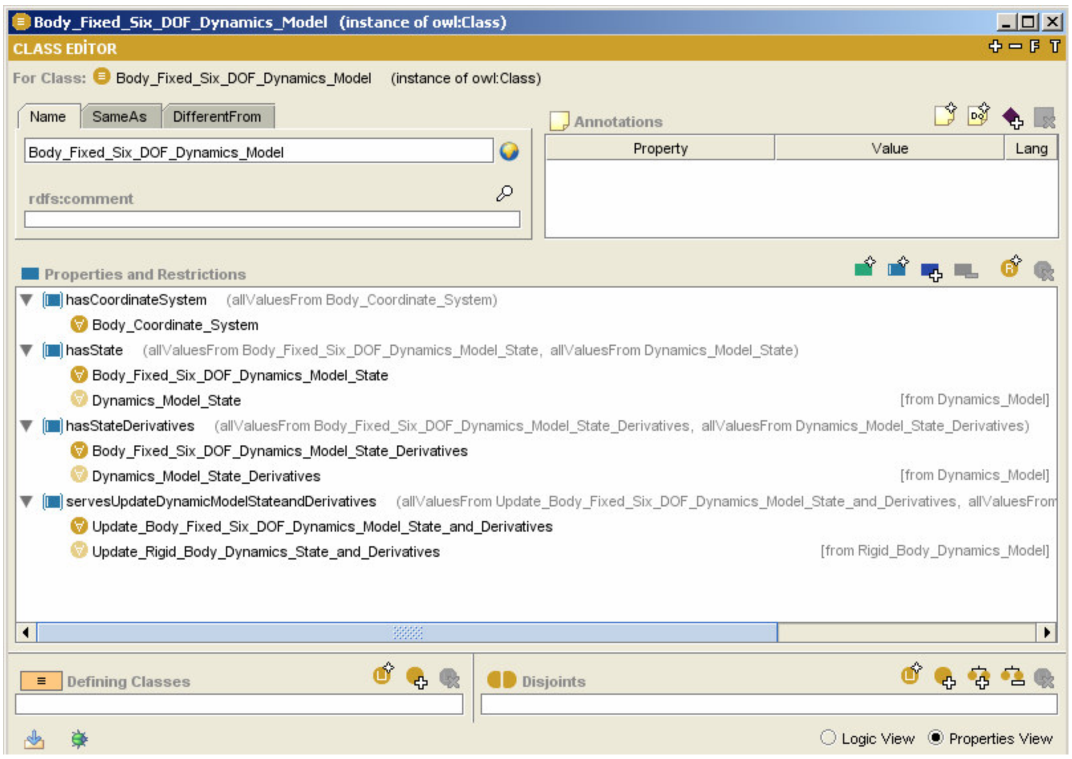
Logic View [✓ Properties View]

Figure 40 Update Thrusted Phase State and Derivatives

Update Thrusted Phase State and Derivatives is one of two functionalities that are provided by a Thrusted Phase. It is a function that uses phase state and computes state derivatives. This function uses number of functions from either parameter classes or models that were listed above to compute the state derivatives. For example it uses of one of Get Physicals function of Physicals classes to get the physical properties of the munition like reference mass or it uses one of Update Dynamics Models State and Derivatives function of Dynamics Model classes to get the Dynamics Model State Derivatives.

TSONT captures these dependencies among the functions on trajectory simulation domain by means of the dependsOn property acting on all functions. As an example, the definition of Update Thrusted Phase State and Derivatives is presented above in Figure 40.

Having discussed some functions and their dependencies, we will proceed with presenting models. Among Trajectory Simulation Models, the Body Fixed Six DOF Dynamics Model from Dynamics Model hierarchy will be discussed in detail to present our approach to the development of TSONT. Four different properties act on this class as restrictions. It should have a coordinate system, which is Body Coordinate System as its name indicates. It should have states and state derivatives. These states and state derivatives are parts of Phase State and State Derivatives which depicts the instantaneous system behavior. Its state is called Body Fixed Six DOF Dynamics Model State and its state derivatives are called Body Fixed Six DOF Dynamics Model State Derivatives. And the last restriction that applies is its service to the simulation. It means the way it is used in the execution of trajectory simulation. Dynamics Models are used to compute systems dynamics state derivatives by using the state. Then as the time passes numeric solver calculates the next time step's state by using these state derivatives. So, Body Fixed Six DOF Dynamics Model updates body fixed six DOF dynamics model state and derivatives. Figure 41 is the definition of Body Fixed Six DOF Dynamics Model in TSONT.


Body_Fixed_Six_DOF_Dynamics_Model (instance of owl:Class)

CLASS EDITOR

For Class: Body_Fixed_Six_DOF_Dynamics_Model (instance of owl:Class)

Name | SameAs | DifferentFrom

Body_Fixed_Six_DOF_Dynamics_Model

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

```
▼ hasCoordinateSystem    (allValuesFrom Body_Coordinate_System)
      Body_Coordinate_System
▼ hasState    (allValuesFrom Body_Fixed_Six_DOF_Dynamics_Model_State, allValuesFrom Dynamics_Model_State)
      Body_Fixed_Six_DOF_Dynamics_Model_State
      Dynamics_Model_State                                                     [from Dynamics_Model]
▼ hasStateDerivatives    (allValuesFrom Body_Fixed_Six_DOF_Dynamics_Model_State_Derivatives, allValuesFrom Dynamics_Model_State_Derivatives)
      Body_Fixed_Six_DOF_Dynamics_Model_State_Derivatives
      Dynamics_Model_State_Derivatives                                         [from Dynamics_Model]
▼ servesUpdateDynamicModelStateandDerivatives    (allValuesFrom Update_Body_Fixed_Six_DOF_Dynamics_Model_State_and_Derivatives, allValuesFrom
      Update_Body_Fixed_Six_DOF_Dynamics_Model_State_and_Derivatives
      Update_Rigid_Body_Dynamics_State_and_Derivatives                         [from Rigid_Body_Dynamics_Model]
```

Defining Classes

Disjoints

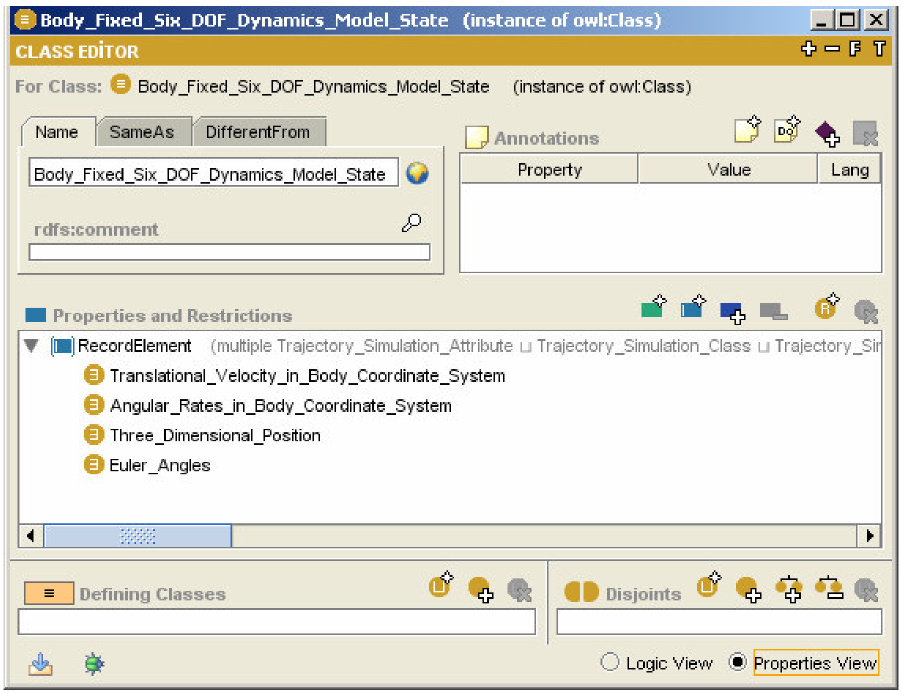
○ Logic View ● Properties View

Figure 41 Body Fixed Six DOF Dynamics Model

If we look at the Body Fixed Six DOF Dynamics Model State, it is defined as a kind of Trajectory Simulation Record composed of:

- Three dimensional translational velocity in body coordinate system
- Angular rates in body coordinate system
- Three dimensional position in earth coordinate system
- Euler angles

TSONT definition of Body Fixed Six DOF Dynamics Model State is presented below in Figure 42.


Body_Fixed_Six_DOF_Dynamics_Model_State (instance of owl:Class)

CLASS EDITOR

For Class: Body_Fixed_Six_DOF_Dynamics_Model_State (instance of owl:Class)

Name | SameAs | DifferentFrom

Body_Fixed_Six_DOF_Dynamics_Model_State

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

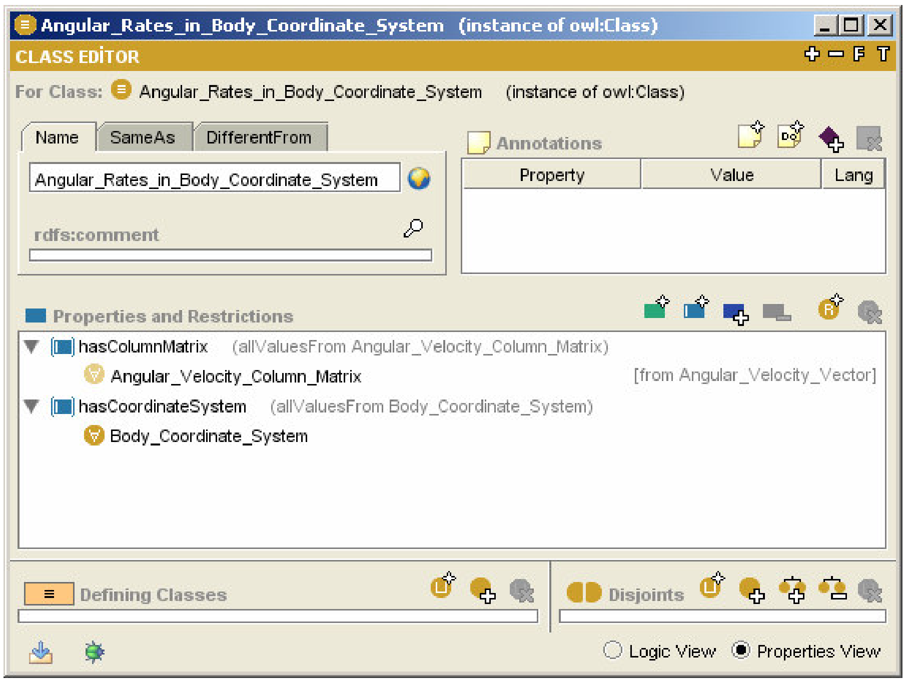
RecordElement (multiple Trajectory_Simulation_Attribute ⊔ Trajectory_Simulation_Class ⊔ Trajectory_Sir
- Translational_Velocity_in_Body_Coordinate_System
- Angular_Rates_in_Body_Coordinate_System
- Three_Dimensional_Position
- Euler_Angles

Defining Classes

Disjoints

Logic View [✓ Properties View]

Figure 42 Body Fixed Six DOF Dynamics Model State


Angular_Rates_in_Body_Coordinate_System (instance of owl:Class)

CLASS EDITOR

For Class: Angular_Rates_in_Body_Coordinate_System (instance of owl:Class)

Name | SameAs | DifferentFrom

Angular_Rates_in_Body_Coordinate_System

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

hasColumnMatrix (allValuesFrom Angular_Velocity_Column_Matrix)
- Angular_Velocity_Column_Matrix [from Angular_Velocity_Vector]

hasCoordinateSystem (allValuesFrom Body_Coordinate_System)
- Body_Coordinate_System

Defining Classes

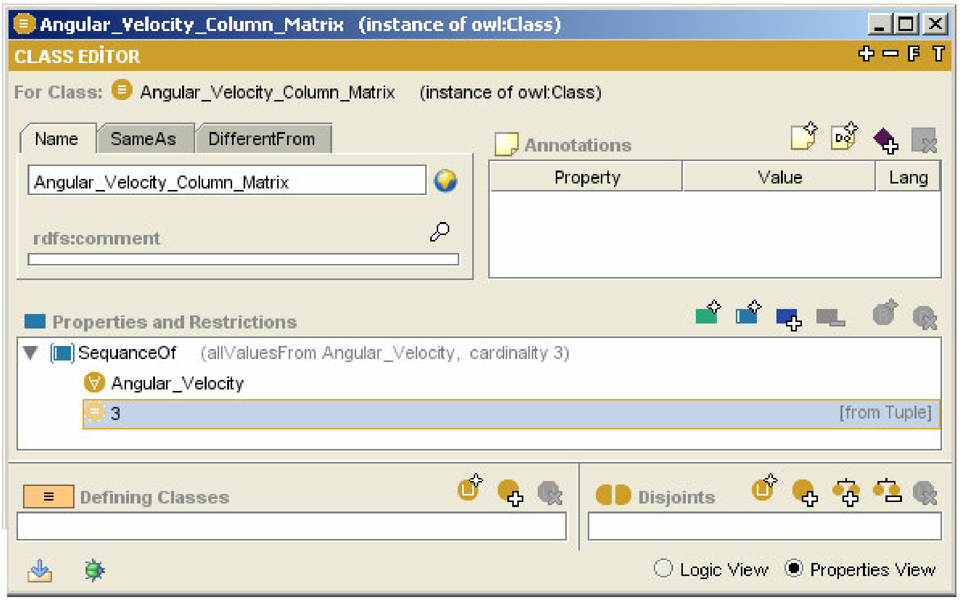
Disjoints

Logic View [✓ Properties View]

Figure 43 Angular Rates in Body Coordinate System

These records are vectors and Vectoral Quantities also have a definition in TSONT. For example Angular Rates in Body Coordinate System is defined in TSONT as depicted above in Figure 43.

As other vectoral quantities, Angular Rates in Body Coordinate System is defined with its coordinate system and its column matrix. Its coordinate system is Body Coordinate System and its column matrix is called Angular Velocity Column Matrix which has a definition in TSONT as given in Figure 44.


Angular_Velocity_Column_Matrix (instance of owl:Class)

CLASS EDITOR

For Class: Angular_Velocity_Column_Matrix (instance of owl:Class)

Name: Angular_Velocity_Column_Matrix

rdfs:comment

Property	Value	Lang

Properties and Restrictions

```
SequenceOf    (allValuesFrom Angular_Velocity, cardinality 3)
    Angular_Velocity
    3                                              [from Tuple]
```

Defining Classes

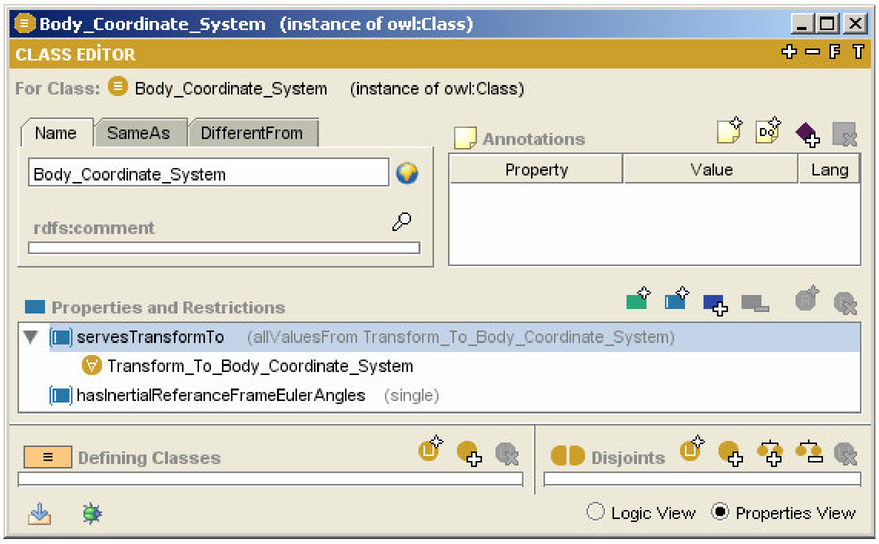
Disjoints

Logic View [✓ Properties View]

Figure 44 Angular Velocity Column Matrix

Angular Velocity Column Matrix is a type of Tuple. It is a sequence of a kind of Scalar Quantity which is Angular Velocity.

Body Coordinate System is one of the four Coordinate Systems mentioned in TSONT. As other coordinate systems, Body Coordinate System is also defined by its orientation with respect to inertial reference frame of the trajectory simulation. It also serves a functionality to transform any vector defined in any coordinate system to itself. Below is the representation of Body Coordinate System in TSONT.


Body_Coordinate_System (instance of owl:Class)

CLASS EDITOR

For Class: Body_Coordinate_System (instance of owl:Class)

Name: Body_Coordinate_System

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

servesTransformTo (allValuesFrom Transform_To_Body_Coordinate_System)
    Transform_To_Body_Coordinate_System
hasInertialReferanceFrameEulerAngles (single)

Defining Classes

Disjoints

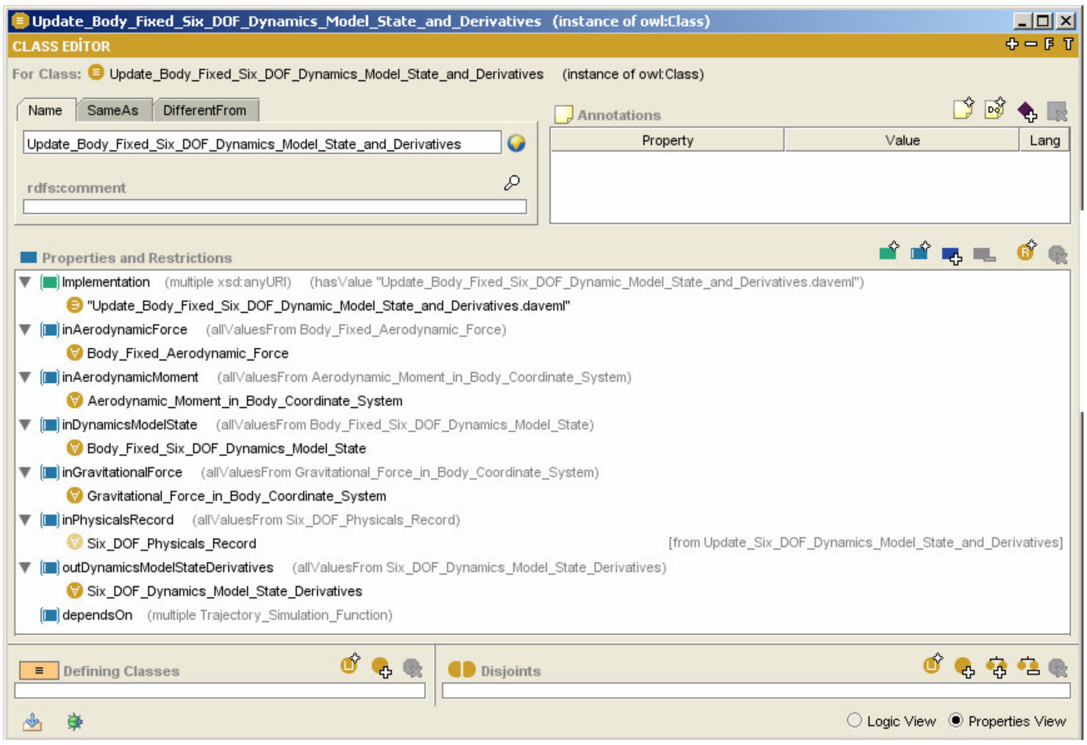
Logic View [✓ Properties View]

Figure 45 Body Coordinate System

If we have a look at Body Fixed Six DOF Dynamics Model State Derivatives which is another property of Body Fixed Six DOF Dynamics Model, we will figure out that it is composed of following items.

- Three dimensional translational acceleration in body coordinate system,
- Angular acceleration in body coordinate system,
- Euler angle rates
- Three dimensional translational velocity in earth coordinate system.

Dynamics Models are used to compute the dynamics of the munition at any time during flight. The implementation of this expression in a continuous simulation domain is to compute the state derivatives which will then used to compute the state of the next time step. So, dynamics model computes the angular and translational accelerations using the instantaneous forces and moments. Then this acceleration is integrated to compute the position and the orientation of the munition. Dynamics Models serves a functionality called Update Dynamics Model State and Derivatives to accomplish this task. As one will Body Fixed Six DOF Dynamics Model that we keep on discussing serves Update Body Fixed Six DOF Dynamics Model State and Derivatives functionality in this respect.



```
Update_Body_Fixed_Six_DOF_Dynamics_Model_State_and_Derivatives  (instance of owl:Class)
CLASS EDITOR
For Class: Update_Body_Fixed_Six_DOF_Dynamics_Model_State_and_Derivatives   (instance of owl:Class)

Name | SameAs | DifferentFrom                              Annotations
Update_Body_Fixed_Six_DOF_Dynamics_Model_State_and_Derivatives    Property | Value | Lang
rdfs:comment

Properties and Restrictions
▼ Implementation   (multiple xsd:anyURI)   (hasValue "Update_Body_Fixed_Six_DOF_Dynamic_Model_State_and_Derivatives.daveml")
      "Update_Body_Fixed_Six_DOF_Dynamic_Model_State_and_Derivatives.daveml"
▼ inAerodynamicForce   (allValuesFrom Body_Fixed_Aerodynamic_Force)
      Body_Fixed_Aerodynamic_Force
▼ inAerodynamicMoment   (allValuesFrom Aerodynamic_Moment_in_Body_Coordinate_System)
      Aerodynamic_Moment_in_Body_Coordinate_System
▼ inDynamicsModelState   (allValuesFrom Body_Fixed_Six_DOF_Dynamics_Model_State)
      Body_Fixed_Six_DOF_Dynamics_Model_State
▼ inGravitationalForce   (allValuesFrom Gravitational_Force_in_Body_Coordinate_System)
      Gravitational_Force_in_Body_Coordinate_System
▼ inPhysicalsRecord   (allValuesFrom Six_DOF_Physicals_Record)
      Six_DOF_Physicals_Record                    [from Update_Six_DOF_Dynamics_Model_State_and_Derivatives]
▼ outDynamicsModelStateDerivatives   (allValuesFrom Six_DOF_Dynamics_Model_State_Derivatives)
      Six_DOF_Dynamics_Model_State_Derivatives
  dependsOn   (multiple Trajectory_Simulation_Function)

Defining Classes                                  Disjoints
                                         ○ Logic View   ● Properties View
```

Figure 46 Update Body Fixed 6 DOF Dynamics Model State and Derivatives

TSONT models all functions in the same manner. It captures the implementation details, in other words the algorithms of the functions using Implementation property. Implementation is a data type property which points to a universal resource identifier to refer a DAVE-ML file. Then the restrictions starting with “in” refer to the input parameters of the function and those starting with “out” refer to the outputs of the process carried out by this functionality. The dependsOn restriction captures the dependencies among functions in trajectory simulation domain. One will figure out that this schema also applies to Update Body Fixed Dynamics Model State and Derivatives function whose definition is presented above in Figure 46.

For the implementation details of Update Body Fixed Dynamics Model State and Derivatives, TSONT refers to a DAVE-ML file. As an example, consider the mathematical model in Update Body Fixed Dynamics Model State and Derivatives for one of the translational accelerations in body coordinate system:

$$\dot{u} = \frac{F_x}{m} - qw + vr \qquad \text{Eq. 1}$$

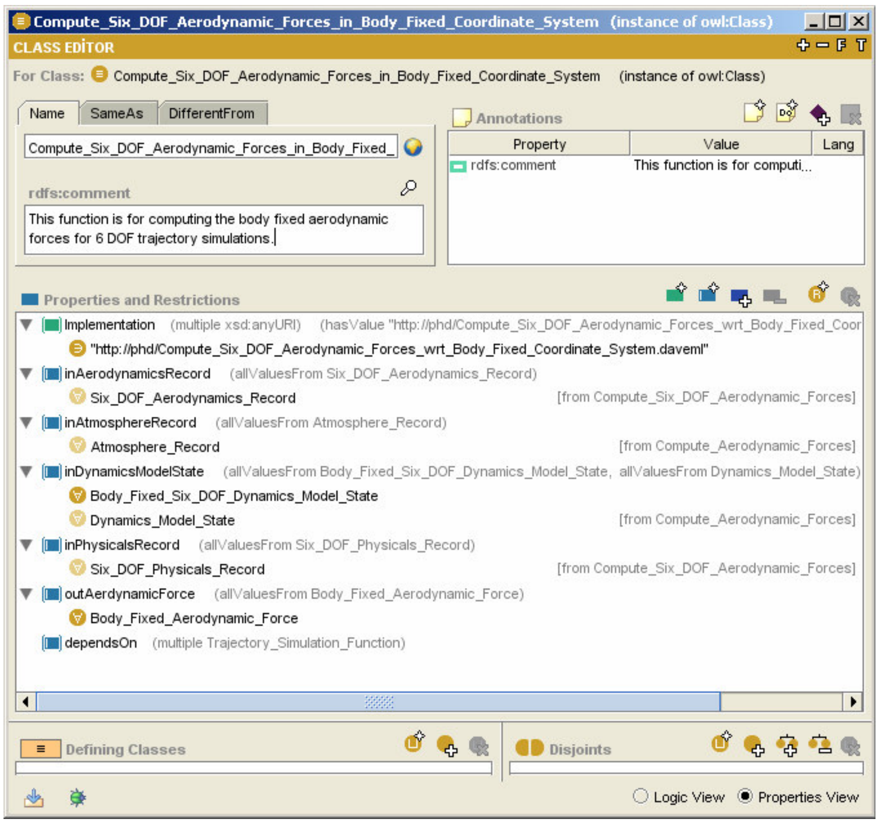
While it should be noted that the full contents of the DAVE-ML file for Update Body Fixed Dynamics Model State and Derivatives is given in APPENDIX C, in this file the above equation is represented as:

```
<variableDef name="udot" varID="udot" units="m/s2">
   <description> Body fixed tranlational acceleration in X </description>
   <calculation>
      <math xmlns='http://www.w3.org/1998/Math/MathML'>
         <apply>
            <eq/>
            <ci>udot</ci>
            <apply>
               <plus/>
               <apply>
                  <times/>
                  <apply>
                     <plus/>
                     <ci>FAX</ci>
                     <ci>FGX</ci>
```

```
                    </apply>
                    <apply>
                        <power/>
                        <ci>mass</ci>
                        <cn type='integer'>-1</cn>
                    </apply>
                </apply>
                <apply>
                    <times/>
                    <ci>r</ci>
                    <ci>v</ci>
                </apply>
                <apply>
                    <times/>
                    <cn type='integer'>-1</cn>
                    <apply>
                        <times/>
                        <ci>q</ci>
                        <ci>w</ci>
                    </apply>
                </apply>
            </apply>
        </apply>
    </math>
  </calculation>
  <isOutput/>
</variableDef>
```

More DAVE-ML samples are given in APPENDIX N.

Parameters classes, as mentioned earlier, are used to supply the required properties of the simulated system to the related models. The way the Parameter classes are modeled and how they relate with Model classes will be presented over an example.


Compute_Six_DOF_Aerodynamic_Forces_in_Body_Fixed_Coordinate_System (instance of owl:Class)

CLASS EDITOR

For Class: Compute_Six_DOF_Aerodynamic_Forces_in_Body_Fixed_Coordinate_System (instance of owl:Class)

Name | SameAs | DifferentFrom

Compute_Six_DOF_Aerodynamic_Forces_in_Body_Fixed_

rdfs:comment

This function is for computing the body fixed aerodynamic forces for 6 DOF trajectory simulations.

Annotations

Property	Value	Lang
rdfs:comment	This function is for computi...	

Properties and Restrictions

```
▼ Implementation   (multiple xsd:anyURI)   (hasValue "http://phd/Compute_Six_DOF_Aerodynamic_Forces_wrt_Body_Fixed_Coor
     "http://phd/Compute_Six_DOF_Aerodynamic_Forces_wrt_Body_Fixed_Coordinate_System.daveml"
▼ inAerodynamicsRecord   (allValuesFrom Six_DOF_Aerodynamics_Record)
     Six_DOF_Aerodynamics_Record                        [from Compute_Six_DOF_Aerodynamic_Forces]
▼ inAtmosphereRecord   (allValuesFrom Atmosphere_Record)
     Atmosphere_Record                                  [from Compute_Aerodynamic_Forces]
▼ inDynamicsModelState   (allValuesFrom Body_Fixed_Six_DOF_Dynamics_Model_State,  allValuesFrom Dynamics_Model_State)
     Body_Fixed_Six_DOF_Dynamics_Model_State
     Dynamics_Model_State                               [from Compute_Aerodynamic_Forces]
▼ inPhysicalsRecord   (allValuesFrom Six_DOF_Physicals_Record)
     Six_DOF_Physicals_Record                           [from Compute_Six_DOF_Aerodynamic_Forces]
▼ outAerdynamicForce   (allValuesFrom Body_Fixed_Aerodynamic_Force)
     Body_Fixed_Aerodynamic_Force
  dependsOn   (multiple Trajectory_Simulation_Function)
```

Defining Classes

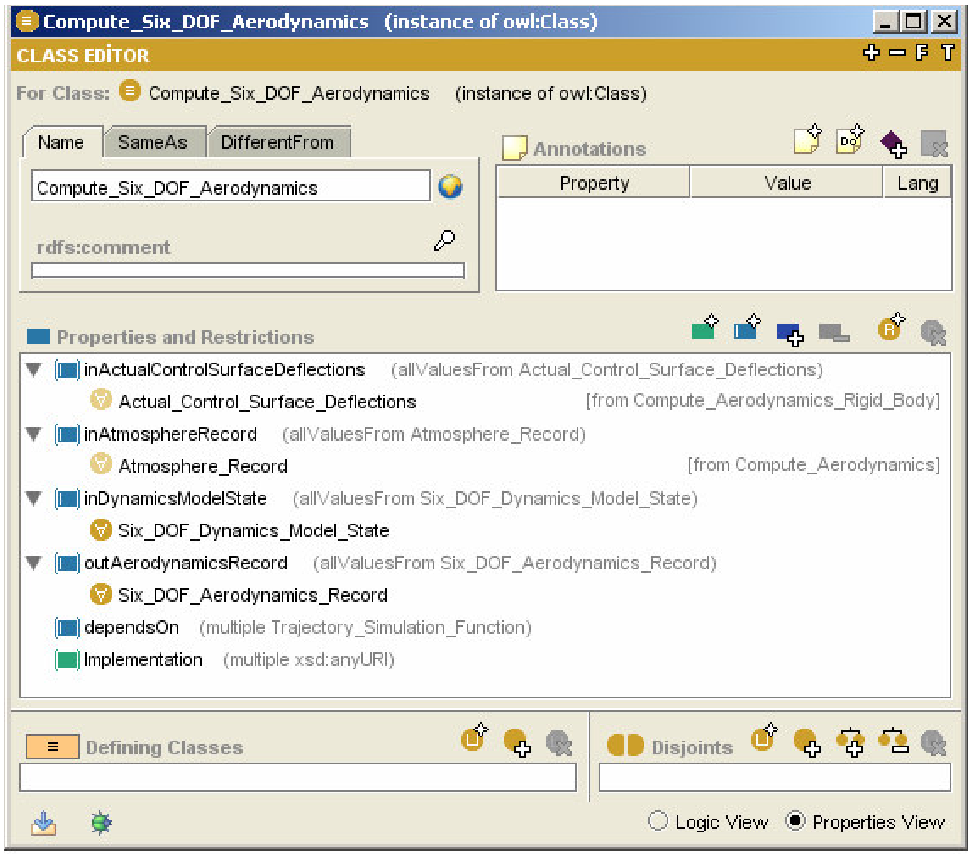
Disjoints

○ Logic View ● Properties View

Figure 47 Compute Six DOF Aerodynamics Forces in Body Fixed Coordinate System

Aerodynamics classes are responsible for computing the aerodynamic coefficients which will be used then by Aerodynamics Models to compute the aerodynamic forces and moments. Aerodynamics Force required above by Update Body Fixed Dynamics Model State and Derivatives functionality is computed by Body Fixed Six DOF Aerodynamics Model. It serves a functionality called Compute Six DOF Aerodynamics in Fixed Coordinate System which requires Six DOF Aerodynamics Record as an input. The definition of Compute Six DOF Aerodynamics Forces in Body Fixed Coordinate System is given above in Figure 47. Six DOF Aerodynamics Record is provided by Six DOF Aerodynamics which is a Parameter class. Six DOF Aerodynamics serves a functionality called Compute Six DOF

Aerodynamics. The definition of Compute Six DOF Aerodynamics is depicted below in Figure 48.


Compute_Six_DOF_Aerodynamics (instance of owl:Class)

CLASS EDITOR

For Class: Compute_Six_DOF_Aerodynamics (instance of owl:Class)

Name: Compute_Six_DOF_Aerodynamics

rdfs:comment

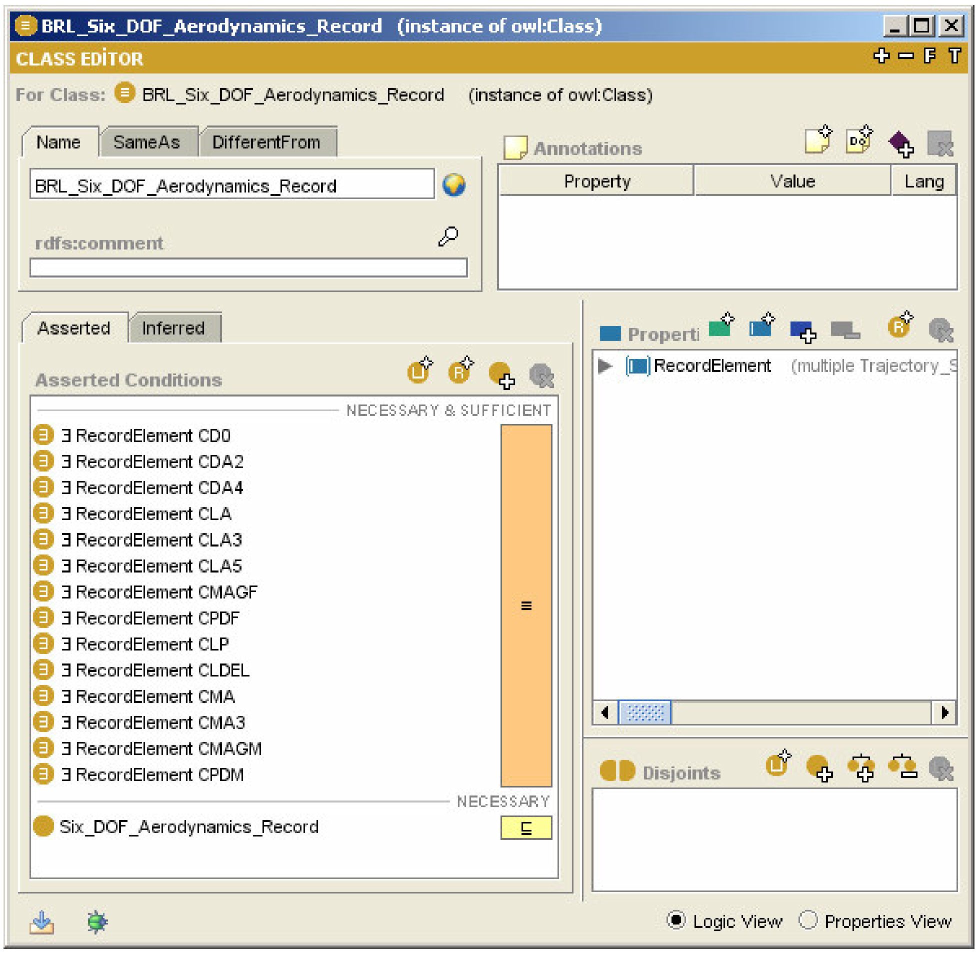
Properties and Restrictions

```
▼ inActualControlSurfaceDeflections   (allValuesFrom Actual_Control_Surface_Deflections)
      Actual_Control_Surface_Deflections          [from Compute_Aerodynamics_Rigid_Body]
▼ inAtmosphereRecord   (allValuesFrom Atmosphere_Record)
      Atmosphere_Record                           [from Compute_Aerodynamics]
▼ inDynamicsModelState   (allValuesFrom Six_DOF_Dynamics_Model_State)
      Six_DOF_Dynamics_Model_State
▼ outAerodynamicsRecord   (allValuesFrom Six_DOF_Aerodynamics_Record)
      Six_DOF_Aerodynamics_Record
  dependsOn   (multiple Trajectory_Simulation_Function)
  Implementation   (multiple xsd:anyURI)
```

Figure 48 Compute Six DOF Aerodynamics

As the above figure represents, Six DOF Aerodynamics Record is the output of Compute Six DOF Aerodynamics. It is Trajectory Simulation Composite Data. Three different ways to represent aerodynamic coefficients for a six degrees of freedom trajectory simulation are captured in TSONT as subclasses of Six DOF Aerodynamics Record. Those representations are Ballistic, Ballistic Research Lab (BRL) and National Advisory Committee for Aeronautics (NACA) representations [17]. Each representation refers to a record definition in TSONT. The definition

BRL Six DOF Aerodynamics Record is presented below in Figure 49 as an example of three.


BRL_Six_DOF_Aerodynamics_Record (instance of owl:Class)

CLASS EDITOR

For Class: BRL_Six_DOF_Aerodynamics_Record (instance of owl:Class)

Name | SameAs | DifferentFrom

BRL_Six_DOF_Aerodynamics_Record

rdfs:comment

Annotations

Property	Value	Lang

Asserted | Inferred

Asserted Conditions

NECESSARY & SUFFICIENT

∃ RecordElement CD0
∃ RecordElement CDA2
∃ RecordElement CDA4
∃ RecordElement CLA
∃ RecordElement CLA3
∃ RecordElement CLA5
∃ RecordElement CMAGF
∃ RecordElement CPDF
∃ RecordElement CLP
∃ RecordElement CLDEL
∃ RecordElement CMA
∃ RecordElement CMA3
∃ RecordElement CMAGM
∃ RecordElement CPDM

NECESSARY

Six_DOF_Aerodynamics_Record ⊑

Properti

RecordElement (multiple Trajectory_S

Disjoints

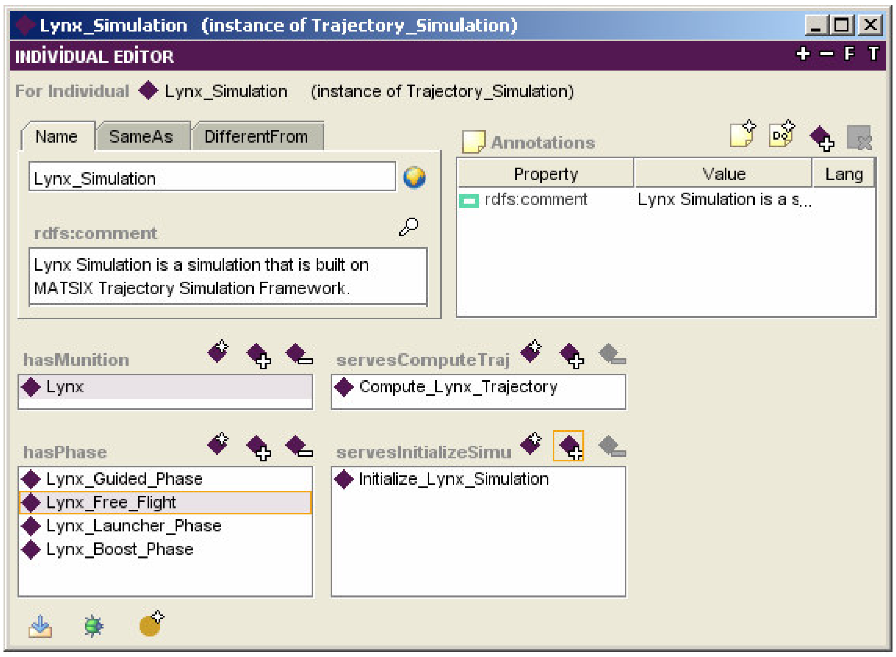
● Logic View ○ Properties View

Figure 49 BRL Six DOF Aerodynamics Record

4.5 TSONT Individuals

The OWL classes of TSONT create a base on which each and every application that is developed using this ontology based reuse infrastructure, is built on. The specific requirements of each application are planned to be added to the ontology as individuals. The domain structure and constraints modeled in TSONT define the relations among these individuals. As the ontology is used in new applications the individuals that were created by the previous projects will also be available for reuse. New domain structures and constraints will be able to be identified as new applications are developed. This commitment adds the ontology constructive nature. It will develop as it is used in the trajectory simulation projects.

Let us consider how TSONT is extended by individuals, as we define a new simulation, and how it guides the development of a trajectory simulation. The individuals of a guided rocket simulation, called Lynx, developed on MATLAB 6 DOF Trajectory Framework (MATSIX) will be presented as the case study. MATSIX was developed based on the design that was obtained by transforming TSONT classes. Then the Lynx Simulation was developed by framework completion referring to the TSONT individuals. This implementation will be discussed in detail in CHAPTER 5.



For Individual Lynx_Simulation (instance of Trajectory_Simulation)

Lynx_Simulation

rdfs:comment

Lynx Simulation is a simulation that is built on MATSIX Trajectory Simulation Framework.

Property	Value	Lang
rdfs:comment	Lynx Simulation is a s...	

hasMunition: Lynx

servesComputeTraj: Compute_Lynx_Trajectory

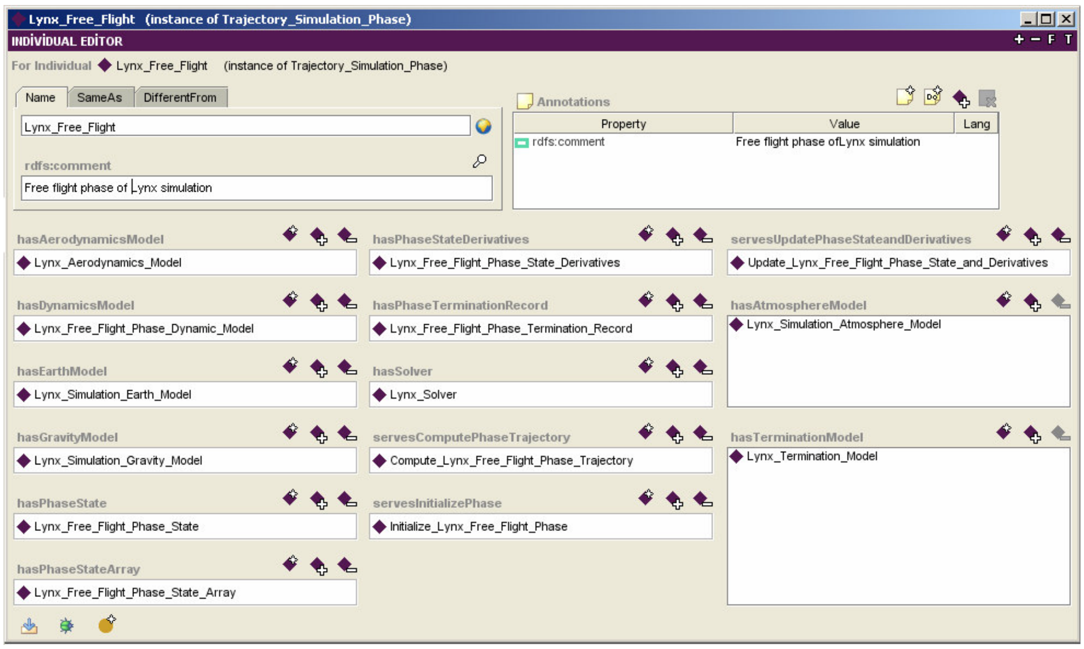
hasPhase: Lynx_Guided_Phase, Lynx_Free_Flight, Lynx_Launcher_Phase, Lynx_Boost_Phase

servesInitializeSimu: Initialize_Lynx_Simulation

Figure 50 Lynx Simulation

As we created a new simulation individual, TSONT asks to define the related properties of the simulation. A simulation as depicted above in Figure 50, is defined by its munition, trajectory, its phases and the functionalities provided.

We referred to individuals of Compute Trajectory and Initialize Simulation functions that are created to specify Lynx in TSONT. Lynx, as an individual of a MGR is set as the munition to be simulated. We defined four different phases for Lynx Simulation. The phases of Lynx Simulation are specified as the individuals of Phase, Guided Phase, Thrusted Phase and In Launcher Phase. If we consider Lynx Free Flight Phases, it is an individual of Phase class. It will be discussed to present how the phase individuals are constructed. To create a phase individual, TSONT forces one to specify the models, services and characteristic properties of that phase. Lynx Free Flight Phase is given in Figure 51.



Lynx_Free_Flight (instance of Trajectory_Simulation_Phase)

INDIVIDUAL EDITOR

For Individual ◆ Lynx_Free_Flight (instance of Trajectory_Simulation_Phase)

Name | SameAs | DifferentFrom

Lynx_Free_Flight

rdfs:comment

Free flight phase of Lynx simulation

Annotations

Property	Value	Lang
rdfs:comment	Free flight phase ofLynx simulation	

hasAerodynamicsModel: ◆ Lynx_Aerodynamics_Model

hasDynamicsModel: ◆ Lynx_Free_Flight_Phase_Dynamic_Model

hasEarthModel: ◆ Lynx_Simulation_Earth_Model

hasGravityModel: ◆ Lynx_Simulation_Gravity_Model

hasPhaseState: ◆ Lynx_Free_Flight_Phase_State

hasPhaseStateArray: ◆ Lynx_Free_Flight_Phase_State_Array

hasPhaseStateDerivatives: ◆ Lynx_Free_Flight_Phase_State_Derivatives

hasPhaseTerminationRecord: ◆ Lynx_Free_Flight_Phase_Termination_Record

hasSolver: ◆ Lynx_Solver

servesComputePhaseTrajectory: ◆ Compute_Lynx_Free_Flight_Phase_Trajectory

servesInitializePhase: ◆ Initialize_Lynx_Free_Flight_Phase

servesUpdatePhaseStateandDerivatives: ◆ Update_Lynx_Free_Flight_Phase_State_and_Derivatives

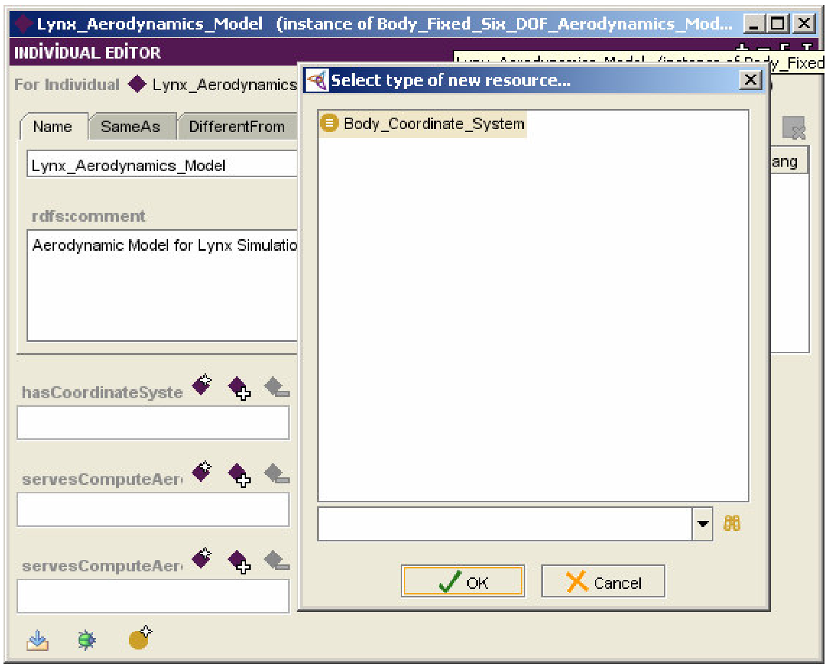
hasAtmosphereModel: ◆ Lynx_Simulation_Atmosphere_Model

hasTerminationModel: ◆ Lynx_Termination_Model

Figure 51 Lynx Free Flight Phase

As it can be followed from the below figure, individuals are defined for dynamics model, aerodynamics model, earth model, gravity model, atmosphere model and termination models. Lynx_Solver is defined as the individual of Runge Kutta 4 solver for the free flight phase of Lynx simulation. And lastly, two individuals are defined for two functions of Lynx_Simulation which refers to the specific implementations of these functions. One can use different algorithms to initialize a trajectory simulation or to compute it.

Lynx Aerodynamics Model will be discussed in this paragraph to create an understanding on how the individuals are used to link the specific trajectory simulation to the TSONT OWL classes. Lynx Aerodynamics Model was defined as an individual of a Body Fixed Six DOF Aerodynamics Model. As an individual, it conforms to all of the constraints of that applies on Body Fixed Six DOF Aerodynamics Model. As an ontology, TSONT, restricts its individuals to conform to their OWL Classes.


Lynx_Aerodynamics_Model (instance of Body_Fixed_Six_DOF_Aerodynamics_Mod...

INDIVIDUAL EDITOR

For Individual Lynx_Aerodynamics

Name: Lynx_Aerodynamics_Model

rdfs:comment: Aerodynamic Model for Lynx Simulatio

hasCoordinateSyste

servesComputeAer

servesComputeAer

Select type of new resource...

Body_Coordinate_System

Figure 52 Lynx Aerodynamics Model

If we look at Body Fixed Six DOF Aerodynamics Model, as depicted above in Figure 52, TSONT specifies how it shall be implemented. When user tries to specify which Coordinate System he wants to use when he computes aerodynamic force and moment, TSONT guides him that the coordinate system he shall use is Body Coordinate System.

As presented above, new simulation setups are defined by adding individuals for different needs. As the ontology, TSONT, is used in new trajectory simulation projects, the number of individuals will increase in number. And, the reuse of these previously captured individuals to define new simulation setups will also be an opportunity. This will enhance the evolution of TSONT as a trajectory simulation knowledge library.

The major motivation of this research is to guide trajectory simulation development efforts in all steps of trajectory simulation projects by providing formally defined reusable artifacts. One of the major motivations of building an ontology in this study was to provide a reusable domain model or trajectory simulation knowledge library to guide the trajectory simulation developer to construct a clear picture of domain concepts for a specific trajectory simulation project. TSONT, as presented above, can guide the trajectory simulation developer on how to construct and relate concepts in the trajectory simulation domain. This ability of TSONT seems to be fulfilling its commitment.

In this chapter, TSONT is presented. After introducing the top level TSONT entities, TSONT hierarchies, classes and individuals are introduced. It would be a good to remember that TSONT is the domain model for the trajectory simulation reuse infrastructure. Specifications of trajectory simulation reuse infrastructure for both object oriented and function oriented paradigms are constructed upon this domain model. Next chapter will introduce this infrastructure.

CHAPTER 5

INFRASTRUCTURE SPECIFICATION AND IMPLEMENTATIONS

In this chapter, object oriented and function oriented reuse infrastructures are built using the knowledge captured in TSONT, presented in the previous chapter. For object oriented reuse infrastructure, first object oriented application frameworks are introduced, and then platform independent trajectory simulation architecture is discussed. Two different case studies are presented for object oriented paradigm. Chapter is concluded with the function oriented reuse infrastructure and its case study.

5.1 Object Oriented Infrastructure Specification and Implementations

5.1.1 Object Oriented Application Frameworks

A common definition of a framework is the reusable design of all or a part of a software system that is accomplished by a set of abstract classes and a prescription of the way their instances interact. It can be regarded as the skeleton of an application that is to be developed in full by an application developer [63]. As a contemporary object oriented reuse technique, different from the earlier techniques based on class libraries, frameworks are targeted for particular application domains such as user interfaces or real-time avionics [64]. The history of framework literature goes back to 80's. Johnson and Foote introduced many basic concepts of application frameworks in their article published in 1988 [63].

Fayad and Schmidt list the benefits of object oriented application frameworks as modularity, reusability, extensibility and inversion of control that they provide to developers [64]. They explain these benefits as follows. Modularity is enhanced by encapsulating volatile implementation details behind stable interfaces. This gives the strength of modularity to frameworks by increasing the quality of product by localizing the impact of design and implementation changes which reduces the effort required to understand and maintain the existing code.

Stable interfaces, furthermore, enable reusability by defining generic components which can be reapplied to create new applications. Leveraging domain knowledge of experienced developers avoids re-creating and revalidating common solutions to reoccurring application requirements and software design challenges. This is the core essence of framework reuse to enhance programmer productivity, and further more quality, reliability and interoperability of software.

Extensibility is enabled in application frameworks by using hook methods. These hook methods decouple the interfaces and the behaviors of the application domain from variations required by a particular application.

Fayad and Schmidt [64] explain the basics as follows. Frameworks are characterized by their run-time architectures, which is known as "inversion of control". Inversion of control works as the framework dispatches related functionality during application processing steps to hook methods, which perform application-specific processing on the events.

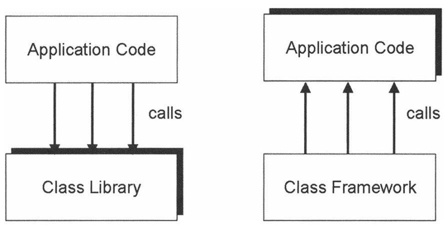


Figure 53 Control Inversion in Frameworks [97].

To summarize, a framework often consists of abstract classes, concrete classes, and predefined interaction among the classes throughout the framework. Developers can then build the application on top of the framework and reduce the development effort through reuse of code and designs provided in the framework. Below Figure 54 provides a high level overview on how an application framework relates to a domain application.

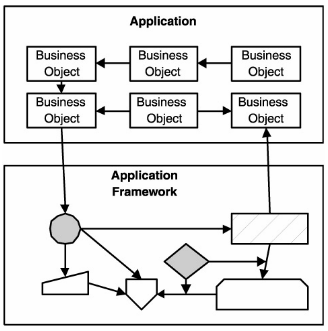


Figure 54 High-level Overview of the Relationship between an Application and the Application Framework [97].

Referring to Chen's book [97], the differences between a framework and a class library can be summarized as follows. A class library consists of a number of ready-to-use components that developers can use to build an application. But, developers must understand the relationships between various components and write process flow code to wire the required components together in the application. On the other hand, a framework encapsulates the control of such process flow by pre-wiring many of its components so that developers do not have to write code to control how the various components interact with each other. Figure 55 illustrates the difference between a class library and a framework.

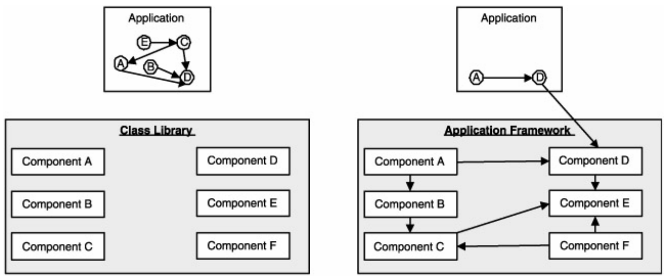


Figure 55 Comparison between a Class Library and an Application Framework [97].

Frameworks are extended using object oriented mechanisms either by inheriting from framework base classes or overriding pre-defined hook methods using patterns, such as the Template Method. The Template Method is presented in Figure 56.

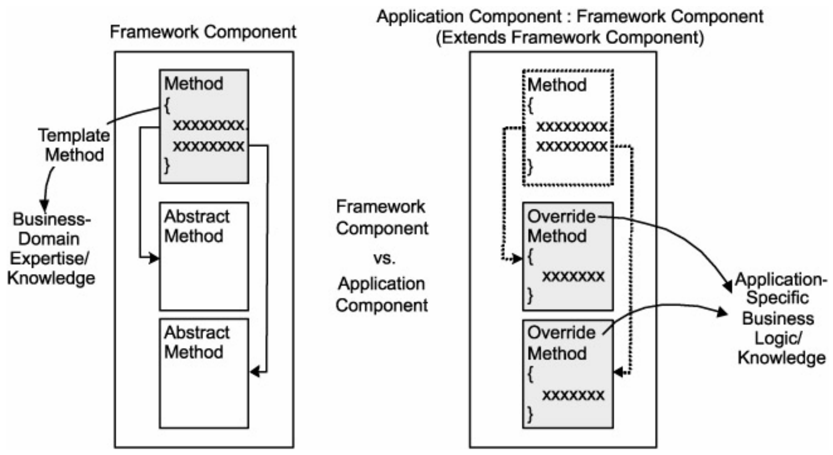

Framework Component
Application Component : Framework Component (Extends Framework Component)
Template Method
Business-Domain Expertise/ Knowledge
Method { xxxxxxxx xxxxxxxx }
Abstract Method
Abstract Method
Framework Component vs. Application Component
Method { xxxxxxxx xxxxxxxx }
Override Method { xxxxxxx }
Override Method { xxxxxxx }
Application-Specific Business Logic/ Knowledge

Figure 56 Template Method [97].

As stated by Akşit et al., although a large number of successful frameworks have been developed during last several years, designing a high quality framework is still an issue. Akşit proposes modeling domain knowledge as an essential step to develop a high quality framework [98]. But currently, there are no widely accepted standards for designing, implementing, documenting and adapting frameworks.

Chen discusses the economics of framework development [97]. He argues that developing an application framework is not an easy and inexpensive effort. In order to develop a highly usable and extensible framework, you need first to find individuals who are not only expert in the application domain, but also expert in software design and development. It is important that those who are developing the framework be competent in both domain knowledge and software development. Without domain expertise, one cannot create the domain-specific framework layers for developers. Without the technical expertise in software development, it will be hard transfer the concept of the framework from theory to the concrete framework code that developers can reuse and extend. How developers can benefit from the services and architecture provided in the framework must be determined by the framework designer. Chen says that some of the work involved in creating a framework can be regarded as abstract and heavily relies on assumptions about how developers will use the framework to build the application. So, it is said that it is difficult to get everything right on the first try, since the designer can only guess at how the final application will look and how it will be built to solve the domain problem. So as a result, in most of the cases, it takes a series of iterations to get the framework right for the applications that will be built on top of it. That makes framework development very much an evolving task, and it demands continual development and support efforts to ensure its relevance.

According to Robert and Johnson, a framework must embody a theory of the domain, and is always the result of domain analysis, whether the domain analysis is explicit and formal or implicit and informal [99]. Here in present research, we emphasize the use of domain engineering practices to construct a reuse infrastructure for trajectory simulation applications. We have an explicit and formal domain model in a form of an ontology. Frameworks, as stated in the previous paragraphs, have been standing as the most promising mechanism for enabling code and design reuse in last 20 years. So, we base our object oriented reuse scenario on framework concepts.

For ontology based object oriented reuse scenario, as given below in Figure 57, a Platform Independent Framework Architecture is proposed. It is an abstract design that is constructed with the guidance of TSONT. It is proposed that abstract design should not have any platform and problem set specific characteristics in order to enable design reuse for a large variety of applications on many different platforms.

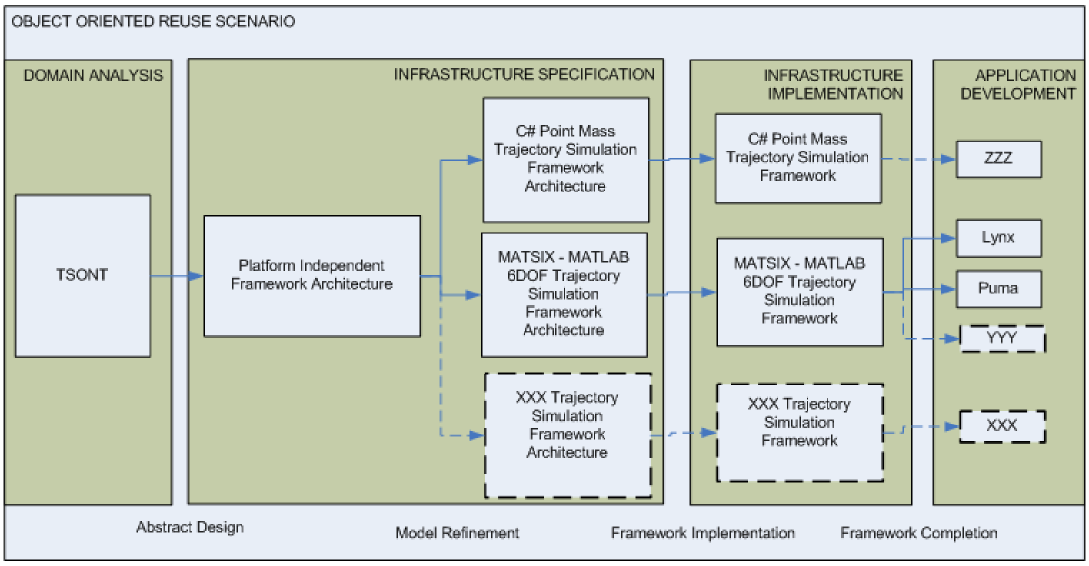

OBJECT ORIENTED REUSE SCENARIO
DOMAIN ANALYSIS
INFRASTRUCTURE SPECIFICATION
INFRASTRUCTURE IMPLEMENTATION
APPLICATION DEVELOPMENT
TSONT
Platform Independent Framework Architecture
C# Point Mass Trajectory Simulation Framework Architecture
MATSIX - MATLAB 6DOF Trajectory Simulation Framework Architecture
XXX Trajectory Simulation Framework Architecture
C# Point Mass Trajectory Simulation Framework
MATSIX - MATLAB 6DOF Trajectory Simulation Framework
XXX Trajectory Simulation Framework
ZZZ
Lynx
Puma
YYY
XXX
Abstract Design
Model Refinement
Framework Implementation
Framework Completion

Figure 57 Object Oriented Reuse Scenario

As the second step of the infrastructure specification activity of domain engineering, this platform independent framework architecture is proposed to be refined to specific platform and problem set. Frameworks, as their nature (they are implemented pieces of code) implies, are platform dependent. They either depend on a programming language like ADA95, a platform like MATLAB or another framework like .NET or EJB. Problem subsetting is also expected to figure in this step of the activity.

In our approach to trajectory simulation development with reuse, we find it favorable to construct new simulations by framework completion, provided, of course, a suitable framework is available. Otherwise one needs first to develop a framework, and then complete it for the particular application. This approach is expected to create a collection of related frameworks addressing different platforms and problem families. So as the framework is first developed in a context of a requirement set, it is obvious that it won't cover the whole domain. For example, it is expected to have body fixed 6 DOF framework with guidance and control models implemented or another framework still 6 DOF but this time it is earth fixed and without any guidance and control models. The former can be a result of requirement of a guided missile development project while the latter can be a requirement of base-bleed artillery projectile development project.

Here, in this research, as examples of object oriented frameworks, we worked on two different frameworks. The first one is 6 DOF framework, namely MATSIX, that was developed on MATLAB's object oriented facilities. Two different applications are built upon this framework. One is LYNX which is a surface to surface rocket simulation and the second one is PUMA which is a guided bomb simulation. There will be presented in the following sections.

The second framework is for point mass trajectory simulations. This one is not fully developed. The focus while developing this case study was the use of code generation facilities of computer aided software engineering tools in out reuse oriented trajectory simulation development methodology. The static structure captured in platform specific framework architecture is used to generate C# code for this framework. This activity will again be presented in the following sections.

As new requirements arise more frameworks can be designed and developed refining the platform independent framework architecture, and more applications can be developed by framework completion.

5.1.2 Platform Independent Trajectory Simulation Framework Architecture

Before going further, the first topic to be discussed is how to specify the platform independent trajectory simulation framework architecture. Typical definition of software architecture is the structure of the components of a program/system, their interrelationship, and the principals and the guidelines governing their design and evolution over time [100].

Referring the definition of software architecture given in previous paragraph, classes are regarded as the components of our object oriented framework. So the structure of the components of our architecture is proposed to be presented by class diagrams that are built depending on domain model which is ontology in our case.

Platform independent framework architecture, as the name implies, must be free of any platform dependencies. It will be the base for the specific framework architectures. So the below constraints apply to the class diagram that represents the platform independent framework architecture. Class diagrams are presented to be in a nature below:

Classes without

- export level (public, protected etc.)
- persistence
- representation details

Operations without

- their export control
- return types
- arguments

Attributes without

- type definition
- export control
- initial value
- containment

Associations only

- generalization
- aggregation/composition

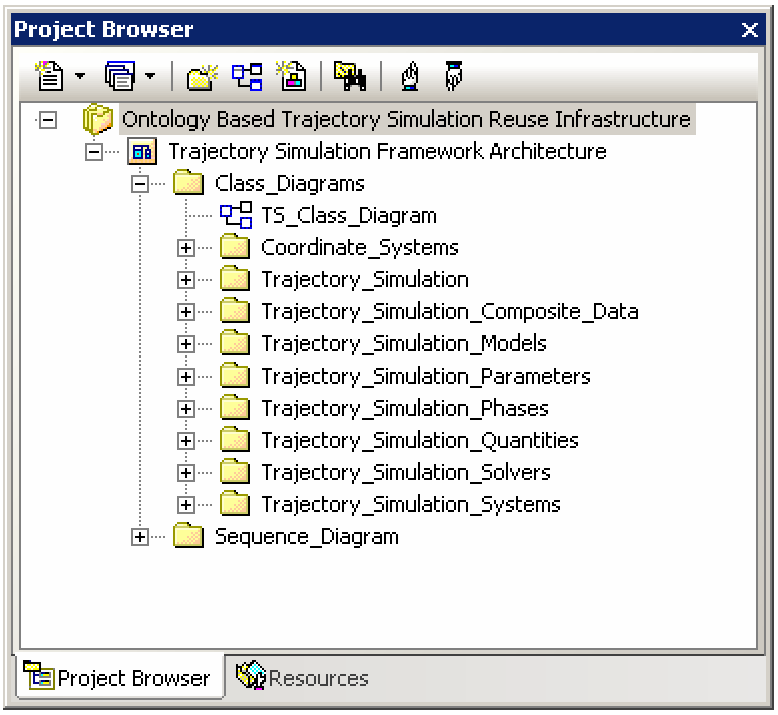
Software architecture, as given in the definition above should arrange the relations of components of the program besides the static structure given as class diagrams. The dynamic relations among the components for the framework are specified by using the UML sequence diagrams.

As the dependency hierarchy is captured in ontology, this information is used to build a top level sequence diagram that will lead the platform dependent developer in designing and developing his simulation.

Sequence diagram is a kind of interaction diagram that lays out the time ordering of messaging. Interaction diagrams in general show interaction, consisting of a set of objects and their relationship, including messages that may be dispatched among them [101].

Class diagrams and the sequence diagram in the present work have been developed by using Enterprise Architect Computer Aided Software Engineering tool of Sparx Systems Inc. [102]

UML class diagrams are constructed in such a way that there is a general view diagram which shows the relations among topmost classes in the generalization hierarchy. With this top level diagram, there are packages for each generalization hierarchy. Each package has another class diagram that shows the generalization hierarchy of the classes in that package. The project view of infrastructure specification is given below in Figure 58. The top level diagram is TS Class Diagram and the packages are the ones with folder icons.



Project Browser

- Ontology Based Trajectory Simulation Reuse Infrastructure
  - Trajectory Simulation Framework Architecture
    - Class_Diagrams
      - TS_Class_Diagram
      - Coordinate_Systems
      - Trajectory_Simulation
      - Trajectory_Simulation_Composite_Data
      - Trajectory_Simulation_Models
      - Trajectory_Simulation_Parameters
      - Trajectory_Simulation_Phases
      - Trajectory_Simulation_Quantities
      - Trajectory_Simulation_Solvers
      - Trajectory_Simulation_Systems
    - Sequence_Diagram

Project Browser | Resources

Figure 58 Trajectory Simulation Framework Architecture Project View

This recently mentioned top level class diagram of infrastructure specification which is given below in Figure 59, presents trajectory simulation developer which top level classes will exist in his simulation framework.

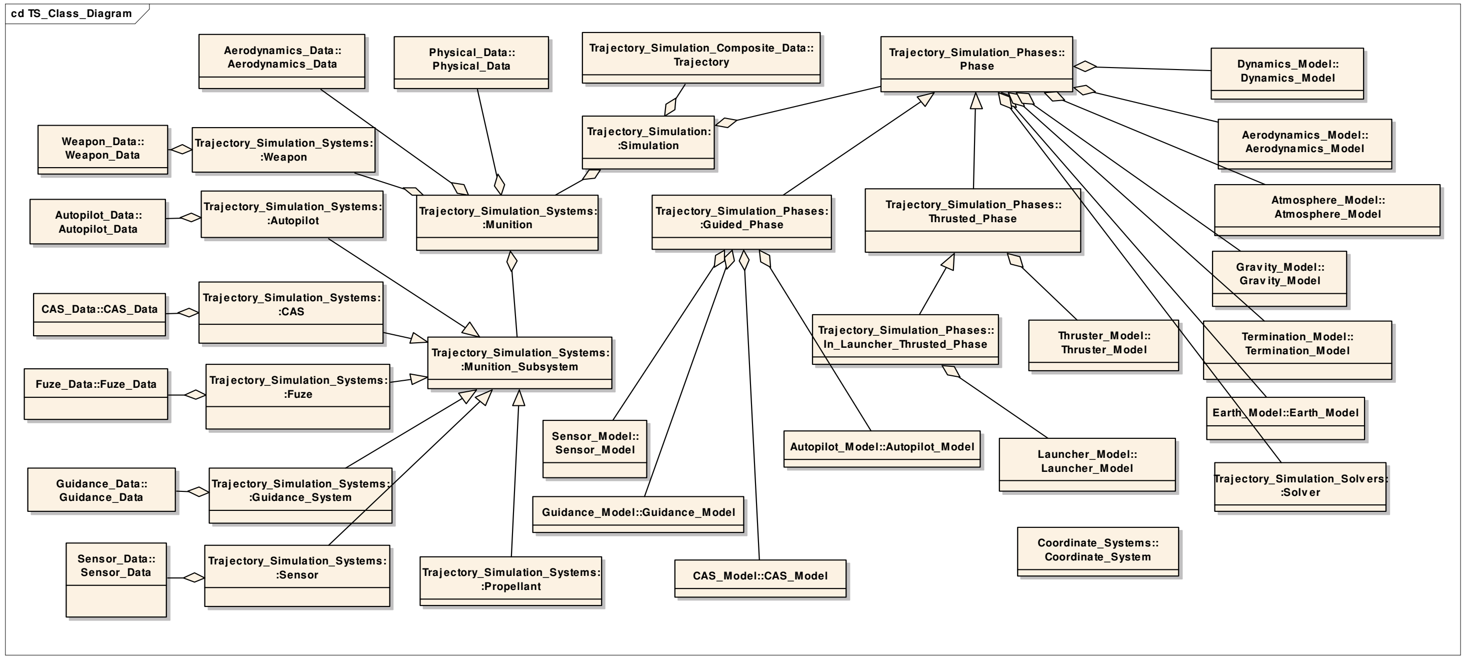
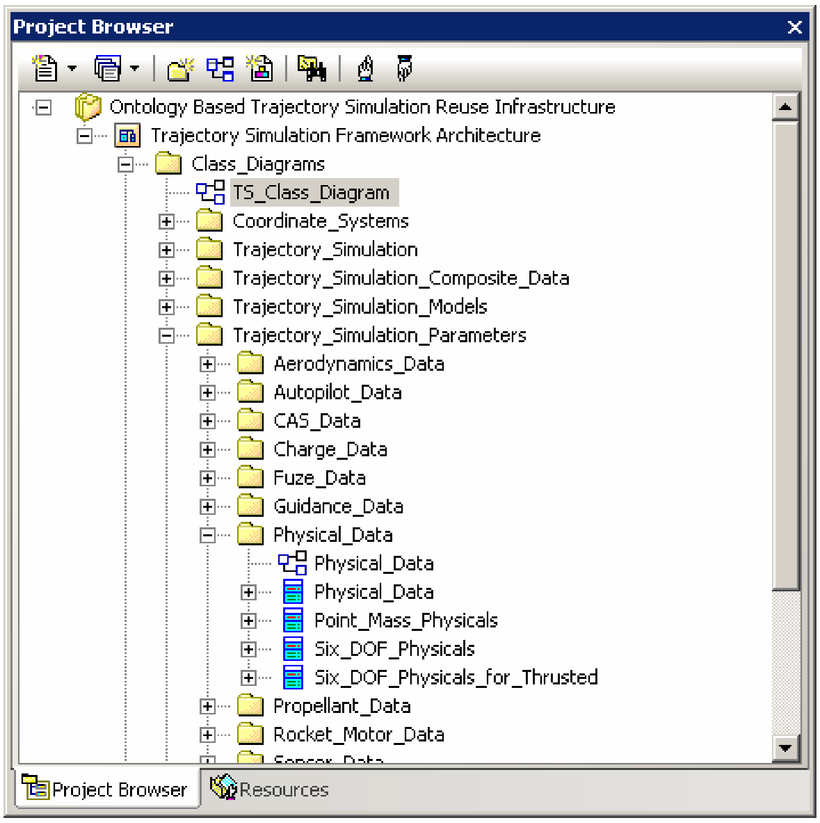

cd TS_Class_Diagram
Aerodynamics_Data::Aerodynamics_Data
Physical_Data::Physical_Data
Trajectory_Simulation_Composite_Data::Trajectory
Trajectory_Simulation_Phases::Phase
Dynamics_Model::Dynamics_Model
Weapon_Data::Weapon_Data
Trajectory_Simulation_Systems::Weapon
Trajectory_Simulation::Simulation
Aerodynamics_Model::Aerodynamics_Model
Autopilot_Data::Autopilot_Data
Trajectory_Simulation_Systems::Autopilot
Trajectory_Simulation_Systems::Munition
Trajectory_Simulation_Phases::Guided_Phase
Trajectory_Simulation_Phases::Thrusted_Phase
Atmosphere_Model::Atmosphere_Model
Gravity_Model::Gravity_Model
CAS_Data::CAS_Data
Trajectory_Simulation_Systems::CAS
Trajectory_Simulation_Phases::In_Launcher_Thrusted_Phase
Thruster_Model::Thruster_Model
Termination_Model::Termination_Model
Fuze_Data::Fuze_Data
Trajectory_Simulation_Systems::Fuze
Trajectory_Simulation_Systems::Munition_Subsystem
Earth_Model::Earth_Model
Sensor_Model::Sensor_Model
Autopilot_Model::Autopilot_Model
Launcher_Model::Launcher_Model
Trajectory_Simulation_Solvers::Solver
Guidance_Data::Guidance_Data
Trajectory_Simulation_Systems::Guidance_System
Guidance_Model::Guidance_Model
Coordinate_Systems::Coordinate_System
Sensor_Data::Sensor_Data
Trajectory_Simulation_Systems::Sensor
Trajectory_Simulation_Systems::Propellant
CAS_Model::CAS_Model

Figure 59 Top Level Class Diagram of Infrastructure Specification

Figure 59 also shows the aggregation/composition and generalization associations among these classes. Top level classes are presented in their related packages. In that related package's class diagram, it is presented with its full specification (complete with operations and attributes) and the other classes in that package inherit from it. The top level associations of classes are designed to be dispatched to the child classes of each package at run time by making use of the polymorphism capability of object oriented programming.

Two examples will be discussed here in this section to give the reader a clear idea about infrastructure specification. These examples will be Physical Data class hierarchy and Phase class hierarchy. Class diagrams of a couple of other packages will be given in APPENDIX D. The whole project is given in APPENDIX N.



```
Ontology Based Trajectory Simulation Reuse Infrastructure
  Trajectory Simulation Framework Architecture
    Class_Diagrams
      TS_Class_Diagram
      Coordinate_Systems
      Trajectory_Simulation
      Trajectory_Simulation_Composite_Data
      Trajectory_Simulation_Models
      Trajectory_Simulation_Parameters
        Aerodynamics_Data
        Autopilot_Data
        CAS_Data
        Charge_Data
        Fuze_Data
        Guidance_Data
        Physical_Data
          Physical_Data
          Physical_Data
          Point_Mass_Physicals
          Six_DOF_Physicals
          Six_DOF_Physicals_for_Thrusted
        Propellant_Data
        Rocket_Motor_Data
```

Figure 60 Physical Data Package

As given in Figure 60, Trajectory Simulation Parameters package has sub packages for each data class hierarchy. Physical Data is one of them.

Each package has a class diagram in the name of the package. Physical Data diagram which is given in Figure 61 is the class diagram for Physical Data package.

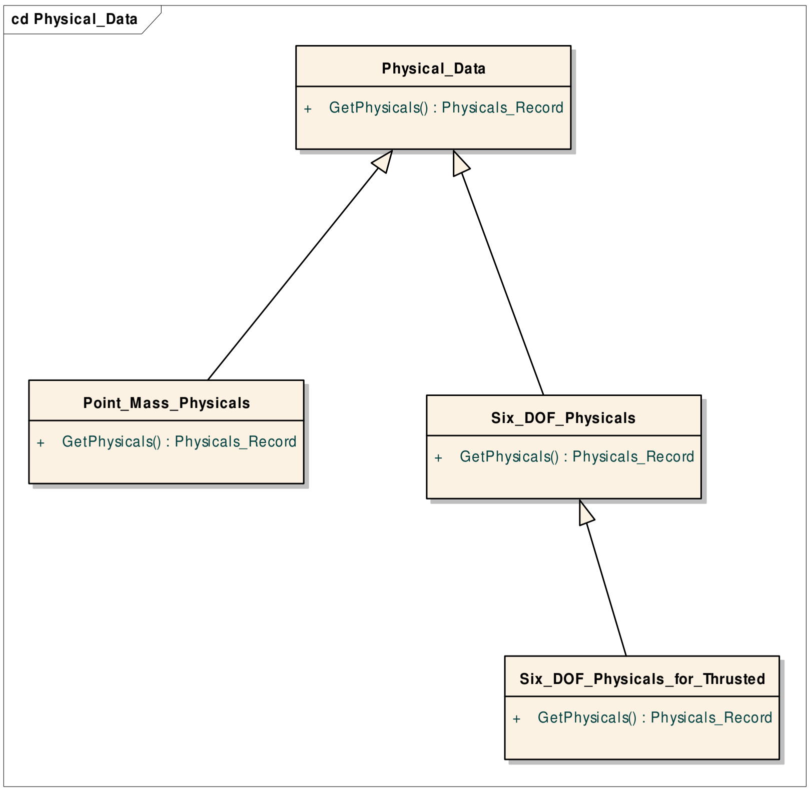

cd Physical_Data
Physical_Data
+ GetPhysicals() : Physicals_Record
Point_Mass_Physicals
+ GetPhysicals() : Physicals_Record
Six_DOF_Physicals
+ GetPhysicals() : Physicals_Record
Six_DOF_Physicals_for_Thrusted
+ GetPhysicals() : Physicals_Record

Figure 61 Physical Data Class Diagram

The idea presented in this diagram is that, presented schema will be the class hierarch for the framework one will develop using this platform independent design. For a framework that supports 6 DOF simulations, framework user, who actually is the application engineer, will use Six DOF Physical or Six DOF Physicals for Thrusted to inherit his own classes.

In the second example we will discuss the conformance of this representation to the form of platform independent framework architecture that was discussed at the beginning of this section. The representation of Phase hierarchy in the infrastructure specification is given below in Figure 62. Here in this diagram, there is no platform dependent information. Like, classes do not have implementation details, operations do not have specific parameters or export levels and attributes do not have any export control.

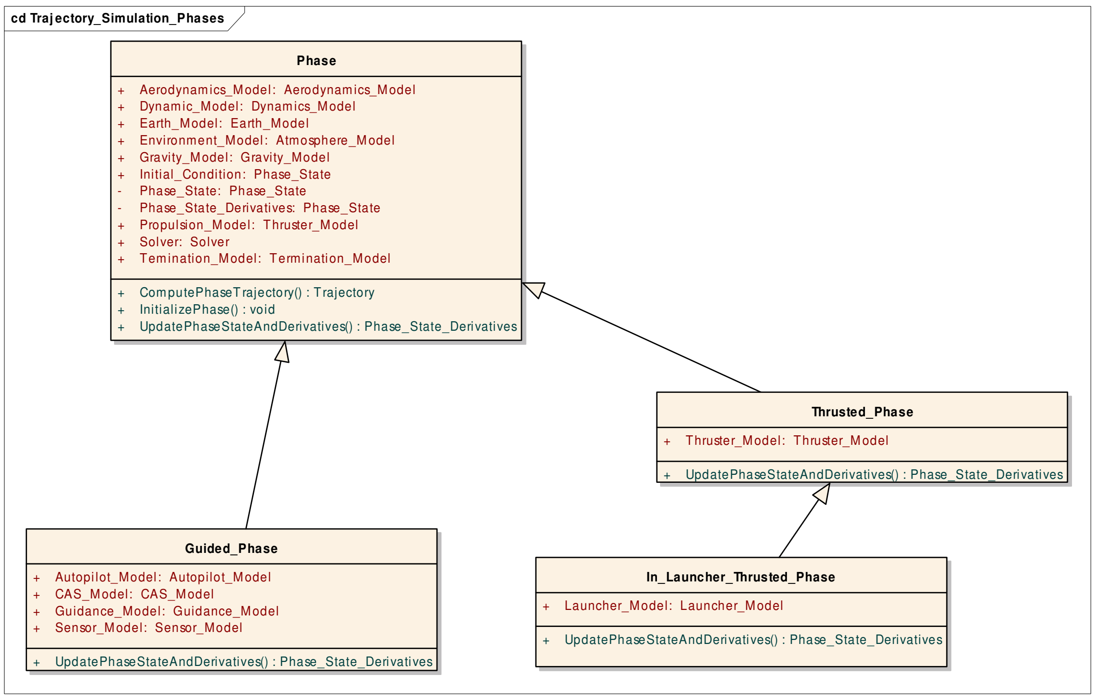

cd Trajectory_Simulation_Phases
Phase
+ Aerodynamics_Model: Aerodynamics_Model
+ Dynamic_Model: Dynamics_Model
+ Earth_Model: Earth_Model
+ Environment_Model: Atmosphere_Model
+ Gravity_Model: Gravity_Model
+ Initial_Condition: Phase_State
- Phase_State: Phase_State
- Phase_State_Derivatives: Phase_State
+ Propulsion_Model: Thruster_Model
+ Solver: Solver
+ Temination_Model: Termination_Model
+ ComputePhaseTrajectory() : Trajectory
+ InitializePhase() : void
+ UpdatePhaseStateAndDerivatives() : Phase_State_Derivatives
Thrusted_Phase
+ Thruster_Model: Thruster_Model
+ UpdatePhaseStateAndDerivatives() : Phase_State_Derivatives
Guided_Phase
+ Autopilot_Model: Autopilot_Model
+ CAS_Model: CAS_Model
+ Guidance_Model: Guidance_Model
+ Sensor_Model: Sensor_Model
+ UpdatePhaseStateAndDerivatives() : Phase_State_Derivatives
In_Launcher_Thrusted_Phase
+ Launcher_Model: Launcher_Model
+ UpdatePhaseStateAndDerivatives() : Phase_State_Derivatives

Figure 62 Phase Class Diagram

These diagrams in infrastructure specification are for guiding the developer to an abstraction schema that was captured in the ontology.

UML sequence diagram is added to platform independent framework architecture to give the user an idea about how the objects interact to accomplish a trajectory simulation. The whole sequence diagram is huge to be presented here in the body of the dissertation. A small portion of the sequence will be presented here and the whole sequence diagram is given in APPENDIX N.

The below Figure 63 illustrates that user starts a simulation by calling the service Compute Trajectory. This service starts a phase loop. For each phase, Compute Trajectory first initializes a phase by calling its Initialize Phase service. Then it calls Compute Phase Trajectory function to make phase compute its trajectory. Compute Phase Trajectory has a trajectory loop. For each time step Compute Phase Trajectory calls Integrate Step function of Solver. Solver integrates step by calling Update Phase State and Derivatives function of the Phase. To update phase state and derivatives, Phase first needs to access the data related to that state. It calls Compute Aerodynamics function of Aerodynamics Data to get the aerodynamic coefficients at that position and velocity of munition. This sequence then continues until whole trajectory of munition is computed.

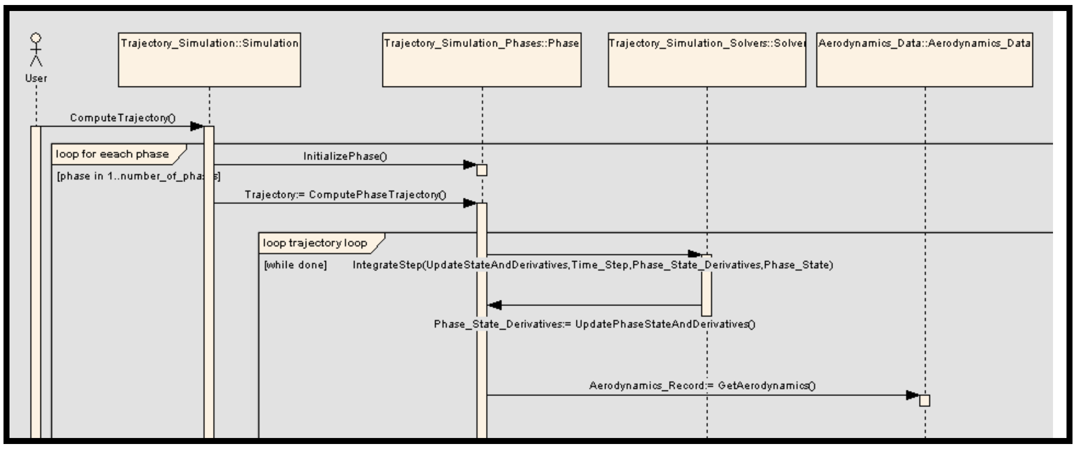

User
Trajectory_Simulation::Simulation
Trajectory_Simulation_Phases::Phase
Trajectory_Simulation_Solvers::Solver
Aerodynamics_Data::Aerodynamics_Data
ComputeTrajectory()
loop for eeach phase
[phase in 1..number_of_phases]
InitializePhase()
Trajectory:= ComputePhaseTrajectory()
loop trajectory loop
[while done]
IntegrateStep(UpdateStateAndDerivatives,Time_Step,Phase_State_Derivatives,Phase_State)
Phase_State_Derivatives:= UpdatePhaseStateAndDerivatives()
Aerodynamics_Record:= GetAerodynamics()

Figure 63 A Portion of Trajectory Simulation Sequence Diagram

Here in this section, one will figure out that the behavior of the simulation is implemented in the framework. So, all the simulations that will be developed with framework completion will have this behavior.

In the platform independent framework architecture, we expect the package/class names mostly match the class names in the ontology. The traceability of top level packages/classes to OWL classes in TSONT is given below in Table 1. A more comprehensive traceability table is presented at APPENDIX E. This will increase the understanding of the model to a developer who is familiar with the ontology and enhance the chance of tracing back to ontology.

Table 1 Class Diagram Packages – Ontology Traceability

Package Name	Entity In TSONT
Coordinate System	Coordinate System
Trajectory Simulation	Trajectory Simulation
Trajectory Simulation Composite Data	Trajectory Simulation Composite Data
Trajectory Simulation Models	Model
Trajectory Simulation Parameters	Parameter
Trajectory Simulation Phases	Trajectory Simulation Phase
Trajectory Simulation Quantities	Trajectory Simulation Quantity
Trajectory Simulation Solvers	Trajectory Simulation Solver
Trajectory Simulation Systems	Trajectory Simulation Object

5.1.3 6 DOF Trajectory Simulation Framework in MATLAB

5.1.3.1 MATSIX, An Introduction

MATSIX is a 6 DOF trajectory simulation which is developed by using MATLAB. This effort aims to present an example on implementation of the platform independent framework architecture that was presented in the previous section. So rather than developing new models, efforts from different researches are leveraged.

Mathematical models of this simulation framework are mostly based on the research that was carried out by Tiryaki [92]. Beyond most of the models, launcher dynamics is based on efforts of Mahmutyazıcıoğlu, atmosphere tables are from Public Domain Aeronautical Software web site and thrust model is based on STANAG. 4355 [93, 103 and 23].

This simulation framework supports trajectory simulations with:

- Standard atmosphere models with no wind profile
- Constant gravitational acceleration
- In launcher and 6 DOF dynamics models for munitions with a rotational symmetry
- Cubic, parabolic and 2D proportional navigation guidance models
- A specific autopilot model from [92].
- Canard control
- Round earth and flat earth
- Non rotating earth
- Solid rocket motors
- Launched from either a rocket launchers or an aircraft

While referring the related publications, implementations of significant models are discussed below in Notes on MATSIX Implementation section.

5.1.3.2 MATLAB Object Oriented Facilities

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation [104]. Object oriented programming among other approaches can be a way to develop software in MATLAB. Short advocacy of object oriented development in MATLAB product documentation says, when using well-designed classes, object-oriented programming can significantly increase code reuse and make your programs easier to maintain and extend.

Programming with classes and objects differs from ordinary structured programming in some important ways. These differences are listed in MATLAB product documentation [104] as follows:

Function and operator overloading. Existing MATLAB functions can be overridden. One should call such a function with user-defined object as an argument. Then MATLAB first checks to see if there is a method defined for the object's class. If there is, MATLAB calls it, rather than the normal MATLAB function.

Encapsulation of data and methods. One can not access object properties from the command line. They are only accessible within class methods.

Inheritance. One can create class hierarchies in MATLAB. The child class inherits data fields and methods from the parent. Single inheritance (A child class can inherit from one parent) or multiple inheritance (A child class can inherit from many parents) is supported. Using inheritance, sharing common parent functions and enforcing common behavior among all child classes is possible.

Aggregation. An object can contain other objects. This is called aggregation, which is also supported by MATLAB.

Although these definition seems to be very similar to common definition of object oriented there are some differences in the implementation of the methodology in MATLAB. These differences are listed in MATLAB product documentation [104] as follows:

- Method dispatching is not syntax based in MALAB. MATLAB uses the left-most object to select the method to call, when the argument list contains objects of equal precedence.

- There is no equivalent to a destructor method in MATLAB. One should use the *clear* function to remove an object from the workspace.

- MATLAB data types are constructed at runtime rather than compile time. To register an object as belonging to a class, one should call the class function.

- The inheritance relationship is established in the child class by creating the parent object, and then calling the class function in MATLAB.

- The child object contains a parent object in a property with the name of the parent class in MATLAB.

- There is no passing of variables by reference in MATLAB. One should pass back the updated object and use an assignment statement to write methods that update an object.

- There is no equivalent to an abstract class in MATLAB.

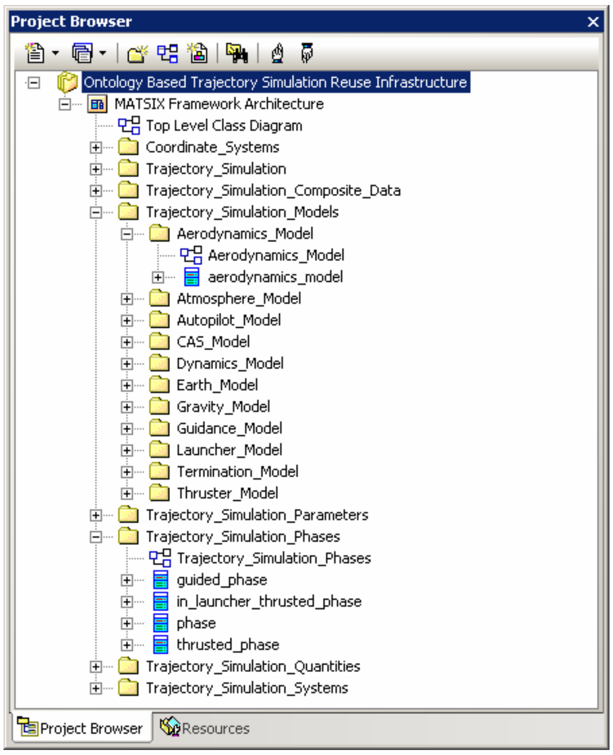
- There is no equivalent to the C++ scoping operator in MATLAB.

- There is no virtual inheritance or virtual base classes in MATLAB.

- There is no equivalent to C++ templates in MATLAB.

5.1.3.3 MATSIX Architecture

Framework architecture of MATSIX implementation is a part of infrastructure implementation. This design is based on the abstract design that is presented in the Platform Independent Framework Architecture. The platform specific constraints are applied on this abstract design and a detailed design is constructed. Meanwhile this MATSIX Architecture is subset of Platform Independent Trajectory Simulation Framework Architecture since it only concentrates on 6 DOF trajectory simulations and specifically the models presented in the previous section.



```
Ontology Based Trajectory Simulation Reuse Infrastructure
  MATSIX Framework Architecture
    Top Level Class Diagram
    Coordinate_Systems
    Trajectory_Simulation
    Trajectory_Simulation_Composite_Data
    Trajectory_Simulation_Models
      Aerodynamics_Model
        Aerodynamics_Model
        aerodynamics_model
      Atmosphere_Model
      Autopilot_Model
      CAS_Model
      Dynamics_Model
      Earth_Model
      Gravity_Model
      Guidance_Model
      Launcher_Model
      Termination_Model
      Thruster_Model
    Trajectory_Simulation_Parameters
    Trajectory_Simulation_Phases
      Trajectory_Simulation_Phases
      guided_phase
      in_launcher_thrusted_phase
      phase
      thrusted_phase
    Trajectory_Simulation_Quantities
    Trajectory_Simulation_Systems
```

Figure 64 MATSIX Project View

The project view is very similar to the one in platform independent framework architecture. There is again a top level class diagram and packages with distinct class diagrams inside. Figure 64 gives the MATSIX project view. Here in this section, only Aerodynamics Model and Trajectory Simulation Phases will be introduced. Couple of other class diagrams will be given in APPENDIX F. The whole MATSIX Project and implementation are given in APPENDIX N.

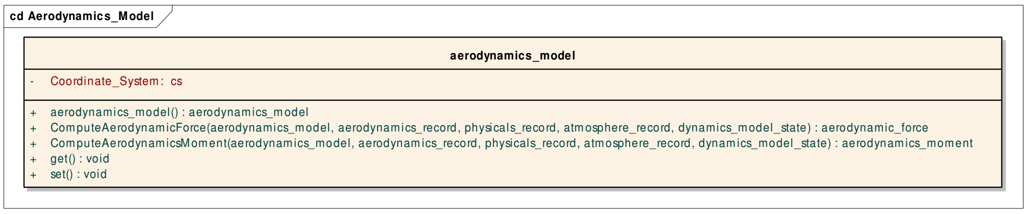

cd Aerodynamics_Model
aerodynamics_model
- Coordinate_System: cs
+ aerodynamics_model() : aerodynamics_model
+ ComputeAerodynamicForce(aerodynamics_model, aerodynamics_record, physicals_record, atmosphere_record, dynamics_model_state) : aerodynamic_force
+ ComputeAerodynamicsMoment(aerodynamics_model, aerodynamics_record, physicals_record, atmosphere_record, dynamics_model_state) : aerodynamics_moment
+ get() : void
+ set() : void

Figure 65 Aerodynamics Model of MATSIX Architecture

As depicted in Figure 65, there is only one Aerodynamics Model class in MATSIX Architecture. This is due to the fact that the platform, this time MATLAB, does not support abstract classes. So the hierarchy defined in abstract design hasn't been implemented, rather updated considering the constraints of the platform.

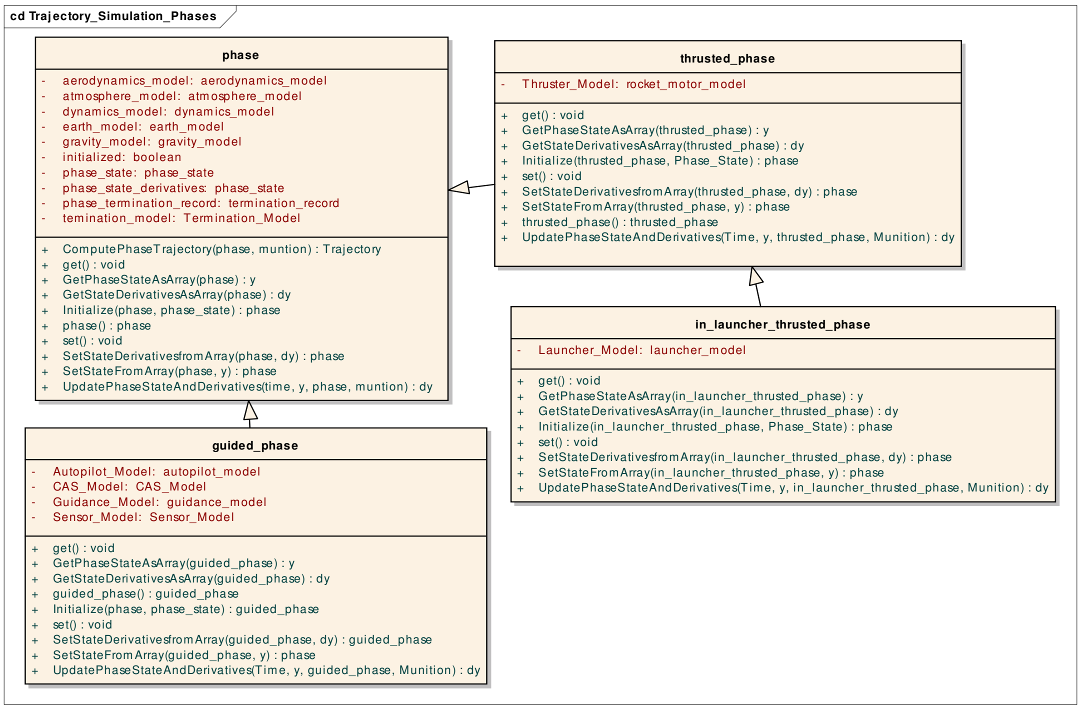

cd Trajectory_Simulation_Phases
phase
- aerodynamics_model: aerodynamics_model
- atmosphere_model: atmosphere_model
- dynamics_model: dynamics_model
- earth_model: earth_model
- gravity_model: gravity_model
- initialized: boolean
- phase_state: phase_state
- phase_state_derivatives: phase_state
- phase_termination_record: termination_record
- temination_model: Termination_Model
+ ComputePhaseTrajectory(phase, muntion) : Trajectory
+ get() : void
+ GetPhaseStateAsArray(phase) : y
+ GetStateDerivativesAsArray(phase) : dy
+ Initialize(phase, phase_state) : phase
+ phase() : phase
+ set() : void
+ SetStateDerivativesfromArray(phase, dy) : phase
+ SetStateFromArray(phase, y) : phase
+ UpdatePhaseStateAndDerivatives(time, y, phase, muntion) : dy
thrusted_phase
- Thruster_Model: rocket_motor_model
+ get() : void
+ GetPhaseStateAsArray(thrusted_phase) : y
+ GetStateDerivativesAsArray(thrusted_phase) : dy
+ Initialize(thrusted_phase, Phase_State) : phase
+ set() : void
+ SetStateDerivativesfromArray(thrusted_phase, dy) : phase
+ SetStateFromArray(thrusted_phase, y) : phase
+ thrusted_phase() : thrusted_phase
+ UpdatePhaseStateAndDerivatives(Time, y, thrusted_phase, Munition) : dy
in_launcher_thrusted_phase
- Launcher_Model: launcher_model
+ get() : void
+ GetPhaseStateAsArray(in_launcher_thrusted_phase) : y
+ GetStateDerivativesAsArray(in_launcher_thrusted_phase) : dy
+ Initialize(in_launcher_thrusted_phase, Phase_State) : phase
+ set() : void
+ SetStateDerivativesfromArray(in_launcher_thrusted_phase, dy) : phase
+ SetStateFromArray(in_launcher_thrusted_phase, y) : phase
+ UpdatePhaseStateAndDerivatives(Time, y, in_launcher_thrusted_phase, Munition) : dy
guided_phase
- Autopilot_Model: autopilot_model
- CAS_Model: CAS_Model
- Guidance_Model: guidance_model
- Sensor_Model: Sensor_Model
+ get() : void
+ GetPhaseStateAsArray(guided_phase) : y
+ GetStateDerivativesAsArray(guided_phase) : dy
+ guided_phase() : guided_phase
+ Initialize(phase, phase_state) : guided_phase
+ set() : void
+ SetStateDerivativesfromArray(guided_phase, dy) : guided_phase
+ SetStateFromArray(guided_phase, y) : phase
+ UpdatePhaseStateAndDerivatives(Time, y, guided_phase, Munition) : dy

Figure 66 Phases of MATSIX Architecture

When the diagram given in Figure 66 is considered, one will figure out that the classes involve implementation details like export levels, arguments and return types.

All attributes are private. This is another MATLAB constraint. So all classes have “get” and “set” functions to enable the class users manipulate the private attributes.

MATLAB requires all classes to have a constructer in the name of the class. As an example, guided phase has a guided phase service that returns a guided phase object.

There is no parameter passing by reference in MATLAB. As one of the consequence of this, services that change the state of an object have objects as one of its return value.

The design for MATSIX Architecture mentioned above was implemented. The framework involves 48 classes, which amount to 3579 SLOC (source lines of code).

5.1.3.4 Notes on MATSIX Implementation

This section presents some of the significant model implementations in MATSIX framework. Coordinate systems, dynamics model, aerodynamics model, guidance models, autopilot model and thruster model are mentioned below. Rather than the derivations of the equations, only the implemented results are given. Further details about the models can be found in the related references.

5.1.3.4.1 Coordinate Systems

MATSIX uses two different right handed and orthogonal coordinate frames. The first one is the earth fixed reference frame, $\Im_E(X,Y,Z)$. Its origin is fixed to the earth’s surface with its X axis pointing towards north, Y axis pointing towards east and Z axis pointing towards down to the centre of the earth. Non-rotating earth assumption is used. Hence the earth fixed reference frame is taken to be inertial.

The second reference frame $\Im_B(x, y, z)$ is the munition body frame. Its origin is at the centre of gravity of the munition. Its x axis points from the centre of gravity to the nose of the munition, y axis points towards the right of the munition looking from rear, z axis points down, forming a right handed orthogonal coordinate system.

Vector quantities are represented as a column vector with a coordinate system. The coordinate system transformations are carried out by using a transformation matrix.

$$\bar{r}^{(a)} = \hat{C}^{(a,b)}\bar{r}^{(b)} \qquad \text{Eq. 2}$$

Coordinate systems are defined by their Euler angles which are ψ, θ and ϕ (yaw angle, pitch angle and roll angle respectively) with respect to the inertial frame of the simulation. 3-2-1 rotated frame based Euler transformation sequence are used for rotational transformations. The transformation matrix from any (X) frame to the inertial frame (I) is obtained as:

$$\hat{C}^{(I,X)} = \begin{bmatrix} c\theta c\psi & s\phi s\theta c\psi - c\phi s\psi & c\phi s\theta c\psi + s\phi s\psi \\ c\theta s\psi & s\phi s\theta s\psi + c\phi c\psi & c\phi s\theta s\psi - s\phi c\psi \\ -s\theta & s\phi c\theta & c\phi c\theta \end{bmatrix} \qquad \text{Eq. 3}$$

where ‘c’ denotes the cosine and ‘s’ denotes the sine of the angle.

5.1.3.4.2 Dynamics Model

MATSIX dynamics model consists of equations of motion which relate the forces and moments which are being applied to the munition to the translational and rotational accelerations. Two different dynamics modeling is used in MATSIX. The first one models the equations of motion of a munition in launcher and the second one is the six degrees of freedom equations of motion of the munition in three dimensional space.

5.1.3.4.2.1 In Launcher Dynamics Model

The launcher is modeled as straight rail that reinforces no spin to the rocket. Launcher constraints the motion of the munition by forcing its elevation and azimuth during in launcher phase [93].

$$\dot{u} = \frac{F_x}{m} \qquad \text{Eq. 4}$$

$$\dot{v} = 0 \qquad \text{Eq. 5}$$

$$\dot{w} = 0 \qquad \text{Eq. 6}$$

where;

$$F_{x=}F_{ax} + F_{tx} + mg_x \qquad \text{Eq. 7}$$

and

$$\dot{p} = 0 \qquad \text{Eq. 8}$$

$$\dot{q} = 0 \qquad \text{Eq. 9}$$

$$\dot{r} = 0 \qquad \text{Eq. 10}$$

Using the body to earth transformation matrix $\hat{C}^{(E,B)}$, body frame translational velocity components can be related to earth frame velocity components as follows:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \hat{C}^{(E,B)} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \qquad \text{Eq. 11}$$

Here u, v, w are the translational velocity components in the munition body frame. X, Y, Z are the coordinates of the centre of gravity of the munition in the earth frame.

It is assumed that no angular rates can be introduced to the munition in launcher, so the rotational kinematic equations implemented are as follows:

$$\dot{\psi} = 0 \qquad \text{Eq. 12}$$

$$\dot{\theta} = 0 \qquad \text{Eq. 13}$$

$$\dot{\phi} = 0 \qquad \text{Eq. 14}$$

5.1.3.4.2.2 Six DOF Dynamics Model

Six degrees of freedom equations of motion are implemented as follows;

$$\dot{u} = \frac{F_x}{m} - qw + vr \qquad \text{Eq. 15}$$

$$\dot{v} = \frac{F_y}{m} - ur + pw \qquad \text{Eq. 16}$$

$$\dot{w} = \frac{F_z}{m} + uq - pv \qquad \text{Eq. 17}$$

where;

$$F_{x=}F_{ax} + F_{tx} + mg_x \qquad \text{Eq. 18}$$

$$F_{y=}F_{ay} + F_{ty} + mg_y \qquad \text{Eq. 19}$$

$$F_{z=}F_{az} + F_{tz} + mg_z \qquad \text{Eq. 20}$$

and

$$\dot{p} = L / I_x \qquad \text{Eq. 21}$$

$$\dot{q} = M / I_y + r.p.(I_y - I_x) / I_y \qquad \text{Eq. 22}$$

$$\dot{r} = N / I_y + p.q.(I_x - I_y) / I_y \qquad \text{Eq. 23}$$

where;

$$L = L_a + L_t \qquad \text{Eq. 24}$$

$$M = M_a + M_t \qquad \text{Eq. 25}$$

$$N = N_a + N_t \qquad \text{Eq. 26}$$

Here, note that $I_z = I_y$ due to rotational symmetry of the munition.

Body frame translational velocity components are related to earth frame velocity components as follows:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \hat{C}^{(E,B)} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \qquad \text{Eq. 27}$$

The rotational kinematics equations implemented are as follows:

$$\dot{\psi} = \left(q.\sin\phi + r.\cos\phi\right) / \cos\theta \qquad \text{Eq. 28}$$

$$\dot{\theta} = q.\cos\phi - r.\sin\phi \qquad \text{Eq. 29}$$

$$\dot{\phi} = p + \left(q.\sin\phi + r.\cos\phi\right).\tan\theta \qquad \text{Eq. 30}$$

5.1.3.4.3 Aerodynamics Model

Aerodynamic model of MATSIX concentrates on the determination of aerodynamic forces and moments acting on a munition which are produced by the relative motion of the munition with respect to the air and depend on the orientation of the munition with respect to the airflow.

The orientation angles are the angle of attack (α) and the sideslip angle (β). These angles are expressed as follows:

$$\alpha = \tan^{-1}\left(\frac{w}{u}\right) \qquad \text{Eq. 31}$$

$$\beta = \sin^{-1}\left(\frac{v}{V}\right) \qquad \text{Eq. 32}$$

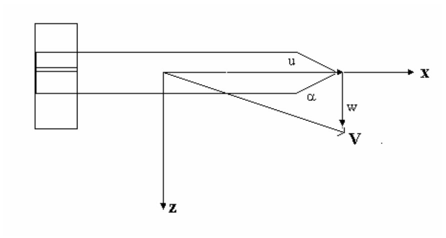


Figure 67 Munition Velocity Components - Side View

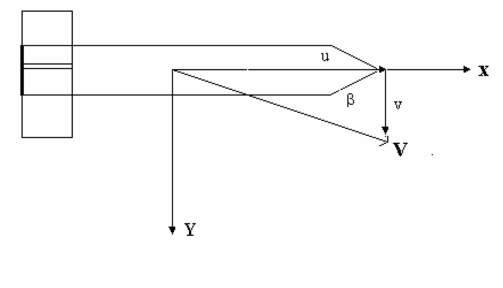


Figure 68 Munition Velocity Components - Top View

The aerodynamic forces and moments acting on the munition are expressed in column representation as:

$$\begin{bmatrix} F_{ax} \\ F_{ay} \\ F_{az} \end{bmatrix} = Q_d.A.\begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} \qquad \text{Eq. 33}$$

$$\begin{bmatrix} L_a \\ M_a \\ N_a \end{bmatrix} = Q_d.A.d.\begin{bmatrix} C_l \\ C_m \\ C_n \end{bmatrix} \qquad \text{Eq. 34}$$

Q_d is the free stream dynamic pressure. It is expressed as:

$$Q_d = \frac{1}{2}\rho V^2 \qquad \text{Eq. 35}$$

ρ is the air density. V is the velocity of the munition and implemented as following regarding the no wind case:

$$V = \sqrt{u^2 + v^2 + w^2} \qquad \text{Eq. 36}$$

"A" is the reference area. It is the maximum cross sectional area of the munition. "d", on the other hand, is the diameter of the munition.

5.1.3.4.4 Aerodynamics Parameters

Aerodynamic coefficients are defined as; C_x, axial force coefficient, C_y, side force coefficient, C_z, normal force coefficient, C_l, rolling moment coefficient, C_m, pitching moment coefficient and C_n, yawing moment coefficient.

These coefficients are expressed as a function of angle of attack, sideslip angle, control surface deflections (for guided munition), and Mach number. The effective control surface deflections are said to be $\delta_e, \delta_r, \delta_a$ for the pitch, yaw and roll planes respectively. Mach number is defined as:

$$M = \frac{V}{V_s} \qquad \text{Eq. 37}$$

Where V is the total velocity of the munition and V_s is the local speed of sound

Force and moment coefficients are implemented as following in MATSIX referring the assumptions from Tiryaki's work [92].

Force Coefficients:

$$C_x = C_{x0}(M) \qquad \text{Eq. 38}$$

$$C_y = C_{y_\beta}(M)\beta + C_{y_\delta}(M)\delta_r + C_{y_r}(M)r\frac{d}{2V} \qquad \text{Eq. 39}$$

$$C_z = C_{z_\alpha}(M)\alpha + C_{z_\delta}(M)\delta_e + C_{z_q}(M)q\frac{d}{2V} \qquad \text{Eq. 40}$$

Moment Coefficients:

$$C_l = C_{l_\delta}(M)\delta_a + C_{l_p}(M)p\frac{d}{2V} \qquad \text{Eq. 41}$$

$$C_m = C_{m_\alpha}(M)\alpha + C_{m_\delta}(M)\delta_e + C_{m_q}(M)q\frac{d}{2V} \qquad \text{Eq. 42}$$

$$C_n = C_{n_\beta}(M)\beta + C_{n_\delta}(M)\delta_r + C_{n_r}(M)r\frac{d}{2V} \qquad \text{Eq. 43}$$

Equalities due to rotational symmetry:

$$C_{z\alpha} = C_{y\beta} \qquad \text{Eq. 44}$$

$$C_{z\delta} = C_{y\delta} \qquad \text{Eq. 45}$$

$$C_{zq} = -C_{yr} \qquad \text{Eq. 46}$$

$$C_{m\alpha} = -C_{n\beta} \qquad \text{Eq. 47}$$

$$C_{m\delta} = -C_{n\delta} \qquad \text{Eq. 48}$$

$$C_{mq} = C_{nr} \qquad \text{Eq. 49}$$

5.1.3.4.5 Atmosphere Model

Table 2 ICAO Standard Atmosphere Table

Altiture	Temperature	Pressure	Density	Speed of Sound
km	K	N/sq.m	kg/cu.m	m/s
-2	301.2	1.28E+05	1.48E+00	347.9
0	288.1	1.01E+05	1.23E+00	340.3
2	275.2	7.95E+04	1.01E+00	332.5
4	262.2	6.17E+04	8.19E-01	324.6
6	249.2	4.72E+04	6.60E-01	316.5
8	236.2	3.57E+04	5.26E-01	308.1
10	223.3	2.65E+04	4.14E-01	299.5
12	216.6	1.94E+04	3.12E-01	295.1
14	216.6	1.42E+04	2.28E-01	295.1
16	216.6	1.04E+04	1.67E-01	295.1
18	216.6	7.57E+03	1.22E-01	295.1
20	216.6	5.53E+03	8.89E-02	295.1
22	218.6	4.05E+03	6.45E-02	296.4
24	220.6	2.97E+03	4.69E-02	297.7
26	222.5	2.19E+03	3.43E-02	299.1
28	224.5	1.62E+03	2.51E-02	300.4
30	226.5	1.20E+03	1.84E-02	301.7
32	228.5	8.89E+02	1.36E-02	303
34	233.7	6.63E+02	9.89E-03	306.5
36	239.3	4.99E+02	7.26E-03	310.1
38	244.8	3.77E+02	5.37E-03	313.7
40	250.4	2.87E+02	4.00E-03	317.2
42	255.9	2.20E+02	3.00E-03	320.7
44	261.4	1.70E+02	2.26E-03	324.1
46	266.9	1.31E+02	1.71E-03	327.5
48	270.6	1.02E+02	1.32E-03	329.8
50	270.6	7.98E+01	1.03E-03	329.8

Standard atmosphere model of MATSIX is implemented in this case study as a simple table look up. The values represent the ICAO standard atmosphere. The values listed above in Table 2 are obtained from the web site of Public Domain Aeronautical Software (PDAS) [103].

5.1.3.4.6 Guidance Model

5.1.3.4.6.1 Cubic Guidance

Cubic Guidance Law is one of three guidance laws implemented in MATSIX. In Cubic Guidance Law, flight path angle rate commands are computed in order to keep the munition on a cubic trajectory [92]. This trajectory is regarded to be tangent to the instantaneous munition velocity vector. It will have two end points, munition's centre of mass and the target point. One can use four conditions to express a cubic polynomial. In addition to satisfying three of them given above, one more condition can be defined on the cubic trajectory. This condition is taken to be the impact angle of the munition.

The mathematical formulation of cubic guidance law is expressed as follows:

The trajectories for $0 < \xi < x_f - x$ is expressed below as:

$$y(x+\xi) = A_y\xi^3 + B_y\xi^2 + C_y\xi + D_y \qquad \text{Eq. 50}$$

$$z(x+\xi) = A_z\xi^3 + B_z\xi^2 + C_z\xi + D_z \qquad \text{Eq. 51}$$

In order to find four unknowns which are the coefficients of the cubic polynomials, we need four conditions on these equations. These conditions will be taken as:

- Starting point condition
- Starting slope condition

- Hit point condition
- Hit slope condition

By using four conditions presented above, these coefficients are found to be;

$$S_{y_i} = \frac{\dot{Y}_i}{\dot{X}_i}$$ Eq. 52

$$S_{y_i} = \frac{\dot{Z}_i}{\dot{X}_i}$$ Eq. 53

$$D_y = y$$ Eq. 54

$$D_z = z$$ Eq. 55

$$C_z = S_{z_i}$$ Eq. 56

$$S_{y_f} = \tan \eta_f$$ Eq. 57

$$S_{z_f} = -\tan \gamma_f / \cos \eta_f$$ Eq. 58

$$B_y = -\frac{(S_{y_f} + 2S_{y_i})(x_f - x) + 3y}{(x_f - x)^2}$$ Eq. 59

$$B_z = -\frac{(S_{z_f} + 2S_{z_i})(x_f - x) + 3z}{(x_f - x)^2}$$ Eq. 60

Desired rates are

$$\dot{\eta}^* = 2B_y V \cos \gamma \cos^3 \eta$$ Eq. 61

$$\dot{\gamma}^* = -2B_z V \cos^3 \gamma \cos^2 \eta + 2C_z B_y V \cos^3 \gamma \cos^3 \eta \sin \eta$$ Eq. 62

where,

$\eta = \arctan(S_{y_i})$ Eq. 63

$\gamma = \arctan(-S_{z_i} \cos\eta)$ Eq. 64

5.1.3.4.6.2 Parabolic Guidance

Parabolic Guidance Law as the second guidance algorithm of MATSIX generates the necessary commands on either the rates of the flight path angles or the normal acceleration components that keep the munition on a parabolic trajectory [92]. This trajectory is kept tangent to the munition's current velocity vector and pass through the munition's centre of mass and the target point at all instants.

The trajectories for $0 < \xi < x_f - x$ is expressed below as:

$y(x+\xi) = C_y + B_y\xi - A_y\xi^2 / 2$ Eq. 65

$z(x+\xi) = C_z + B_z\xi - A_z\xi^2 / 2$ Eq. 66

Instantaneous parabolic trajectories are described by these equations. Three unknowns in these equations which are the coefficients of the second order polynomials are solved by three conditions which are:

- Starting point condition
- Starting slope condition
- Hit point condition

By using three conditions presented above, these coefficients are found to be;

$C_y = y$ Eq. 67

$C_z = z$ Eq. 68

$$B_y = \tan\eta \qquad \text{Eq. 69}$$

$$B_z = -\frac{\tan\gamma}{\cos\eta} \qquad \text{Eq. 70}$$

$$A_y = \frac{2(y + \tan\eta(x_f - x))}{(x_f - x)^2} \qquad \text{Eq. 71}$$

$$A_z = \frac{2(z - (\tan\gamma / \cos\eta)(x_f - x))}{(x_f - x)^2} \qquad \text{Eq. 72}$$

Desired rates are

$$\dot{\eta}^* = -A_y V \cos\gamma \cos^3\eta \qquad \text{Eq. 73}$$

$$\dot{\gamma}^* = (A_z \cos\gamma + A_y \sin\gamma \sin\eta) V (\cos\gamma \cos\eta)^2 \qquad \text{Eq. 74}$$

where,

$$\eta = \arctan(S_{y_i}) \qquad \text{Eq. 75}$$

$$\gamma = \arctan(-S_{z_i} \cos\eta) \qquad \text{Eq. 76}$$

5.1.3.4.6.3 Proportional Navigation in 2D

The last guidance law that we implemented in MATSIX is Proportional Navigation in 2D. Proportional navigation guidance as one of the first guidance laws developed for tactical missiles is popular by its simplicity, effectiveness and ease of implementation [15].

Proportional navigation guidance law generates acceleration command which is proportional to the line of sight rate and the closing velocity. Mathematically it can be expressed as [92]:

$$n_c = N \dot{\lambda} V_c$$ Eq. 77

where, N is a unitless effective navigation, $\dot{\lambda}$ is the line of sight rate, V_c is the closing velocity and n_c is the command acceleration.

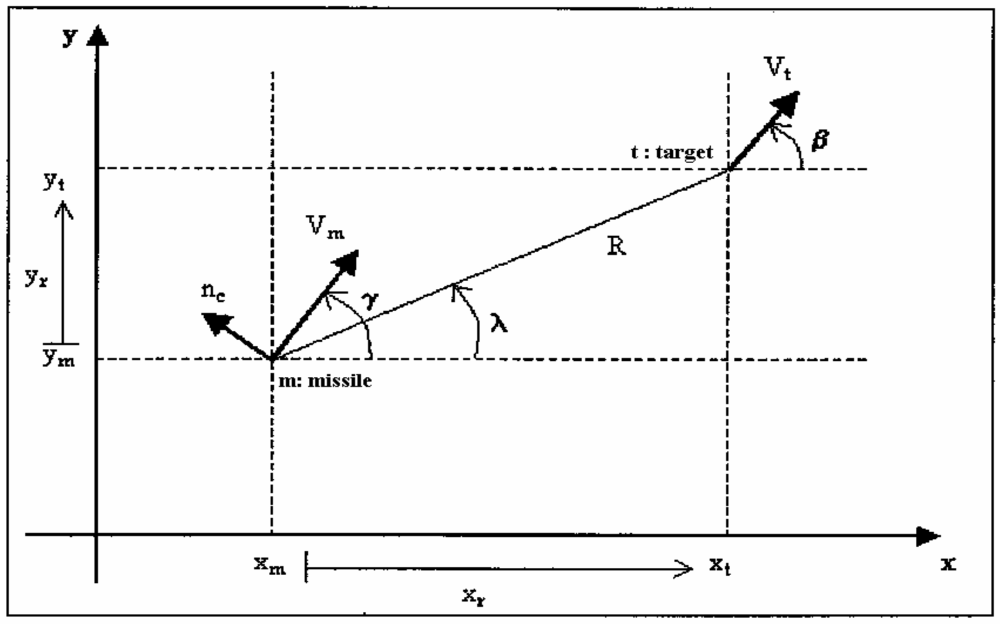


Figure 69 2-D Missile-Target Kinematics

$x_m, y_m, z_m, x_t, y_t, z_t$, $V_{mx}, V_{my}, V_{mz}, V_{tx}, V_{ty}, V_{tz}$ as basic missile and target parameters, are presented in Figure 69.

$$\dot{\lambda} = \frac{(x_t - x_m)(V_{ty} - V_{my}) - (y_t - y_m)(V_{tx} - V_{mx})}{(x_t - x_m)^2 + (y_t - y_m)^2}$$ Eq. 78

$$V_c = -\frac{(x_t - x_m)(V_{tx} - V_{mx}) + (y_t - y_m)(V_{ty} - V_{my})}{\sqrt{(x_t - x_m)^2 + (y_t - y_m)^2}}$$ Eq. 79

5.1.3.4.7 Autopilot Model

5.1.3.4.7.1 Pitch Autopilot

The aim of pitch autopilot model is to find the commanded fin deflections to stabilize the longitudinal dynamics and keep $\dot{\gamma}=\dot{\gamma}^*$, where $\dot{\gamma}^*$ is the command rate provided by the guidance law.

Pitch autopilot of MATSIX is implemented using equations work that follow [92]:

$$b_{\gamma\alpha} = -Q_d A C_{z_\alpha} / (Mass V_t) \qquad \text{Eq. 80}$$

$$b_{\gamma\delta} = Q_d A C_{z\delta} / (Mass V_t) \qquad \text{Eq. 81}$$

$$b_{q\alpha} = Q_d A d C_{m\alpha} / I_y \qquad \text{Eq. 82}$$

$$b_{q\delta} = -Q_d A d C_{m\delta} / I_y \qquad \text{Eq. 83}$$

$$B_{q\delta} = b_{q\delta}\, b_{\gamma\alpha} - b_{q\alpha}\, b_{\gamma\delta} \qquad \text{Eq. 84}$$

$$d_0 = \mu\omega_n^3 \qquad \text{Eq. 85}$$

$$d_1 = (1+2\xi\mu)\omega_n^2 \qquad \text{Eq. 86}$$

$$d_2 = (2\xi+\mu)\omega_n \qquad \text{Eq. 87}$$

$$f_g = b_{q\alpha} \cos\gamma g / V \qquad \text{Eq. 88}$$

$$k_0 = (b_{\gamma\delta} - B_{q\delta} / b_{q\delta} + (b_{q\delta} / B_{q\delta}) * b_{q\alpha}) / (d_1 + b_{q\alpha} + (B_{q\delta} / b_{q\delta}) * (b_{\gamma\delta} - d_2) - (b_{q\delta} / B_{q\delta}) * d_0) \qquad \text{Eq. 89}$$

$$k_1 = (d_2 - b_{\gamma\alpha} - k_0) / b_{q\delta} \qquad \text{Eq. 90}$$

$$k_3 = (d_0 + b_{q\alpha} k_0) / B_{q\delta} \qquad \text{Eq. 91}$$

$$k_2 = k_3 - k_0 k_1 \qquad \text{Eq. 92}$$

$$c_3 = d_0 / B_{q\delta} \qquad \text{Eq. 93}$$

$$c_1 = (d_1 - b_{q\delta} c_3) / B_{q\delta} \qquad \text{Eq. 94}$$

$$c_2 = c_3 - k_0 c_1 \qquad \text{Eq. 95}$$

$$h_3 = k_0 / B_{q\delta} \qquad \text{Eq. 96}$$

$$h_1 = (1 - b_{q\delta} h_3) / B_{q\delta} \qquad \text{Eq. 97}$$

$$h_2 = h_3 - k_0 h_1 \qquad \text{Eq. 98}$$

$$\delta_e^{\prime} = -k_1 q + c_1 \dot{\gamma}^* - h_1 f_g \qquad \text{Eq. 99}$$

$$\delta_e(s) = \delta_e^{\prime}(s) + \delta_e^{\prime\prime}(s) \qquad \text{Eq. 100}$$

$$\dot{\delta}_e^{\prime\prime} + k_0 \delta_e^{\prime\prime} = -k_2 q + c_2 \dot{\gamma}^* - h_2 f_g \qquad \text{Eq. 101}$$

5.1.3.4.7.2 Yaw Autopilot

The aim of yaw autopilot model is to find the commanded fin deflections to stabilize the yawing dynamics and keep $\dot{\eta} \cong \dot{\eta}^*$, where $\dot{\eta}^*$ is the commanded horizontal flight path angle rate by the guidance law. The following set of equations are implemented in MATSIX to simulate the yaw autopilot [92].

$$b_{\eta\beta} = -Q_d A C_{z_\alpha} / (Mass V_t) \qquad \text{Eq. 102}$$

$$b_{\eta\delta} = -Q_d A C_{z\delta} / (Mass V_t) \qquad \text{Eq. 103}$$

$$b_{r\beta} = -Q_d.A.d.C_{m\alpha} / I_y \qquad \text{Eq. 104}$$

$$b_{r\delta} = -Q_d.A.d.C_{m\delta} / I_y \qquad \text{Eq. 105}$$

$$B_{\eta\delta} = b_{r\delta}\, b_{\eta\beta} - b_{r\beta}\, b_{r\delta} \qquad \text{Eq. 106}$$

$$d_0 = \mu\omega_n{}^3 \qquad \text{Eq. 107}$$

$$d_1 = (1 + 2\xi\mu)\omega_n{}^2 \qquad \text{Eq. 108}$$

$$d_2 = (2\xi + \mu)\omega_n \qquad \text{Eq. 109}$$

$$f_g = b_{q\alpha} \cos\gamma g / V \qquad \text{Eq. 110}$$

$$k_0 = (b_{\eta\beta} - B_{r\delta} / b_{r\delta} + (b_{r\delta} / B_{r\delta}) b_{r\beta}) / (d_1 + b_{r\beta} + (B_{r\delta} / b_{r\delta})(b_{\eta\beta} - d_2) - (b_{r\delta} / B_{r\delta}) d_0) \qquad \text{Eq. 111}$$

$$k_1 = (d_2 - b_{\eta\beta} - k_0) / b_{r\delta} \qquad \text{Eq. 112}$$

$$k_3 = (d_0 - b_{r\beta} k_0) / B_{r\delta} \qquad \text{Eq. 113}$$

$$k_2 = k_3 - k_0 k_1 \qquad \text{Eq. 114}$$

$$c_3 = d_0 / B_{r\delta}$$ Eq. 115

$$c_1 = (d_1 - b_{r\delta} c_3) / B_{r\delta}$$ Eq. 116

$$c_2 = c_3 - k_0 * c_1$$ Eq. 117

$$h_3 = k_0 / B_{r\delta}$$ Eq. 118

$$h_1 = (1 - b_{r\delta} h_3) / B_{r\delta}$$ Eq. 119

$$h_2 = h_3 - k_0 h_1$$ Eq. 120

$$\delta_r = \delta_r' + \delta_r''$$ Eq. 121

$$\delta_r' = -k_1 r + c_1 \dot{\eta}^*$$ Eq. 122

$$\dot{\delta}_r'' + k_0 \delta_r'' = -k_2 r + c_2 \dot{\eta}^*$$ Eq. 123

5.1.3.4.7.3 Roll Autopilot

The autopilot model is used to keep $p \cong 0$ so that the lateral autopilots can work properly. Implemented roll autopilot model of MATSIX is as follows [92]:

$$L_p = Q_d A C_{lp} (d/(2V_t))/(Mass V_t) \quad \text{Eq. 124}$$

$$L_\delta = Q_d A d C_{l\delta} / I_x \quad \text{Eq. 125}$$

$$K_p = (2\xi_r \omega_{nr} + L_p)/L_\delta \quad \text{Eq. 126}$$

$$K_\phi = \omega_{nr}^{\ 2} / L_\delta \quad \text{Eq. 127}$$

$$\delta_a = -K_p p - K_\phi \phi \quad \text{Eq. 128}$$

5.1.3.4.7.4 CAS Model

MATSIX supports the simulation of canard-controlled guided munition with four control surfaces that are 90° apart from each other. A rear view of the munition with the body frame axes on it is seen in Figure 70.

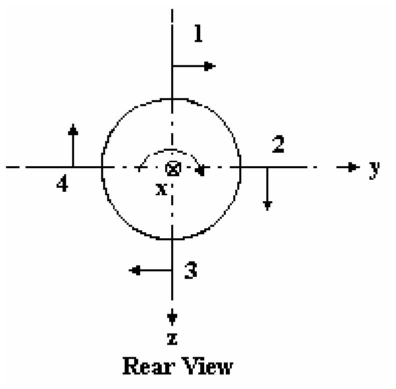


Figure 70 Positive Control Surface Deflection Convention

The positive deflections are as follows. For δ_2 and δ_4 right hand rotations about (-y) and (+y) axis and for δ_1 and δ_3 right hand rotations about the (-z) and (+z) axes of body frame respectively.

δ_e (elevator), δ_r (rudder) and δ_a (aileron) are the apparent control surface deflections. These deflections are defined in terms of the control surface deflections $\delta_1, \delta_2, \delta_3, \delta_4$ as follows [92]:

$$\delta_e = \frac{\delta_2 - \delta_4}{2} \qquad \text{Eq. 129}$$

$$\delta_r = \frac{\delta_1 - \delta_3}{2} \qquad \text{Eq. 130}$$

$$\delta_a = \frac{\delta_1 + \delta_2 + \delta_3 + \delta_4}{4} \qquad \text{Eq. 131}$$

Autopilot is modeled to send pitch, yaw, roll commands defined to the actuators. They are separated into individual fin commands as follows:

$$\delta_{1_c} = \delta_a + \delta_r \qquad \text{Eq. 132}$$

$$\delta_{2_c} = \delta_a + \delta_e \qquad \text{Eq. 133}$$

$$\delta_{3_c} = \delta_a - \delta_r \qquad \text{Eq. 134}$$

$$\delta_{4_c} = \delta_a - \delta_e \qquad \text{Eq. 135}$$

The values indicated above as the fin commands δ_c to the control actuation system are converted into an actual surface deflection δ. Here the response of the fin actuator is modeled by a second order transfer function with natural frequency of $\omega_{n_{cas}}$ and damping ξ_{cas} as follows:

$$\frac{\delta_i(s)}{\delta_{i_c}(s)} = \frac{\omega_{n_{cas}}{}^2}{s^2 + 2\xi_{cas}\omega_{n_{cas}} s + \omega_{n_{cas}}{}^2} \qquad \text{Eq. 136}$$

Then the actual control surface deflections will be the outcomes of the following differential equation.

$$\ddot{\delta}_i + 2\xi_{cas}\omega_{n_{cas}}\dot{\delta}_i + \omega_{n_{cas}}{}^2\delta_i = \omega_{n_{cas}}{}^2\delta_{i_c} \qquad i = 1,2,3,4 \qquad \text{Eq. 137}$$

5.1.3.4.8 Thruster Model

Thrust model of MATSIX is responsible to compute the thrust force and thrust moment acting on the rocket at any time of boost phase. Mass flow values are supplied to the model for any instant of time and thrust force and thrust moment are computed using the following equations [23, 93].

$$F_t = \dot{m}I_{sp} + (P_{ref} - P)A_{Exit} \qquad \text{Eq. 138}$$

$$F_{tx} = F_t \cos\delta_1 \qquad \text{Eq. 139}$$

$$F_{ty} = -F_t \sin\delta_1 \sin\delta_2 \qquad \text{Eq. 140}$$

$$F_{tz} = -F_t \sin\delta_1 \cos\delta_2 \quad \text{Eq. 141}$$

$$L_t = 0 \quad \text{Eq. 142}$$

$$M_t = -T_y(l - X_{cg}) \quad \text{Eq. 143}$$

$$N_t = T_z(l - X_{cg}) \quad \text{Eq. 144}$$

where δ_1, δ_2 are thrust misalignments.

During boost phase, inertia and the center of gravity of the munition change with respect to time due to the burning or the propellant. Below model is used to approximate the instantaneous center of gravity and the inertia [93].

$$x_{cg_t} = x_{cg_{ref}} + (x_{cg_0} - x_{cg_{ref}})\frac{(m_{fuel_0} - m_{fuel_t})}{m_{fuel_0}} \quad \text{Eq. 145}$$

$$I_t = I_{ref} + (I_0 - I_{ref})\frac{(m_{fuel_0} - m_{fuel_t})}{m_{fuel_0}} \quad \text{Eq. 146}$$

5.1.3.5 MATSIX Applications

5.1.3.5.1 LYNX – A Surface to Surface Guided Rocket Simulation

LYNX is a surface to surface guided rocket simulation. The operation concept of the simulated system is designed as given below in Figure 71. The fictive rocket system that was used for simulation is fired from a launcher. It has a solid rocket motor so flies through a boost phase. After boost, guidance system does not start till a predefined range in trajectory. This phase of the flight is called free flight. The last phase is guided flight.

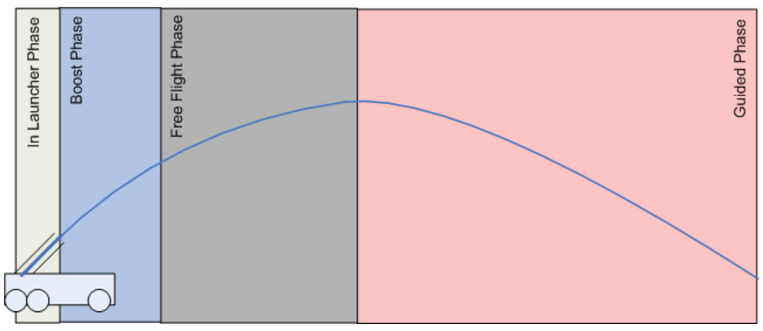
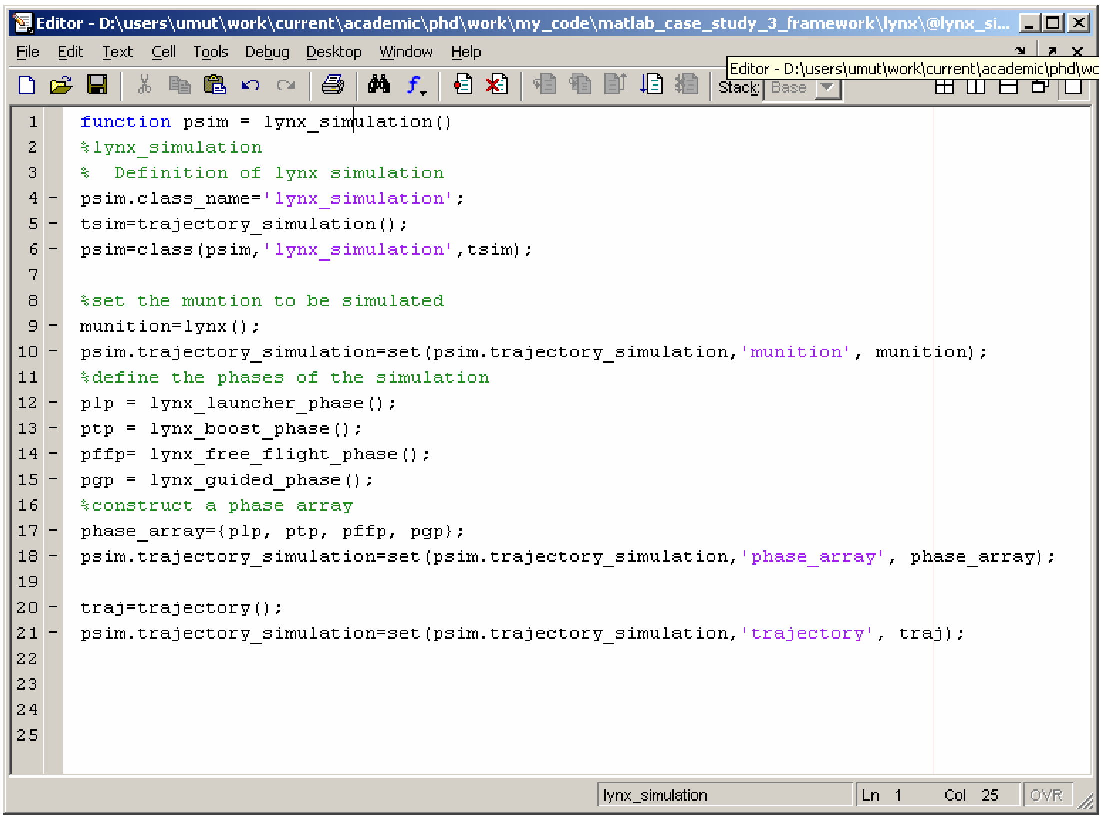


Figure 71 LYNX Concept of Operation

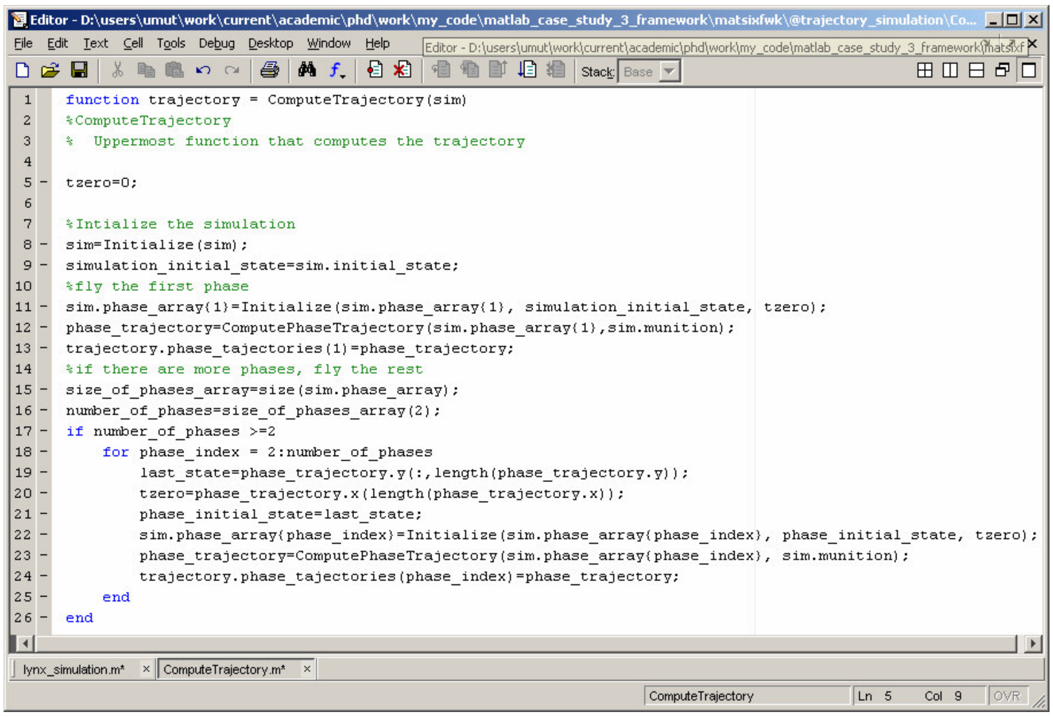

Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study_3_framework\lynx\@lynx_si...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
lynx_simulation Ln 1 Col 25 OVR

```
function psim = lynx_simulation()
%lynx_simulation
%  Definition of lynx simulation
psim.class_name='lynx_simulation';
tsim=trajectory_simulation();
psim=class(psim,'lynx_simulation',tsim);

%set the muntion to be simulated
munition=lynx();
psim.trajectory_simulation=set(psim.trajectory_simulation,'munition', munition);
%define the phases of the simulation
plp = lynx_launcher_phase();
ptp = lynx_boost_phase();
pffp= lynx_free_flight_phase();
pgp = lynx_guided_phase();
%construct a phase array
phase_array={plp, ptp, pffp, pgp};
psim.trajectory_simulation=set(psim.trajectory_simulation,'phase_array', phase_array);

traj=trajectory();
psim.trajectory_simulation=set(psim.trajectory_simulation,'trajectory', traj);
```

Figure 72 LYNX Simulation Class

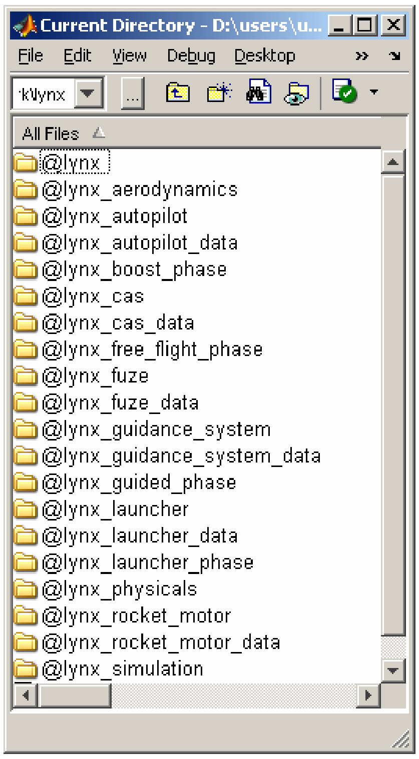
This operational concept together with the data used to represent the rocket system is used to develop the simulation by framework completion. LYNX has been implemented by adding 20 new classes, all derived from framework classes, with 1137 SLOC. The implementation is given in APPENDIX N.

The code that is given above in Figure 72 shows how the operation of the rocket system is reflected to code while completing the framework. One just derives a new simulation class from the base trajectory simulation class and defines the phases and the sequence of phases. The computation of trajectory is implemented in the Trajectory Simulation class beforehand. Once the application engineer inherits a new simulation class from Trajectory Simulation, he owes all the trajectory simulation mechanism. Below in Figure 73, code that simulates the flight of the munition though all phases, is given. It is the Compute Trajectory service of Trajectory Simulation class.


Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study_3_framework\matsixfwk\@trajectory_simulation\Co...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
lynx_simulation.m*
ComputeTrajectory.m*
ComputeTrajectory
Ln 5 Col 9
OVR

```
function trajectory = ComputeTrajectory(sim)
%ComputeTrajectory
%  Uppermost function that computes the trajectory

tzero=0;

%Intialize the simulation
sim=Initialize(sim);
simulation_initial_state=sim.initial_state;
%fly the first phase
sim.phase_array{1}=Initialize(sim.phase_array{1}, simulation_initial_state, tzero);
phase_trajectory=ComputePhaseTrajectory(sim.phase_array{1},sim.munition);
trajectory.phase_tajectories(1)=phase_trajectory;
%if there are more phases, fly the rest
size_of_phases_array=size(sim.phase_array);
number_of_phases=size_of_phases_array(2);
if number_of_phases >=2
    for phase_index = 2:number_of_phases
        last_state=phase_trajectory.y(:,length(phase_trajectory.y));
        tzero=phase_trajectory.x(length(phase_trajectory.x));
        phase_initial_state=last_state;
        sim.phase_array{phase_index}=Initialize(sim.phase_array{phase_index}, phase_initial_state, tzero);
        phase_trajectory=ComputePhaseTrajectory(sim.phase_array{phase_index}, sim.munition);
        trajectory.phase_tajectories(phase_index)=phase_trajectory;
    end
end
```

Figure 73 Compute Trajectory Service of Trajectory Simulation Class


Current Directory - D:\users\u...

File Edit View Debug Desktop

All Files

```
@lynx
@lynx_aerodynamics
@lynx_autopilot
@lynx_autopilot_data
@lynx_boost_phase
@lynx_cas
@lynx_cas_data
@lynx_free_flight_phase
@lynx_fuze
@lynx_fuze_data
@lynx_guidance_system
@lynx_guidance_system_data
@lynx_guided_phase
@lynx_launcher
@lynx_launcher_data
@lynx_launcher_phase
@lynx_physicals
@lynx_rocket_motor
@lynx_rocket_motor_data
@lynx_simulation
```

Figure 74 LYNX Classes

The list of the derived classes for LYNX simulation is given above in Figure 74. There will be a section on how the framework is completed with these classes to develop LYNX.

LYNX simulation uses simulation parameters that define the surface to surface guided rocket system that it simulates. This data includes physicals like its mass and reference area, its aerodynamic coefficients, its motor properties and so on. This data used in LYNX Simulation is given in APPENDIX G.

Number of sample simulation runs was done with the developed code. Below is presents the results of one of them with 711 mills elevation. More plots from the sample runs of LYNX Simulation are given in APPENDIX H.

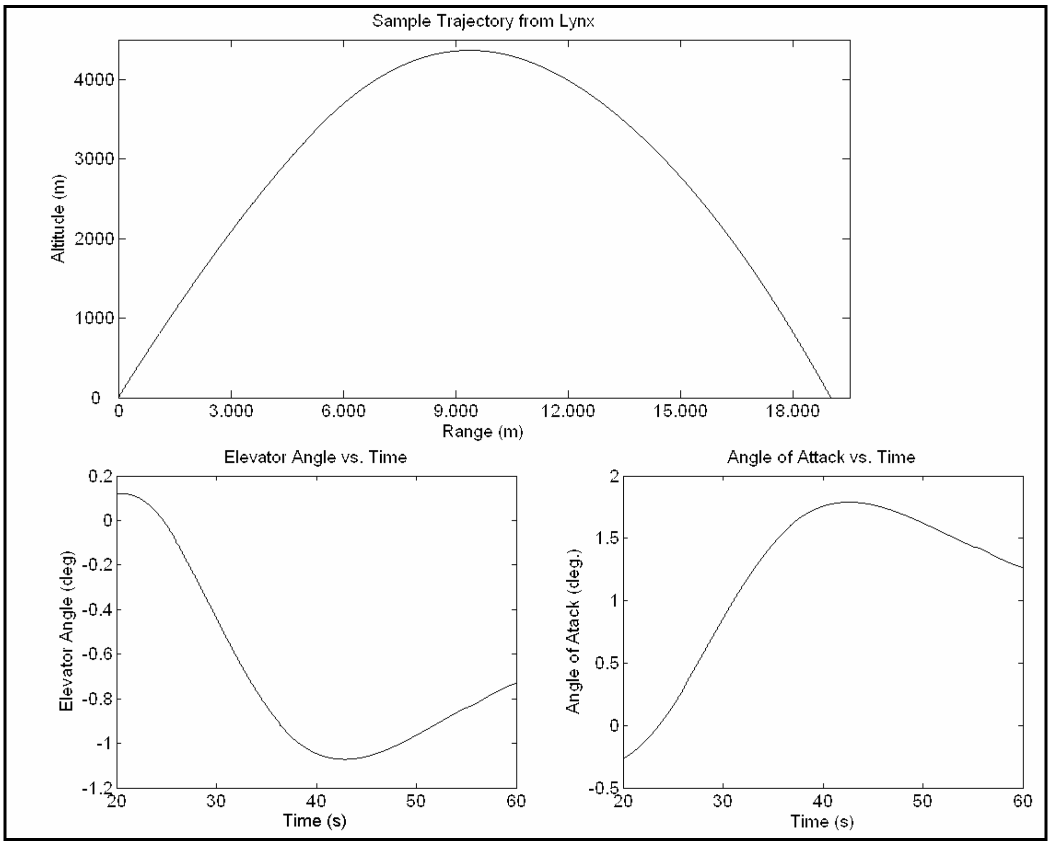


Figure 75 Plots from a Sample LYNX Simulation Run

5.1.3.5.2 PUMA – An Air to Ground Guided Bomb Simulation

PUMA is a guided bomb simulation. The concept bomb that was used for simulation is released from a bomber aircraft. Its guidance system does not operate for the first short period after release for a safe separation. Then guidance and control system starts to navigate the bomb. A number different guidance laws are used as the guidance algorithm. The operation concept of this system is designed as given below in Figure 76.

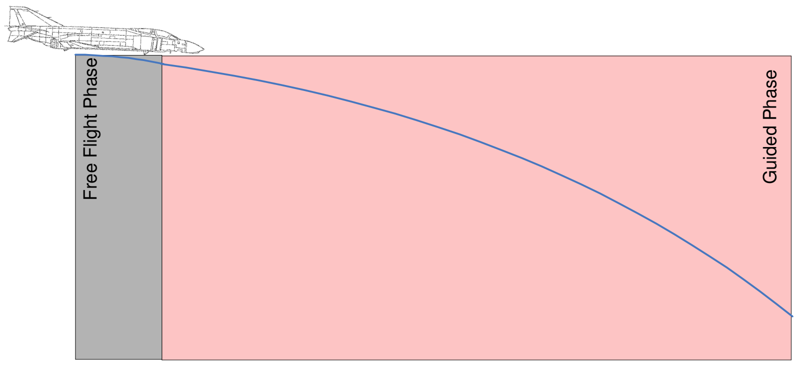
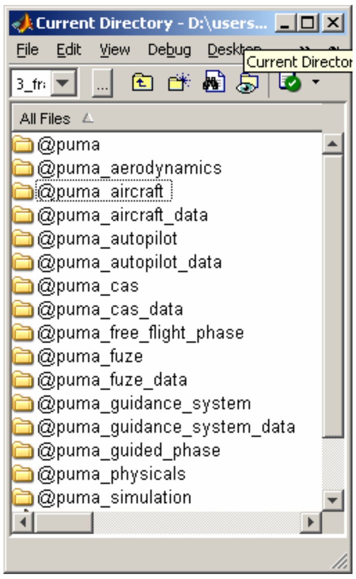


Figure 76 PUMA Concept of Operation

Like LYNX, PUMA was also developed by completing the MATLAB 6DOF Trajectory Simulation Framework. PUMA has been implemented by adding 16 new classes, all derived from framework classes, with 733 SLOC. The list of the derived classes for PUMA simulation is given below in Figure 77. The implementation is given in APPENDIX N.



@puma
@puma_aerodynamics
@puma_aircraft
@puma_aircraft_data
@puma_autopilot
@puma_autopilot_data
@puma_cas
@puma_cas_data
@puma_free_flight_phase
@puma_fuze
@puma_fuze_data
@puma_guidance_system
@puma_guidance_system_data
@puma_guided_phase
@puma_physicals
@puma_simulation

Figure 77 PUMA Classes

The data used in PUMA Simulation is given in APPENDIX I. Sample runs were carried out using the developed system. Below is from the results of a sample for a release from 1100m height with 250m/s True Air Speed. More plots from the sample runs of PUMA Simulation are given in APPENDIX J.

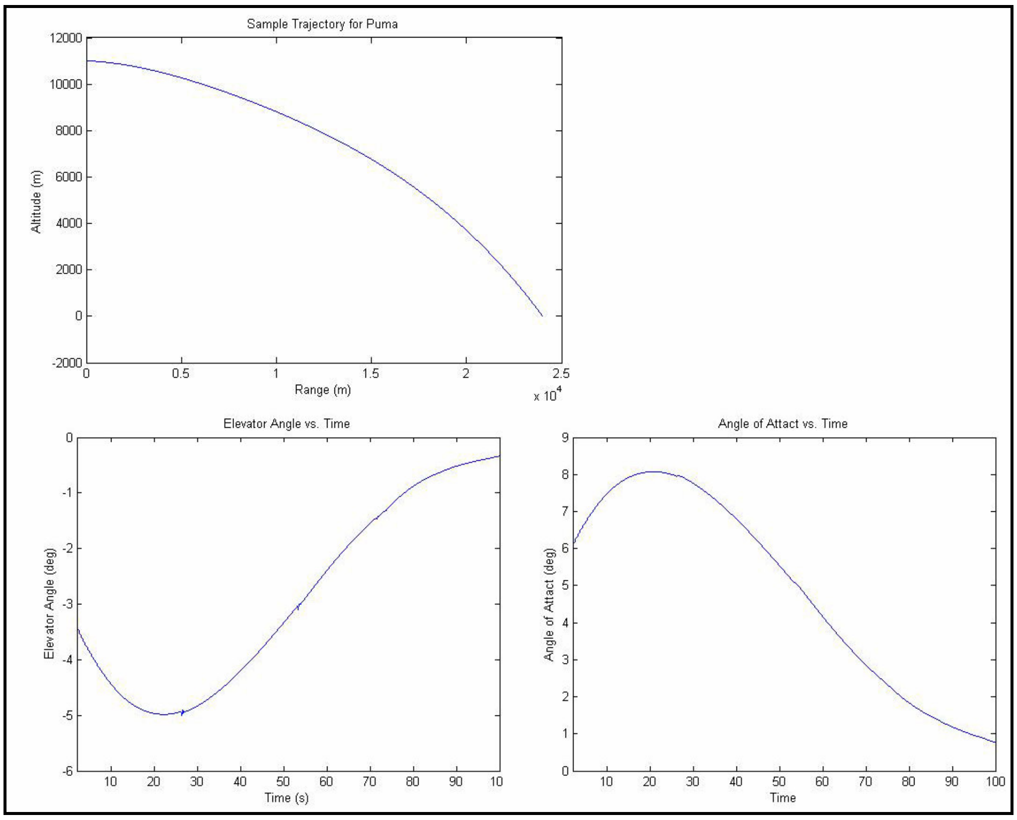
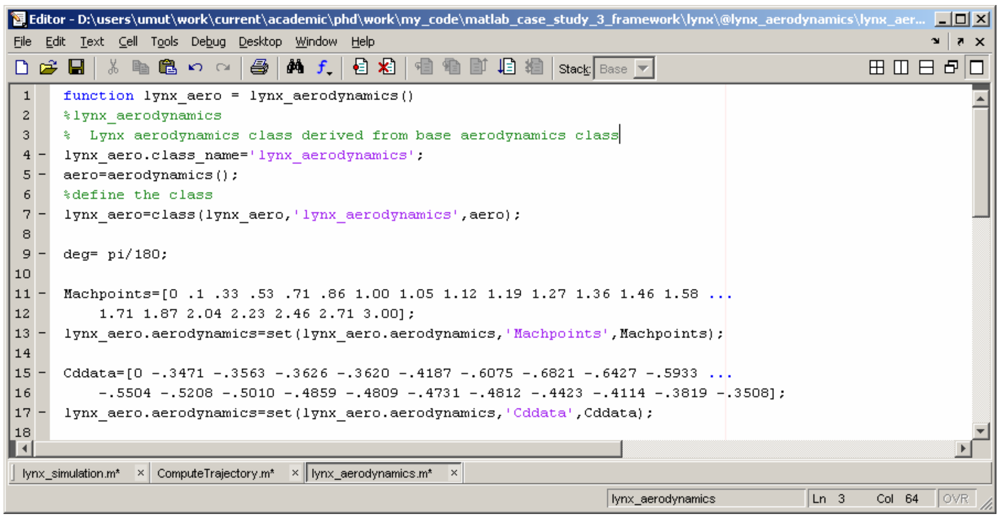


Figure 78 Plots from a Sample PUMA Simulation Run

5.1.3.6 Framework Completion Process

In this section, we will investigate how we complete the framework in detail. While completing the framework, one basically follows the steps listed below;

1. Derive you data classes from related base data classes and implement the mechanism to read data. Below, Figure 79 is a part of LYNX Aerodynamics class code. This class is derived form Aerodynamics class. Aerodynamic coefficients are hard coded in LYNX simulation.



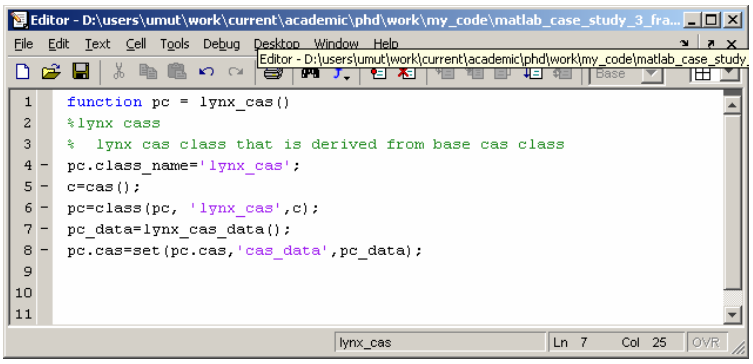
Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study_3_framework\lynx\@lynx_aerodynamics\lynx_aer...

```
function lynx_aero = lynx_aerodynamics()
%lynx_aerodynamics
%  Lynx aerodynamics class derived from base aerodynamics class
lynx_aero.class_name='lynx_aerodynamics';
aero=aerodynamics();
%define the class
lynx_aero=class(lynx_aero,'lynx_aerodynamics',aero);

deg= pi/180;

Machpoints=[0 .1 .33 .53 .71 .86 1.00 1.05 1.12 1.19 1.27 1.36 1.46 1.58 ...
    1.71 1.87 2.04 2.23 2.46 2.71 3.00];
lynx_aero.aerodynamics=set(lynx_aero.aerodynamics,'Machpoints',Machpoints);

Cddata=[0 -.3471 -.3563 -.3626 -.3620 -.4187 -.6075 -.6821 -.6427 -.5933 ...
    -.5504 -.5208 -.5010 -.4859 -.4809 -.4731 -.4812 -.4423 -.4114 -.3819 -.3508];
lynx_aero.aerodynamics=set(lynx_aero.aerodynamics,'Cddata',Cddata);
```

File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
lynx_simulation.m* ComputeTrajectory.m* lynx_aerodynamics.m*
lynx_aerodynamics Ln 3 Col 64 OVR

Figure 79 LYNX Aerodynamics Class

2. Derive your munition sub system classes from related base classes and implement the association of subsystem classes with their data classes.



Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study_3_fra...

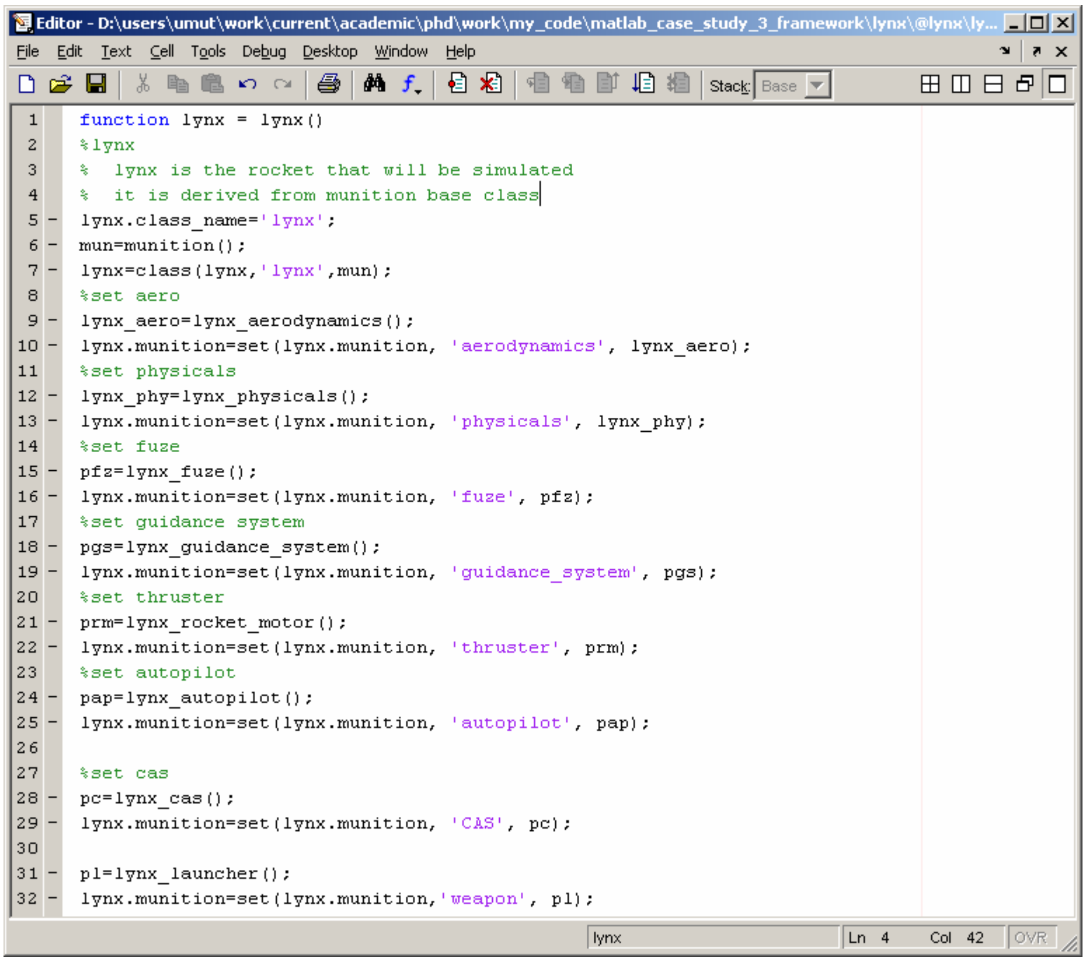
Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study

```
function pc = lynx_cas()
%lynx cass
%  lynx cas class that is derived from base cas class
pc.class_name='lynx_cas';
c=cas();
pc=class(pc, 'lynx_cas',c);
pc_data=lynx_cas_data();
pc.cas=set(pc.cas,'cas_data',pc_data);
```

Figure 80 LYNX CAS Class

As an example LYNX CAS class code is presented in Figure 80. This class is derived form CAS class of the framework. As seen in line 7 LYNX CAS class is associated with LYNX CAS data class.

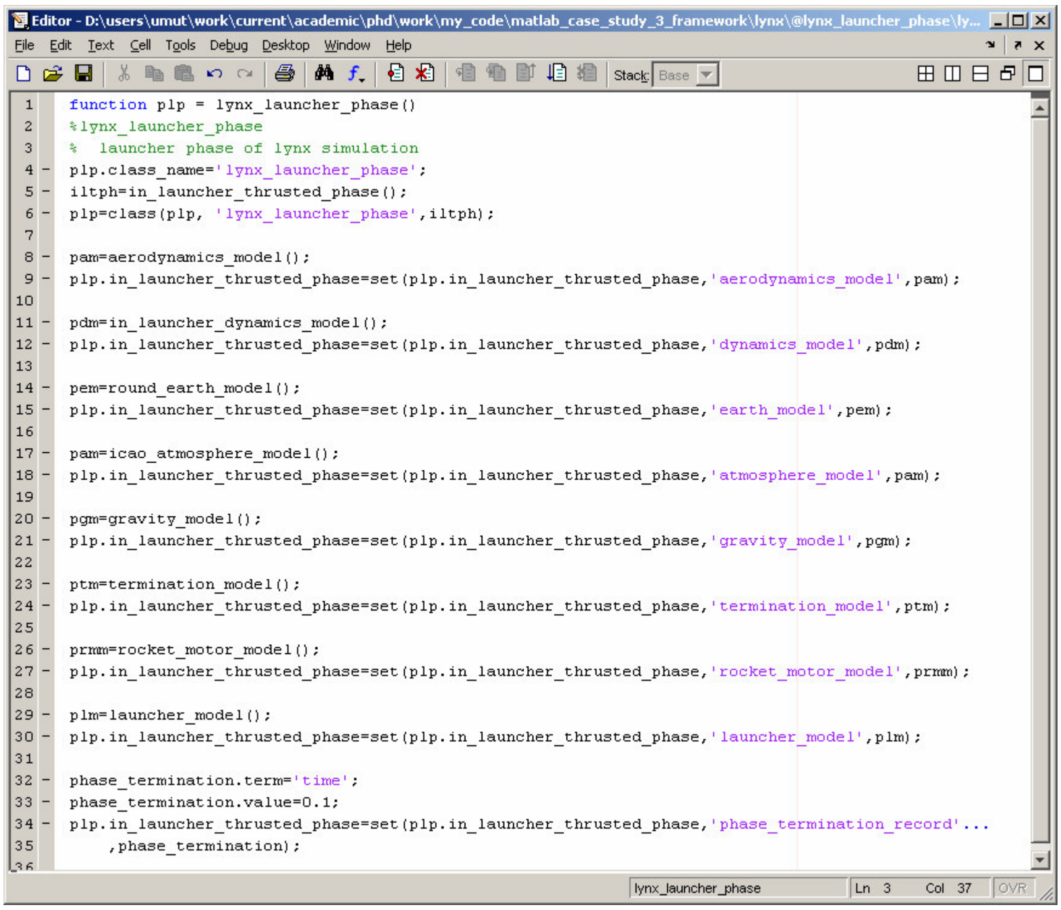
3. Derive your munition class from the base munition class and implement the association of munition with its data and its subsystems. Below LYNX class is given as an example in Figure 81. Here as you see, all the sub systems and the related data are associated with LYNX class that derived from munition base class.


Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study_3_framework\lynx\@lynx\ly...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
lynx
Ln 4
Col 42
OVR

```
function lynx = lynx()
%lynx
%  lynx is the rocket that will be simulated
%  it is derived from munition base class
lynx.class_name='lynx';
mun=munition();
lynx=class(lynx,'lynx',mun);
%set aero
lynx_aero=lynx_aerodynamics();
lynx.munition=set(lynx.munition, 'aerodynamics', lynx_aero);
%set physicals
lynx_phy=lynx_physicals();
lynx.munition=set(lynx.munition, 'physicals', lynx_phy);
%set fuze
pfz=lynx_fuze();
lynx.munition=set(lynx.munition, 'fuze', pfz);
%set guidance system
pgs=lynx_guidance_system();
lynx.munition=set(lynx.munition, 'guidance_system', pgs);
%set thruster
prm=lynx_rocket_motor();
lynx.munition=set(lynx.munition, 'thruster', prm);
%set autopilot
pap=lynx_autopilot();
lynx.munition=set(lynx.munition, 'autopilot', pap);

%set cas
pc=lynx_cas();
lynx.munition=set(lynx.munition, 'CAS', pc);

pl=lynx_launcher();
lynx.munition=set(lynx.munition,'weapon', pl);
```

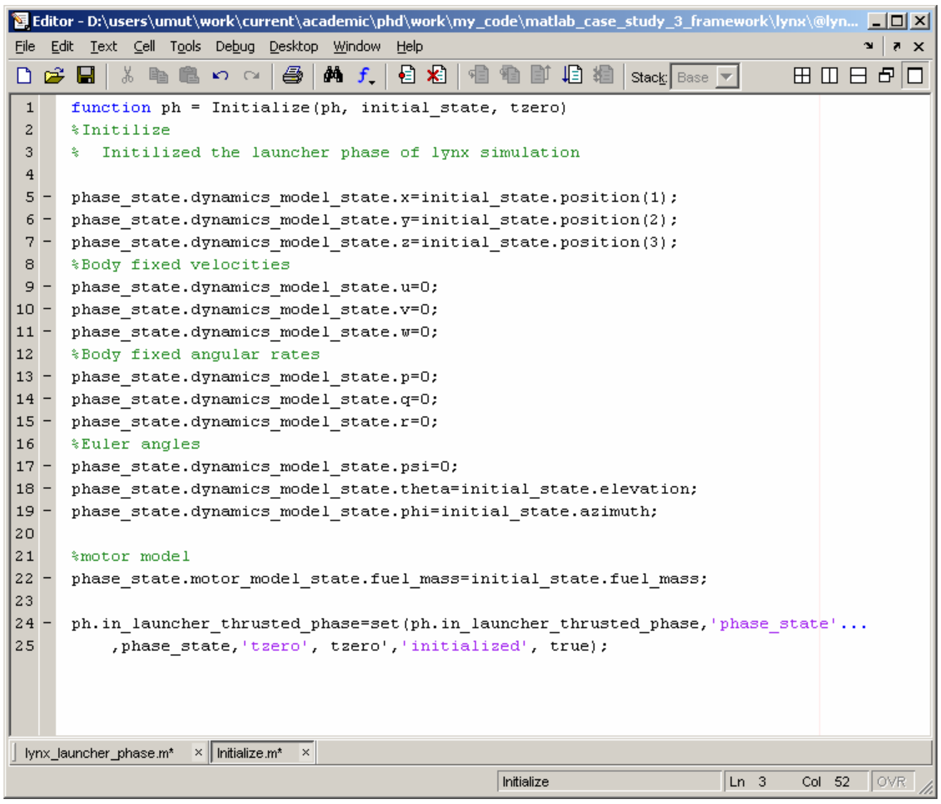
Figure 81 LYNX Class

4. Derive required phase classes from the base phase classes, implement the initialization services of the derived phases and specify the models that will be used during simulation and the phase termination conditions. LYNX Launcher Phase class and its Initialize service are presented below in Figure 82 and Figure 83 as an example of this step. In the class definition related models and phase termination conditions are associated with the class. Initialize service on the other hand implements how the initial phase state is set.


Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study_3_framework\lynx\@lynx_launcher_phase\ly...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
lynx_launcher_phase
Ln 3 Col 37 OVR

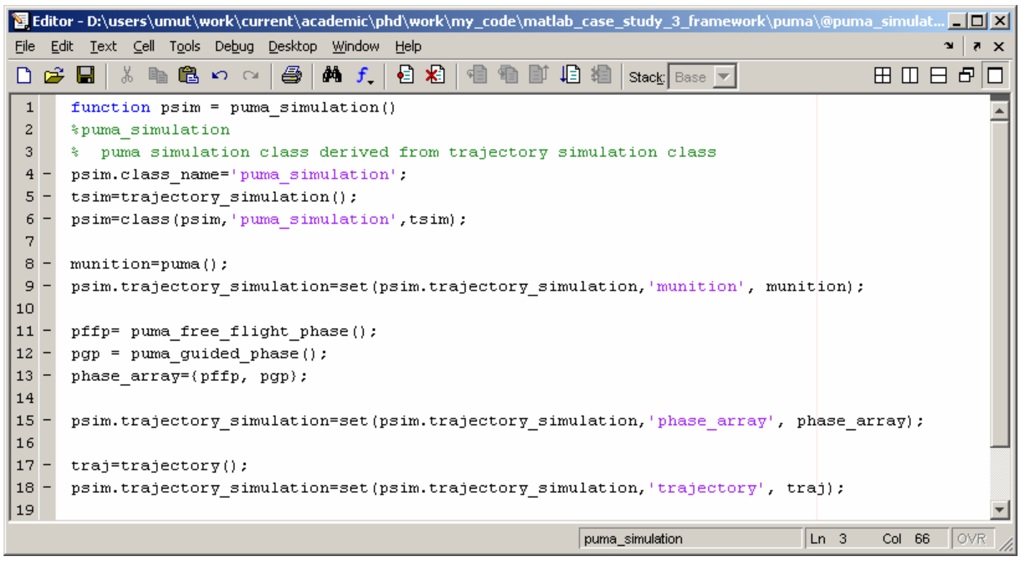
```
function plp = lynx_launcher_phase()
%lynx_launcher_phase
%  launcher phase of lynx simulation
plp.class_name='lynx_launcher_phase';
iltph=in_launcher_thrusted_phase();
plp=class(plp, 'lynx_launcher_phase',iltph);

pam=aerodynamics_model();
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'aerodynamics_model',pam);

pdm=in_launcher_dynamics_model();
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'dynamics_model',pdm);

pem=round_earth_model();
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'earth_model',pem);

pam=icao_atmosphere_model();
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'atmosphere_model',pam);

pgm=gravity_model();
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'gravity_model',pgm);

ptm=termination_model();
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'termination_model',ptm);

prmm=rocket_motor_model();
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'rocket_motor_model',prmm);

plm=launcher_model();
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'launcher_model',plm);

phase_termination.term='time';
phase_termination.value=0.1;
plp.in_launcher_thrusted_phase=set(plp.in_launcher_thrusted_phase,'phase_termination_record'...
    ,phase_termination);
```

Figure 82 LYNX Launcher Phase Class


Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study_3_framework\lynx\@lyn...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
lynx_launcher_phase.m*
Initialize.m*
Initialize
Ln 3 Col 52 OVR

```
function ph = Initialize(ph, initial_state, tzero)
%Initilize
%   Initilized the launcher phase of lynx simulation

phase_state.dynamics_model_state.x=initial_state.position(1);
phase_state.dynamics_model_state.y=initial_state.position(2);
phase_state.dynamics_model_state.z=initial_state.position(3);
%Body fixed velocities
phase_state.dynamics_model_state.u=0;
phase_state.dynamics_model_state.v=0;
phase_state.dynamics_model_state.w=0;
%Body fixed angular rates
phase_state.dynamics_model_state.p=0;
phase_state.dynamics_model_state.q=0;
phase_state.dynamics_model_state.r=0;
%Euler angles
phase_state.dynamics_model_state.psi=0;
phase_state.dynamics_model_state.theta=initial_state.elevation;
phase_state.dynamics_model_state.phi=initial_state.azimuth;

%motor model
phase_state.motor_model_state.fuel_mass=initial_state.fuel_mass;

ph.in_launcher_thrusted_phase=set(ph.in_launcher_thrusted_phase,'phase_state'...
    ,phase_state,'tzero', tzero','initialized', true);
```

Figure 83 Initialize Service of LYNX Launcher Phase


Editor - D:\users\umut\work\current\academic\phd\work\my_code\matlab_case_study_3_framework\puma\@puma_simulat...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
puma_simulation
Ln 3 Col 66 OVR

```
function psim = puma_simulation()
%puma_simulation
%   puma simulation class derived from trajectory simulation class
psim.class_name='puma_simulation';
tsim=trajectory_simulation();
psim=class(psim,'puma_simulation',tsim);

munition=puma();
psim.trajectory_simulation=set(psim.trajectory_simulation,'munition', munition);

pffp= puma_free_flight_phase();
pgp = puma_guided_phase();
phase_array={pffp, pgp};

psim.trajectory_simulation=set(psim.trajectory_simulation,'phase_array', phase_array);

traj=trajectory();
psim.trajectory_simulation=set(psim.trajectory_simulation,'trajectory', traj);

```

Figure 84 PUMA Simulation Class

5. Finally, derive your simulation class from the base Trajectory Simulation class and specifying the phases and their order. PUMA class is presented above in Figure 84 as an example.

5.1.4 C# Point Mass Trajectory Simulation Framework

After presenting a full scale ontology based reuse infrastructure development and its use by means of framework development and framework completion for MATLAB platform, we would like to present another case for basically two reasons. First we would like to exercise to design a framework for a different platform. Then we would like to present the use of code generation capabilities of computer aided software engineering tools with ontology based trajectory simulation reuse infrastructure. We selected .NET platform of Microsoft. C# was chosen as the language to develop this framework.

In this case study, we designed a platform specific framework architecture and generated source code from this design specification. Full implementation of the framework and framework completion for specific applications is planned to be carried out by different developers from the target reuse group. This will be a step towards institutionalization of the ontology based reuse infrastructure development process in TUBITAK-SAGE.

Platform and the language will not be discussed in detail here but it will be good to give a brief background. C# is said to be designed to provide a simple, safe, modern, object-oriented, internet-centric, high performance language for .NET development. It is a new language, but it is said to be drawing on the lessons learned over the past three decades. C# influences from Java, C++, Visual Basic (VB), and other languages. The .NET platform on the other hand is, in essence, a new development framework that provides a fresh application programming interface (API) to the services and APIs of classic Windows operating systems, especially Windows 2000, while bringing together a number of cutting edge technologies that emerged from Microsoft during the late 1990s. Currently, the .NET Framework consists of:

- Four official languages: C#, VB .NET, Managed C++, and JScript .NET
- The Common Language Runtime (CLR), an object-oriented platform for Windows and web development that all these languages share
- A number of related class libraries, collectively known as the Framework Class Library (FCL) [105].

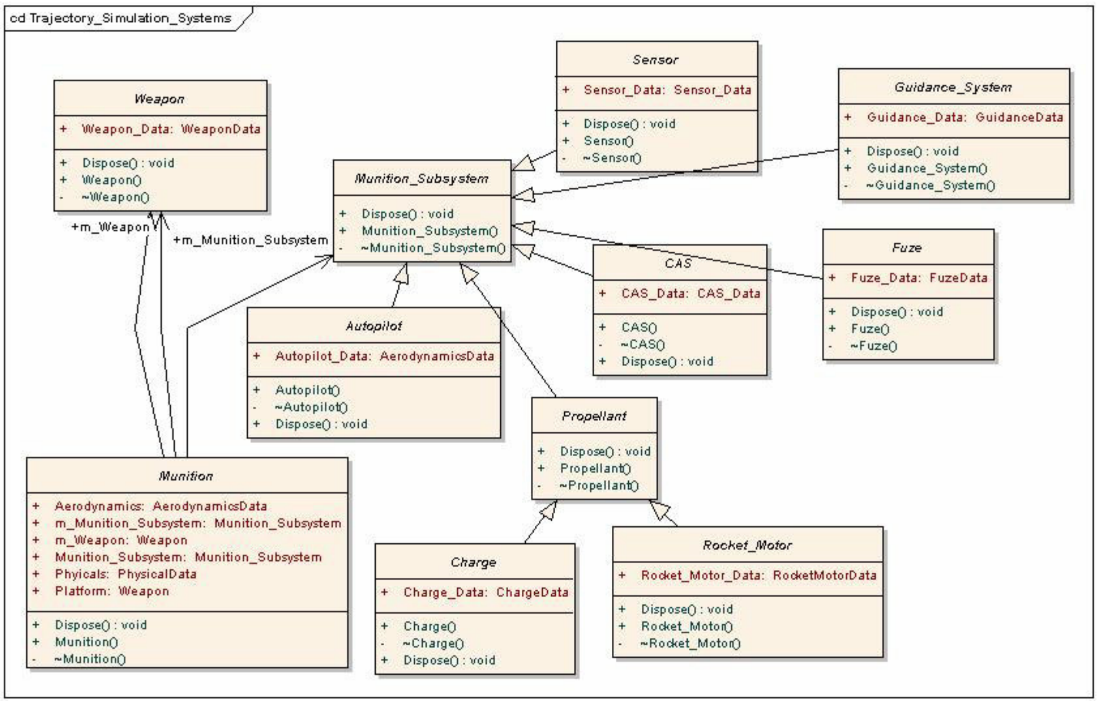
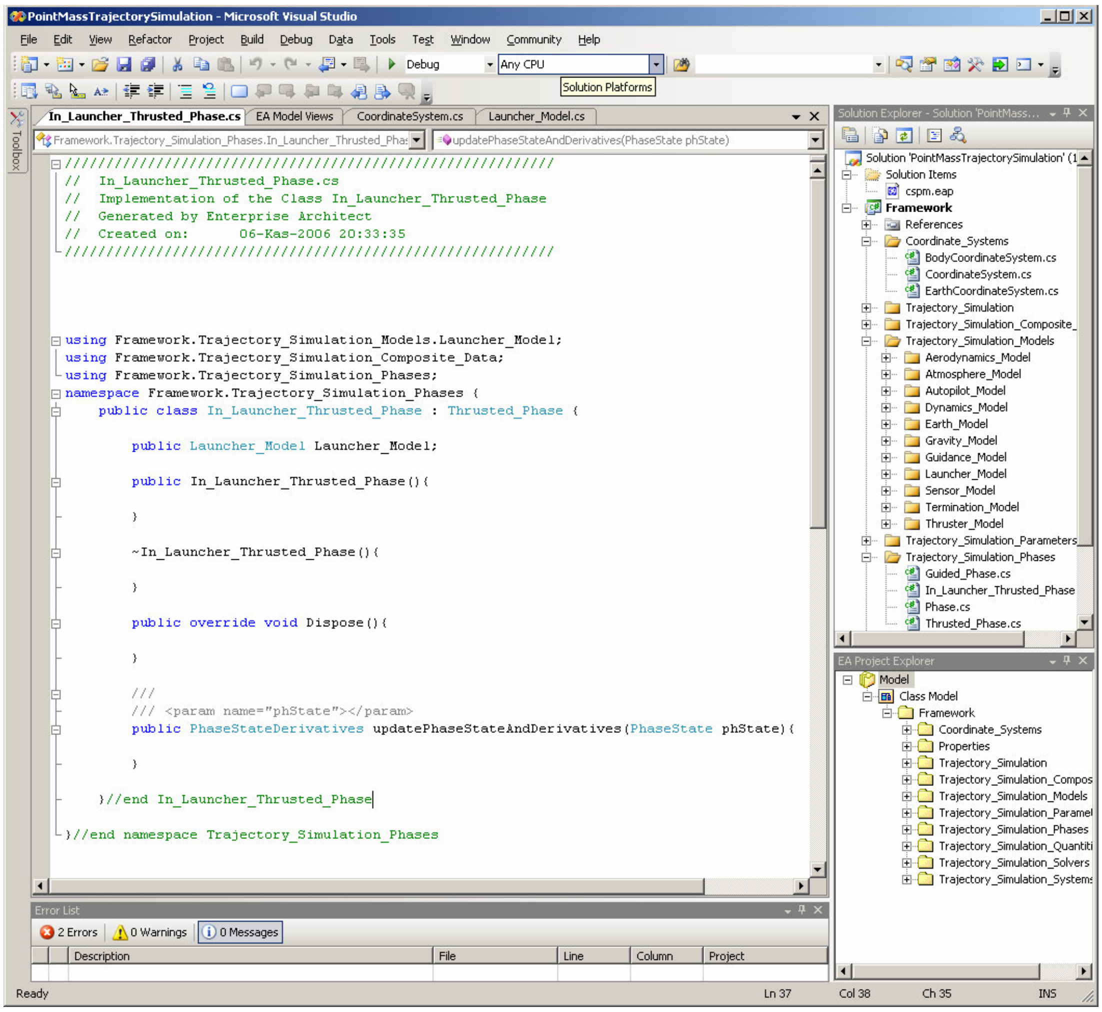


Figure 85 Trajectory Simulation Systems

Above is a design schema from platform specific framework architecture As we have done in MATLAB framework architecture design, here we take the platform independent framework architecture and subset it for point mass trajectory simulation supporting guided and thrusted munitions. Then we applied the platform specific design constraints on that subset and generated a platform specific framework architecture..



```
//////////////////////////////////////////////////////////////
//  In_Launcher_Thrusted_Phase.cs
//  Implementation of the Class In_Launcher_Thrusted_Phase
//  Generated by Enterprise Architect
//  Created on:      06-Kas-2006 20:33:35
//////////////////////////////////////////////////////////////




using Framework.Trajectory_Simulation_Models.Launcher_Model;
using Framework.Trajectory_Simulation_Composite_Data;
using Framework.Trajectory_Simulation_Phases;
namespace Framework.Trajectory_Simulation_Phases {
    public class In_Launcher_Thrusted_Phase : Thrusted_Phase {

        public Launcher_Model Launcher_Model;

        public In_Launcher_Thrusted_Phase(){

        }

        ~In_Launcher_Thrusted_Phase(){

        }

        public override void Dispose(){

        }

        ///
        /// <param name="phState"></param>
        public PhaseStateDerivatives updatePhaseStateAndDerivatives(PhaseState phState){

        }

    }//end In_Launcher_Thrusted_Phase

}//end namespace Trajectory_Simulation_Phases
```

PointMassTrajectorySimulation - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Data Tools Test Window Community Help
Debug
Any CPU
Solution Platforms
In_Launcher_Thrusted_Phase.cs
EA Model Views
CoordinateSystem.cs
Launcher_Model.cs
Framework.Trajectory_Simulation_Phases.In_Launcher_Thrusted_Phas
updatePhaseStateAndDerivatives(PhaseState phState)
Toolbox
Solution Explorer - Solution 'PointMass...
Solution 'PointMassTrajectorySimulation' (1
Solution Items
cspm.eap
Framework
References
Coordinate_Systems
BodyCoordinateSystem.cs
CoordinateSystem.cs
EarthCoordinateSystem.cs
Trajectory_Simulation
Trajectory_Simulation_Composite_
Trajectory_Simulation_Models
Aerodynamics_Model
Atmosphere_Model
Autopilot_Model
Dynamics_Model
Earth_Model
Gravity_Model
Guidance_Model
Launcher_Model
Sensor_Model
Termination_Model
Thruster_Model
Trajectory_Simulation_Parameters
Trajectory_Simulation_Phases
Guided_Phase.cs
In_Launcher_Thrusted_Phase
Phase.cs
Thrusted_Phase.cs
EA Project Explorer
Model
Class Model
Framework
Coordinate_Systems
Properties
Trajectory_Simulation
Trajectory_Simulation_Compos
Trajectory_Simulation_Models
Trajectory_Simulation_Paramet
Trajectory_Simulation_Phases
Trajectory_Simulation_Quantiti
Trajectory_Simulation_Solvers
Trajectory_Simulation_System
Error List
2 Errors
0 Warnings
0 Messages
Description
File
Line
Column
Project
Ready
Ln 37
Col 38
Ch 35
INS

Figure 86 Sample Code Snapshot From IDE

Enterprise Architect of Sparx Systems is being used as the computer aided software engineering tool for forward and reverse engineering during development of this framework. Using this tool's forward engineering capabilities, code generation process was executed. About 2600 SLOC was produced automatically. Visual

Studio Team System is being used as the IDE (Integrated Development Environment). Above, an example code snapshot from this produced code is presented in Figure 86. More sample diagrams from this platform specific framework design and sample code snapshots will be provided in APPENDIX K. Besides, APPENDIX N presents the model and code projects.

5.2 Function Oriented Infrastructure Specification and Implementation

5.2.1 Function Oriented Programming and Reuse

Function oriented programming as stated by Sommerville relies on decomposing the system into a set of interacting functions with a centralized system state shared by these functions. Function-oriented design has been used informally since the programming has begun. Programs have been decomposed into subroutines which were functional in nature [65].

One way to develop reuse infrastructure for function oriented paradigm is to develop a function library in a structured language like Fortran or C. Numerical Recipes is an example of such a function library. It is one of the most famous function libraries in scientific computing society [106 and 107]. The other way is to use MATLAB Simulink and develop a function oriented blocksets. We selected to do this one since such an ontology based blockset reuse practice using MATLAB Simulink is more likely to be used by target reuse group then a Fortran or C function library approach.

As presented below in Figure 87, in a function oriented reuse scenario as in the object oriented scenario, we still propose a platform independent abstract design as the first step of the infrastructure specification activity of domain engineering.

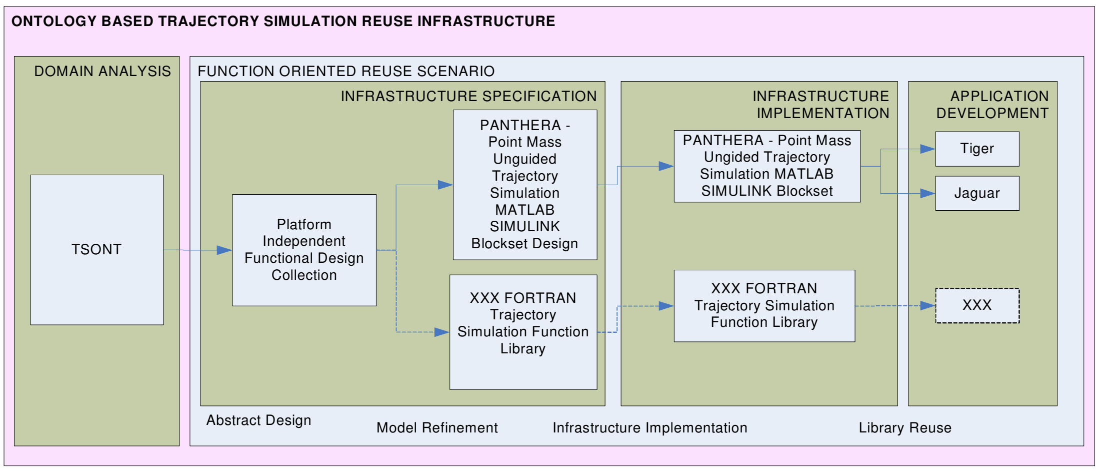

ONTOLOGY BASED TRAJECTORY SIMULATION REUSE INFRASTRUCTURE
DOMAIN ANALYSIS
TSONT
FUNCTION ORIENTED REUSE SCENARIO
INFRASTRUCTURE SPECIFICATION
Platform Independent Functional Design Collection
PANTHERA - Point Mass Unguided Trajectory Simulation MATLAB SIMULINK Blockset Design
XXX FORTRAN Trajectory Simulation Function Library
INFRASTRUCTURE IMPLEMENTATION
PANTHERA - Point Mass Ungided Trajectory Simulation MATLAB SIMULINK Blockset
XXX FORTRAN Trajectory Simulation Function Library
APPLICATION DEVELOPMENT
Tiger
Jaguar
XXX
Abstract Design
Model Refinement
Infrastructure Implementation
Library Reuse

Figure 87 Function Oriented Reuse Scenario

Data flow diagrams are treated as the tools for abstract function oriented design. As presented in the famous software engineering book of Sommerville, data flow diagrams are concerned with designing a sequence of functional transformations that convert system inputs into the required outputs. These diagrams illustrate how data flows through a system and how the output is derived from the input through a sequence of functional transformations [65].

Different from our object oriented scenario, we do not propose a single abstract design that covers whole domain. Rather, we propose a collection of data flow diagrams for different problem sets, like, point mass data flow diagrams that we will present in the following sections or a modified point mass projectile simulation data flow diagrams. This collection of abstract designs will be the reuse assets for the future projects.

Platform specific design will be the refinement of these abstract designs. We, in our case study, refined or transformed the data flow diagrams to the block diagrams of MATLAB.

5.2.2 Platform Independent Point Mass Unguided Trajectory Simulation Abstract Software Design

The functions with their functionalities and their interfaces are captured in the ontology. Besides, dependencies of these functions are also being captured in TSONT. Here in this case study, we tried to show how ontology is helpful when the software development paradigm changes from object oriented to function oriented. We used function definitions in the ontology to draw our data flow diagrams.

Below, in Figure 88, data flow diagram of Compute Point Mass Phase Trajectory service is given. Here in the data flow, the functions to be executed to compute the point mass phase's trajectory and the data flow among functions are captured in abstract fashion. The whole set of data flow diagrams are attached at APPENDIX L and APPENDIX N.

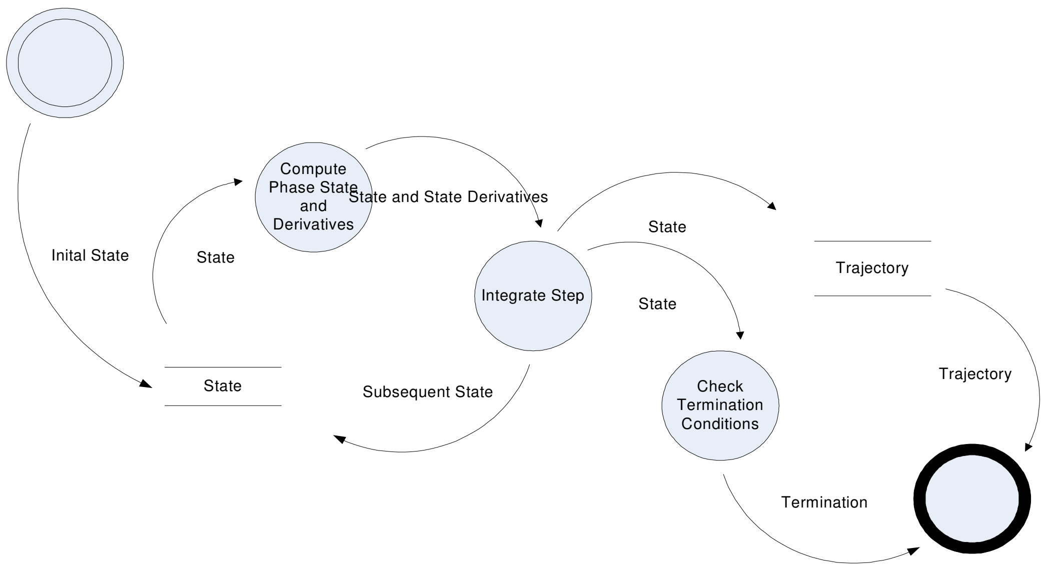


Figure 88 Compute Point Mass Phase Trajectory Data Flow Diagram

5.2.3 Design of PANTHERA

PANTHERA is a function oriented MATLAB Simulink blockset for point mass unguided trajectory simulation. Models used in PANTHERA are kept as simple as possible. The aim of this case study is to present the use of ontology based reuse infrastructure in function oriented software development paradigm.

We, in this research used MATLAB Simulink in a function oriented fashion. Blocks are used to represent the functions and their ports are used to represent function interfaces. Blocksets are set of blocks. Blocks are the elements from which MATLAB Simulink models are built. One can model virtually any dynamic system by creating and interconnecting blocks in appropriate ways.

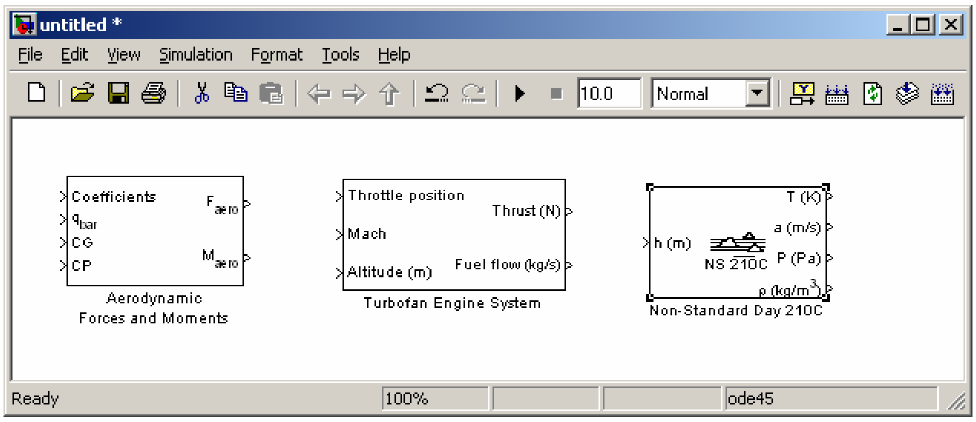


Figure 89 Some Blocks from Aerospace Blockset

Aerospace system modeling and simulation community is familiar with Aerospace Blockset of Mathworks Inc. The way the Aerospace Blockset is constructed can be named as actor-oriented approach [108]. It contains the basic actors in an aerospace simulation as blocks and hides the functionality of the actors underneath block interfaces. Above in Figure 89, some blocks from Aerospace Blockset are presented.

In our approach, we defined the blocks regarding the functionality they serve. Platform Independent Point Mass Unguided Trajectory Simulation Abstract Software Design that is presented in the previous section is used to structure the blockset. It will be good to be reminded that the definitions of functions with their interfaces and dependencies modeled in Platform Independent Point Mass Unguided Trajectory Simulation Abstract Software Design are based on TSONT. By structuring the MATLAB Simulink blocks of PANTHERA for representing functions and their dependencies, this approach is classified as the function oriented use of MATLAB Simulink. Subsystems of PANTHERA are presented below.

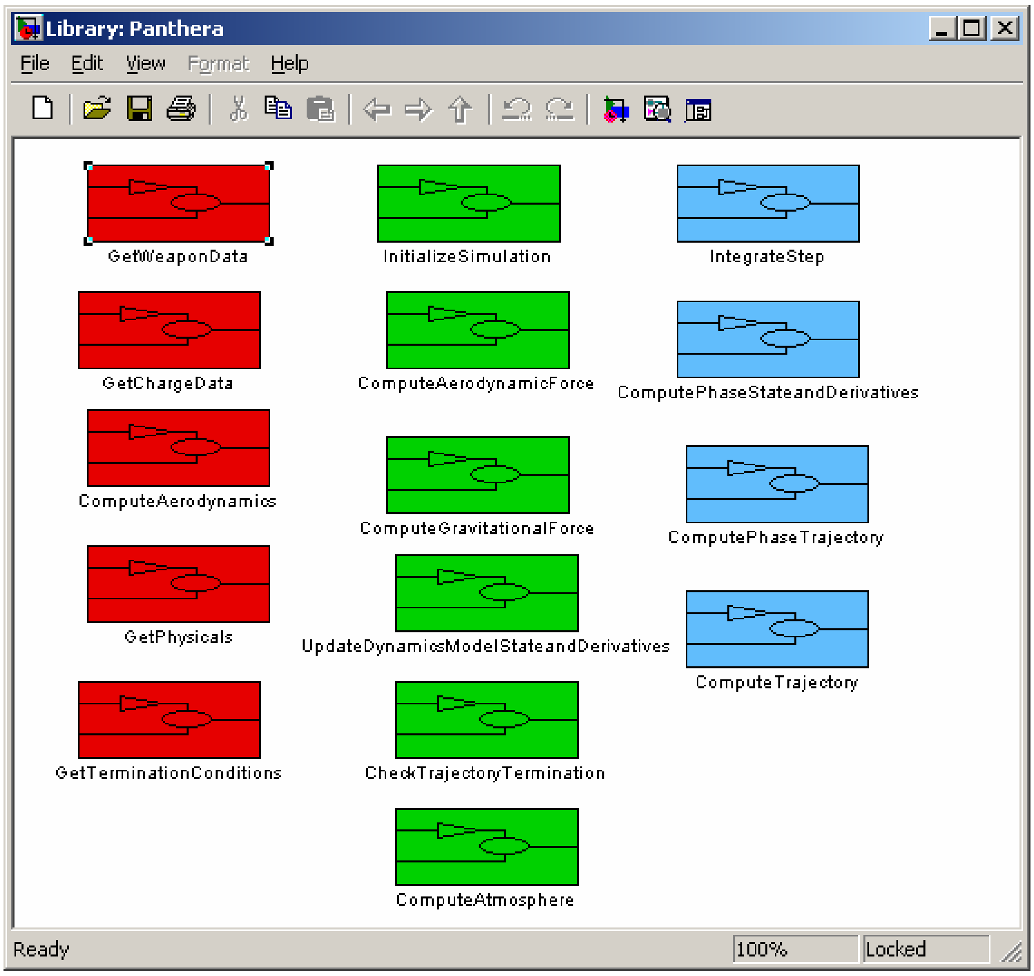


Figure 90 Subsystem of PANTHERA

Problem set is selected as simple as possible since the aim of this case is not to develop a fully functional large and complex blockset but rather to show that MATLAB Simulink Blocksets are opportunities to enable a function oriented reuse scenario starting from TSONT.

Subsystems in the left most column of the blockset are data sources. Mid column has the computation oriented subsystems and the right most column has the aggregate subsystems that use left two columns to accomplish their task.

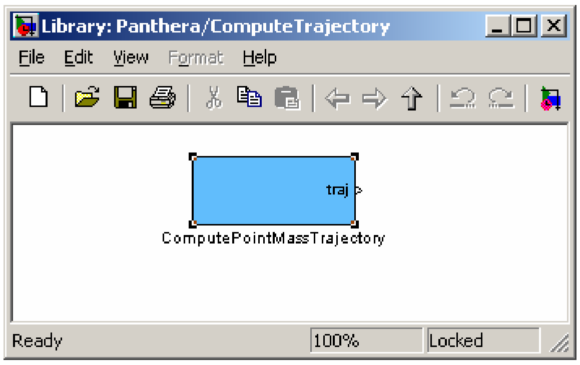

Library: Panthera/ComputeTrajectory
File
Edit
View
Format
Help
traj
ComputePointMassTrajectory
Ready
100%
Locked

Figure 91 Compute Trajectory Subsystem

Compute Trajectory Subsystem only has Compute Point Mass Trajectory block. This block has only one output port which is trajectory as defined in data flow diagrams.

When we look under the mask of Compute Point Mass Trajectory block, we will see that Initialize Point Mass Simulation Service initializes the simulation by the data it obtained from Get Point Mass Weapon Data and Get Point Mass Data services. Check Trajectory Termination, on the other hand, checks the termination condition on the state of the simulation computed by Compute Point Mass Phase Trajectory block.

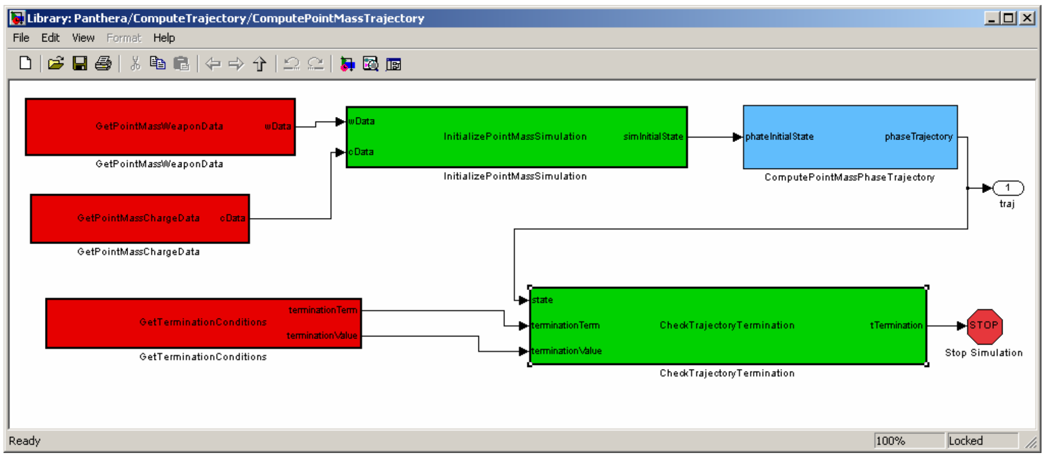


Figure 92 Compute Point Mass Trajectory Block of PANTHERA

Compute Point Mass Phase Trajectory block has only two blocks. One computes the phase state derivatives and the second one integrates the step. To compute the state derivatives, Compute Point Mass Phase State and Derivatives block computes forces and use forces to compute the accelerations.

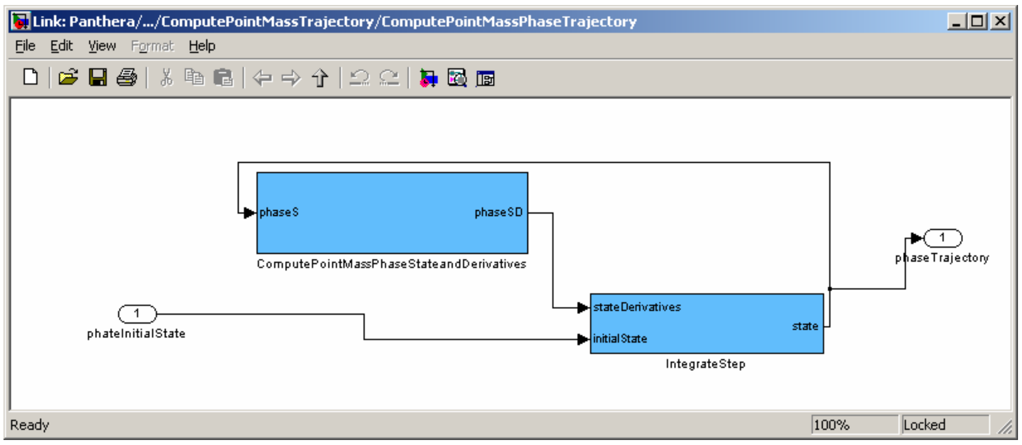

Link: Panthera/.../ComputePointMassTrajectory/ComputePointMassPhaseTrajectory
File Edit View Format Help
phaseS
phaseSD
ComputePointMassPhaseStateandDerivatives
1
phateInitialState
stateDerivatives
initialState
state
IntegrateStep
1
phaseTrajectory
Ready
100%
Locked

Figure 93 Compute Point Mass Phase Trajectory Block

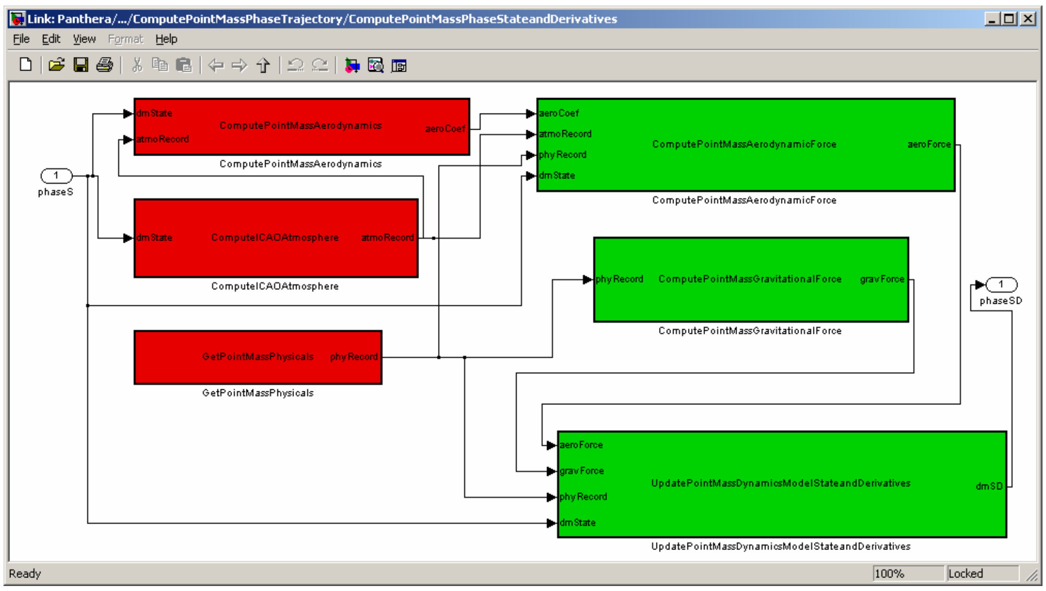
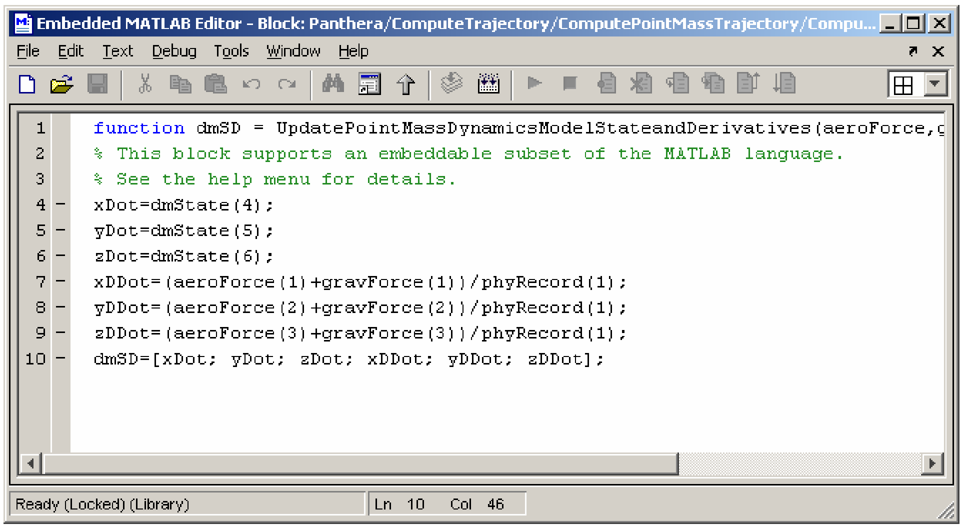

Link: Panthera/.../ComputePointMassPhaseTrajectory/ComputePointMassPhaseStateandDerivatives
File Edit View Format Help
1
phaseS
dmState
atmoRecord
ComputePointMassAerodynamics
aeroCoef
ComputePointMassAerodynamics
dmState
ComputeICAOAtmosphere
atmoRecord
ComputeICAOAtmosphere
GetPointMassPhysicals
phyRecord
GetPointMassPhysicals
aeroCoef
atmoRecord
phyRecord
dmState
ComputePointMassAerodynamicForce
aeroForce
ComputePointMassAerodynamicForce
phyRecord
ComputePointMassGravitationalForce
gravForce
ComputePointMassGravitationalForce
1
phaseSD
aeroForce
gravForce
phyRecord
dmState
UpdatePointMassDynamicsModelStateandDerivatives
dmSD
UpdatePointMassDynamicsModelStateandDerivatives
Ready
100%
Locked

Figure 94 Compute Phase Point Mass State and Derivatives Block

All the blocks used in the blockset are MATLAB Embedded Function Blocks. They are implemented by developing functions in MATLAB Scripting language. These functions conform to the interface requirements.


Embedded MATLAB Editor - Block: Panthera/ComputeTrajectory/ComputePointMassTrajectory/Compu...
File Edit Text Debug Tools Window Help
Ready (Locked) (Library)
Ln 10 Col 46

```
function dmSD = UpdatePointMassDynamicsModelStateandDerivatives(aeroForce,g
% This block supports an embeddable subset of the MATLAB language.
% See the help menu for details.
xDot=dmState(4);
yDot=dmState(5);
zDot=dmState(6);
xDDot=(aeroForce(1)+gravForce(1))/phyRecord(1);
yDDot=(aeroForce(2)+gravForce(2))/phyRecord(1);
zDDot=(aeroForce(3)+gravForce(3))/phyRecord(1);
dmSD=[xDot; yDot; zDot; xDDot; yDDot; zDDot];
```

Figure 95 Update Point Mass Dynamic Model State and Derivatives Block

As an example let us look at Update Point Mass Dynamic Model State and Derivatives block. The implemented code in MATLAB m file is given above in Figure 95. One can have a look the whole implementation which is given in APPENDIX N.

5.2.4 Notes on PANTHERA Implementation

PANTHERA uses an earth fixed reference frame, $\Im_E(X,Y,Z)$. Its origin is taken to be fixed to the earth's surface with its X axis pointing towards north, Y axis pointing toward up and Z axis pointing towards east. Non-rotating and flat earth assumptions are used. Hence the earth fixed reference frame is assumed to be inertial.

Update Point Mass Dynamics Model State and Derivatives implements a point mass dynamics model. Acceleration of the munition is computed as follows [4]:

$$\dot{\vec{V}} = \frac{\Sigma\vec{F}}{m} \qquad \text{Eq. 147}$$

where

$$\Sigma\vec{F} = \vec{D} + \vec{G} \qquad \text{Eq. 148}$$

Only aerodynamics and gravitational forces are taken into account. Compute Point Mass Aerodynamics Force computes a drag force. Drag force is computed using the following equation.

$$\vec{D} = 0.5\rho V^2 C_D S\vec{u}_v \qquad \text{Eq. 149}$$

where

$$\vec{u}_v = \frac{\vec{V}}{V} \qquad \text{Eq. 150}$$

The gravitational force is computed by Compute Point Mass Gravitational Force block assuming constant gravitational acceleration as follows:

$$\vec{G} = m\vec{g} \tag{Eq. 151}$$

where

$$\vec{g} = \begin{bmatrix} 0 \\ -9.81 \\ 0 \end{bmatrix} \tag{Eq. 152}$$

The initial state of the simulation is set by using Get Point Mass Weapon Data block which provides initial position, elevation and the azimuth of fire and Get Point Mass Charge Data that provides the muzzle velocity to the simulation.

Standard ICAO atmosphere is supported by this Blockset. ICAO atmosphere is implemented by Compute ICAO Atmosphere block uses the values presented in Table 2.

5.2.5 Sample Blockset Implementations

To show how one can use PANTHERA to develop a simulation, we will present two examples. TIGER and JAGUAR. They both use the same data set but the way they use the block set differs. The data used for these simulations will be given in APPENDIX M. The implementations are given in APPENDIX N.

TIGER uses the top most block to develop the simulation. MATLAB Simulink block diagram is given below.

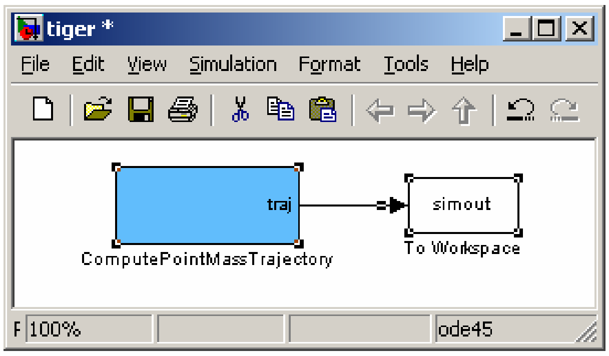

tiger *
File Edit View Simulation Format Tools Help
traj
ComputePointMassTrajectory
simout
To Workspace
100%
ode45

Figure 96 TIGER Block Diagram

JAGUAR on the other hand uses the low level function blocks to implement the same simulation. Besides, some functionality in JAGUAR is developed by the developer like Get Jaguar Weapon Data service. Figure 97 depicts the block diagram of JAGUAR.

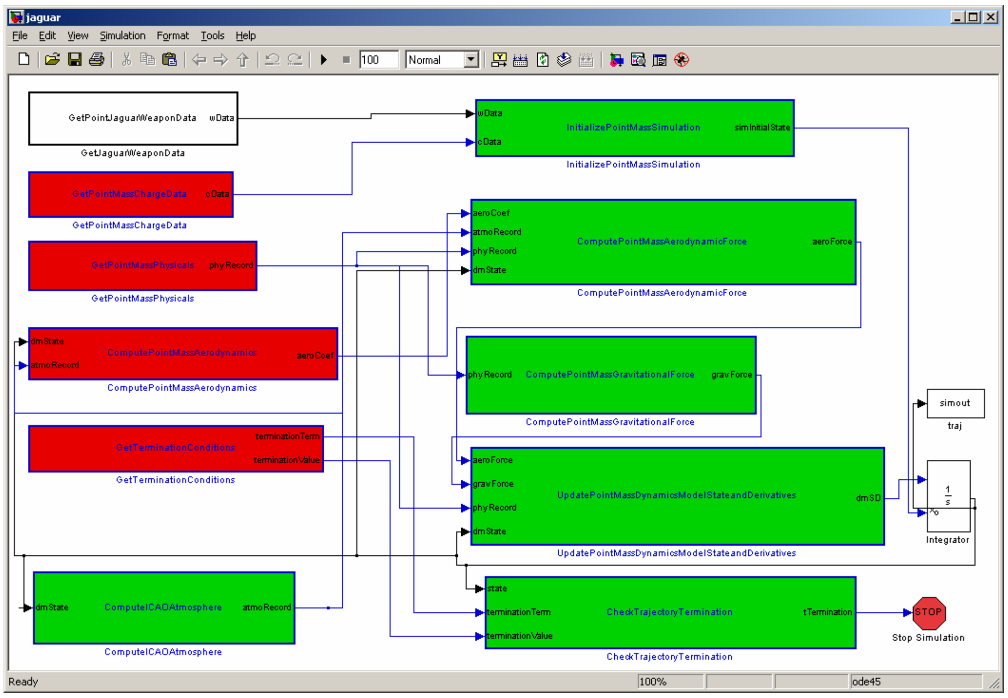


Figure 97 JAGUAR Block Diagram

The data set used for TIGER and JAGUAR is based on 81mm mortar. Below is a range versus altitude graph from a sample run of TIGER for 800 mils elevation.

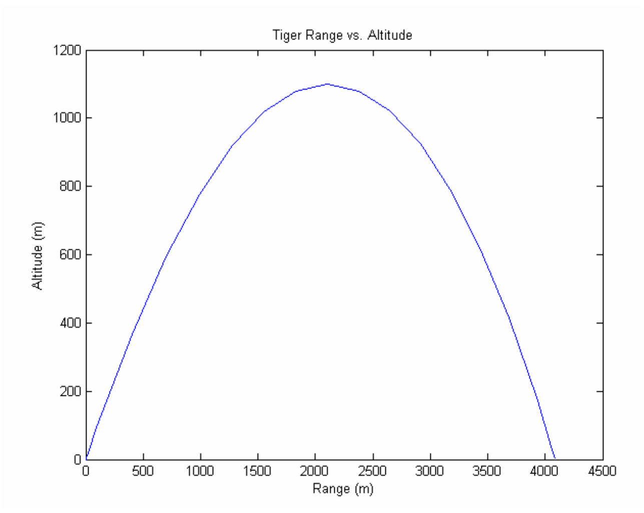


Figure 98 Range vs. Altitude for a Sample TIGER Run

This chapter presented trajectory simulation reuse infrastructure that was developed for the object oriented and function oriented programming paradigms. Platform independent reusable designs are discussed and the platform and problem family specific designs and reusable codes are introduced as case studies. Sample applications built upon these reusable codes are introduced. Next chapter will consist of discussions on the results of this reseach.

CHAPTER 6

CONCLUSION

In this research, we developed an ontology based reuse infrastructure for trajectory simulations and investigated the use of ontologies and domain engineering practices to develop a formalized methodology to make use of the experience and knowledge leveraged from the past trajectory simulation projects. Trajectory simulation is defined as a computational tool to calculate the flight path and other parameters of munition like its orientation or angular rates during its operation. To develop a trajectory simulation, one requires mechanical engineering, modeling and simulation and software engineering body of knowledge. In this thesis, engineering knowledge in the mentioned areas that is needed to simulate the trajectory of a munition is captured in an ontology called TSONT. TSONT consists of the concepts of trajectory simulation and the relation among these concepts. Then TSONT is presented as a knowledge library that is available for reuse. It is the domain model of the reuse infrastructure.

After formalizing the domain knowledge for reuse, we concentrated on building an infrastructure to enable the reuse of software artifacts. Two main programming practices were considered when developing an infrastructure. Object oriented programming and function oriented programming. We used "platform independent model" and "platform specific model" concepts to present the specification of the reuse infrastructure. It enabled us to present a specification of trajectory simulation in a platform independent fashion to enable reuse for different platforms like MATLAB or Java and in a platform specific way to construct a detailed design for a specific platform. We make use of UML and application frameworks when constructing an object oriented infrastructure. First, a platform independent framework architecture is constructed. Then, two different trajectory simulation frameworks are designed using this abstract design. The same abstract design is reused by two different platform specific designs. This is presented as evidence of abstract design reuse enabled by the infrastructure. MATSIX, which is one of these two frameworks, is developed. Then, two different trajectory simulations are developed using framework completion which is a formal reuse practice of application frameworks. This showed the code reuse capabilities of the infrastructure. With these two simulations, we presented all the way through from knowledge reuse to code reuse in object oriented paradigm.

Data flow diagrams are used to formalize the design of the function oriented simulations to compute the trajectory of munition. A platform independent design is constructed for a point mass unguided trajectory simulation using TSONT. As we used TSONT for both in object oriented framework design and function oriented simulation design, we had a chance to speak out the evidence of cross paradigm reuse of the knowledge captured in TSONT about how to develop trajectory simulations. A MATLAB Simulink Blockset is developed using the design presented in data flow diagrams. Point mass mortar simulations are developed using this Blockset as case studies. With these mortar simulations, we again presented all the way through from knowledge reuse to code reuse, this time in function oriented paradigm.

In this research, we had the chance to show that ontologies can be a useful instrument for knowledge sharing and reuse. While developing TSONT, we experienced the construction of an ontology for a real-life industrial application. As we started to use TSONT for specification of reuse infrastructure, we had a chance to see the practical role of ontologies as mechanisms for knowledge sharing and reuse.

One of the biggest challenges that we had to overcome as we developed TSONT for a real-life industrial application was its scale. As TSONT get bigger and bigger, it became harder to resolve the complex relations among the concepts of trajectory simulation. We used an iterative approach to ontology development which enabled us to reconsider all the structure as we tried to capture new bunch of knowledge in TSONT. The second challenge to be mentioned here was in determining the scope and structuring TSONT. TSONT, as mentioned, is neither a complete nor has the only correct structure to capture trajectory simulation domain knowledge. It is scoped reflecting the experience of target reuse group. Rather than an effort to capture all the available knowledge on trajectory simulation in literature, it is aimed to formalize what is available among target reuse group. The knowledge is structured in a way that the target reuse group abstracts the trajectory simulation domain. Here another challenge arises. As the organization that uses the ontology for knowledge sharing evolves, the shared vocabulary and shared conceptualization also evolves, so ontology needs an active maintenance effort.

To develop an ontology that will enable an organization wide knowledge sharing and reuse, it should be institutionalized and owned by all shareholders. During the development of TSONT, we tried to construct this sense of ownership by using peer review mechanism. As TSONT evolved during this research, peer reviews are handled with the target reuse group to align the conceptualization in a collaborative manner.

Ontology specifies the shared conceptualization in a formal way that enable human and machine readability. As we practiced the role of ontologies as mechanisms for knowledge sharing and reuse, we used and presented the human readability of the TSONT. We reused ontology to construct two different abstract software designs to developed trajectory simulations. This reuse processes were human in the loop type. We read TSONT and reflected the concepts and the relations among these concepts captured in the ontology to software design constructs like classes and associations or functions and functional flows. Besides these case studies that are presented in this thesis, we also experienced automated means of reuse of TSONT to construct abstract software design. Two collaborative research efforts have been carried out with M.Sc. students from Computer Engineering Department on model driven engineering practices that will enable us to transform machine readable TSONT to a software design using model transformation. One concluded with the transformation of TSONT to UML class diagram that represents an abstract object oriented design. The second one concluded with transformations of TSONT to MATLAB Simulink blocks. These two efforts gave us strong clues that show the promise on successful use of model transformation practices with this ontology based approach.

While this research is not the first time that ontologies are used in mechanical engineering, it is one of the small numbers of studies going on about using ontologies for knowledge sharing and reuse in mechanical engineering. For its specific focus, this research is a frontier in using ontology in the field of trajectory simulation. Besides, in the field of modeling and simulation, this research is one of the first studies that try to formalize the domain knowledge in a form of ontology and make it available for developing simulations. These efforts are called ontology driven simulation. The extensions of this research on combining ontology driven simulation with model transformation practices are avant-garde. Their preliminary results are exciting in a way they show new horizons on automatic transformation of domain knowledge to executable simulations.

Ontology based reuse infrastructure for trajectory simulations is composed of a domain model, an infrastructure definition and infrastructure implementations. This definition with the methodology used is based on domain engineering practices. Use of ontologies as domain model was first pronounced in early 2000's. We based our approach on this literature and developed TSONT as our domain model. We contributed to this approach by selecting OWL as the ontology definition language. This enabled us to extent the domain engineering practices with integrating them to model driven engineering practices. It means, by using OWL, we had a chance to make use of the results of efforts on Ontology Definition Metamodel (ODM) and Model Driven Architecture (MDA) of Object Management Group (OMG). Using Meta Object Facility (MOF), future efforts on matching UML Metamodel and Ontology Definition Metamodel will be reflected to our ontology based reuse infrastructure. Tools for automatic transformation of domain model to infrastructure definition and infrastructure implementations will be available. In this research, we proposed two different levels of abstractions for infrastructure specification. It is composed of a platform independent abstract design and a platform specific detailed design. By this decision, we accomplished two different goals. First of all, we produced an abstract software design apart from platform specific details so that it can be reused for different trajectory simulations that will target different platforms. And we matched the artifacts of reuse infrastructure with the levels of MDA that will enable us to leverage the future enhancements on it. Domain model is matched with Computation Independent Model (CIM) of MDA, platform independent abstract design of infrastructure specification is matched to Platform Independent Model (PIM) of MDA and platform specific detailed design is matched to Platform Specific Model (PSM) of MDA.

Reuse attempts on trajectory simulations in literature have been focused on code and mathematical model reuse. With this research, we proposed knowledge reuse and design reuse and code reuse from the ontology based reuse infrastructure. This creates a distinction for this research in trajectory simulation reuse literature. Infrastructure, as it is, serves number of artifacts that can be reused for trajectory simulations like TSONT, Platform Independent Framework Architecture, MATSIX Framework Architecture and MATSIX Code. With these artifacts, we also proposed a methodology to produce reusable artifacts in future projects. As target reuse group gains experience with the future projects, this experience can be formalized by enhancing TSONT. The enhancements in TSONT can be reflected to Platform Independent Framework Architecture to enable design reuse. As new frameworks developed for different projects targeting different platforms and problem sets, infrastructure developed in this thesis can be expanded by adding new framework architectures and framework implementations.

The scope of this thesis is bounded to focus on trajectory simulation reuse targeting the future projects. There is another spot that can be focused during a future research. That is the legacy trajectory simulations. Semantic matching of the concepts and designs implemented in those simulations to TSONT seems a promising research objective. This semantic matching will enhance the interoperability of different simulations that make use of TSONT and will also strengthen and enhance TSONT by making it refer to legacy implementations and make use of the knowledge which is transformed to a product in those projects.

Measuring reuse is as important as developing a reuse infrastructure. Institutionalizing the reuse infrastructure developed in this thesis, making it used in future projects is one of the challenges that will be handled in the post thesis period. As it is used in number of projects, it will be possible to measure reuse. It will enable us to validate the proof of the expected increase in productivity and decrease of the risk of the projects that depend on reuse infrastructure that is built in this thesis.

TSONT captures the domain knowledge about design and development of trajectory simulations. Here, there is another hard question. How can we make use of TSONT to write the requirements of a specific trajectory simulation project? Developing methodologies and tools to transform the domain knowledge captured in TSONT to software or simulation requirements is another spot that can be listed in the future research agenda of this thesis.

The reuse infrastructure in this thesis as we mentioned before focused on two main paradigms, namely object oriented programming and function oriented programming. Developing infrastructures for emerging paradigms like aspect oriented programming, actor oriented programming, agent based programming and distributed simulation can be added to future research agenda as new challenges.

As TSONT is transformed to design and code manually, there is an arguable issue about the traceability of the knowledge captured in ontology to the foregoing artifacts. In this thesis, we prepared a table that tries to capture this traceability as far as possible. But as the automatic transformation of domain knowledge to design and code is being studied, new tool or methodologies can be developed to trace the flow of the knowledge from domain to code. This is another future research challenge.

This reuse infrastructure that consists of reusable domain knowledge, designs and code can be regarded as the foundations of a trajectory simulation software product line. Evolving this infrastructure to a software product line for trajectory simulation is another future research focuss.

Trajectory simulation is one of the computational fields of mechanical engineering. It can be pronounced as an application of system dynamics. Mechanical engineering has a number of other computation intensive application areas on which number of software development is carried out. These fields include structural mechanics, computation fluid dynamics, computer integrated manufacturing, computer aided design and robotics. The methodology proposed in this thesis can be applied in those fields to make reuse work. So, developing reuse infrastructures for other fields of mechanical engineering can be proposed as a future research topic.

Besides enabling reuse in mechanical engineering software, ontologies may work as a glue to enable different engineering software work together to accomplish a goal. Ontologies that will be developed for different computer aided engineering tools may enable to develop collaborative design environments and integrate the design and manufacturing processes seamlessly. Interoperability using ontologies in the field of mechanical engineering is another future research direction.

Formalizing mechanical engineering body of knowledge is not only helpful on reuse and interoperability but can also be used for problem solving. Ontology based problem solving methodologies for mechanical engineering problems seem to be an interesting research direction.

In this thesis, we developed and used ontology in trajectory simulation problem of mechanical engineering. There is still lot to do either for trajectory simulation problem or other problems of the mechanical engineering by using ontologies.

Besides all future research topics mentioned above, it is time to remember the quote of J.R.R. Tolkien. "There is nothing like looking, if you want to find something. You certainly usually find something, if you look, but it is not always quite the something you were after."

REFERENCES

1. Kelton, W.D., Sadowski, R.P. and Sadowski D.A., *Simulation with Arena,* WCB/McGraw-Hill, Boston, 1998.

2. Matko, D., Karba, R. and Zupancic, B., *Simulation and Modeling of Continious Systems – A Case Study Approach,* Prentice Hall International (UK) Ltd, New York, 1992.

3. Neelamkavil, F., *Computer Simulation and Modeling*, John Wiley, London, 1987.

4. *MIL-HDBK 1211 Missile Flight Simulation Part One Surface-to-Air Missiles*. U.S. Department of Defense, 1995.

5. Zipfel, P.H., *Modeling and Simulation of Aerospace Vehicle Dynamics*, American Institute of Aeronautics and Astronautics, Inc., Reston, 2000.

6. *Modeling and Simulation – Linking Entertainment and Defense*, National Academy Press, Washington D.C., 1997.

7. *EADSIM Executive Summary*, URL: http://www.eadsim.com/EADSIMExecSum.pdf, 19 June 2007.

8. Korkmaz, S., Mahmutyazicioglu, G., Tiftikci, H. and Gokhan, T., *FMCAD - Flight Mechanics Computer Aided Design Software,* 37th Aerospace Sciences Meeting and Exhibit, Reno, NV, 1999.

9. Rea, M. M., Baird, A. M., Batchelder, F. E., Belrose, F. M. and Holt, W. C., *Missile System Simulation at the Advanced Simulation Center*, Technical Report RD-82-11, Systems Simulation and Development Directorate, Advanced Simulation Center, US Army Missile Laboratory, US Army Missile Command, Redstone Arsenal, AL, 1982.

10. Fleeman, E.L., *Tactical Missile Design*, AIAA Education Series, Reston, VA, 2001.

11. *MIL-HDBK 799 Fire Control Systems – General,* U.S. Department of Defense, 1996.

12. Sowa J.A., Lieske, R.F., Matts J.A. and Miller J.L., *Ballistic Kernels Sharable Fire Control and Ballistic Simulation Software*, Firing Tables and Aeroballistics Branch Information Report, FTAB-IR-32, 1997.

13. Durak,U., Dayanç, K., Elaldı, F., Anlağan, Ö., *Topçu Roketlerinin Atış Kontrol Sistemleri için Yeni Nesil Balistik Çözücü* USMOS 2005, Ankara, 2005.

14. Aytar Ortaç, S., Durak, U., Kutluay, Ü., Küçük, K. and Candan, C., *NABK Based Next Generation Ballistic Table Toolkit*, 23rd International Symposium on Ballistics, Tarragona, Spain,2007.

15. Zarchan, P., *Tactical and Strategic Missile Guidance*, Vol. 157, Progress in Aeronautics and Astronautics, AIAA, Washington DC, 1994.

16. Shampine, L.F., *Numerical Solution of Ordinary Differential Equations*, Champman & Hall Inc., New York, 1994.

17. McCoy, R.L., *Modern exterior Ballistics: The Launch and Flight Dynamics of Symetric Projectiles*, Schiffer Publishing Ltd, Atglen, PA, 1999.

18. Jackson, E.B., *Results of a Flight Simulation Software Methods Survey*, AIAA Flight Simulation Technologies Conference, Baltimore, MD, 1995.

19. Durak, U., Oğuztüzün, H. and İder, S.K., *Ontology Based Trajectory Simulation Framework*, Journal of Computing and Information Science in Engineering -ACCEPTED-

20. Griss, M.L.; *Software Reuse: Architecture, Process and Organization for Business Success,* Technology of Object-Oriented Languages, TOOLS 26. Proceedings, 1998.

21. Tracs, W., *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*, Addison Wesley Publishing Company, Reading, Massachusetts, 1995.

22. Lieske, R.F., Reiter, M.L., *Equations of Motion for a Modified Point Mass Trajectory*, Ballistic Research Laboratories, Report No.1314, 1966.

23. *STANAG. 4355 The Modified Point Mass and Five Degrees Of Freedom Trajectory Models*, Draft Edition 5.0A, 2003.

24. Kaplan, J.A., Chappell, A.R. and McManus. J.W., *The Analysis of a Generic air-to-Air Missile Simulation Model*. NASA TM-109057, 1994.

25. Gorecki, R.M., *A Baseline 6 Degree of Freedom (DOF) Mathematical Model of Generic Missile*, DSTO System Sciences Laboratory, DSTO-TR-0931, 2003.

26. Kalaver, S. A., Pritchett, A.R., *Development of Generic Dynamic Models in Aerospace Simulation*, AIAA Modeling and Simulation Technologies Conference and Exhibit, 2005.

27. McCarthy, T.R., Campbell, T.T., and Moseley, P.E., *A Generic Common Simulation Framework based Starting Point for Missile 6DOF Simulations*, AIAA Modeling and Simulation Technologies Conference and Exhibit, 2005.

28. Berndt, J.S., *JSBSim: An Open Source Flight Dynamics Model in C++,* AIAA Modeling and Simulation Technologies Conference and Exhibit, 2004.

29. *Aerospace Blockset User's Guide*, The MathWorks, Inc., 2005.

30. Durak, U., Oğuztüzün, H. and Mahmutyazıcıoğlu, G., *Domain Analysis for Reusable Trajectory Simulation*, Euro-SIW 2005, Toulouse, France, 2005.

31. Durak, U., Oğuztüzün, H. and İder, K., *An Ontology for Trajectory Simulation*, WinterSim06, Monterey, CA, USA, 2006.

32. Durak, U., Güler, S., Oğuztüzün, H., and İder, K., *An Exercise In Ontology Driven Trajectory Simulation with MATLAB Simulink*, 21st EUROPEAN Conference on Modelling and Simulation, Prague, Czech Republic, 2007.

33. Özdikiş, Ö, Oğuztüzün, H, and Durak, U., *OWL to UML : Transforming Domain Models to Framework Architectures*, Manuscript.

34. Mili, M., Mili A., Yacoub, S., and Addy, E., *Reuse Based Software Engineering*, John Wiley & Sons Inc., 2002.

35. Arango. G., *Domain Analysis: From Art to Engineering Discipline*, Proceedings of 5th International Workshop on Software Specification and Design, 1989.

36. McIlroy, M.D., *Mass-Produced Software Components*, Proceedings of the NATO Conference on Software Engineering, 1969

37. Favaro, J., *Technical Report on Reuse*, European Software Institute, 1995.

38. Sodhi J., Sodhi P., *Software Reuse: Domain Analysis and Design Process*, McGraw-Hill, 1999.

39. *Reuse-Driven Software Processes Guidebook*, Tech. Report SPC-92019-CMC, Software Productivity Consortium, Herndon, VA., 1993.

40. White, S., Edwards, M., *Domain Engineering: The Challenge, Status, and Trends*, IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96), 1996.

41. Arango, G., Prieto-Diaz, R., *Domain Analysis Concepts and Research Directions*, in Domain Analysis and Software Systems Modeling, IEEE Computer Society Press, 1991.

42. *Proceedings of Third International Conference on Software Reuse, Advances in Software Reusability*, IEEE Computer Society Press, Los Alamitos, CA., 1994.

43. *Systematic Reuse, Theme Issue of IEEE Software*, Vol. 11, No. 5, IEEEE Computer Society Press, Los Alamitos, CA., 1994.

44. Palmer, C., *A CAMP Update*, AIAA-1989-3144, 7th AIAA Computers in Aerospace Conference, Monterey, CA, 1989.

45. Diaz, R.P., *Domain Analysis: An Introduction*, ACMSIG Software Engineering Notes, 1990.

46. Arango, G., *Domain Analysis Methods*, in Software Reusability, Editors: W. Schaefer, R . Prieto-Diaz, and M. Matsumoto, Ellis Horwood, New York, 1994.

47. Falbo, R.A., Guizzardi, G. and Duarte. K.C., *An Ontological Approach to Domain Engineering,* International Conference on Software Engineering and Knowledge Enginnering, Ischia, Italy, 2002.

48. Falbo, R.A., Guizzardi, G., Duarte. K.C., Candida, A., and Natali, C., *Developing Software for and With Reuse: An Ontological Approach.* ACIS International Conference on Computer Science, Software Engineering, Information Technology, e-Business and Applications, Brazil, 2002.

49. Calero C., Ruiz, F., Piattini, M., *Ontologies for Software Engineering and Software Technology*, Springer-Vergal Berlin Heidelberg, 2006.

50. Knublauch, H., *Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protégé/OWL*, International Workshop on the Model-Driven Semantic Web, Monterey, CA, 2004.

51. Neighbors, J., *Software Construction Using Components*, Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1980.

52. *Webster's New World Dictionary of American English*, 3rd Collage Edition, Prentice Hall, New York, 1993.

53. Uschold, M., *Knowledge Level Modelling: Concepts and Terminology*, Knowledge Engineering Review, 13 (1), 1998.

54. Falbo, R.A., Menezes, C.S., and Rocha, A.R.C., *A Systematic Approach for Building Ontologies*, in Proceedings of the IBERAMIA'98, Lisbon, Portugal, 1998.

55. Mizoguchi, R., *Ontological Engineering: Foundations of the Next Generation Knowledge Processing*, Web Intelligence 2001, Maebashi City, Japan, 2001.

56. Borst, W.N., Akkermans, J.M., Pos, A. and Top, J. L., *The PhysSys Ontology for Physical Systems*. In B. Bredeweg (Ed.), Working Papers of the Ninth International Workshop on Qualitative Reasoning QR'95, University of Amsterdam, 1995.

57. Borst, W. N., Akkermans, J. M., *Engineering Ontologies*, International Journal of Human-Computer Studies, 46 (2/3):365-406, 1997.

58. Borst, W. N., *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*, Ph.D. Thesis, University of Twente, 1997.

59. Schreiber, G., Wielinga, B., and Jansweijer, W., *The KACTUS View on the 'O' Word*, In Proceedings of IJCAI95 Workshop on Basic Ontological Issues in Knowledge Sharing. Montreal, Canada, 1995.

60. Ciocoiu, M., Gruninger, M., and Nau, D.S., *Ontologies for Integrating Engineering Applications*, Journal of Computing and Information Science in Engineering, (1) 1:12-22, 2001.

61. Stahovich, T., Davis, R. and Shrobe, H., *An Ontology of Mechanical Devices*, Working Notes, Reasoning about Function, AAAI-93, pp. 137-140, 1993.

62. Gruber, T.R., Olsen, G.R., *An Ontology for Engineering Mathematics*, Fourth International Conference on Principles of Knowledge Representation and Reasoning, Gustav Stresemann Institut, Bonn, Germany, Morgan Kaufmann, 1994.

63. Johnson, R.E., *Components, Frameworks, Patterns*, ACM SIGSOFT Symposium on Software Reusability, 1997.

64. Fayad, M., Schmidt, D.C., *Object-Oriented Application Frameworks*. Communications of the ACM, 40(10):32-38, 1997.

65. Sommerville, I., *Software Engineering*, Addison Wesley Longman Publishing, Redwood City, CA, USA, 1995.

66. Neches, R., Fikes, R.E., Finin, T., Gruber, T.R., Senator, T. and Swartout W.R., *Enabling Technology for Knowledge Sharing*, AI Magazine, 12 (3):36-56, 1991.

67. Gennari, J.H., Musen, M.A., Fergerson, R.W., Grosso, W.E., Crubzy, M., Eriksson, H., Noy, N.F., and Tu, S.W., *The Evolution of Protégé: An Environment for Knowledge-Based Systems Development,* International Journal of Human-Computer Studies, 58 (1):89-123, 2003

68. Gruber, T.R., *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*, Technical Report KSL 93-04, Knowledge Systems Laboratory, Stanford University, 1993.

69. Struder. R., Benjamins, V.R., Fensel, D., *Knowledge Engineering: Principles and Methods*, IEEE Transactions on Knowledge and Data Engineering 25 (1-2):161-167, 1998.

70. Gruber, R., *A Translational Approach to Portable Ontology Specifications*, Knowledge Acquisition, 5(2), 1993.

71. Corcho, O, Perez, A.G., *Evaluating Knowledge Representation and Reasoning Capabilities of Ontology Specification Languages*, ECAI'00 Workshop on Applications of Ontologies and Problem Solving Methods. Berlin, Germany, 2000.

72. McGuinness, D., van Harmelen, F. (eds) *OWL Web Ontology Overview*, URL: http://www.w3.org/TR/2003/WD-owl-features-20030331/, 19 June 2007.

73. Dean, M., Schreiber, G., (eds), van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P. and Stein, L., *OWL Web Ontology Language Reference*, URL: http://www.w3.org/TR/2003/WD-owl-ref-20030331/, 19 June 2007.

74. Smith, M., Welty, C. and McGuinness, D., *OWL Web Ontology Language Guide*, URL: http://www.w3.org/TR/2003/WD-owl-guide-20030331/, 19 June 2007.

75. Antoniou, G., van Harmelen, F., *Web Ontology Language: OWL*, Handbook on Ontologies, International Handbooks on Information Systems, Springer, 2004.

76. Horridge, M., Knublauch, H., Rector, A., Stevens, R. and Wroe, C., *A Practical Guide To Building OWL Ontologies Using The Protégé - OWL Plugin and CO-ODE Tools Edition 1.0.*, URL: http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf, 19 June 2007.

77. Jackson, E., Hildreth, B., York, B. and Cleveland, W., *Evaluation of a Candidate Flight Dynamics Model Simulation Standard Exchange Format*,

AIAA Modeling and Simulation Technologies Conference and Exhibit, Providence, Rhode Island, 2004.

78. AIAA Simulation Standards Working Group, *Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) Reference, Version 1.7b1*, URL: http://daveml.nasa.gov/DTDs/1p7b1/DAVE-ML_ref.pdf, 19 June 2007.

79. Niles, I., Pease, A., *Towards a Standard Upper Ontology*, In Proceedings of the 2nd International Conference on Formal Ontology in Information Sys-tems (FOIS-2001), Chris Welty and Barry Smith, eds, Ogunquit, Maine, 2001.

80. Fitzgerald, J., Larsen, P.G., *Modelling Systems: Practical Tools and Techniques in Software Development*, Cambridge University Press, 1998.

81. *DOD Department of Defense Dictionary of Military and Associated Terms*, Joint Publication 1-02, 2001.

82. *AOP-29 NATO Indirect Fire Ammunition Interchangeability,* Allied Ordinance Publications 29, 1999.

83. *DoD 101 – Introduction to Millitary*, URL: http://www.fas.org/man/dod-101/index.html, 19 June 2007.

84. *Model Designation of Military Aerospace Vehicles*, Department of Defense Publication 4120.15-L, 2004.

85. *STANAG. 6022 Adoption of a Standard Gridded Data Meteorological Message*, Draft Edition 1.0, 2007.

86. *STANAG 4082 LAND Adoption of a Standard Artillery Computer Meteorological Message*, Edition No.2, 1969.

87. *STANAG 4061 LAND Adoption of a Standard Ballistic Meteorological Message*, Edition No.4, 2000.

88. *Manual of the ICAO Standard Atmosphere (extended to 80 kilometres (262 500 feet))*, ICAO Doc 7488-CD, Third Edition, 1993.

89. *AOP-37 NATO Armaments Ballistic Kernel (NABK) Library*, Allied Ordinance Publications 37, 2007.

90. *Tactics, Techniques and Procedures for Field Artillery Manual Gunnery*, Field Manual No. 6-40, Marine Corps Warfighting Publication No 3-1.6.19, Headquarters Department of the Army, U.S. Marine Corps, 1996.

91. Blakelock, J. H. , *Automatic Control of Aircraft and Missiles*, John Wiley and Sons Publications, 1991.

92. Tiryaki, K., *Polynomial Guidance Laws and Dynamic Flight Simulation Studies*, M.Sc. Thesis, Middle East Technical University, Türkiye, 2002.

93. Mahmutyazıcıoğlu, G., *Dynamics and Control Simulation of an Inertially Guided Missile*, M.Sc. Thesis, Middle East Technical University, Türkiye, 1994.

94. Özkan, B., *Dynamics Modeling, Guidance and Control of Homing Missiles*, Ph.D. Thesis, Middle East Technical University, Türkiye, 2005.

95. *The International System of Units,* 8th Edition, Organisation Intergouvernementale de la Convention du Mètre, 2006.

96. Watt, D.A., *Programming Language Concepts and Paradigms*, Prentice Hall Series in Computer Science, 1990.

97. Chen X., *Developing Application Frameworks in .NET*, APress, 2004.

98. Aksit, M., Marcelloni, F. and Tekinerdogan, B., *Developing Object-Oriented Frameworks Using Domain Models*, ACM Computing Surveys, 2000.

99. Robert, D., Johnson, R., *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, Pattern Languages of Program Design 3, Addison Wesley, 1997.

100. Perry, D.E., Wolf, A.E., *Foundations for the Study of Software Architectures*, Software Engineering Notes, (17)40, 1992.

101. Booch, G., Rumbaugh, J. and Jakobsen, I., *The Unified Modeling Language User Guide*, Addison-Wesley, Boston, 1999.

102. *Enterprise Architect 6.5 Reviewer's Guide*, URL: http://www.sparxsystems.com.au/downloads/whitepapers/Reviewers_Guide_EA_6_5.pdf, 19 June 2007.

103. *Public Domain Aeronautical Software*, URL: http://www.pdas.com/m1.htm, 19 June 2007.

104. *MATLAB Technical Documentation*, URL: http://www.mathworks.com/access/helpdesk/help/helpdesk.html, 19 June 2007.

105. Liberty, J., *Programming C#*, O'Reilly, 2002.

106. Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipies in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

107. Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipies in FORTRAN 77: The Art of Scientific Computing*, Cambridge University Press, 1992.

108. Lee, E.A., *Model-Driven Development - From Object-Oriented Design to Actor-Oriented Design*, Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, Chicago, 2003.

APPENDIX A

TSONT MUNITION TAXONOMY

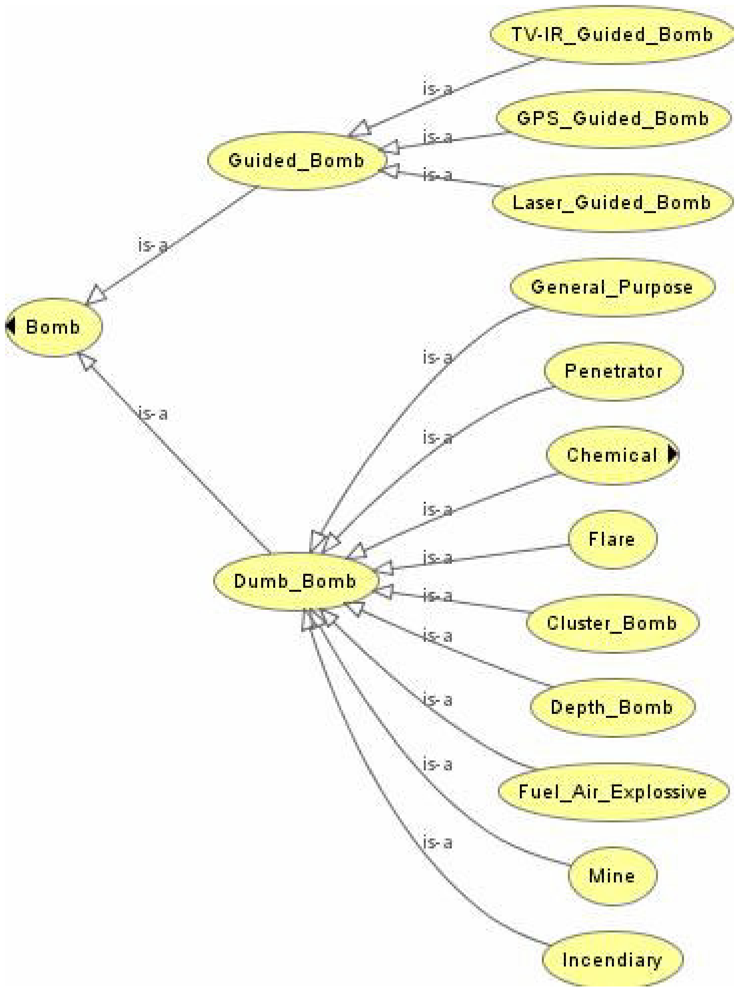


Figure 99 TSONT Bomb Classification

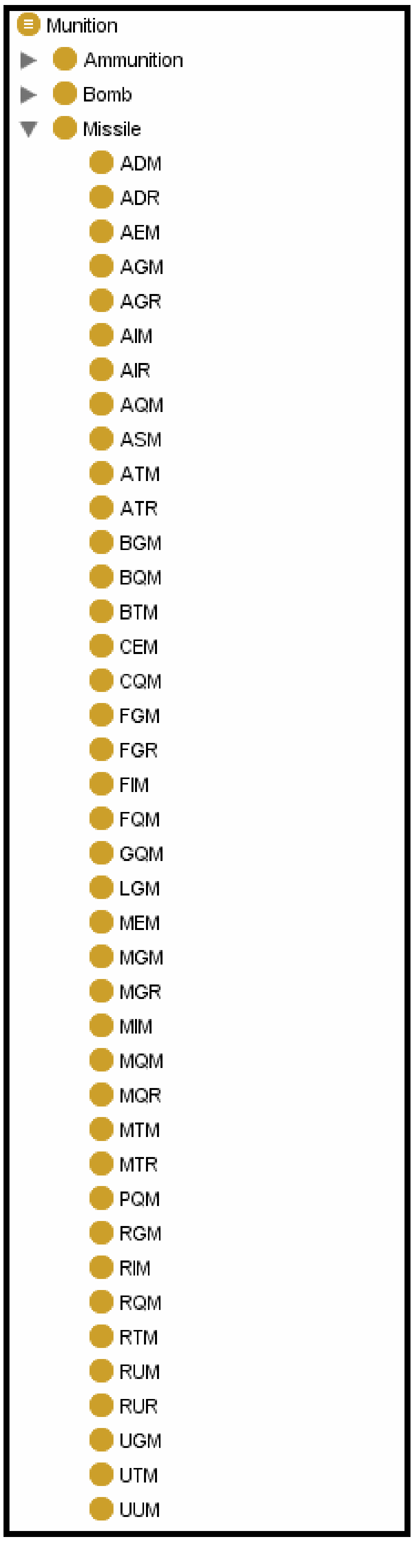

Munition
- Ammunition
- Bomb
- Missile
  - ADM
  - ADR
  - AEM
  - AGM
  - AGR
  - AIM
  - AIR
  - AQM
  - ASM
  - ATM
  - ATR
  - BGM
  - BQM
  - BTM
  - CEM
  - CQM
  - FGM
  - FGR
  - FIM
  - FQM
  - GQM
  - LGM
  - MEM
  - MGM
  - MGR
  - MIM
  - MQM
  - MQR
  - MTM
  - MTR
  - PQM
  - RGM
  - RIM
  - RQM
  - RTM
  - RUM
  - RUR
  - UGM
  - UTM
  - UUM

Figure 100 TSONT Missile Classification

The letters of the acronyms in the missile classification is as follows. The first letter is the launch environment of the vehicle:

A - Air

B - Multiple

C - Coffin (non-hardened container)

F - Individual

G - Runway or Ground

H - Silo Stored

L - Silo Launched

M - Ground Launched, Mobile

P - Soft Pad

R - Surface Ship

S - Space

U - Underwater

Second letter is the purpose of the vehicle:

C - Transport

D - Decoy

E - Special Electronics, Communication

G - Surface Attack

I - Interception

L - Launch Detection

M - Scientific Measurements, Calibration

N - Navigation

Q - Drone, UAV

S - Space Operations Support

T - Training

U - Underwater Attack

W - Weather (probes or satellites gathering and/or distributing meteorological data)

The last letter defines the type of vehicle:

B - Booster

M - Guided Missile, Drone, UAV

N - Probe (suborbital sounding rocket)

R - Rocket (unguided vehicle)

S - Satellite

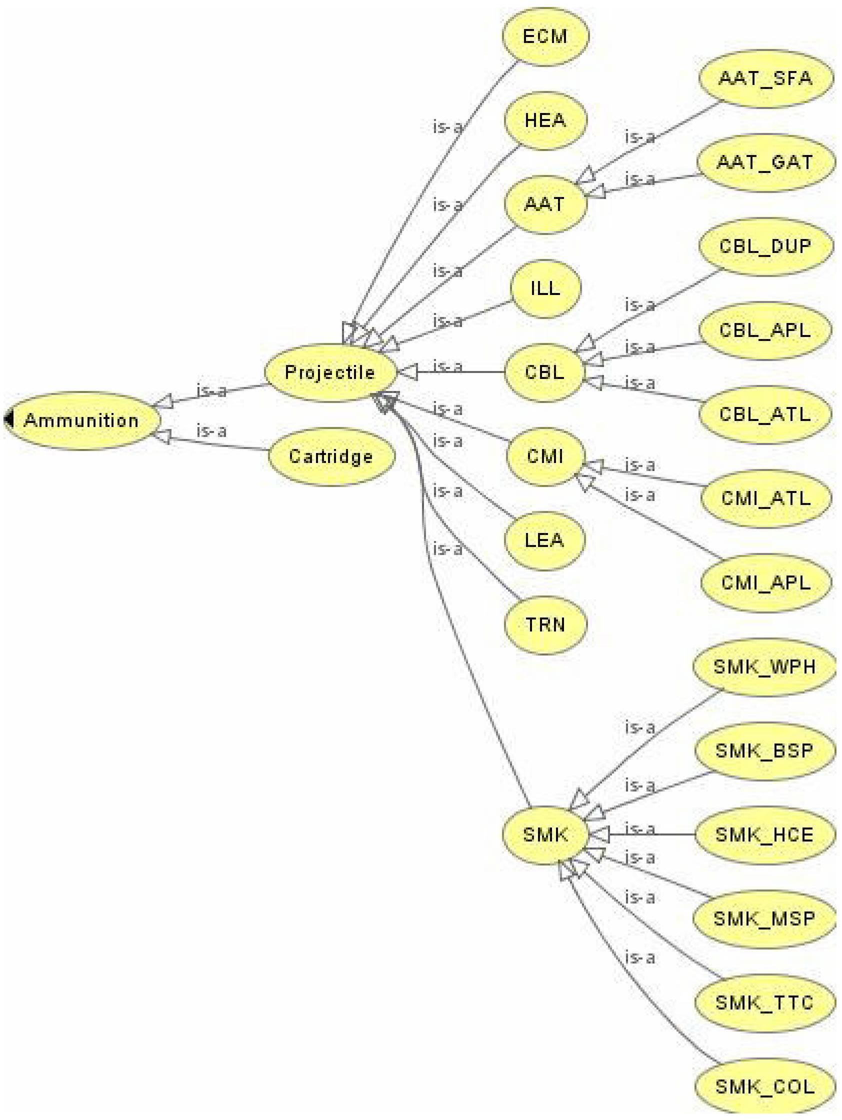


Figure 101 TSONT Ammunition Classification

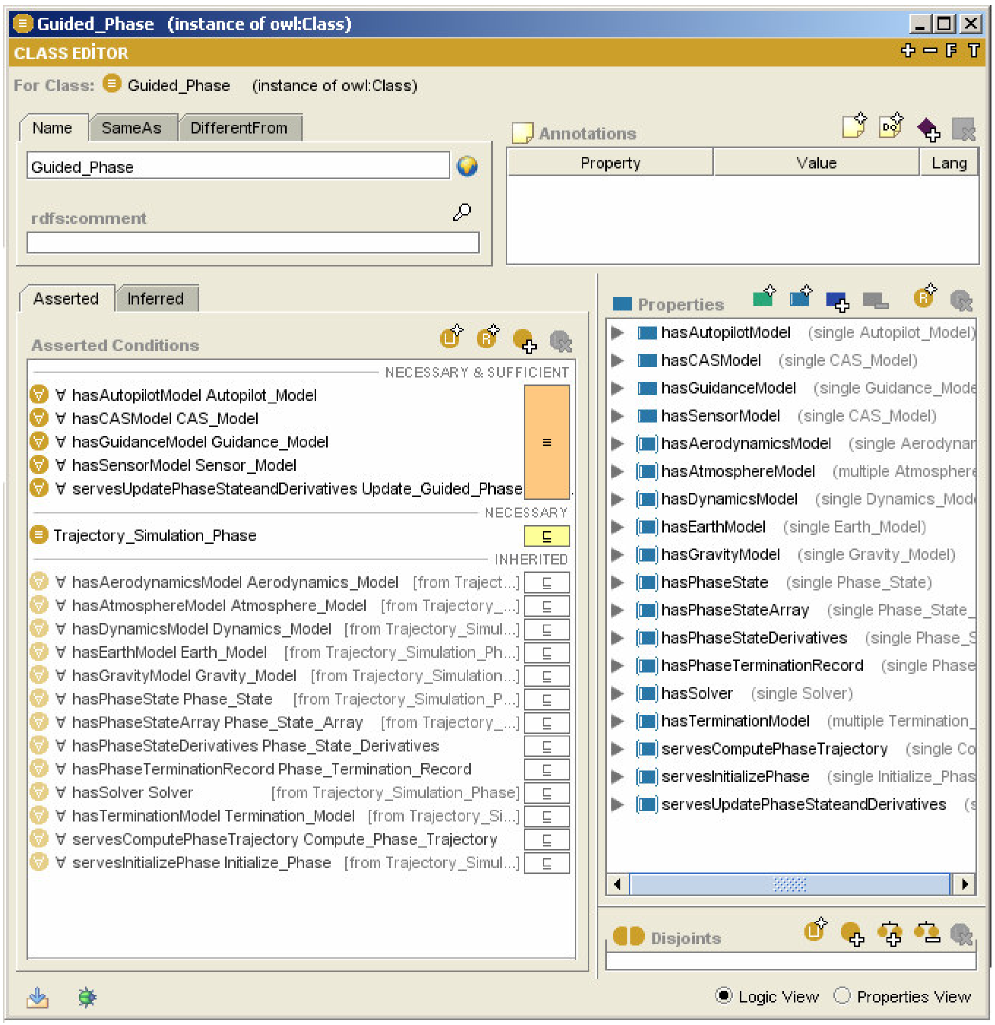
The acronyms presented above in the taxonomy are as follows.

Table 3 Acronyms of Ammunition Classification

Projectile		Description
Type	Subtype	
HEA		High Explosive Round
SMK	HCE	Hexa-Chlorethane
	WPH	White Phosphorus
	TTC	Titanium Terra Chloride
	MSP	Multi-Spectral
	COL	Colored
	BSP	Bi-Spectral
ILL		Illumination Round
CBL	APL	Bomblet – Antipersonnel
	ATK	Bomblet – Antitank
	DUP	Bomblet - Dual Purpose
CMI	ATK	Mines – Antitank
	APL	Mines – Antipersonnel
AAT	GAT	Guided Antitank
	SFA	Sensor Fuzed
ECM		Electronic Countermeasure
LEA		Leaflet
TRN		Training
OTH		Other type of projectiles

APPENDIX B

SAMPLE CLASS DEFINITIONS FROM TSONT


Guided_Phase (instance of owl:Class)

CLASS EDITOR

For Class: Guided_Phase (instance of owl:Class)

Name: Guided_Phase

rdfs:comment

Asserted Conditions

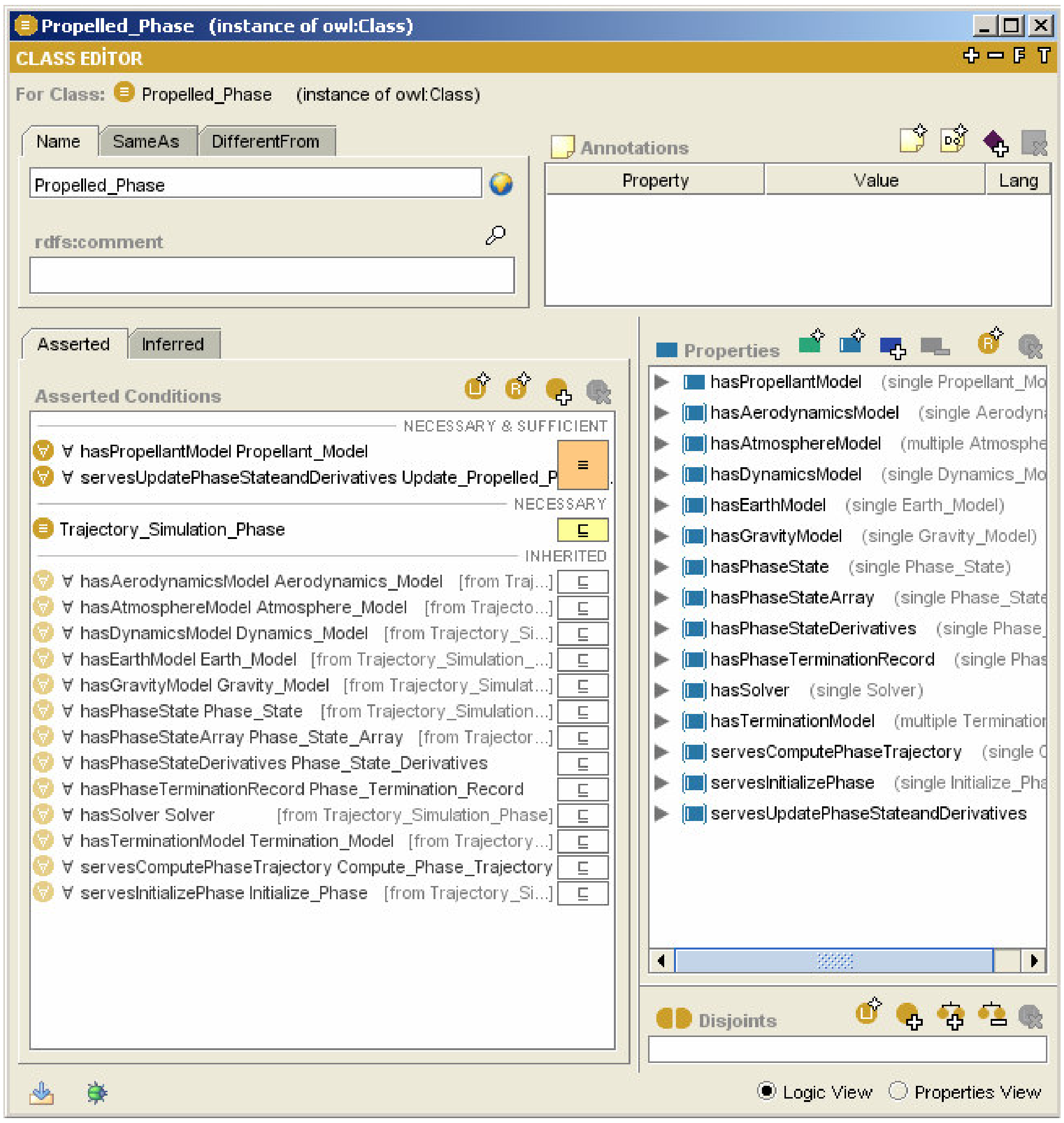
NECESSARY & SUFFICIENT
∀ hasAutopilotModel Autopilot_Model
∀ hasCASModel CAS_Model
∀ hasGuidanceModel Guidance_Model
∀ hasSensorModel Sensor_Model
∀ servesUpdatePhaseStateandDerivatives Update_Guided_Phase

NECESSARY
Trajectory_Simulation_Phase

INHERITED
∀ hasAerodynamicsModel Aerodynamics_Model [from Traject...]
∀ hasAtmosphereModel Atmosphere_Model [from Trajectory_...]
∀ hasDynamicsModel Dynamics_Model [from Trajectory_Simul...]
∀ hasEarthModel Earth_Model [from Trajectory_Simulation_Ph...]
∀ hasGravityModel Gravity_Model [from Trajectory_Simulation...]
∀ hasPhaseState Phase_State [from Trajectory_Simulation_P...]
∀ hasPhaseStateArray Phase_State_Array [from Trajectory_...]
∀ hasPhaseStateDerivatives Phase_State_Derivatives
∀ hasPhaseTerminationRecord Phase_Termination_Record
∀ hasSolver Solver [from Trajectory_Simulation_Phase]
∀ hasTerminationModel Termination_Model [from Trajectory_Si...]
∀ servesComputePhaseTrajectory Compute_Phase_Trajectory
∀ servesInitializePhase Initialize_Phase [from Trajectory_Simul...]

Properties
hasAutopilotModel (single Autopilot_Model)
hasCASModel (single CAS_Model)
hasGuidanceModel (single Guidance_Mode
hasSensorModel (single CAS_Model)
hasAerodynamicsModel (single Aerodynar
hasAtmosphereModel (multiple Atmosphere
hasDynamicsModel (single Dynamics_Mode
hasEarthModel (single Earth_Model)
hasGravityModel (single Gravity_Model)
hasPhaseState (single Phase_State)
hasPhaseStateArray (single Phase_State_
hasPhaseStateDerivatives (single Phase_S
hasPhaseTerminationRecord (single Phase
hasSolver (single Solver)
hasTerminationModel (multiple Termination_
servesComputePhaseTrajectory (single Co
servesInitializePhase (single Initialize_Phas
servesUpdatePhaseStateandDerivatives (s

Figure 102 TSONT Guided Phase


Propelled_Phase (instance of owl:Class)

CLASS EDITOR

For Class: Propelled_Phase (instance of owl:Class)

Name | SameAs | DifferentFrom

Propelled_Phase

rdfs:comment

Annotations

Property	Value	Lang

Asserted | Inferred

Asserted Conditions

NECESSARY & SUFFICIENT
∀ hasPropellantModel Propellant_Model
∀ servesUpdatePhaseStateandDerivatives Update_Propelled_P

NECESSARY
Trajectory_Simulation_Phase

INHERITED
∀ hasAerodynamicsModel Aerodynamics_Model [from Traj...]
∀ hasAtmosphereModel Atmosphere_Model [from Trajecto...]
∀ hasDynamicsModel Dynamics_Model [from Trajectory_Si...]
∀ hasEarthModel Earth_Model [from Trajectory_Simulation_...]
∀ hasGravityModel Gravity_Model [from Trajectory_Simulat...]
∀ hasPhaseState Phase_State [from Trajectory_Simulation...]
∀ hasPhaseStateArray Phase_State_Array [from Trajector...]
∀ hasPhaseStateDerivatives Phase_State_Derivatives
∀ hasPhaseTerminationRecord Phase_Termination_Record
∀ hasSolver Solver [from Trajectory_Simulation_Phase]
∀ hasTerminationModel Termination_Model [from Trajectory...]
∀ servesComputePhaseTrajectory Compute_Phase_Trajectory
∀ servesInitializePhase Initialize_Phase [from Trajectory_Si...]

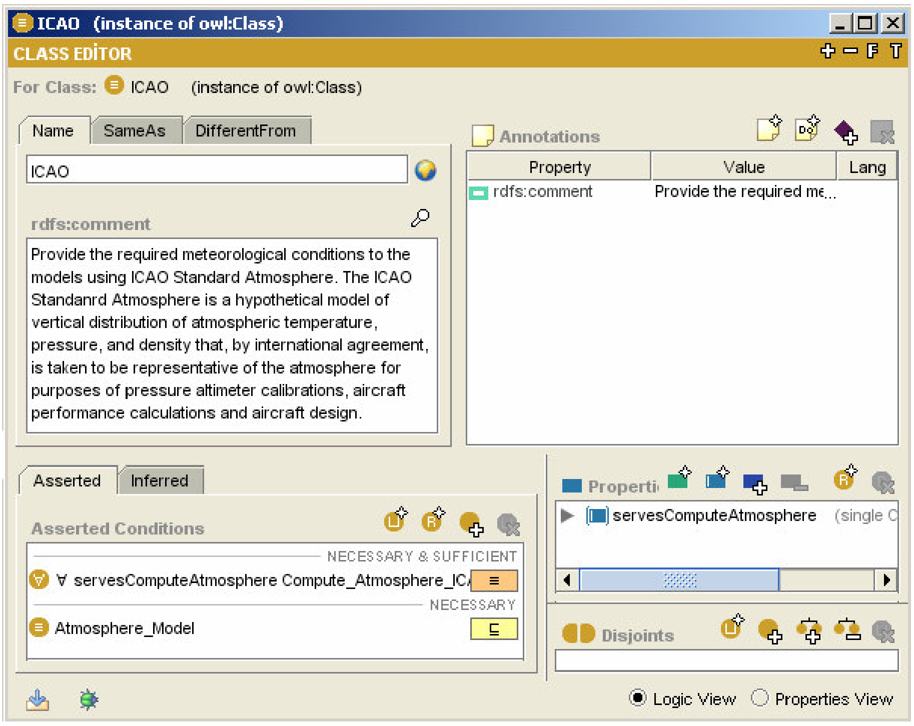
Properties

hasPropellantModel (single Propellant_Mo
hasAerodynamicsModel (single Aerodyn
hasAtmosphereModel (multiple Atmosphe
hasDynamicsModel (single Dynamics_Mo
hasEarthModel (single Earth_Model)
hasGravityModel (single Gravity_Model)
hasPhaseState (single Phase_State)
hasPhaseStateArray (single Phase_State
hasPhaseStateDerivatives (single Phase_
hasPhaseTerminationRecord (single Phas
hasSolver (single Solver)
hasTerminationModel (multiple Terminatio
servesComputePhaseTrajectory (single C
servesInitializePhase (single Initialize_Pha
servesUpdatePhaseStateandDerivatives

Disjoints

[● Logic View] ○ Properties View

Figure 103 TSONT Propelled Phase


ICAO (instance of owl:Class)

CLASS EDITOR

For Class: ICAO (instance of owl:Class)

Name: ICAO

rdfs:comment: Provide the required meteorological conditions to the models using ICAO Standard Atmosphere. The ICAO Standanrd Atmosphere is a hypothetical model of vertical distribution of atmospheric temperature, pressure, and density that, by international agreement, is taken to be representative of the atmosphere for purposes of pressure altimeter calibrations, aircraft performance calculations and aircraft design.

Property	Value	Lang
rdfs:comment	Provide the required me...	

Asserted Conditions

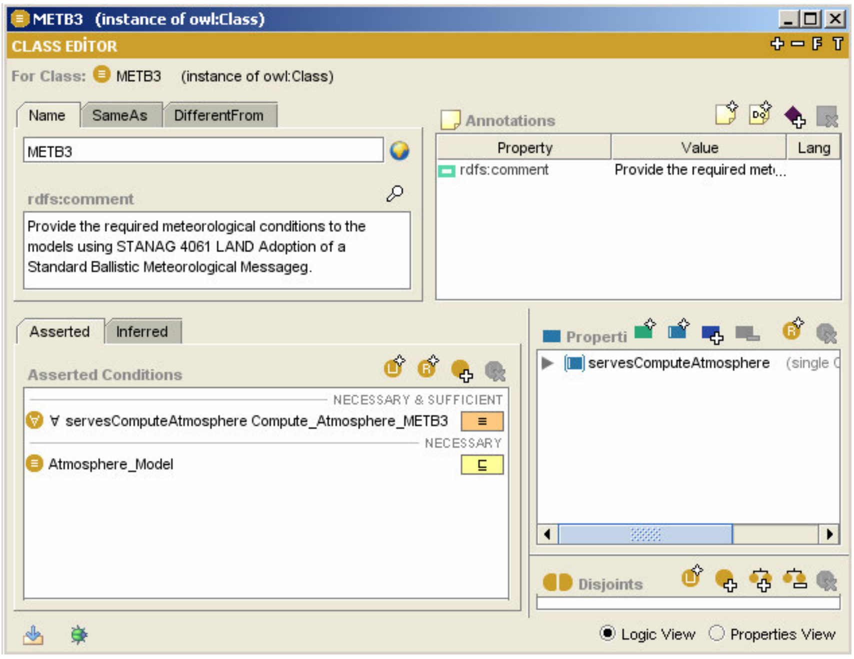
NECESSARY & SUFFICIENT
∀ servesComputeAtmosphere Compute_Atmosphere_ICA ≡

NECESSARY
Atmosphere_Model ⊑

Properties: servesComputeAtmosphere (single C

Logic View / Properties View

Figure 104 TSONT ICAO


METB3 (instance of owl:Class)

CLASS EDITOR

For Class: METB3 (instance of owl:Class)

Name: METB3

rdfs:comment: Provide the required meteorological conditions to the models using STANAG 4061 LAND Adoption of a Standard Ballistic Meteorological Messageg.

Property	Value	Lang
rdfs:comment	Provide the required met...	

Asserted Conditions

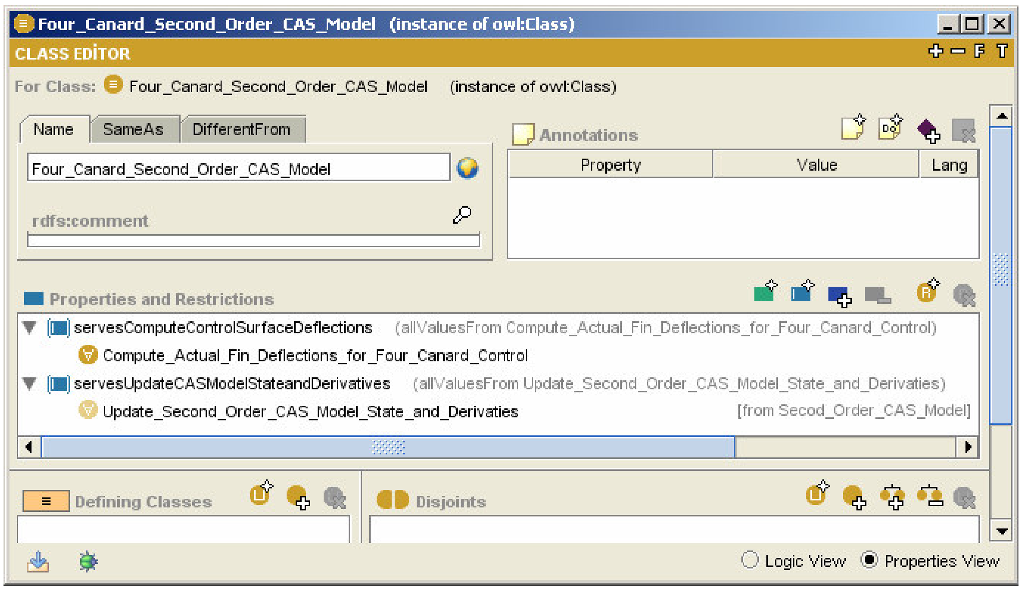
NECESSARY & SUFFICIENT
∀ servesComputeAtmosphere Compute_Atmosphere_METB3 ≡

NECESSARY
Atmosphere_Model ⊑

Properties: servesComputeAtmosphere (single C

Logic View / Properties View

Figure 105 TSONT METB3


Four_Canard_Second_Order_CAS_Model (instance of owl:Class)

CLASS EDITOR

For Class: Four_Canard_Second_Order_CAS_Model (instance of owl:Class)

Name: Four_Canard_Second_Order_CAS_Model

rdfs:comment

Annotations

Property	Value	Lang

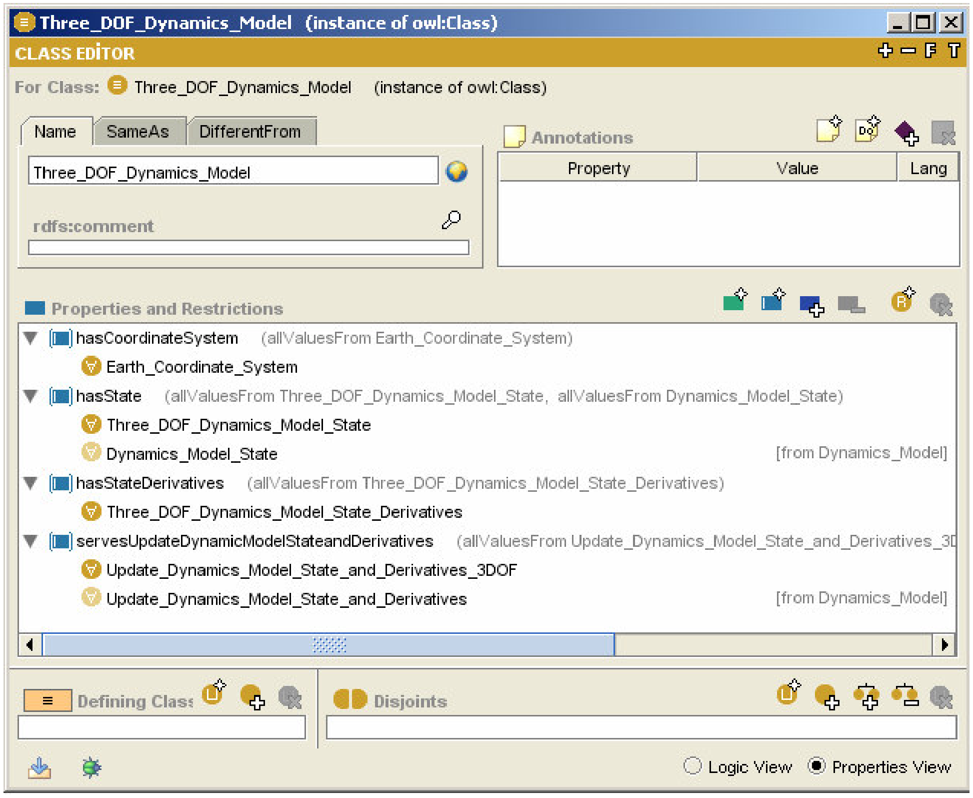
Properties and Restrictions

```
▼ servesComputeControlSurfaceDeflections    (allValuesFrom Compute_Actual_Fin_Deflections_for_Four_Canard_Control)
      Compute_Actual_Fin_Deflections_for_Four_Canard_Control
▼ servesUpdateCASModelStateandDerivatives    (allValuesFrom Update_Second_Order_CAS_Model_State_and_Derivaties)
      Update_Second_Order_CAS_Model_State_and_Derivaties                     [from Secod_Order_CAS_Model]
```

Defining Classes | Disjoints

○ Logic View ● Properties View

Figure 106 TSONT Four Canard Second Order CAS Model


Three_DOF_Dynamics_Model (instance of owl:Class)

CLASS EDITOR

For Class: Three_DOF_Dynamics_Model (instance of owl:Class)

Name: Three_DOF_Dynamics_Model

rdfs:comment

Annotations

Property	Value	Lang

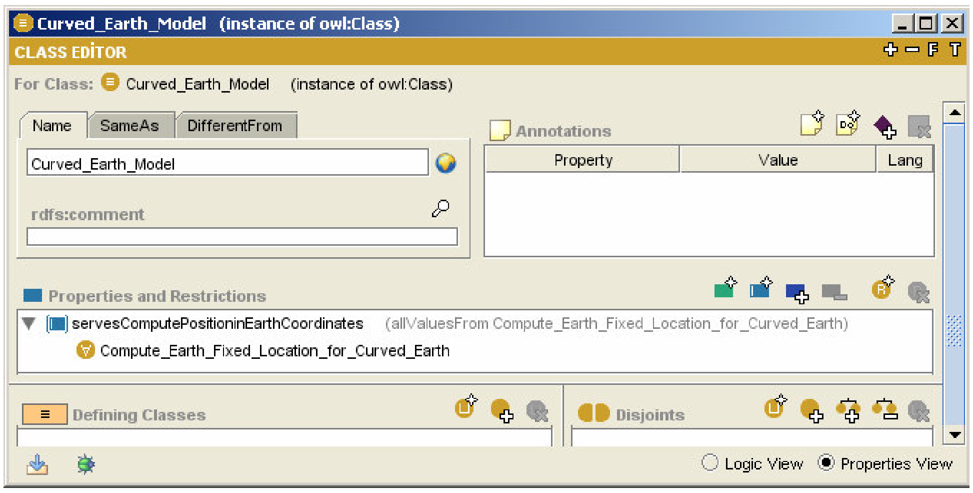
Properties and Restrictions

```
▼ hasCoordinateSystem    (allValuesFrom Earth_Coordinate_System)
      Earth_Coordinate_System
▼ hasState    (allValuesFrom Three_DOF_Dynamics_Model_State, allValuesFrom Dynamics_Model_State)
      Three_DOF_Dynamics_Model_State
      Dynamics_Model_State                                   [from Dynamics_Model]
▼ hasStateDerivatives    (allValuesFrom Three_DOF_Dynamics_Model_State_Derivatives)
      Three_DOF_Dynamics_Model_State_Derivatives
▼ servesUpdateDynamicModelStateandDerivatives    (allValuesFrom Update_Dynamics_Model_State_and_Derivatives_3I
      Update_Dynamics_Model_State_and_Derivatives_3DOF
      Update_Dynamics_Model_State_and_Derivatives            [from Dynamics_Model]
```

Defining Class | Disjoints

○ Logic View ● Properties View

Figure 107 TSONT Three DOF Dynamics Model



Curved_Earth_Model (instance of owl:Class)

CLASS EDITOR

For Class: Curved_Earth_Model (instance of owl:Class)

Name | SameAs | DifferentFrom

Curved_Earth_Model

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

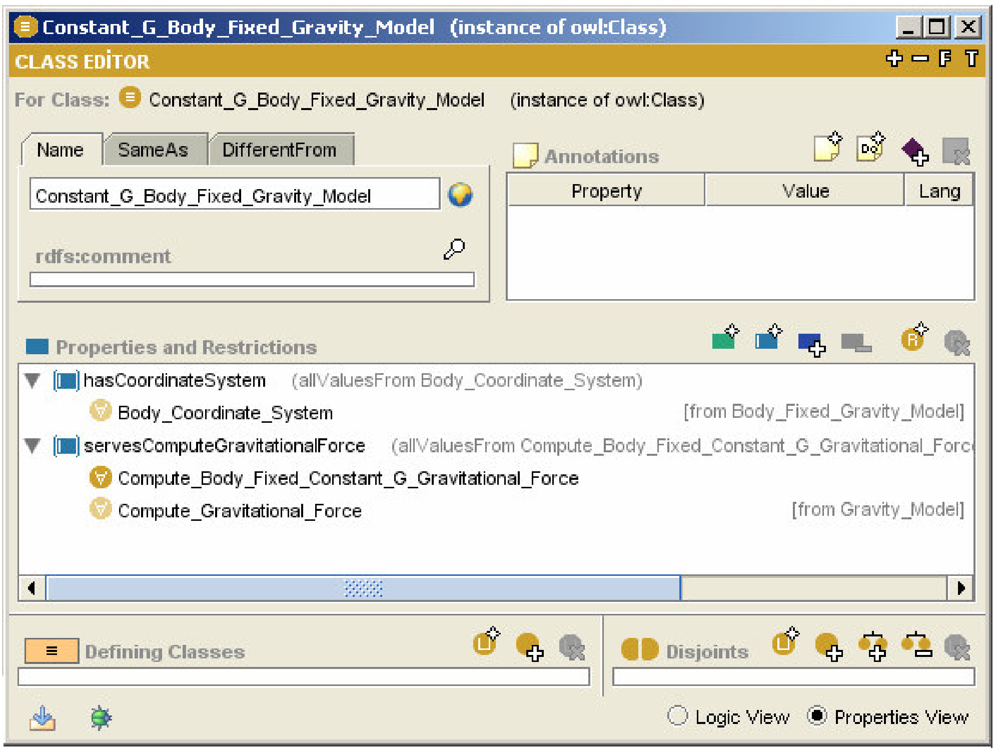
servesComputePositioninEarthCoordinates (allValuesFrom Compute_Earth_Fixed_Location_for_Curved_Earth)
Compute_Earth_Fixed_Location_for_Curved_Earth

Defining Classes

Disjoints

( ) Logic View (●) Properties View

Figure 108 TSONT Curved Earth Model



Constant_G_Body_Fixed_Gravity_Model (instance of owl:Class)

CLASS EDITOR

For Class: Constant_G_Body_Fixed_Gravity_Model (instance of owl:Class)

Name | SameAs | DifferentFrom

Constant_G_Body_Fixed_Gravity_Model

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

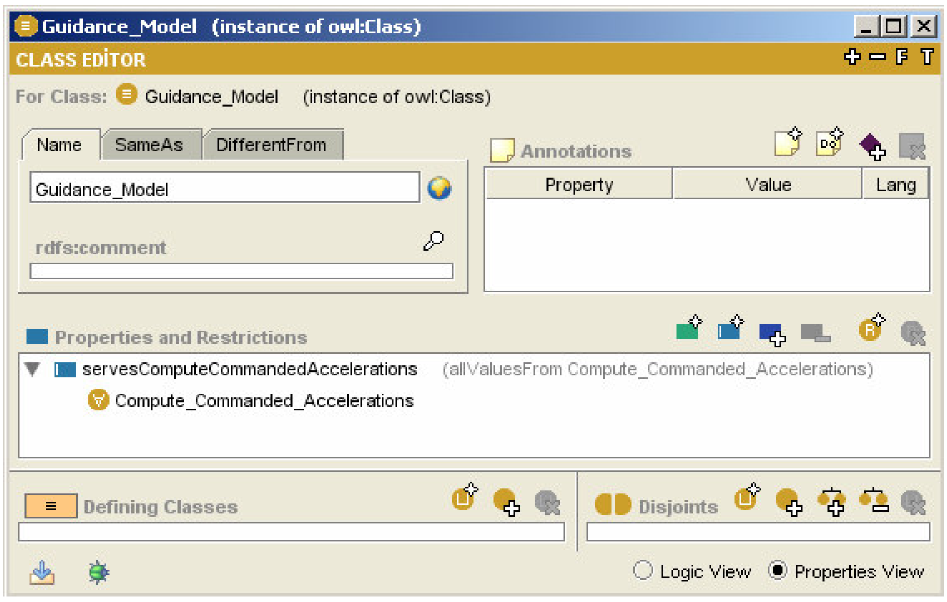
hasCoordinateSystem (allValuesFrom Body_Coordinate_System)
Body_Coordinate_System [from Body_Fixed_Gravity_Model]
servesComputeGravitationalForce (allValuesFrom Compute_Body_Fixed_Constant_G_Gravitational_Forc
Compute_Body_Fixed_Constant_G_Gravitational_Force
Compute_Gravitational_Force [from Gravity_Model]

Defining Classes

Disjoints

( ) Logic View (●) Properties View

Figure 109 TSONT Constant G Body Fixed Gravity Model



Guidance_Model (instance of owl:Class)

CLASS EDITOR

For Class: Guidance_Model (instance of owl:Class)

Name: Guidance_Model

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

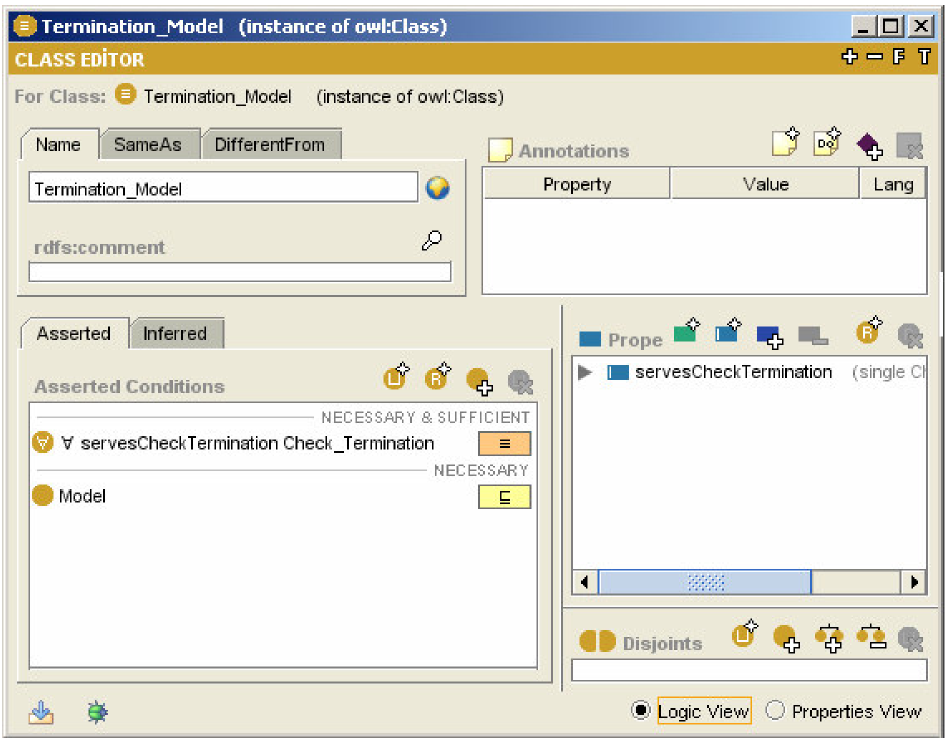
servesComputeCommandedAccelerations (allValuesFrom Compute_Commanded_Accelerations)
Compute_Commanded_Accelerations

Defining Classes

Disjoints

Logic View [✓ Properties View]

Figure 110 TSONT Guidance Model



Termination_Model (instance of owl:Class)

CLASS EDITOR

For Class: Termination_Model (instance of owl:Class)

Name: Termination_Model

rdfs:comment

Annotations

Property	Value	Lang

Asserted Conditions

NECESSARY & SUFFICIENT
∀ servesCheckTermination Check_Termination ≡

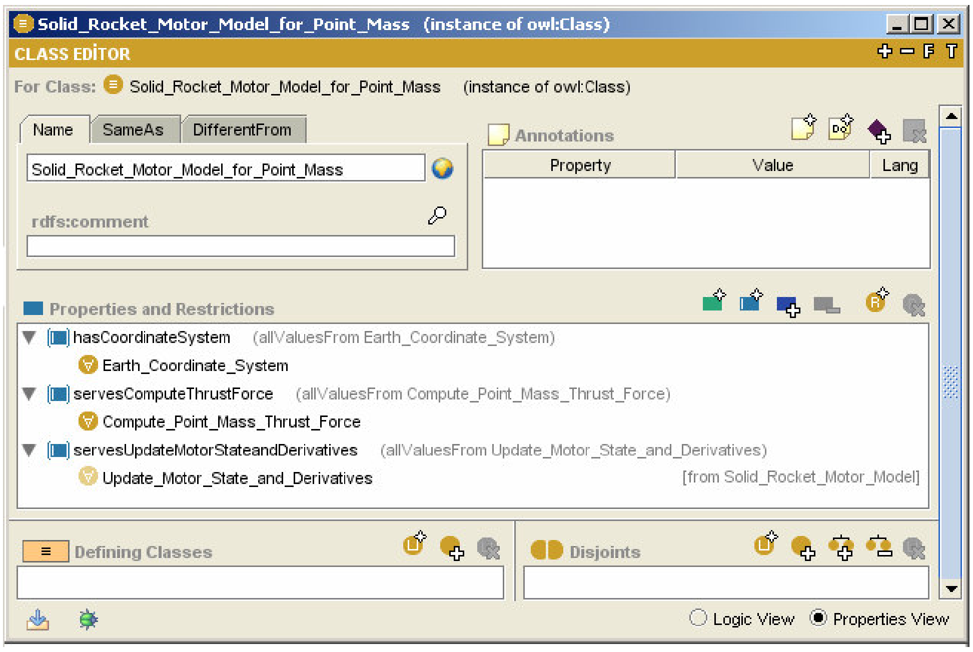
NECESSARY
Model ⊑

Prope
servesCheckTermination (single Cl

Disjoints

[✓ Logic View] Properties View

Figure 111 TSONT Termination Model


Solid_Rocket_Motor_Model_for_Point_Mass (instance of owl:Class)

CLASS EDITOR

For Class: Solid_Rocket_Motor_Model_for_Point_Mass (instance of owl:Class)

Name | SameAs | DifferentFrom

Solid_Rocket_Motor_Model_for_Point_Mass

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

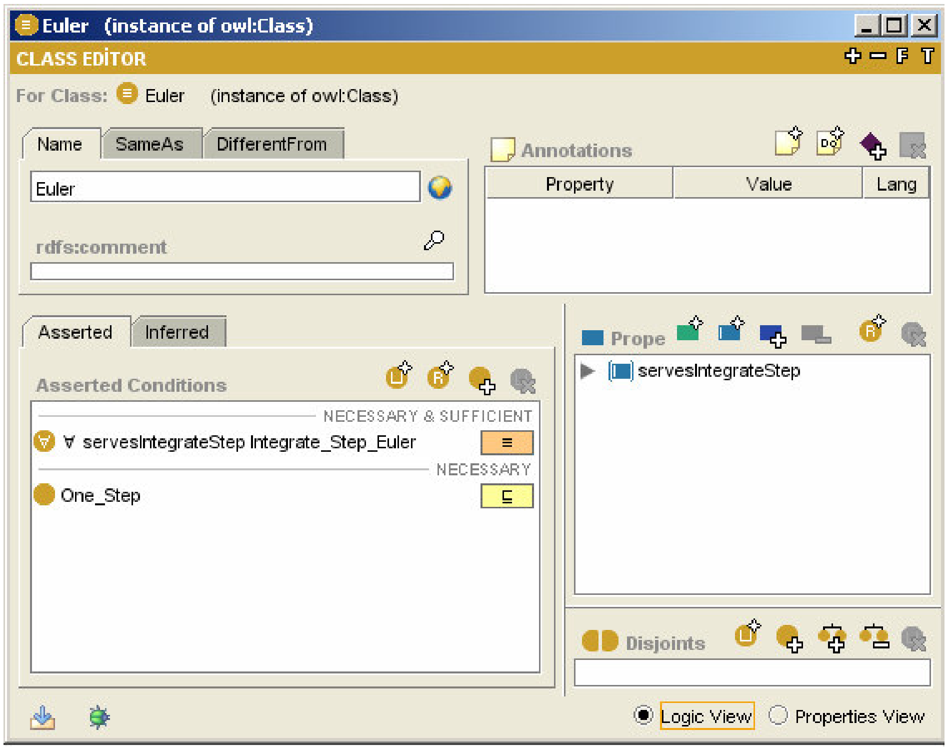
```
▼ hasCoordinateSystem   (allValuesFrom Earth_Coordinate_System)
    Earth_Coordinate_System
▼ servesComputeThrustForce   (allValuesFrom Compute_Point_Mass_Thrust_Force)
    Compute_Point_Mass_Thrust_Force
▼ servesUpdateMotorStateandDerivatives   (allValuesFrom Update_Motor_State_and_Derivatives)
    Update_Motor_State_and_Derivatives                 [from Solid_Rocket_Motor_Model]
```

Defining Classes

Disjoints

Logic View [✓ Properties View]

Figure 112 TSONT Solid Rocket Motor Model for Point Mass


Euler (instance of owl:Class)

CLASS EDITOR

For Class: Euler (instance of owl:Class)

Name | SameAs | DifferentFrom

Euler

rdfs:comment

Annotations

Property	Value	Lang

Asserted | Inferred

Asserted Conditions

```
NECESSARY & SUFFICIENT
∀ servesIntegrateStep Integrate_Step_Euler   ≡
NECESSARY
One_Step   ⊑
```

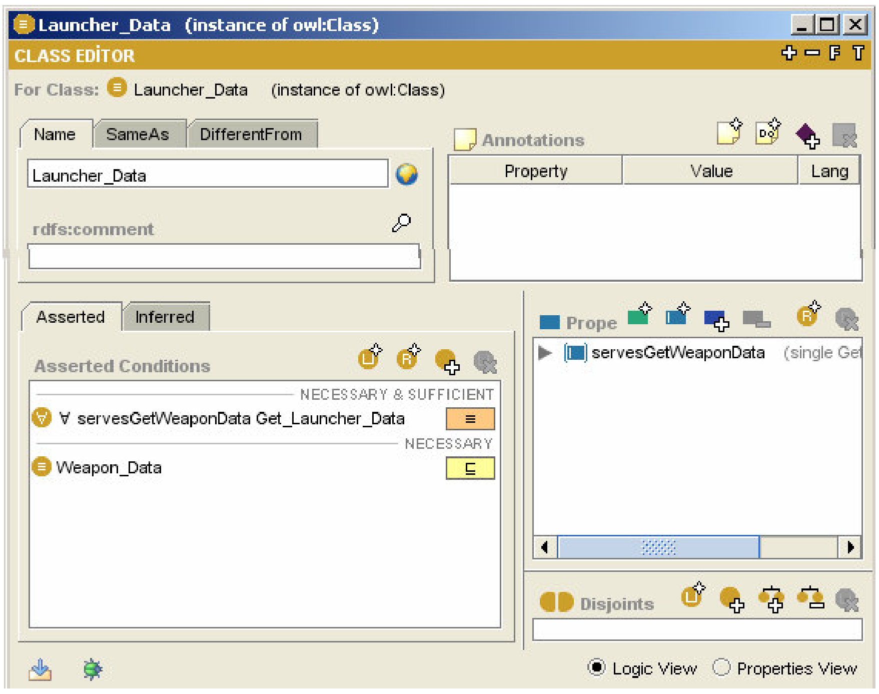
Prope

▶ servesIntegrateStep

Disjoints

[✓ Logic View] Properties View

Figure 113 TSONT Euler Solver



Launcher_Data (instance of owl:Class)

For Class: Launcher_Data (instance of owl:Class)

Name: Launcher_Data

rdfs:comment:

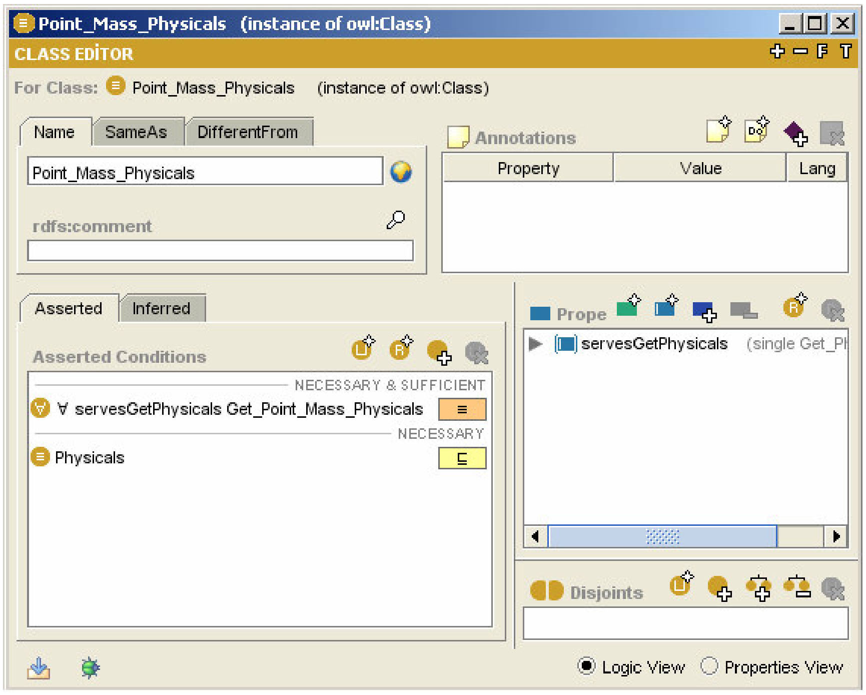
∀ servesGetWeaponData Get_Launcher_Data ≡

Weapon_Data ⊑

servesGetWeaponData (single Get

[✓ Logic View] Properties View

Figure 114 TSONT Launcher Data



Point_Mass_Physicals (instance of owl:Class)

For Class: Point_Mass_Physicals (instance of owl:Class)

Name: Point_Mass_Physicals

rdfs:comment:

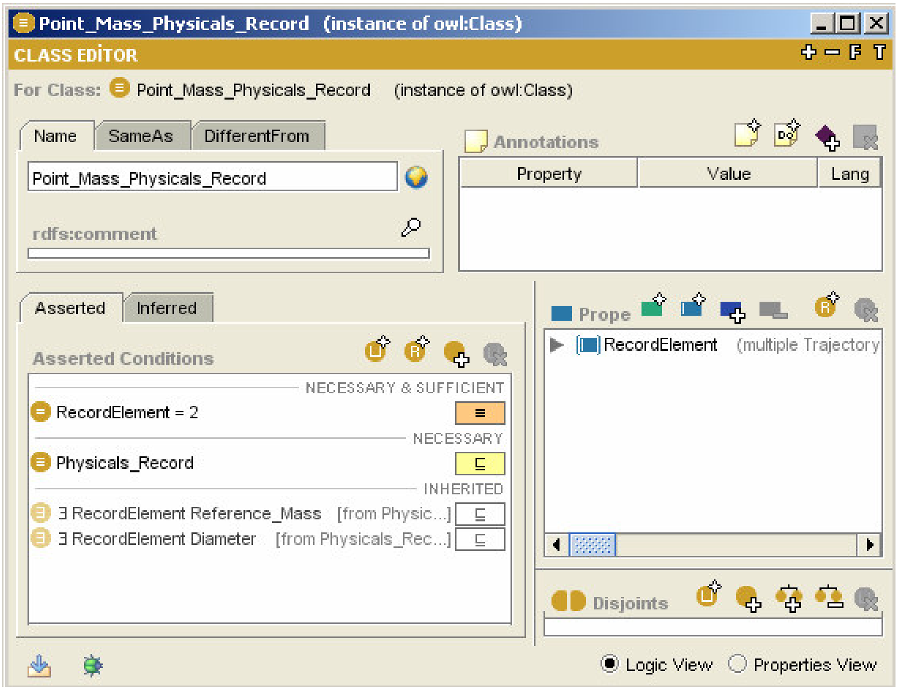
∀ servesGetPhysicals Get_Point_Mass_Physicals ≡

Physicals ⊑

servesGetPhysicals (single Get_Pl

[✓ Logic View] Properties View

Figure 115 TSONT Point Mass Physicals


Point_Mass_Physicals_Record (instance of owl:Class)

CLASS EDITOR

For Class: Point_Mass_Physicals_Record (instance of owl:Class)

Name: Point_Mass_Physicals_Record

rdfs:comment

Annotations

Property	Value	Lang

Asserted Conditions

NECESSARY & SUFFICIENT
RecordElement = 2 ≡

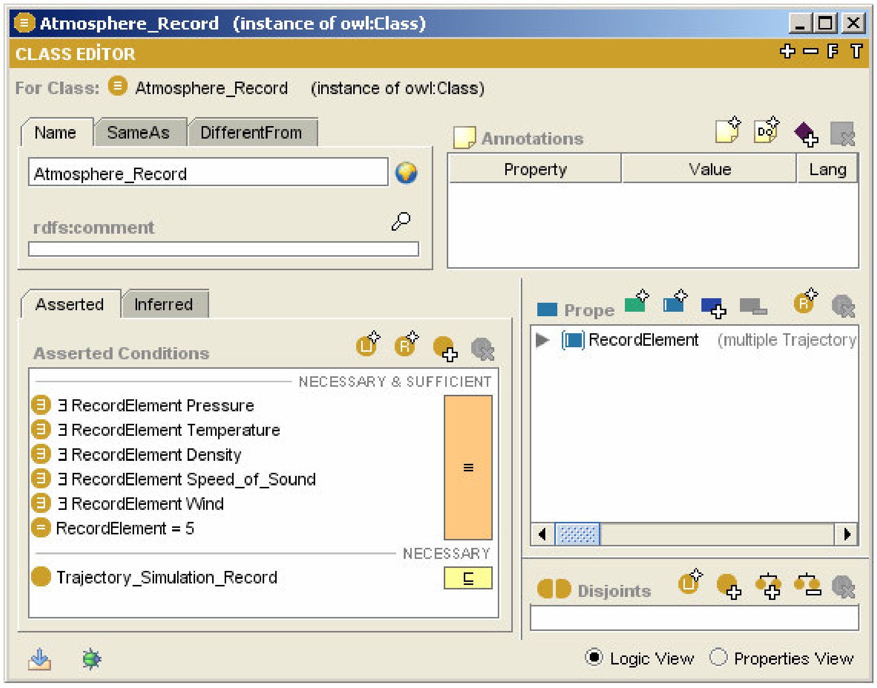
NECESSARY
Physicals_Record ⊑

INHERITED
∃ RecordElement Reference_Mass [from Physic...] ⊑
∃ RecordElement Diameter [from Physicals_Rec...] ⊑

Prope
RecordElement (multiple Trajectory

Disjoints

Figure 116 TSONT Point Mass Physicals Record


Atmosphere_Record (instance of owl:Class)

CLASS EDITOR

For Class: Atmosphere_Record (instance of owl:Class)

Name: Atmosphere_Record

rdfs:comment

Annotations

Property	Value	Lang

Asserted Conditions

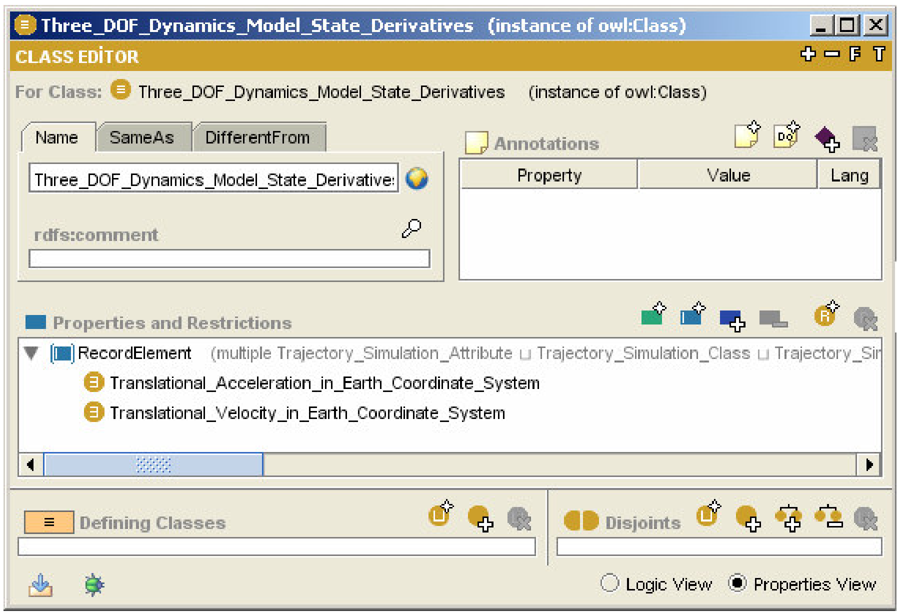
NECESSARY & SUFFICIENT
∃ RecordElement Pressure
∃ RecordElement Temperature
∃ RecordElement Density
∃ RecordElement Speed_of_Sound
∃ RecordElement Wind
RecordElement = 5 ≡

NECESSARY
Trajectory_Simulation_Record ⊑

Prope
RecordElement (multiple Trajectory

Disjoints

Figure 117 TSONT Atmosphere Record


Three_DOF_Dynamics_Model_State_Derivatives (instance of owl:Class)

CLASS EDITOR

For Class: Three_DOF_Dynamics_Model_State_Derivatives (instance of owl:Class)

Name: Three_DOF_Dynamics_Model_State_Derivative:

rdfs:comment

Property	Value	Lang

Properties and Restrictions

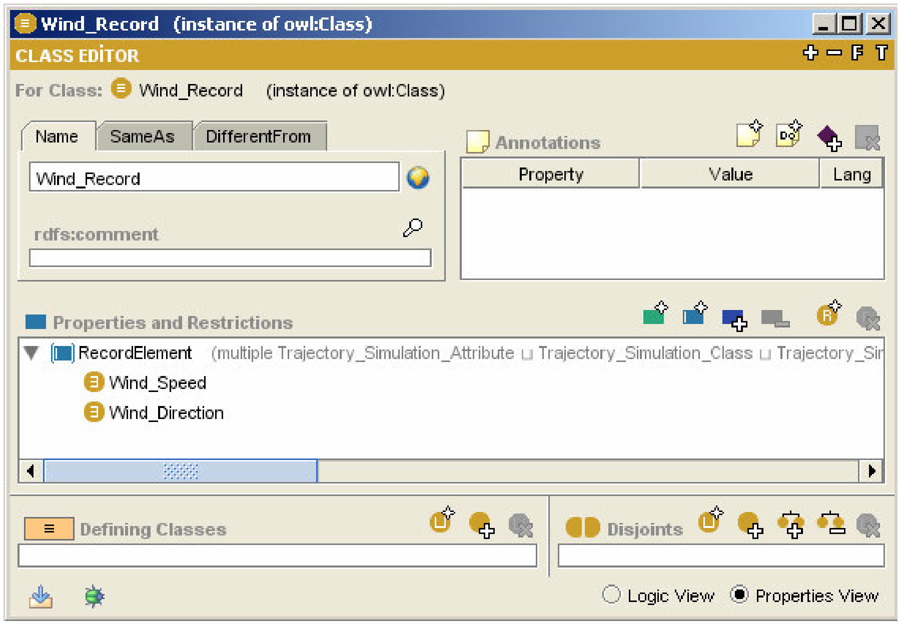
RecordElement (multiple Trajectory_Simulation_Attribute ⊔ Trajectory_Simulation_Class ⊔ Trajectory_Sir
- Translational_Acceleration_in_Earth_Coordinate_System
- Translational_Velocity_in_Earth_Coordinate_System

Defining Classes

Disjoints

( ) Logic View (●) Properties View

Figure 118 TSONT Three DOF Dynamics Models State Derivatives


Wind_Record (instance of owl:Class)

CLASS EDITOR

For Class: Wind_Record (instance of owl:Class)

Name: Wind_Record

rdfs:comment

Property	Value	Lang

Properties and Restrictions

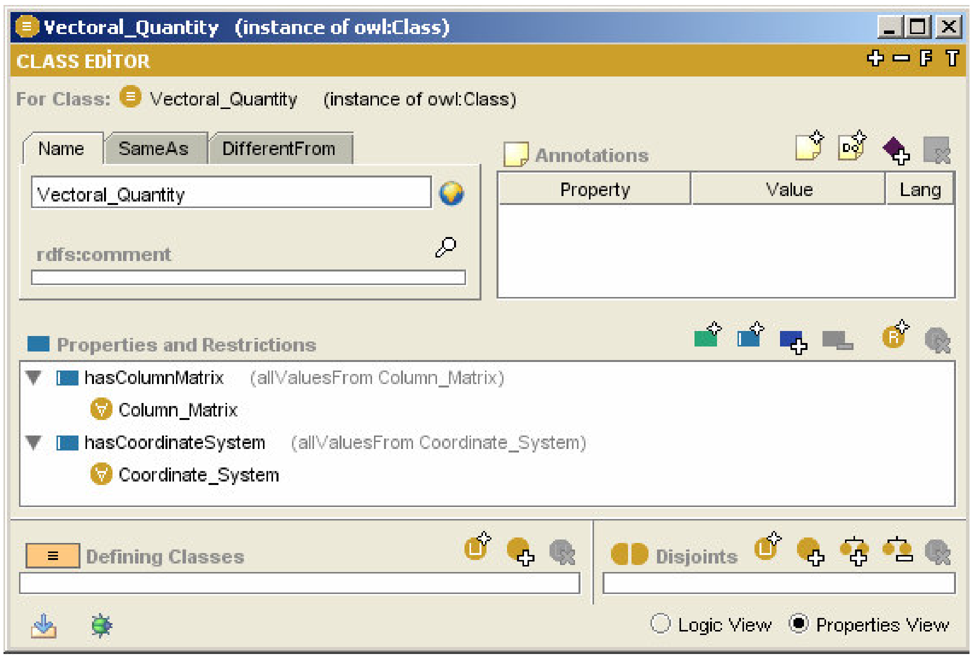
RecordElement (multiple Trajectory_Simulation_Attribute ⊔ Trajectory_Simulation_Class ⊔ Trajectory_Sir
- Wind_Speed
- Wind_Direction

Defining Classes

Disjoints

( ) Logic View (●) Properties View

Figure 119 TSONT Wind Record


Vectoral_Quantity (instance of owl:Class)

CLASS EDITOR

For Class: Vectoral_Quantity (instance of owl:Class)

Name | SameAs | DifferentFrom

Vectoral_Quantity

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

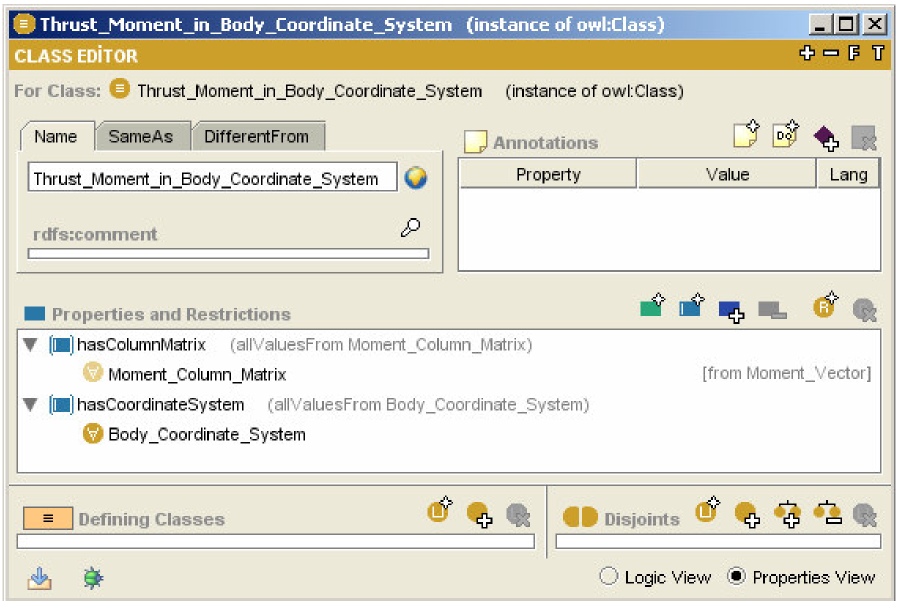
- hasColumnMatrix (allValuesFrom Column_Matrix)
  - Column_Matrix
- hasCoordinateSystem (allValuesFrom Coordinate_System)
  - Coordinate_System

Defining Classes

Disjoints

Logic View [✓ Properties View]

Figure 120 TSONT Vectoral Quantity


Thrust_Moment_in_Body_Coordinate_System (instance of owl:Class)

CLASS EDITOR

For Class: Thrust_Moment_in_Body_Coordinate_System (instance of owl:Class)

Name | SameAs | DifferentFrom

Thrust_Moment_in_Body_Coordinate_System

rdfs:comment

Annotations

Property	Value	Lang

Properties and Restrictions

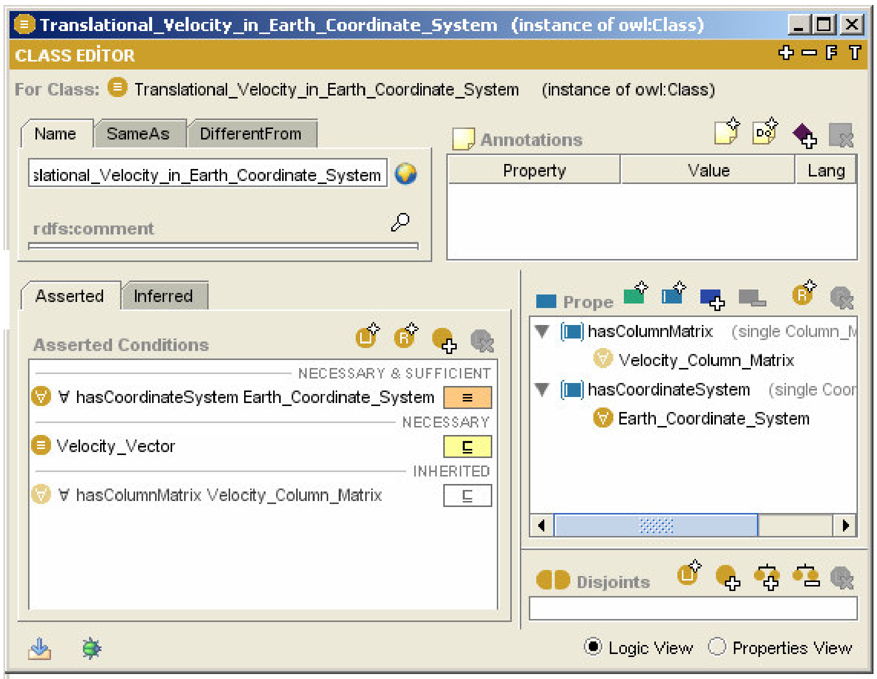
- hasColumnMatrix (allValuesFrom Moment_Column_Matrix)
  - Moment_Column_Matrix [from Moment_Vector]
- hasCoordinateSystem (allValuesFrom Body_Coordinate_System)
  - Body_Coordinate_System

Defining Classes

Disjoints

Logic View [✓ Properties View]

Figure 121 TSONT Thrust Moment in Body Coordinate System


Translational_Velocity_in_Earth_Coordinate_System (instance of owl:Class)

CLASS EDITOR

For Class: Translational_Velocity_in_Earth_Coordinate_System (instance of owl:Class)

Name | SameAs | DifferentFrom

slational_Velocity_in_Earth_Coordinate_System

rdfs:comment

Annotations

Property	Value	Lang

Asserted | Inferred

Asserted Conditions

NECESSARY & SUFFICIENT
∀ hasCoordinateSystem Earth_Coordinate_System ≡

NECESSARY
Velocity_Vector ⊑

INHERITED
∀ hasColumnMatrix Velocity_Column_Matrix ⊑

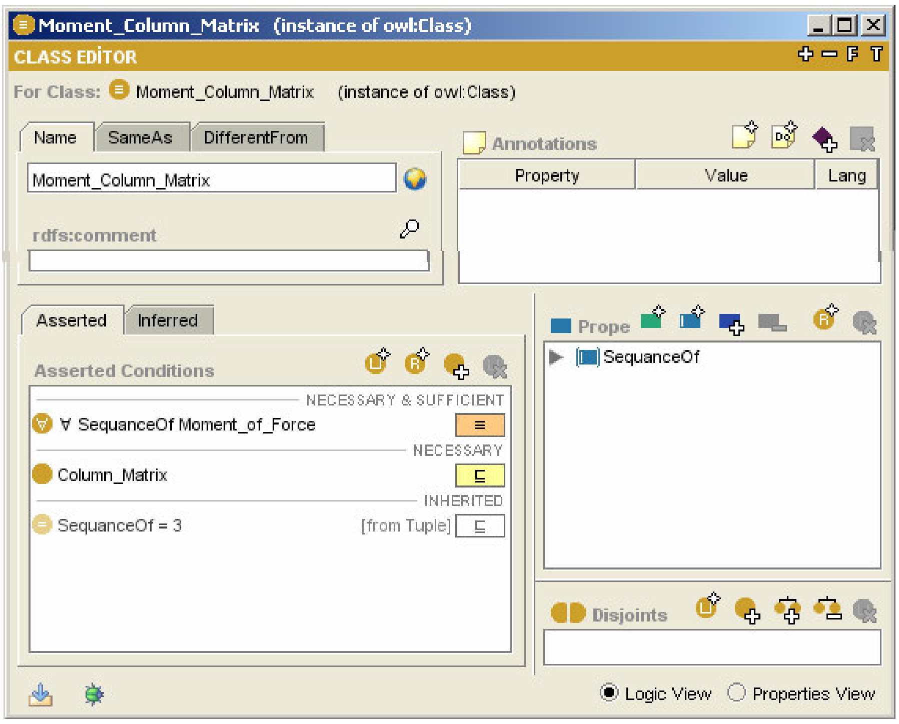
Prope

hasColumnMatrix (single Column_M
Velocity_Column_Matrix
hasCoordinateSystem (single Coor
Earth_Coordinate_System

Disjoints

[✓ Logic View] Properties View

Figure 122 TSONT Translational Velocity in Earth Coordinate System


Moment_Column_Matrix (instance of owl:Class)

CLASS EDITOR

For Class: Moment_Column_Matrix (instance of owl:Class)

Name | SameAs | DifferentFrom

Moment_Column_Matrix

rdfs:comment

Annotations

Property	Value	Lang

Asserted | Inferred

Asserted Conditions

NECESSARY & SUFFICIENT
∀ SequanceOf Moment_of_Force ≡

NECESSARY
Column_Matrix ⊑

INHERITED
SequanceOf = 3 [from Tuple] ⊑

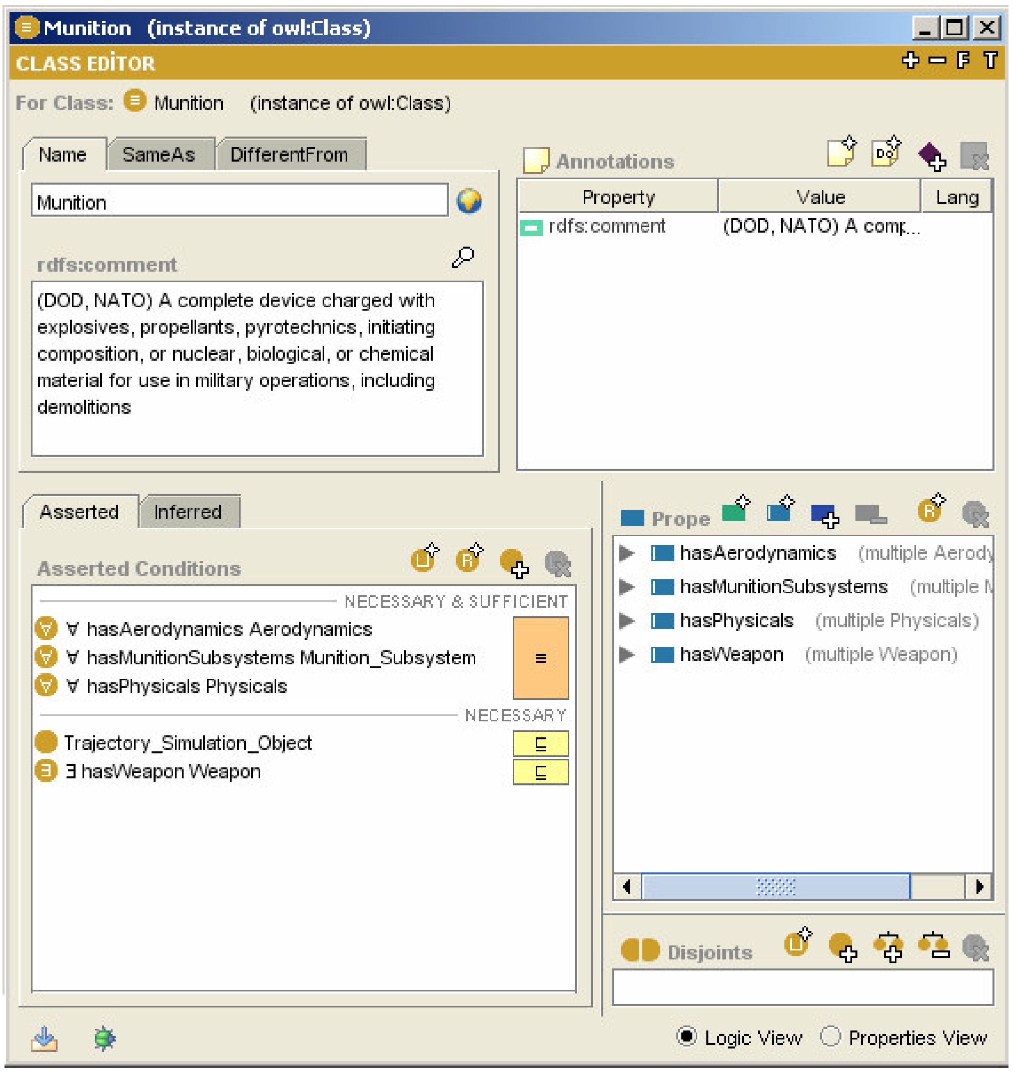
Prope

SequanceOf

Disjoints

[✓ Logic View] Properties View

Figure 123 TSONT Moment Column Matrix


Munition (instance of owl:Class)

CLASS EDITOR

For Class: Munition (instance of owl:Class)

Name: Munition

rdfs:comment

(DOD, NATO) A complete device charged with explosives, propellants, pyrotechnics, initiating composition, or nuclear, biological, or chemical material for use in military operations, including demolitions

Annotations

Property	Value	Lang
rdfs:comment	(DOD, NATO) A comp...	

Asserted Conditions

NECESSARY & SUFFICIENT
∀ hasAerodynamics Aerodynamics
∀ hasMunitionSubsystems Munition_Subsystem
∀ hasPhysicals Physicals

NECESSARY
Trajectory_Simulation_Object
∃ hasWeapon Weapon

Prope
hasAerodynamics (multiple Aerody
hasMunitionSubsystems (multiple M
hasPhysicals (multiple Physicals)
hasWeapon (multiple Weapon)

Disjoints

[● Logic View] Properties View

Figure 124 TSONT Munition

APPENDIX C

A DAVE-ML EXAMPLE

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DAVEfunc SYSTEM "DAVEfunc.dtd">
<!-- $Revision: 2.3 $ -->
<DAVEfunc>
    <fileHeader name="Update Body Fixed Six DOF Dynamic Model State and Derivatives">
        <author name="Umut DURAK" org="TUBITAK-SAGE" xns="@bjax"/>
        <fileCreationDate date="24/10/2005"/>
        <description> This daveml fuction defines the model to calculate state derivatives of
Six
            DOF Body Fixed Dynamics Model </description>
        <!-- ================== -->
        <!-- References         -->
        <!-- ================== -->
        <reference  refID="[MIL95]"  author="N/A"  title="MIL-HDBK  1211"  accession="TÜBİTAK-SAGE
Kütüphanesi" date="1995"/>
        <modificationRecord modID="A">
            <author name="Umut DURAK" org="TUBITAK-SAGE" email="udurak@sage.tubitak.gov.tr"/>
            <description> First Creation </description>
        </modificationRecord>
    </fileHeader>
    <!-- ===================-->
    <!-- Input variables                -->
    <!-- ===================-->
    <!-- ===================-->
    <!-- Ballistic Record             -->
    <!-- ===================-->
    <variableDef name="Mass" varID="mass" units="kg" symbol="mass">
        <description> Mass in kg </description>
    </variableDef>
    <variableDef name="Ix" varID="Ix" units="kgm2" symbol="Ix">
        <description> Axial Moment of Inertia </description>
    </variableDef>
    <variableDef name="Iy" varID="Iy" units="kgm2" symbol="Ix">
        <description> Transverse Moment of Inertia in Y axis </description>
    </variableDef>
    <variableDef name="Iz" varID="Iz" units="kgm2" symbol="Iz">
        <description> Transverse Moment of Inertia in Z axis </description>
    </variableDef>
    <!-- ===================-->
    <!--State                                  -->
    <!-- ===================-->
    <variableDef name="phi" varID="phi" units="rad" symbol="phi">
        <description> Roll atitude </description>
    </variableDef>
    <variableDef name="theta" varID="theta" units="rad" symbol="theta">
        <description> Pitch atitude </description>
    </variableDef>
    <variableDef name="psi" varID="psi" units="rad" symbol="psi">
        <description> Yaw atitude </description>
    </variableDef>
    <variableDef name="p" varID="p" units="rad/s" symbol="p">
        <description> Roll rate </description>
    </variableDef>
    <variableDef name="q" varID="q" units="rad/s" symbol="q">
        <description> Pitch rate </description>
    </variableDef>
    <variableDef name="r" varID="r" units="rad/s" symbol="r">
        <description> Yaw Rate </description>
    </variableDef>
    <variableDef name="u" varID="u" units="m/s" symbol="u">
        <description> Body Fixed Velocity in X </description>
    </variableDef>
    <variableDef name="v" varID="v" units="m/s" symbol="v">
        <description> Body Fixed Velocity in Y </description>
    </variableDef>
    <variableDef name="w" varID="w" units="m/s" symbol="w">
        <description> Body Fixed Velocity in Z </description>
    </variableDef>
```

```
<!-- ================== -->
<!-- Forces and Moments     -->
<!-- ================== -->
<variableDef name="FAX" varID="FAX" units="N" symbol="FAX">
    <description> Aerodynamic Force in X </description>
</variableDef>
<variableDef name="FAY" varID="FAY" units="N" symbol="FAY">
    <description> Aerodynamic Force in Y </description>
</variableDef>
<variableDef name="FAZ" varID="FAZ" units="N" symbol="FAZ">
    <description> Aerodynamic Force in Z </description>
</variableDef>

<variableDef name="FGX" varID="FGX" units="N" symbol="FGX">
    <description> Gravitational Force in X </description>
</variableDef>
<variableDef name="FGY" varID="FGY" units="N" symbol="FGY">
    <description> Gravitational Force in Y </description>
</variableDef>
<variableDef name="FGZ" varID="FGZ" units="N" symbol="FGZ">
    <description> Gravitational Force in Z </description>
</variableDef>

<variableDef name="LA" varID="LA" units="Nm" symbol="LA">
    <description> Aerodynamic Moment in X </description>
</variableDef>
<variableDef name="MA" varID="MA" units="Nm" symbol="MA">
    <description> Aerodynamic Moment in Y </description>
</variableDef>
<variableDef name="NA" varID="NA" units="Nm" symbol="NA">
    <description> Aerodynamic Moment in Z </description>
</variableDef>

<!-- ================== -->
<!--  Output variables             -->
<!-- ================== -->
<variableDef name="udot" varID="udot" units="m/s2">
    <description> Body fixed tranlational acceleration in X </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>udot</ci>
                <apply>
                    <plus/>
                    <apply>
                        <times/>
                        <apply>
                            <plus/>
                            <ci>FAX</ci>
                            <ci>FGX</ci>
                        </apply>
                        <apply>
                            <power/>
                            <ci>mass</ci>
                            <cn type='integer'>-1</cn>
                        </apply>
                    </apply>
                    <apply>
                        <times/>
                        <ci>r</ci>
                        <ci>v</ci>
                    </apply>
                    <apply>
                        <times/>
                        <cn type='integer'>-1</cn>
                        <apply>
                            <times/>
                            <ci>q</ci>
                            <ci>w</ci>
                        </apply>
                    </apply>
                </apply>
            </apply>
        </math>
    </calculation>
    <isOutput/>
</variableDef>
```

```
<variableDef name="vdot" varID="vdot" units="m/s2">
    <description> Body fixed tranlational acceleration in Y </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>vdot</ci>
                <apply>
                    <plus/>
                    <apply>
                        <times/>
                        <apply>
                            <plus/>
                            <ci>FAY</ci>
                            <ci>FGY</ci>
                        </apply>
                        <apply>
                            <power/>
                            <ci>mass</ci>
                            <cn type='integer'>-1</cn>
                        </apply>
                    </apply>
                    <apply>
                        <times/>
                        <cn type='integer'>-1</cn>
                        <apply>
                            <times/>
                            <ci>r</ci>
                            <ci>u</ci>
                        </apply>
                    </apply>
                    <apply>
                        <times/>
                        <ci>p</ci>
                        <ci>w</ci>
                    </apply>
                </apply>
            </apply>
        </math>
    </calculation>
        <isOutput/>
</variableDef>

<variableDef name="wdot" varID="wdot" units="m/s2">
    <description> Body fixed tranlational acceleration in Z </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>wdot</ci>
                <apply>
                    <plus/>
                    <apply>
                        <times/>
                        <apply>
                            <plus/>
                            <ci>FAZ</ci>
                            <ci>FGZ</ci>
                        </apply>
                        <apply>
                            <power/>
                            <ci>mass</ci>
                            <cn type='integer'>-1</cn>
                        </apply>
                    </apply>
                    <apply>
                        <times/>
                        <ci>q</ci>
                        <ci>u</ci>
                    </apply>
                    <apply>
                        <times/>
                        <cn type='integer'>-1</cn>
                        <apply>
                            <times/>
                            <ci>p</ci>
                            <ci>v</ci>
                        </apply>
                    </apply>
                </apply>
            </apply>
        </math>
    </calculation>
        <isOutput/>
</variableDef>

<variableDef name="pdot" varID="pdot" units="rad/s2">
    <description> Body fixed angular acceleration in X </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>pdot</ci>
                <apply>
                    <times/>
                    <apply>
                        <plus/>
                        <ci>LA</ci>
                        <apply>
                            <times/>
                            <cn type='integer'>-1</cn>
                            <apply>
                                <times/>
                                <apply>
                                    <plus/>
                                    <ci>Iz</ci>
                                    <apply>
                                        <times/>
                                        <cn type='integer'>-1</cn>
                                        <ci>Iy</ci>
                                    </apply>
                                </apply>
                                <ci>q</ci>
                                <ci>r</ci>
                            </apply>
                        </apply>
                    </apply>
                    <apply>
                        <power/>
                        <ci>Ix</ci>
                        <cn type='integer'>-1</cn>
                    </apply>
                </apply>
            </apply>
        </math>
    </calculation>
    <isOutput/>
</variableDef>
<variableDef name="qdot" varID="qdot" units="rad/s2">
    <description> Body fixed angular acceleration in Y </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>qdot</ci>
                <apply>
                    <times/>
                    <apply>
                        <plus/>
                        <ci>MA</ci>
                        <apply>
                            <times/>
                            <cn type='integer'>-1</cn>
                            <apply>
                                <times/>
                                <apply>
                                    <plus/>
                                    <ci>Ix</ci>
                                    <apply>
                                        <times/>
                                        <cn type='integer'>-1</cn>
                                        <ci>Iz</ci>
                                    </apply>
                                </apply>
                                <ci>p</ci>
                                <ci>r</ci>
                            </apply>
                        </apply>
                    </apply>
                    <apply>
                        <power/>
                        <ci>Iy</ci>
                        <cn type='integer'>-1</cn>
                    </apply>
                </apply>
            </apply>
        </math>
    </calculation>
    <isOutput/>
</variableDef>

<variableDef name="rdot" varID="rdot" units="rad/s2">
    <description> Body fixed angular acceleration in Z </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>rdot</ci>
                <apply>
                    <times/>
                    <apply>
                        <plus/>
                        <ci>NA</ci>
                        <apply>
                            <times/>
                            <cn type='integer'>-1</cn>
                            <apply>
                                <times/>
                                <apply>
                                    <plus/>
                                    <ci>Iy</ci>
                                    <apply>
                                        <times/>
                                        <cn type='integer'>-1</cn>
                                        <ci>Ix</ci>
                                    </apply>
                                </apply>
                                <ci>p</ci>
                                <ci>q</ci>
                            </apply>
                        </apply>
                    </apply>
                    <apply>
                        <power/>
                        <ci>Iz</ci>
                        <cn type='integer'>-1</cn>
                    </apply>
                </apply>
            </apply>
        </math>
    </calculation>
    <isOutput/>
</variableDef>

<variableDef name="phidot" varID="phidot" units="rad/s">
    <description> Rate of change of roll attitute </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>phidot</ci>
                <apply>
                    <plus/>
                    <ci>p</ci>
                    <apply>
                        <times/>
                        <apply>
                            <plus/>
                            <apply>
                                <times/>
                                <ci>r</ci>
                                <apply>
                                    <cos/>
                                    <ci>phi</ci>
                                </apply>
                            </apply>
                            <apply>
```

```
                                <times/>
                                <ci>q</ci>
                                <apply>
                                    <sin/>
                                    <ci>phi</ci>
                                </apply>
                            </apply>
                        </apply>
                        <apply>
                            <tan/>
                            <ci>theta</ci>
                        </apply>
                    </apply>
                </apply>
            </apply>
        </math>
    </calculation>
    <isOutput/>
</variableDef>

<variableDef name="thetadot" varID="thetadot" units="rad/s">
    <description> Rate of change of pitch attitute </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>thetadot</ci>
                <apply>
                    <plus/>
                    <apply>
                        <times/>
                        <ci>q</ci>
                        <apply>
                            <cos/>
                            <ci>phi</ci>
                        </apply>
                    </apply>
                    <apply>
                        <times/>
                        <cn type='integer'>-1</cn>
                        <apply>
                            <times/>
                            <ci>r</ci>
                            <apply>
                                <sin/>
                                <ci>phi</ci>
                            </apply>
                        </apply>
                    </apply>
                </apply>
            </apply>
        </math>
    </calculation>
    <isOutput/>
</variableDef>

<variableDef name="psidot" varID="psidot" units="rad/s">
    <description> Rate of change of yaw attitute </description>
    <calculation>
        <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                <eq/>
                <ci>psidot</ci>
                <apply>
                    <times/>
                    <apply>
                        <plus/>
                        <apply>
                            <times/>
                            <ci>r</ci>
                            <apply>
                                <cos/>
                                <ci>phi</ci>
                            </apply>
                        </apply>
                        <apply>
                            <times/>
                            <ci>q</ci>
                            <apply>
```

```
                                    <sin/>
                                    <ci>phi</ci>
                                </apply>
                            </apply>
                        </apply>
                        <apply>
                            <power/>
                            <apply>
                                <cos/>
                                <ci>theta</ci>
                            </apply>
                            <cn type='integer'>-1</cn>
                        </apply>
                    </apply>
                </apply>
            </math>
        </calculation>
        <isOutput/>
    </variableDef>

</DAVEfunc>
```

APPENDIX D

SAMPLE CLASS DIAGRAMS FROM PLATFORM INDEPENDENT FRAMEWORK ARCHITECTURE

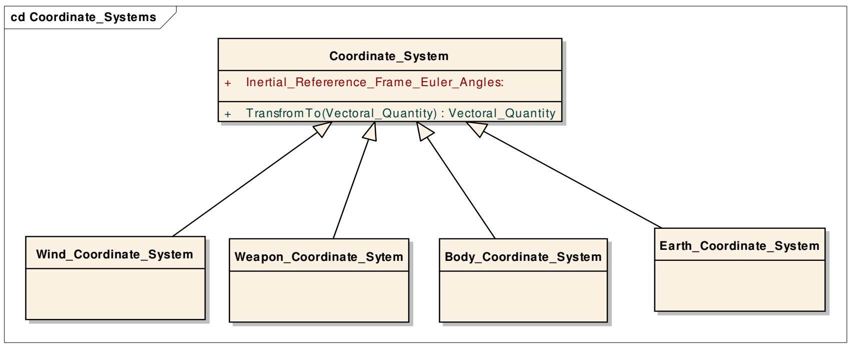


Figure 125 Platform Independent Coordinate System Classes

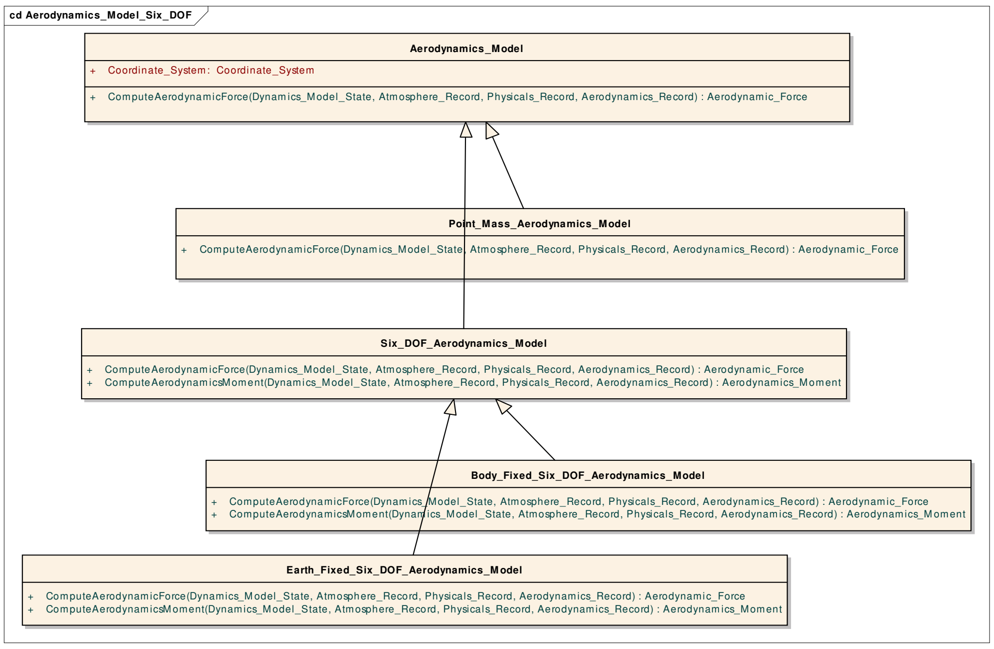

cd Aerodynamics_Model_Six_DOF
Aerodynamics_Model
+ Coordinate_System: Coordinate_System
+ ComputeAerodynamicForce(Dynamics_Model_State, Atmosphere_Record, Physicals_Record, Aerodynamics_Record) : Aerodynamic_Force
Point_Mass_Aerodynamics_Model
+ ComputeAerodynamicForce(Dynamics_Model_State, Atmosphere_Record, Physicals_Record, Aerodynamics_Record) : Aerodynamic_Force
Six_DOF_Aerodynamics_Model
+ ComputeAerodynamicForce(Dynamics_Model_State, Atmosphere_Record, Physicals_Record, Aerodynamics_Record) : Aerodynamic_Force
+ ComputeAerodynamicsMoment(Dynamics_Model_State, Atmosphere_Record, Physicals_Record, Aerodynamics_Record) : Aerodynamics_Moment
Body_Fixed_Six_DOF_Aerodynamics_Model
+ ComputeAerodynamicForce(Dynamics_Model_State, Atmosphere_Record, Physicals_Record, Aerodynamics_Record) : Aerodynamic_Force
+ ComputeAerodynamicsMoment(Dynamics_Model_State, Atmosphere_Record, Physicals_Record, Aerodynamics_Record) : Aerodynamics_Moment
Earth_Fixed_Six_DOF_Aerodynamics_Model
+ ComputeAerodynamicForce(Dynamics_Model_State, Atmosphere_Record, Physicals_Record, Aerodynamics_Record) : Aerodynamic_Force
+ ComputeAerodynamicsMoment(Dynamics_Model_State, Atmosphere_Record, Physicals_Record, Aerodynamics_Record) : Aerodynamics_Moment

Figure 126 Some of Platform Independent Aerodynamics Model Classes

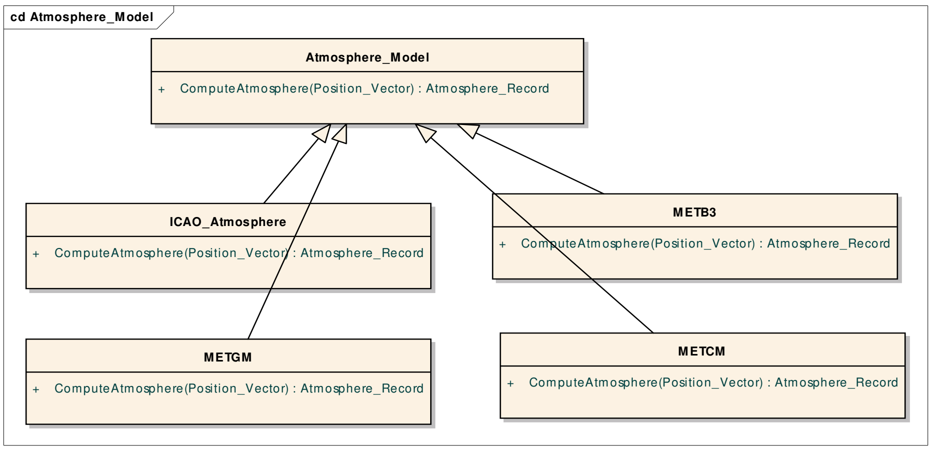

cd Atmosphere_Model
Atmosphere_Model
+ ComputeAtmosphere(Position_Vector) : Atmosphere_Record
ICAO_Atmosphere
+ ComputeAtmosphere(Position_Vector) : Atmosphere_Record
METB3
+ ComputeAtmosphere(Position_Vector) : Atmosphere_Record
METGM
+ ComputeAtmosphere(Position_Vector) : Atmosphere_Record
METCM
+ ComputeAtmosphere(Position_Vector) : Atmosphere_Record

Figure 127 Platform Independent Atmosphere Model Classes

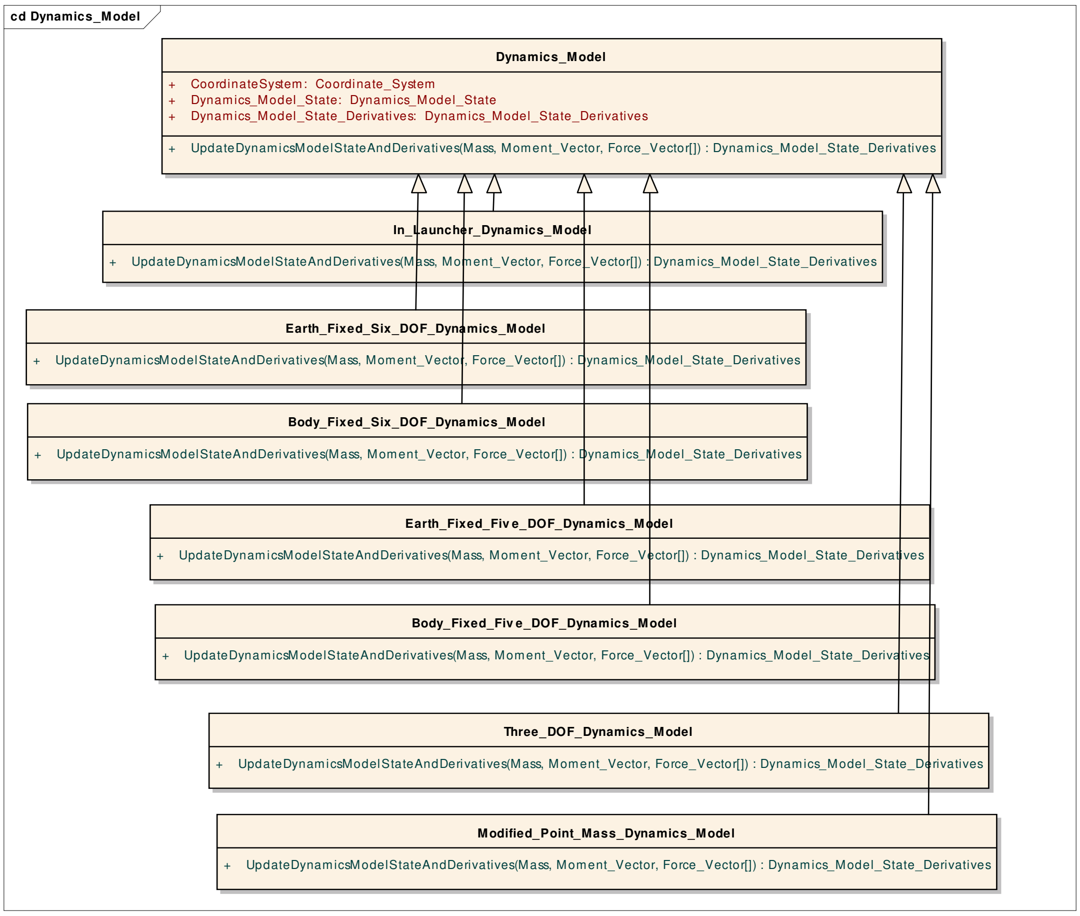

cd Dynamics_Model
Dynamics_Model
+ CoordinateSystem: Coordinate_System
+ Dynamics_Model_State: Dynamics_Model_State
+ Dynamics_Model_State_Derivatives: Dynamics_Model_State_Derivatives
+ UpdateDynamicsModelStateAndDerivatives(Mass, Moment_Vector, Force_Vector[]) : Dynamics_Model_State_Derivatives
In_Launcher_Dynamics_Model
+ UpdateDynamicsModelStateAndDerivatives(Mass, Moment_Vector, Force_Vector[]) : Dynamics_Model_State_Derivatives
Earth_Fixed_Six_DOF_Dynamics_Model
+ UpdateDynamicsModelStateAndDerivatives(Mass, Moment_Vector, Force_Vector[]) : Dynamics_Model_State_Derivatives
Body_Fixed_Six_DOF_Dynamics_Model
+ UpdateDynamicsModelStateAndDerivatives(Mass, Moment_Vector, Force_Vector[]) : Dynamics_Model_State_Derivatives
Earth_Fixed_Five_DOF_Dynamics_Model
+ UpdateDynamicsModelStateAndDerivatives(Mass, Moment_Vector, Force_Vector[]) : Dynamics_Model_State_Derivatives
Body_Fixed_Five_DOF_Dynamics_Model
+ UpdateDynamicsModelStateAndDerivatives(Mass, Moment_Vector, Force_Vector[]) : Dynamics_Model_State_Derivatives
Three_DOF_Dynamics_Model
+ UpdateDynamicsModelStateAndDerivatives(Mass, Moment_Vector, Force_Vector[]) : Dynamics_Model_State_Derivatives
Modified_Point_Mass_Dynamics_Model
+ UpdateDynamicsModelStateAndDerivatives(Mass, Moment_Vector, Force_Vector[]) : Dynamics_Model_State_Derivatives

Figure 128 Platform Independent Dynamics Model Classes

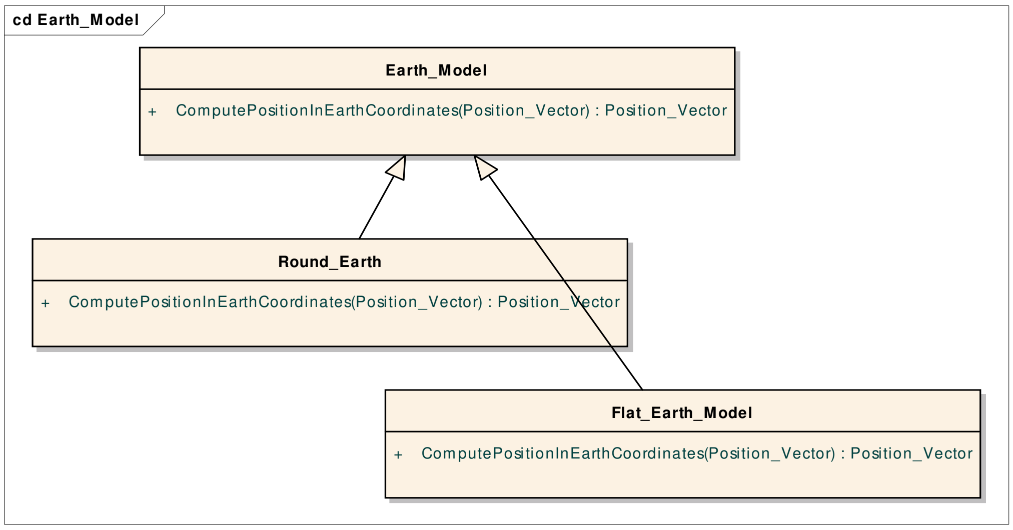

cd Earth_Model
Earth_Model
+ ComputePositionInEarthCoordinates(Position_Vector) : Position_Vector
Round_Earth
+ ComputePositionInEarthCoordinates(Position_Vector) : Position_Vector
Flat_Earth_Model
+ ComputePositionInEarthCoordinates(Position_Vector) : Position_Vector

Figure 129 Platform Independent Earth Model Classes

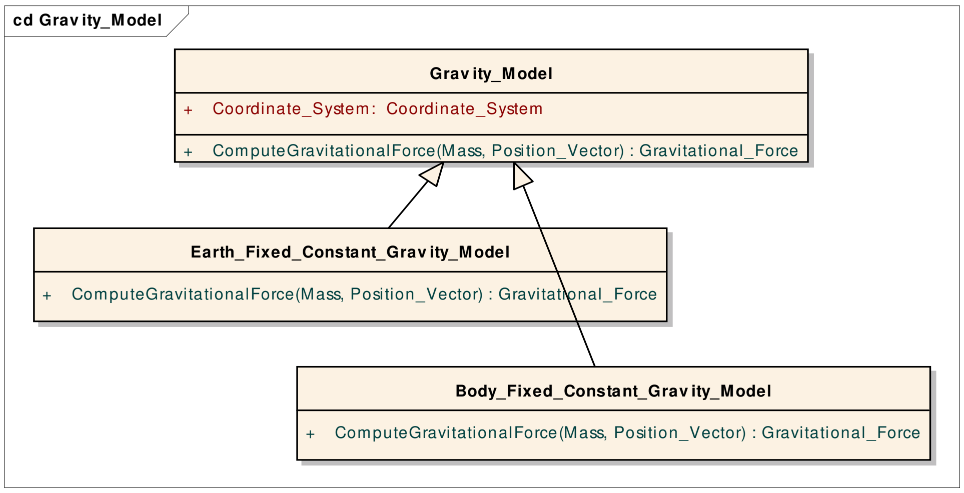

cd Gravity_Model
Gravity_Model
+ Coordinate_System: Coordinate_System
+ ComputeGravitationalForce(Mass, Position_Vector) : Gravitational_Force
Earth_Fixed_Constant_Gravity_Model
+ ComputeGravitationalForce(Mass, Position_Vector) : Gravitational_Force
Body_Fixed_Constant_Gravity_Model
+ ComputeGravitationalForce(Mass, Position_Vector) : Gravitational_Force

Figure 130 Platform Independent Gravity Model Classes

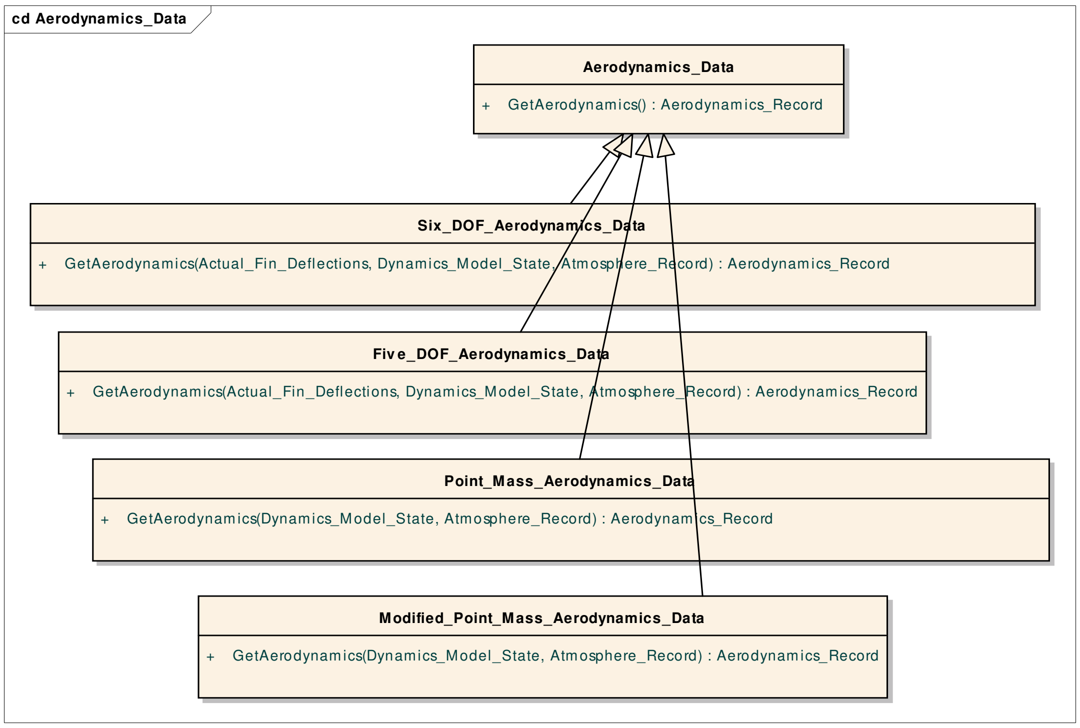

cd Aerodynamics_Data
Aerodynamics_Data
+ GetAerodynamics() : Aerodynamics_Record
Six_DOF_Aerodynamics_Data
+ GetAerodynamics(Actual_Fin_Deflections, Dynamics_Model_State, Atmosphere_Record) : Aerodynamics_Record
Five_DOF_Aerodynamics_Data
+ GetAerodynamics(Actual_Fin_Deflections, Dynamics_Model_State, Atmosphere_Record) : Aerodynamics_Record
Point_Mass_Aerodynamics_Data
+ GetAerodynamics(Dynamics_Model_State, Atmosphere_Record) : Aerodynamics_Record
Modified_Point_Mass_Aerodynamics_Data
+ GetAerodynamics(Dynamics_Model_State, Atmosphere_Record) : Aerodynamics_Record

Figure 131 Platform Independent Aerodynamics Data Classes

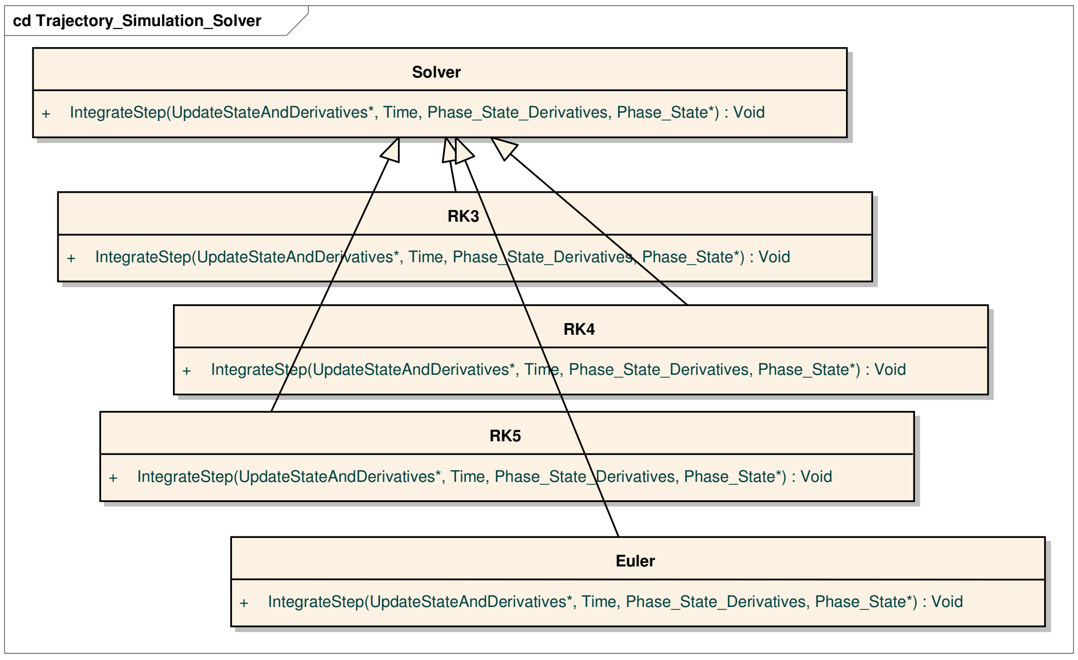

cd Trajectory_Simulation_Solver
Solver
+ IntegrateStep(UpdateStateAndDerivatives*, Time, Phase_State_Derivatives, Phase_State*) : Void
RK3
+ IntegrateStep(UpdateStateAndDerivatives*, Time, Phase_State_Derivatives, Phase_State*) : Void
RK4
+ IntegrateStep(UpdateStateAndDerivatives*, Time, Phase_State_Derivatives, Phase_State*) : Void
RK5
+ IntegrateStep(UpdateStateAndDerivatives*, Time, Phase_State_Derivatives, Phase_State*) : Void
Euler
+ IntegrateStep(UpdateStateAndDerivatives*, Time, Phase_State_Derivatives, Phase_State*) : Void

Figure 132 Platform Independent Solver Classes

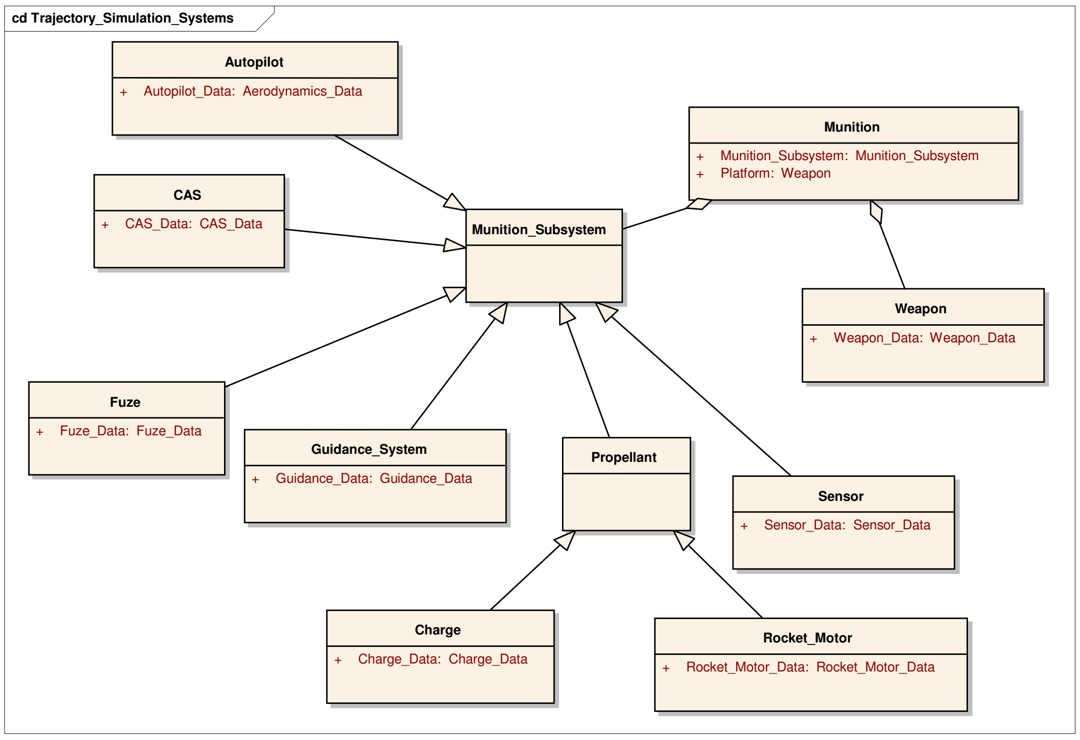

cd Trajectory_Simulation_Systems
Autopilot
+ Autopilot_Data: Aerodynamics_Data
Munition
+ Munition_Subsystem: Munition_Subsystem
+ Platform: Weapon
CAS
+ CAS_Data: CAS_Data
Munition_Subsystem
Weapon
+ Weapon_Data: Weapon_Data
Fuze
+ Fuze_Data: Fuze_Data
Guidance_System
+ Guidance_Data: Guidance_Data
Propellant
Sensor
+ Sensor_Data: Sensor_Data
Charge
+ Charge_Data: Charge_Data
Rocket_Motor
+ Rocket_Motor_Data: Rocket_Motor_Data

Figure 133 Platform Independent Munition Subsystem Classes

APPENDIX E

TSONT TO MATSIX CODE TRACEABILITY

Table 4 TSONT to MATSIX Code Traceability

TSONT	Platform Independent Framework Architecture	MATSIX Architecture	Code
Coordinate System	Coordinate System	cs	@cs
Body_Coordinate_System	Body Coordinate System	bcs	@bcs
Earth_Coordinate_System	Earth Coordinate System	ecs	@ecs
Wind_Coordinate_System	Wind Coordinate System		
Weapon_Coordinate_System	Weapon Coordinate System		
Trajectory_Simulation	Trajectory Simulation	trajectory_simulation	@trajectory_simulation
Trajectory_Simulation_Composite_Data	Trajectory Simulation Composite Data		
Model	Trajectory_Simulation_Models		
Aerodynamics_Model	Aerodynamics_Model		
Body_Fixed_Five_DOF_Aerodynamics_Model	Body_Fixed_Five_DOF_Aerodynamics_Model		
Body_Fixed_Six_DOF_Aerodynamics_Model	Body_Fixed_Six_DOF_Aerodynamics_Model	aerodynamics_model	@aerodynamics_model
Earth_Fixed_Six_DOF_Aerodynamics_Model	Earth_Fixed_Six_DOF_Aerodynamics_Model		
Earth_Fixed_Five_DOF_Aerodynamics_Model	Earth_Fixed_Five_DOF_Aerodynamics_Model		
Five_DOF_Aerodynamics_Model	Five_DOF_Aerodynamics_Model		
Modified_Point_Mass_Aerodynamics_Model	Modified_Point_Mass_Aerodynamics_Model		
Point_Mass_Aerodynamics_Model	Point_Mass_Aerodynamics_Model		
Six_DOF_Aerodynamics_Model	Six_DOF_Aerodynamics_Model		
Three_DOF_Aerodynamics_Model	Three_DOF_Aerodynamics_Model		
Atmosphere_Model	Atmosphere_Model	atmosphere_model	@atmosphere_model
ICAO	ICAO_Atmosphere	icao_atmosphere_model	@icao_atmosphere_model
METB3	METB3		
METCM	METCM		

TSONT	Platform Independent Framework Architecture	MATSIX Architecture	Code
METGM	METGM		
Autopilot_Model	Autopilot_Model	autopilot_model	@autopilot_model
CAS_Model	CAS_Model		
Four_Canard_Second_Order_CAS_Model	Second_Order_CAS_Model	cas_model	@cas_model
Dynamics_Model	Dynamics_Model		
Body_Fixed_Five_DOF_Dynamics_Model	Body_Fixed_Five_DOF_Dynamics_Model		
Body_Fixed_Six_DOF_Dynamics_Model	Body_Fixed_Six_DOF_Dynamics_Model	dynamics_model	@dynamics_model
Earth_Fixed_Five_DOF_Dynamics_Model	Earth_Fixed_Five_DOF_Dynamics_Model		
Earth_Fixed_Six_DOF_Dynamics_Model	Earth_Fixed_Six_DOF_Dynamics_Model		
In_Launcher_Dynamics_Model	In_Launcher_Dynamics_Model	in_launcher_dynamics_model	@in_launcher_dynamics_model
Modified_Point_Mass_Dynamics_Model	Modified_Point_Mass_Dynamics_Model		
Three_DOF_Dynamics_Model	Three_DOF_Dynamics_Model		
Earth_Model	Earth_Model		
Flat_Earth_Model	Flat_Earth_Model	earth_model	@earth_model
Curved_Earth_Model	Round_Earth	round_earth_model	@round_earth_model
Gravity_Model	Gravity_Model		
Constant_G_Body_Fixed_Gravity_Model	Body_Fixed_Constant_Gravity_Model		
Constant_G_Earth_Fixed_Gravity_Model	Earth_Fixed_Constant_Gravity_Model	gravity_model	@gravity_model
Guidance_Model	Guidance_Model	guidance_model	@guidance_model
Cubic_Guidance_Model	Cubic_Guidance_Model	cubic_guidance_model	@cubic_guidance_model
PN_Guidance_Model	PN_Guidance_Model	pn_guidance_model	@pn_guidance_model
Parabolic_Guidance_Model	Parabolic_Guidance_Model	parabolic_guidance_model	@parabolic_guidance_model
Launcher_Model	Launcher_Model		
	Simple_Launcher_Model	launcher_model	@launcher_model
Sensor_Model	Sensor_Model		
Termination_Model	Termination_Model	termination_model	@termination_model
Thruster_Model	Thruster_Model		
Body_Fixed_Six_DOF_Rocket_Motor_Model	Body_Fixed_Six_DOF_Solid_Rocket_Motor_Model		
	Center_Burning_Solid_Rocket_Motor_Model	rocket_motor_model	@rocket_motor_model
	Earth_Fixed_Six_DOF_Solid_Rocket_Motor_Model		@six_dof_rocket_motor_model

TSONT	Platform Independent Framework Architecture	MATSIX Architecture	Code
	End_Burning_Solid_Rocket_Motor_Model		
Solid_Rocket_Motor_Model_for_Point_Mass	Point_Mass_Solid_Rocket_Motor_Model		
Parameter	Trajectory Simulation Parameters		
Aerodynamics	Aerodynamics_Data		
Five_DOF_Aerodynamics	Five_DOF_Aerodynamics_Data		
Modified_Point_Mass_Aerodynamics	Modified_Point_Mass_Aerodynamics_Data		
Point_Mass_Aerodynamics	Point_Mass_Aerodynamics_Data		
Six_DOF_Aerodynamics	Six_DOF_Aerodynamics_Data	aerodynamics	@aerodynamics
Autopilot_Data	Autopilot_Data	autopilot_data	@autopilot_data
CAS_Data	CAS_Data		
Second_Order_CAS_Data	Second_Order_CAS_Data	cas_data	@cas_data
Charge_Data	Charge_Data		
Fuze_Data	Fuze_Data		
Guidance_Data	Guidance_Data	guidance_data	@guidance_system_data
Physical_Data	Physical_Data		
Point_Mass_Physicals	Point_Mass_Physicals		
Six_DOF_Physicals	Six_DOF_Physicals	physicals	@physicals
	Six_DOF_Physicals_for_Thrusted	physicals_for_thrusted	@physicals_for_thrusted
	Propellant_Data		
Solid_Rocket_Motor_Data	Rocket_Motor_Data		
Point_Mass_Solid_Rocket_Motor_Data	Point_Mass_Rocket_Motor_Data		
Rigid_Body_Solid_Rocket_Motor_Data	Rigid_Body_Rocket_Motor_Data	rocket_motor_data	@rocket_motor_data
Sensor_Data	Sensor_Data		
Weapon_Data	Weapon_Data	weapon_data	@weapon_data
Laucher_Data	Laucher_Data	launcher_data	@launcher_data
Trajectory_Simulation_Phase	Trajectory Simulation Phases		
Phase	Phase	phase	@phase
Guided_Phase	Guided_Phase	guided_phase	@guided_phase
In_Launcher_Phase	In_Launcher_Thrusted_Phase	in_launcher_thrusted_phase	@in_launcher_thrusted_phase
Thrusted_Phase	Thrusted_Phase	thrusted_phase	@thrusted_phase
Trajectory Simulation Quantity	Trajectory Simulation Quantities		
Trajectory Simulation Solver	Trajectory Simulation Solvers		

TSONT	Platform Independent Framework Architecture	MATSIX Architecture	Code
Euler	Euler		
Third_Order_RK	RK3		
Fourth_Order_RK	RK4		
Fifth_Order_RK	RK5		
Trajectory Simulation Object	Trajectory Simulation Systems		
Autopilot	Autopilot	autopilot	@autopilot
CAS	CAS	cas	@cas
Charge	Charge		
Fuze	Fuze	fuze	@fuze
Guidance_System	Guidance_System	guidance_system	@guidance_system
Munition	Munition	muntion	@munition
Munition_Subsystem	Munition_Subsystem		
Propellant	Propellant		
Rocket_Motor	Rocket_Motor	rocket_motor	@rocket_motor
Sensor	Sensor		
Weapon	Weapon	weapon	@weapon
		aircraft	@aircraft_data

APPENDIX F

SAMPLE CLASS DIAGRAMS FROM MATSIX ARCHITECTURE

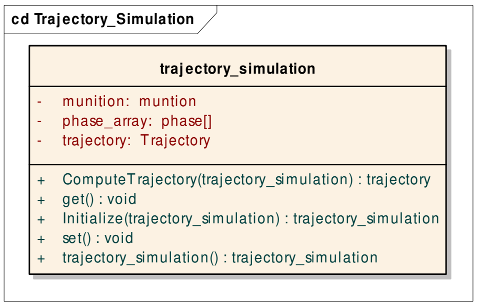

cd Trajectory_Simulation
trajectory_simulation
- munition: muntion
- phase_array: phase[]
- trajectory: Trajectory
+ ComputeTrajectory(trajectory_simulation) : trajectory
+ get() : void
+ Initialize(trajectory_simulation) : trajectory_simulation
+ set() : void
+ trajectory_simulation() : trajectory_simulation

Figure 134 MATSIX Trajectory Simulation

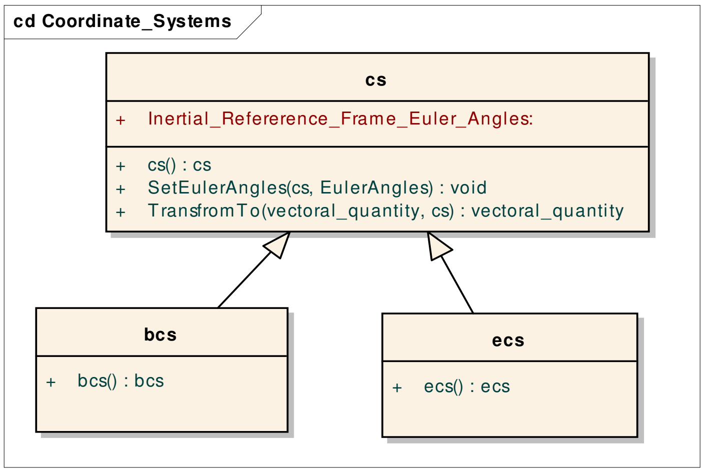

cd Coordinate_Systems
cs
+ Inertial_Refererence_Frame_Euler_Angles:
+ cs() : cs
+ SetEulerAngles(cs, EulerAngles) : void
+ TransfromTo(vectoral_quantity, cs) : vectoral_quantity
bcs
+ bcs() : bcs
ecs
+ ecs() : ecs

Figure 135 MATSIX Coordinate System Classes

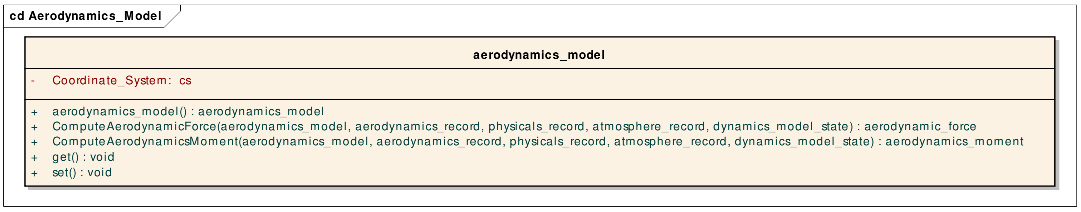

cd Aerodynamics_Model
aerodynamics_model
- Coordinate_System: cs
+ aerodynamics_model() : aerodynamics_model
+ ComputeAerodynamicForce(aerodynamics_model, aerodynamics_record, physicals_record, atmosphere_record, dynamics_model_state) : aerodynamic_force
+ ComputeAerodynamicsMoment(aerodynamics_model, aerodynamics_record, physicals_record, atmosphere_record, dynamics_model_state) : aerodynamics_moment
+ get() : void
+ set() : void

Figure 136 MATSIX Aerodynamics Model

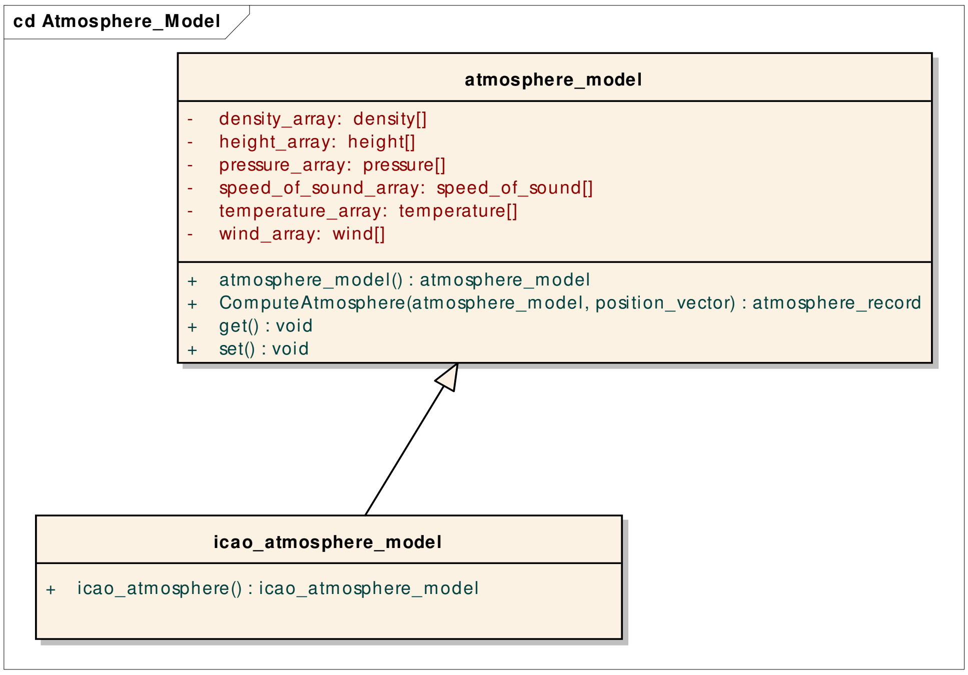

cd Atmosphere_Model
atmosphere_model
- density_array: density[]
- height_array: height[]
- pressure_array: pressure[]
- speed_of_sound_array: speed_of_sound[]
- temperature_array: temperature[]
- wind_array: wind[]
+ atmosphere_model() : atmosphere_model
+ ComputeAtmosphere(atmosphere_model, position_vector) : atmosphere_record
+ get() : void
+ set() : void
icao_atmosphere_model
+ icao_atmosphere() : icao_atmosphere_model

Figure 137 MATSIX Atmosphere Model

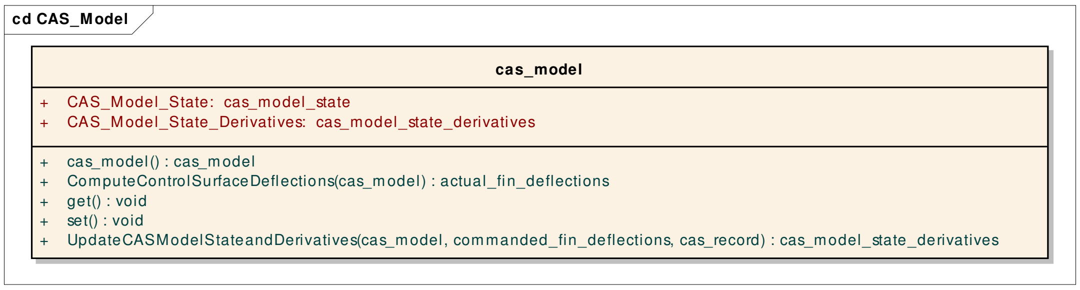

cd CAS_Model
cas_model
+ CAS_Model_State: cas_model_state
+ CAS_Model_State_Derivatives: cas_model_state_derivatives
+ cas_model() : cas_model
+ ComputeControlSurfaceDeflections(cas_model) : actual_fin_deflections
+ get() : void
+ set() : void
+ UpdateCASModelStateandDerivatives(cas_model, commanded_fin_deflections, cas_record) : cas_model_state_derivatives

Figure 138 MATSIX CAS Model

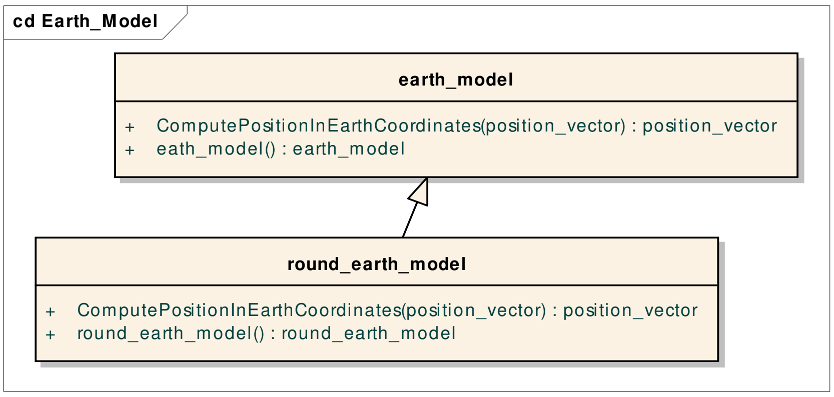

cd Earth_Model
earth_model
+ ComputePositionInEarthCoordinates(position_vector) : position_vector
+ eath_model() : earth_model
round_earth_model
+ ComputePositionInEarthCoordinates(position_vector) : position_vector
+ round_earth_model() : round_earth_model

Figure 139 TSONT Earth Model

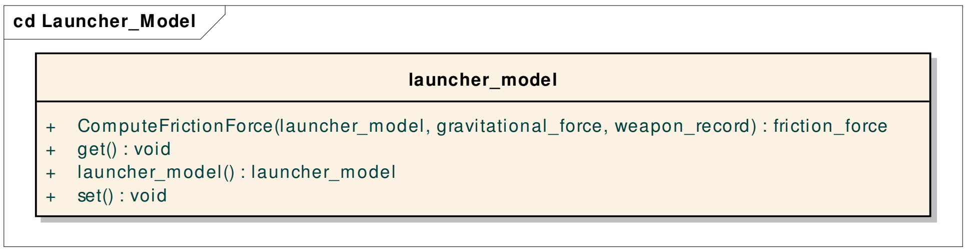

cd Launcher_Model
launcher_model
+ ComputeFrictionForce(launcher_model, gravitational_force, weapon_record) : friction_force
+ get() : void
+ launcher_model() : launcher_model
+ set() : void

Figure 140 MATSIX Launcher Model

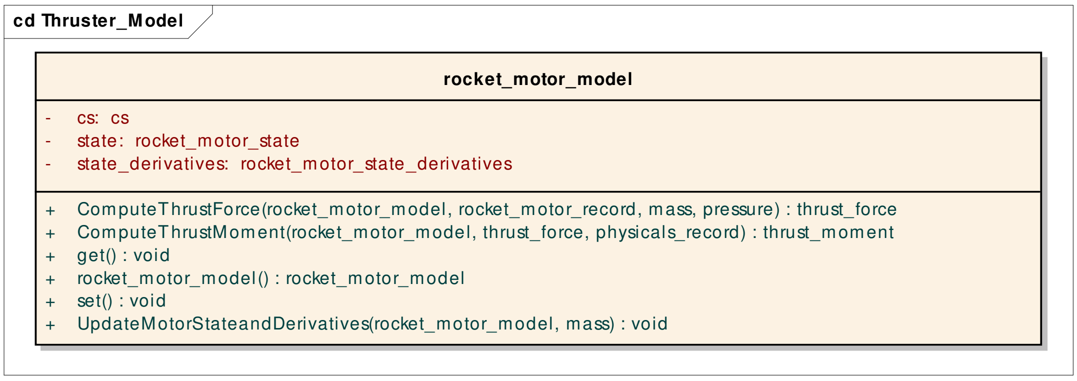

cd Thruster_Model
rocket_motor_model
- cs: cs
- state: rocket_motor_state
- state_derivatives: rocket_motor_state_derivatives
+ ComputeThrustForce(rocket_motor_model, rocket_motor_record, mass, pressure) : thrust_force
+ ComputeThrustMoment(rocket_motor_model, thrust_force, physicals_record) : thrust_moment
+ get() : void
+ rocket_motor_model() : rocket_motor_model
+ set() : void
+ UpdateMotorStateandDerivatives(rocket_motor_model, mass) : void

Figure 141 MATSIX Rocket Motor Model

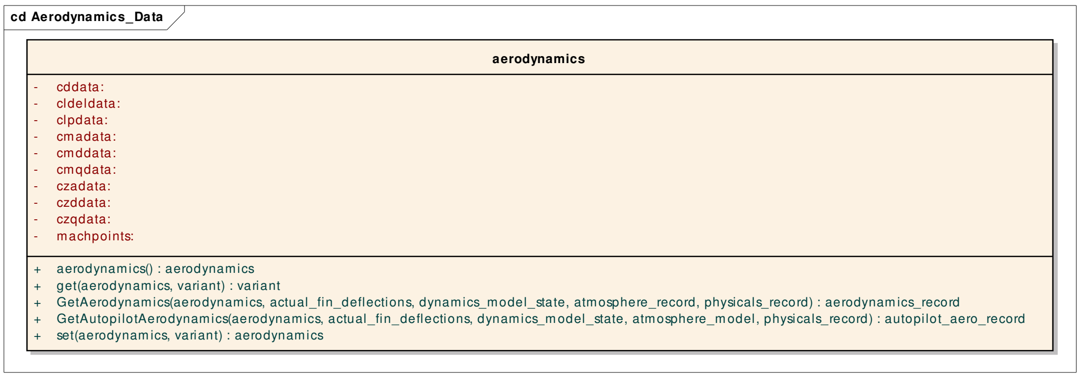

cd Aerodynamics_Data
aerodynamics
- cddata:
- cldeldata:
- clpdata:
- cmadata:
- cmddata:
- cmqdata:
- czadata:
- czddata:
- czqdata:
- machpoints:
+ aerodynamics() : aerodynamics
+ get(aerodynamics, variant) : variant
+ GetAerodynamics(aerodynamics, actual_fin_deflections, dynamics_model_state, atmosphere_record, physicals_record) : aerodynamics_record
+ GetAutopilotAerodynamics(aerodynamics, actual_fin_deflections, dynamics_model_state, atmosphere_model, physicals_record) : autopilot_aero_record
+ set(aerodynamics, variant) : aerodynamics

Figure 142 MATSIX Aerodynamics

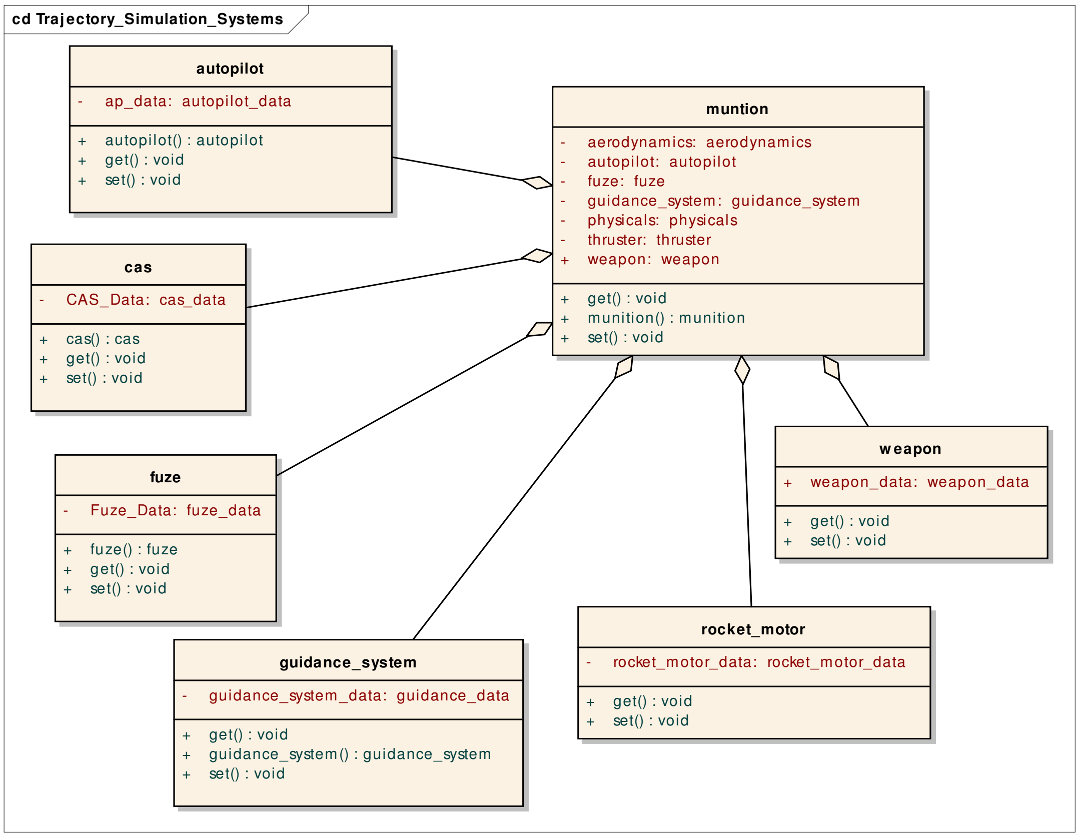


Figure 143 MATSIX Trajectory Simulation Systems

APPENDIX G

LYNX DATA

Table 5 LYNX Data

PHYSICALS	
Diameter	0.227 m
Length	2.7 m
Reference Mass	135.1 kg
Reference CG	1.7 m
Reference Inertia Matrix	[1.16 0 0; 0 125 0; 0 0 125] kgm2
Initial CG	1.97 m
Initial Inertia Matrix	[1.65 0 0; 0 165 0; 0 0 165] kgm2
AERODYNAMICS	
Mach Points	[0 .1 .33 .53 .71 .86 1.00 1.05 1.12 1.19 1.27 1.36 1.46 1.58 1.71 1.87 2.04 2.23 2.46 2.71 3.00]
Cd_0	[0 -.3471 -.3563 -.3626 -.3620 -.4187 -.6075 -.6821 -.6427 -.5933 -.5504 -.5208 -.5010 -.4859 -.4809 -.4731 -.4812 -.4423 -.4114 -.3819 -.3508]
Cz_α	[0 -.2684 -.2731 -.2792 -.2852 -.2993 -.3378 -.3402 -.3368 -.3338 -.3196 -.3110 -.2964 -.2674 -.2506 -.2361 -.2218 -.2062 -.1914 -.1776 -.1639]
Cz_δ	[0 .04710 .04826 .04966 .05160 .05647 .06867 .06111 .05658 .05286 .05045 .04882 .04528 .04482 .04250 .04062 .03865 .03586 .03325 .03077 .02808]

Cm_α	[0 -.2501 -.2462 -.2425 -.2268 -.1908 -.09181 -.2532 -.3466 -.4242 -.3229 -.3217 -.3555 -.2542 -.2178 -.1743 -.1366 -.1065 -.08035 -.05590 -.03951]
Cm_δ	[0 -.2830 -.2933 -.3064 -.3206 -.3537 -.4417 -.4402 -.4299 -.4193 -.3981 -.3715 -.3224 -.2670 -.2314 -.2045 -.1819 -.1596 -.1406 -.1241 -.1074]
Cmq	[0 27.30 29.98 33.68 39.51 30.80 27.09 27.63 26.75 25.30 17.24 16.82 16.20 14.29 13.46 12.67 12.26 11.53 10.50 9.806 9.202]
Clp	[0 .02245 .02261 .02256 .02311 .02529 .02585 .01605 .01136 .007408 .007540 .009875 .01189 .01540 .01822 .02086 .02286 .02316 .02316 .02288 .02215]
Cl_δ	[0 .01194 .01203 .01201 .01232 .01352 .01384 .007377 .004822 .002651 .002744 .004050 .005188 .007141 .008722 .01018 .01127 .01144 .01144 .01129 .01089]
AUTOPILOT DATA	
Pitch and Yaw Autopilot	
Wn	10
Ksi	0.707
Mu	1
Roll Autopilot	
Wn	12
Ksi	0.7
LAUNCHER DATA	
Friction Coefficient	0
Launcher Length	3
ROCKET MOTOR DATA	

Specific Impulse	2100 Ns/kg
Exit Area	0.03m2
Reference Pressure	101325.018Pa
Reference Fuel Mass	108.14595 kg
Mass Flow Data	[0 0;
Time(s) vs	0.049 30.295908145179933;
Mass Flow (kg/s)	0.099 29.502196193265007;
	0.149 28.785543654157355;
	0.199 28.330893118594435;
	0.249 27.953301995838792;
	0.299 27.614240579486783;
	0.349 27.271326192494413;
	0.399 26.970794482546044;
	0.449 26.516143946983124;
	0.499 26.215612237034755;
	0.549 25.83802111427911;
	0.599 25.572166140094012;
	0.649 25.421900285119825;
	0.699 25.15604531093473;
	0.749 25.08283886876782;
	0.799 25.005779455960546;
	0.849 24.85551360098636;
	0.899 24.778454188179087;
	0.949 24.6281883332049;
	0.999 24.589658626801263;
	1.049 24.55498189103799;

	1.099 24.40086306542344;
	1.149 24.289126916852894;
	1.199 24.212067504045617;
	1.249 24.061801649071434;
	1.299 24.250597210449257;
	1.349 23.950065500500884;
	1.399 23.950065500500884;
	1.449 23.87300608769361;
	1.499 23.87300608769361;
	1.549 23.834476381289974;
	1.599 23.761269939123064;
	1.649 23.684210526315788;
	1.699 23.684210526315788;
	1.749 23.684210526315788;
	1.799 23.64568081991215;
	1.849 23.64568081991215;
	1.899 23.495414964937968;
	1.949 23.456885258534328;
	1.999 23.41835555213069;
	2.049 23.26808969715651;
	2.099 22.967557987208135;
	2.149 22.9290282808045;
	2.199 22.817292132233952;
	2.249 22.663173306619402;
	2.299 22.47437774524158;
	2.349 22.362641596671033;

	2.399 22.362641596671033;
	2.449 22.51290745164522;
	2.499 22.51290745164522;
	2.549 22.135316328889573;
	2.599 21.00254296062264;
	2.649 19.037527934037143;
	2.699 16.317330661940357;
	2.749 13.18486553132465;
	2.799 10.726670262772597;
	2.849 8.800184942590738;
	2.899 7.405409570779071;
	2.949 6.157047083301225;
	2.999 5.324805424982661;
	3.049 4.457887030900824;
	3.099 3.702704785389535;
	3.149 3.059258688448794;
	3.199 2.45434229791169;
	3.249 1.92648532018186;
	3.299 1.471834784618941;
	3.349 1.059566926100023;
	3.399 0.678122832704015;
	3.449 0.339061416352007;
	3.499 0.077059412807274;
	3.52 0;]

APPENDIX H

LYNX SAMPLE RUNS

Table 6 1st LYNX Sample Run Parameters

Elevation	40 Deg.
Azimuth	0 Deg.
Guidance Start Range	8000 m
Target Range	25000 m
Vertical Angle of Fall	45 Deg.
Horizontal Angle of Fall	0 Deg.

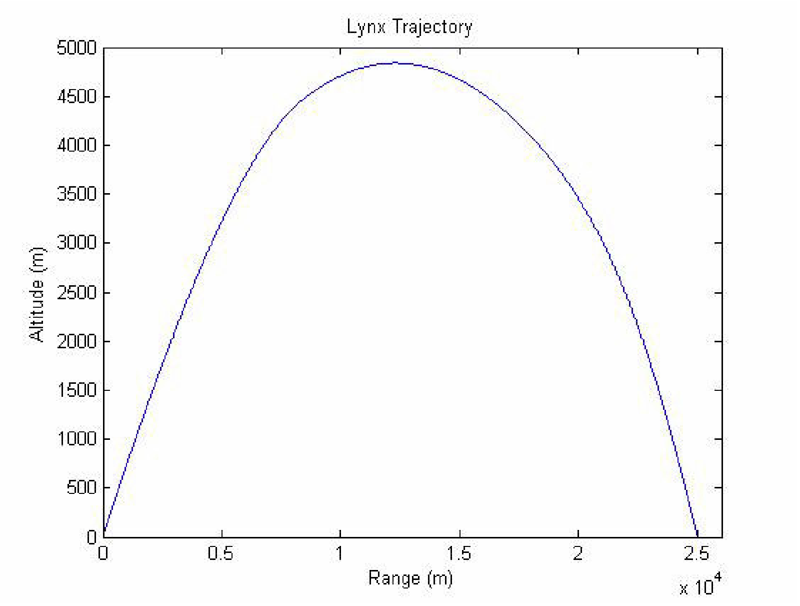


Figure 144 1st LYNX Sample Run Trajectory Plot

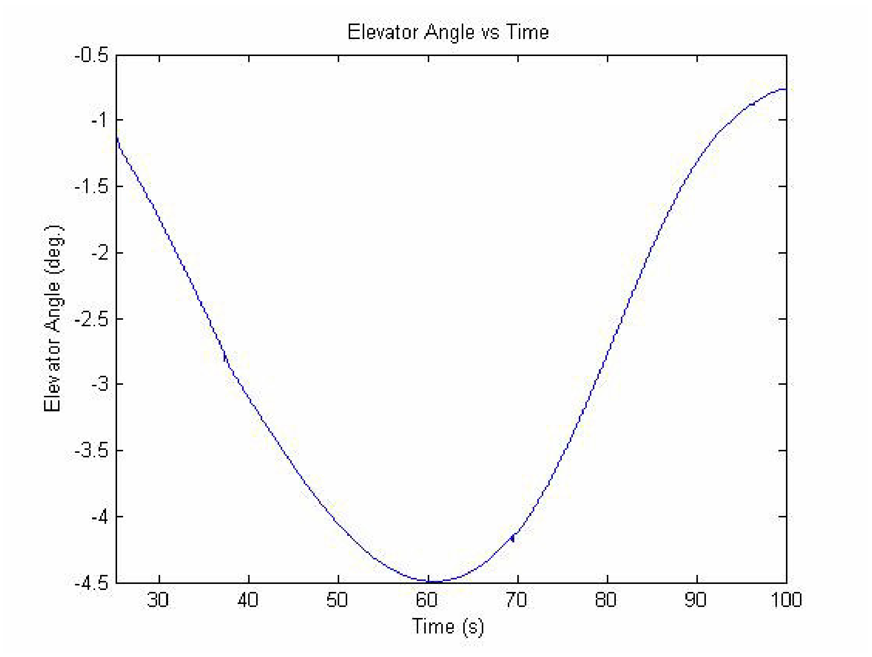


Figure 1451st LYNX Sample Run Elevator Angle vs. Time Plot

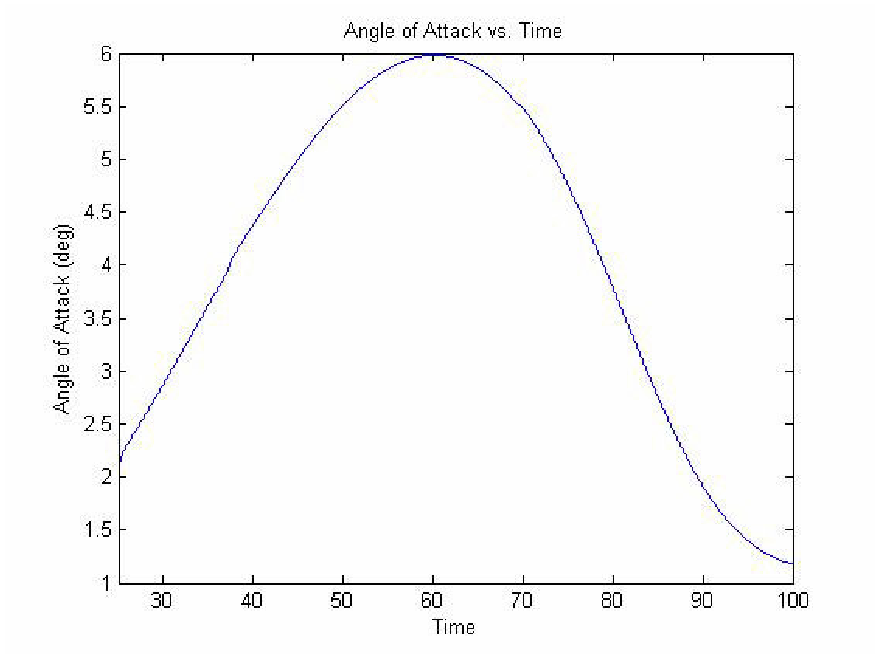


Figure 146 1st LYNX Sample Run Elevator Angle of Attack vs. Time Plot

Table 7 2nd LYNX Sample Run Parameters

Elevation	55 Deg.
Azimuth	0 Deg.
Guidance Start Range	8000 m
Target Range	25000 m
Vertical Angle of Fall	45 Deg.
Horizontal Angle of Fall	0 Deg.

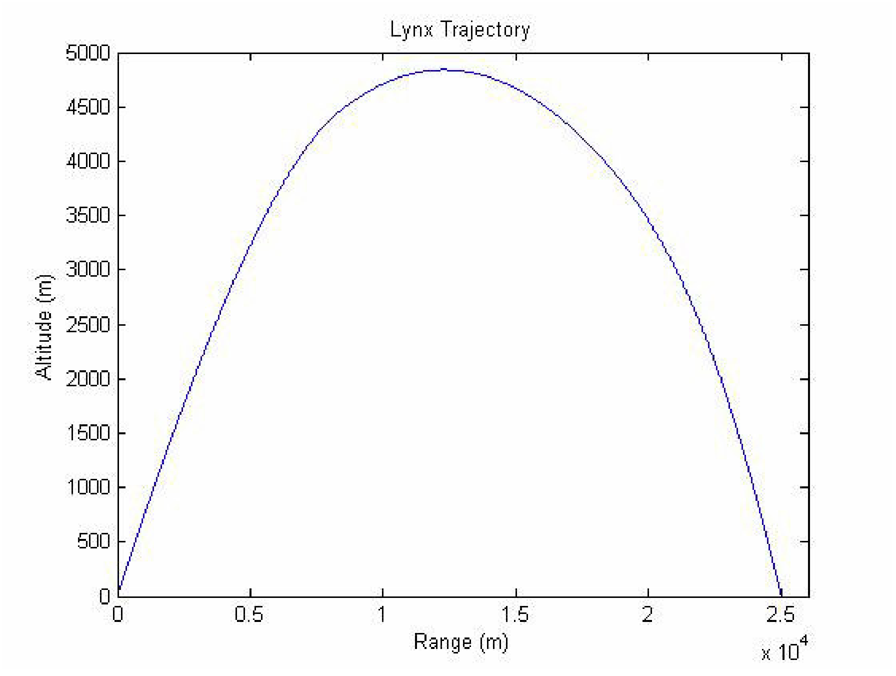


Figure 147 2nd LYNX Sample Run Trajectory Plot

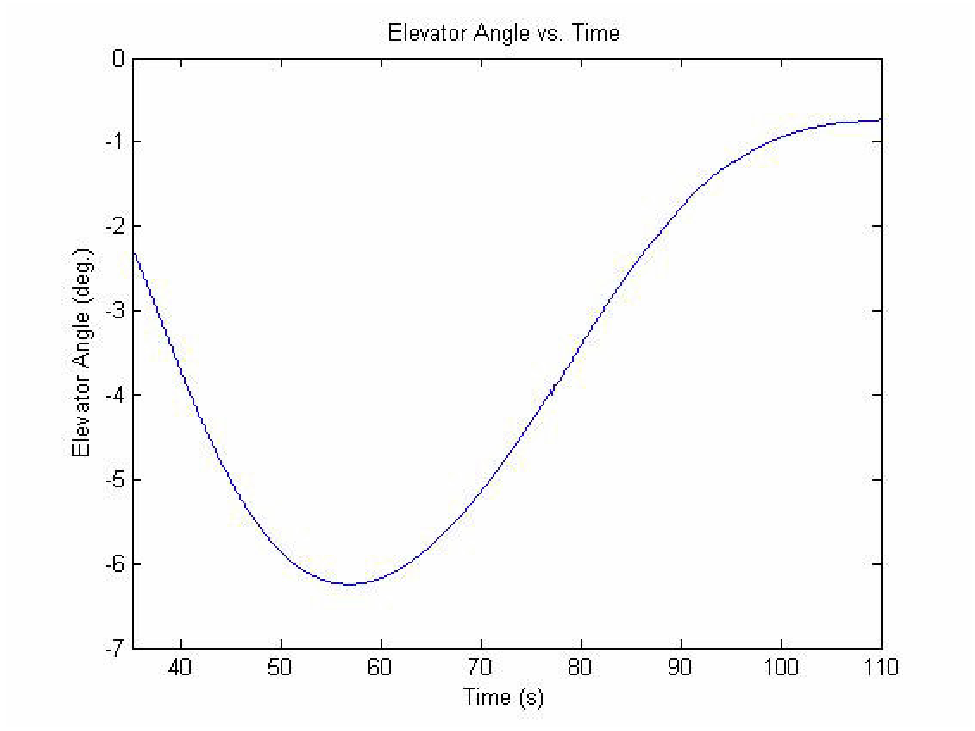


Figure 148 2nd LYNX Sample Run Elevator Angle vs. Time Plot

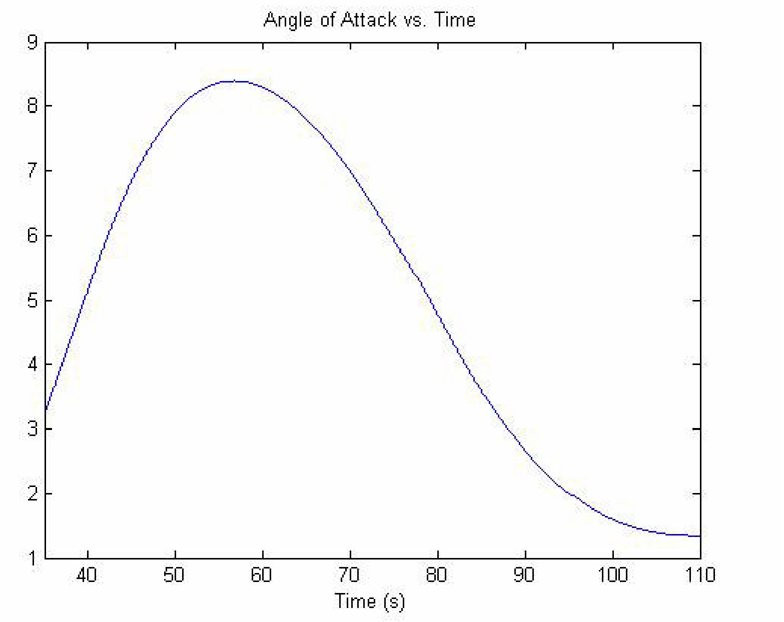


Figure 149 2nd LYNX Sample Run Elevator Angle of Attack vs. Time Plot

APPENDIX I

PUMA DATA

Table 8 PUMA Data

PHYSICALS	
Diameter	0.227 m
Length	2.7 m
Reference Mass	135.1 kg
Reference CG	1.7 m
Reference Inertia Matrix	[1.16 0 0; 0 125 0; 0 0 125] kgm2
AERODYNAMICS	
Mach Points	[0 .1 .33 .53 .71 .86 1.00 1.05 1.12 1.19 1.27 1.36 1.46 1.58 1.71 1.87 2.04 2.23 2.46 2.71 3.00]
Cd_0	[0 -.3471 -.3563 -.3626 -.3620 -.4187 -.6075 -.6821 -.6427 -.5933 -.5504 -.5208 -.5010 -.4859 -.4809 -.4731 -.4812 -.4423 -.4114 -.3819 -.3508]
Cz_α	[0 -.2684 -.2731 -.2792 -.2852 -.2993 -.3378 -.3402 -.3368 -.3338 -.3196 -.3110 -.2964 -.2674 -.2506 -.2361 -.2218 -.2062 -.1914 -.1776 -.1639]
Cz_δ	[0 .04710 .04826 .04966 .05160 .05647 .06867 .06111 .05658 .05286 .05045 .04882 .04528 .04482 .04250 .04062 .03865 .03586 .03325 .03077 .02808]
Cm_α	[0 -.2501 -.2462 -.2425 -.2268 -.1908 -.09181 -.2532 -.3466 -.4242 -.3229 -.3217 -.3555 -.2542 -.2178 -.1743 -.1366 -.1065 -.08035 -.05590 -.03951]

Cm_{δ}	[0 -.2830 -.2933 -.3064 -.3206 -.3537 -.4417 -.4402 -.4299 -.4193 -.3981 -.3715 -.3224 -.2670 -.2314 -.2045 -.1819 -.1596 -.1406 -.1241 -.1074]
Cmq	[0 27.30 29.98 33.68 39.51 30.80 27.09 27.63 26.75 25.30 17.24 16.82 16.20 14.29 13.46 12.67 12.26 11.53 10.50 9.806 9.202]
Clp	[0 .02245 .02261 .02256 .02311 .02529 .02585 .01605 .01136 .007408 .007540 .009875 .01189 .01540 .01822 .02086 .02286 .02316 .02316 .02288 .02215]
Cl_{δ}	[0 .01194 .01203 .01201 .01232 .01352 .01384 .007377 .004822 .002651 .002744 .004050 .005188 .007141 .008722 .01018 .01127 .01144 .01144 .01129 .01089]
AUTOPILOT DATA	
Pitch and Yaw Autopilot	
Wn	10
Ksi	0.707
Mu	1
Roll Autopilot	
Wn	12
Ksi	0.7

APPENDIX J

PUMA SAMPLE RUNS

Table 9 1st PUMA Sample Run Parameters

Altitude	11000 m
Velocity	250 m/s.
Guidance Start Range	200 m
Target Range	20000 m

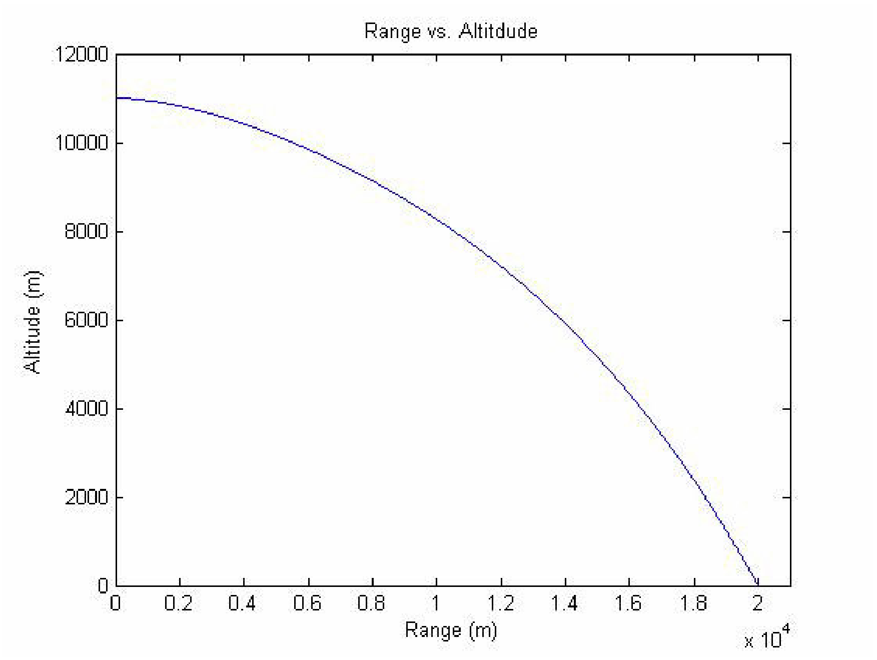


Figure 150 1st PUMA Sample Run Trajectory Plot

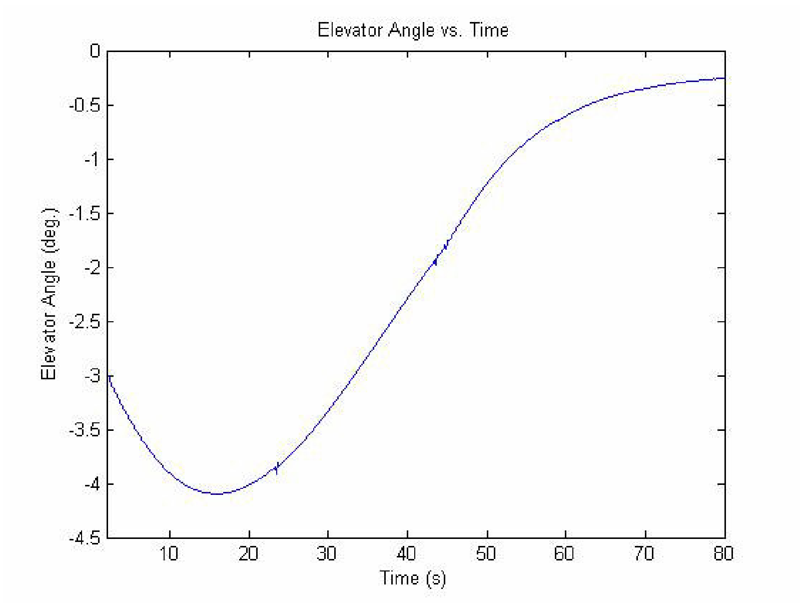


Figure 151 1st PUMA Sample Run Elevator Angle vs. Time Plot

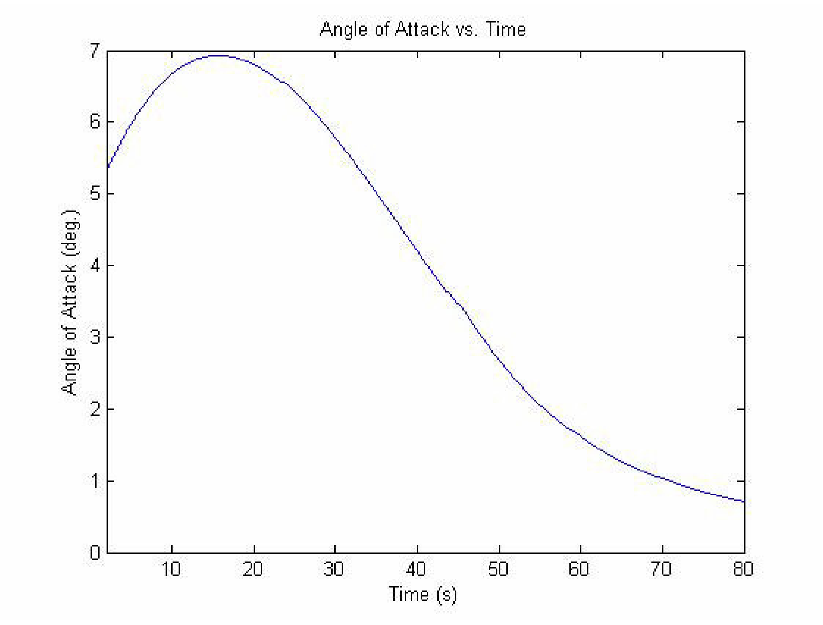


Figure 152 1st PUMA Sample Run Angle of Attack vs. Time

Table 10 2nd PUMA Sample Run Parameters

Altitude	11000 m
Velocity	250 m/s.
Guidance Start Range	200 m
Target Range	10000 m

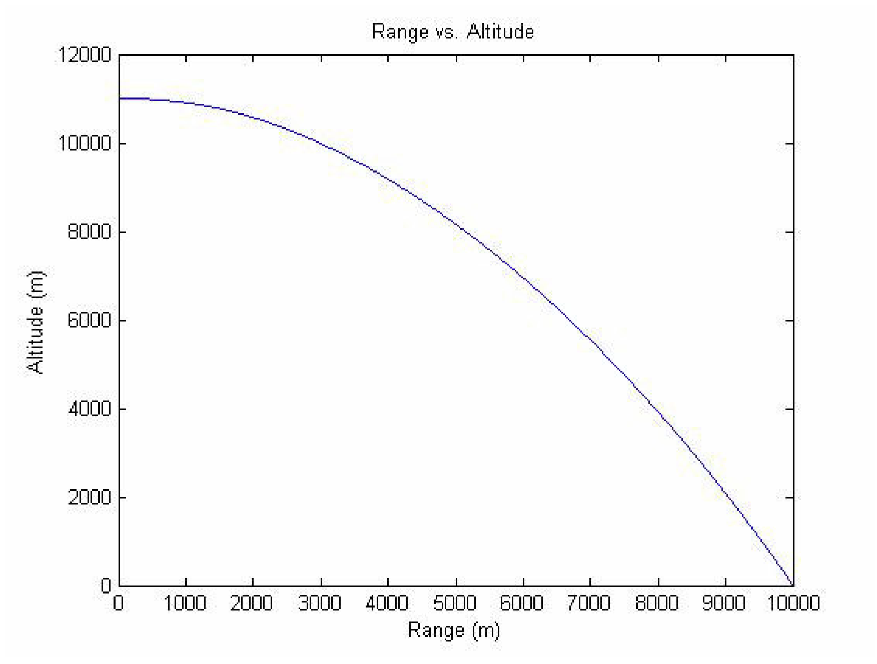


Figure 153 2nd PUMA Sample Run Trajectory Plot

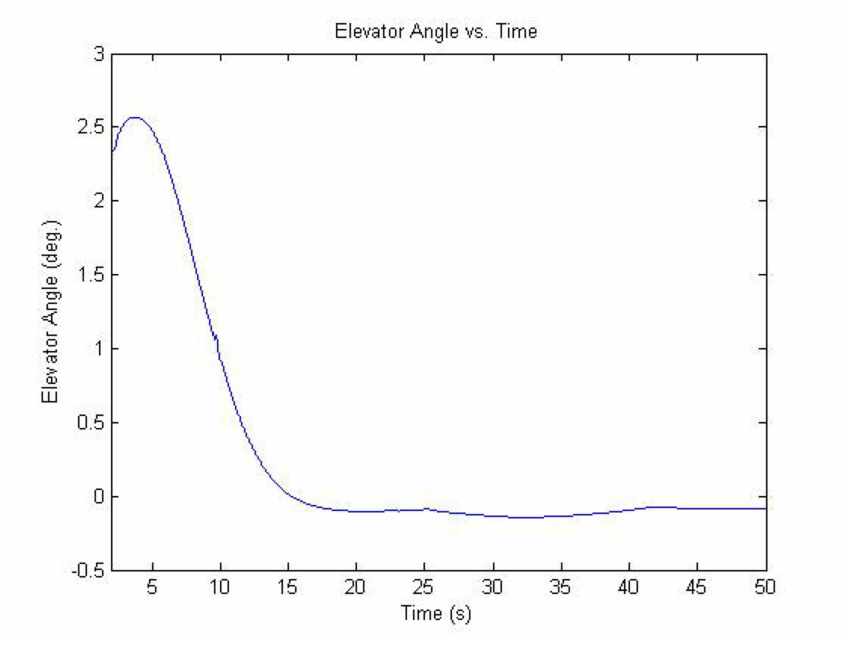


Figure 154 2nd PUMA Sample Run Elevator Angle vs. Time Plot

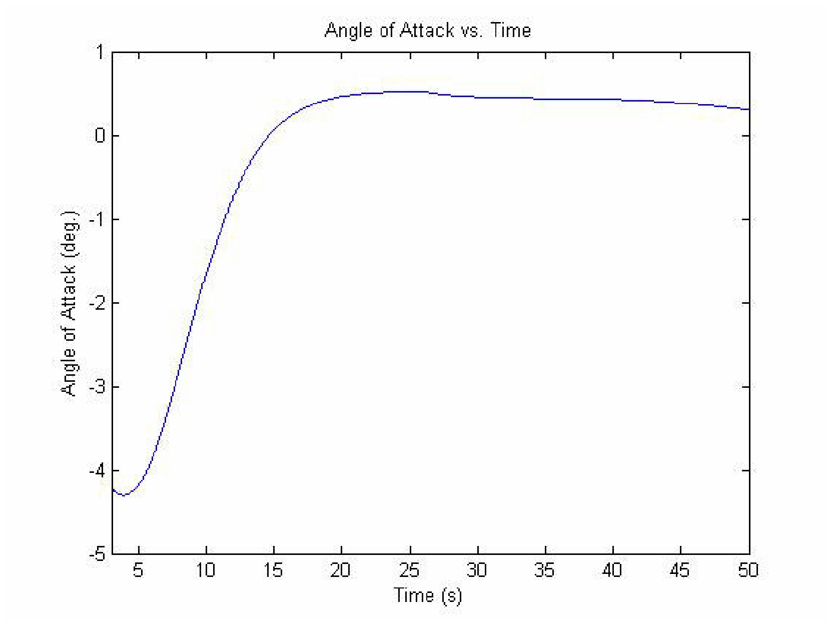


Figure 155 2nd PUMA Sample Run Angle of Attack vs. Time Plot

APPENDIX K

SAMPLE DIAGRAMS AND CODE FORM C# POINT MASS TRAJECTORY SIMULATION FRAMEWORK

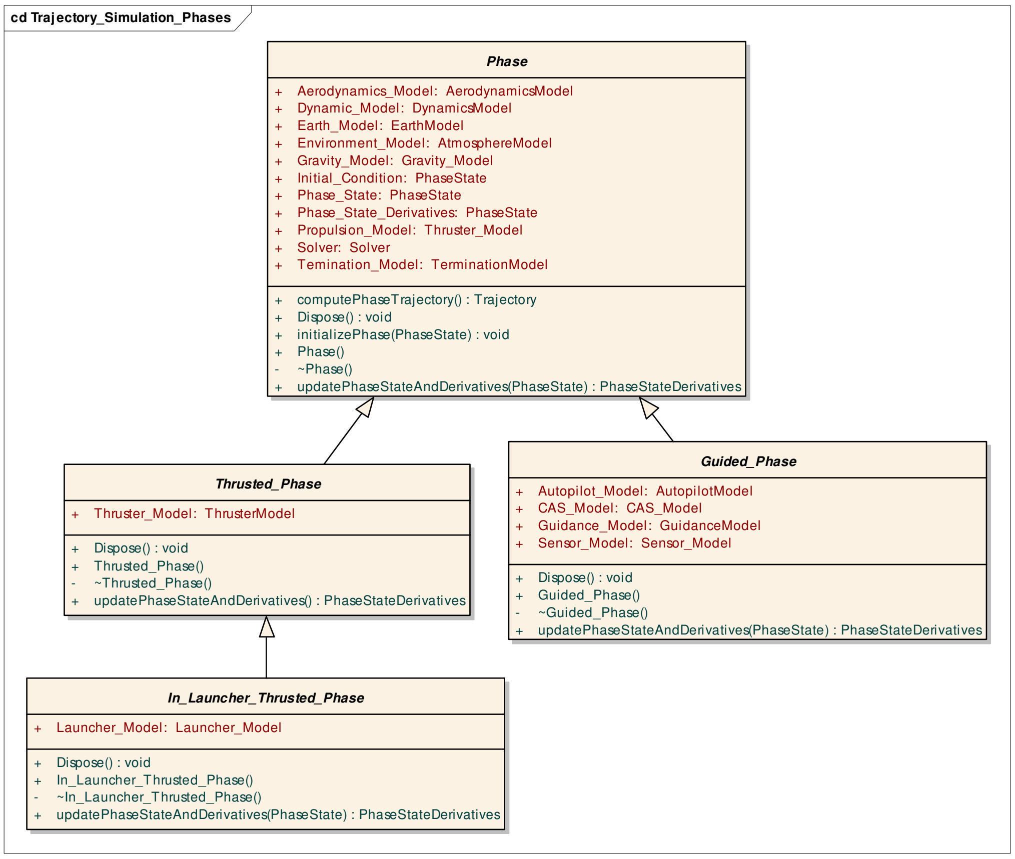

cd Trajectory_Simulation_Phases
Phase
+ Aerodynamics_Model: AerodynamicsModel
+ Dynamic_Model: DynamicsModel
+ Earth_Model: EarthModel
+ Environment_Model: AtmosphereModel
+ Gravity_Model: Gravity_Model
+ Initial_Condition: PhaseState
+ Phase_State: PhaseState
+ Phase_State_Derivatives: PhaseState
+ Propulsion_Model: Thruster_Model
+ Solver: Solver
+ Temination_Model: TerminationModel
+ computePhaseTrajectory() : Trajectory
+ Dispose() : void
+ initializePhase(PhaseState) : void
+ Phase()
- ~Phase()
+ updatePhaseStateAndDerivatives(PhaseState) : PhaseStateDerivatives
Thrusted_Phase
+ Thruster_Model: ThrusterModel
+ Dispose() : void
+ Thrusted_Phase()
- ~Thrusted_Phase()
+ updatePhaseStateAndDerivatives() : PhaseStateDerivatives
Guided_Phase
+ Autopilot_Model: AutopilotModel
+ CAS_Model: CAS_Model
+ Guidance_Model: GuidanceModel
+ Sensor_Model: Sensor_Model
+ Dispose() : void
+ Guided_Phase()
- ~Guided_Phase()
+ updatePhaseStateAndDerivatives(PhaseState) : PhaseStateDerivatives
In_Launcher_Thrusted_Phase
+ Launcher_Model: Launcher_Model
+ Dispose() : void
+ In_Launcher_Thrusted_Phase()
- ~In_Launcher_Thrusted_Phase()
+ updatePhaseStateAndDerivatives(PhaseState) : PhaseStateDerivatives

Figure 156 Phase Hierarchy of C# Point Mass Trajectory Simulation Framework

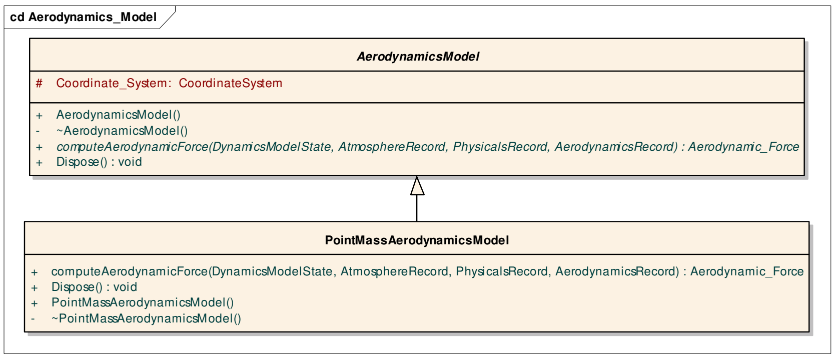
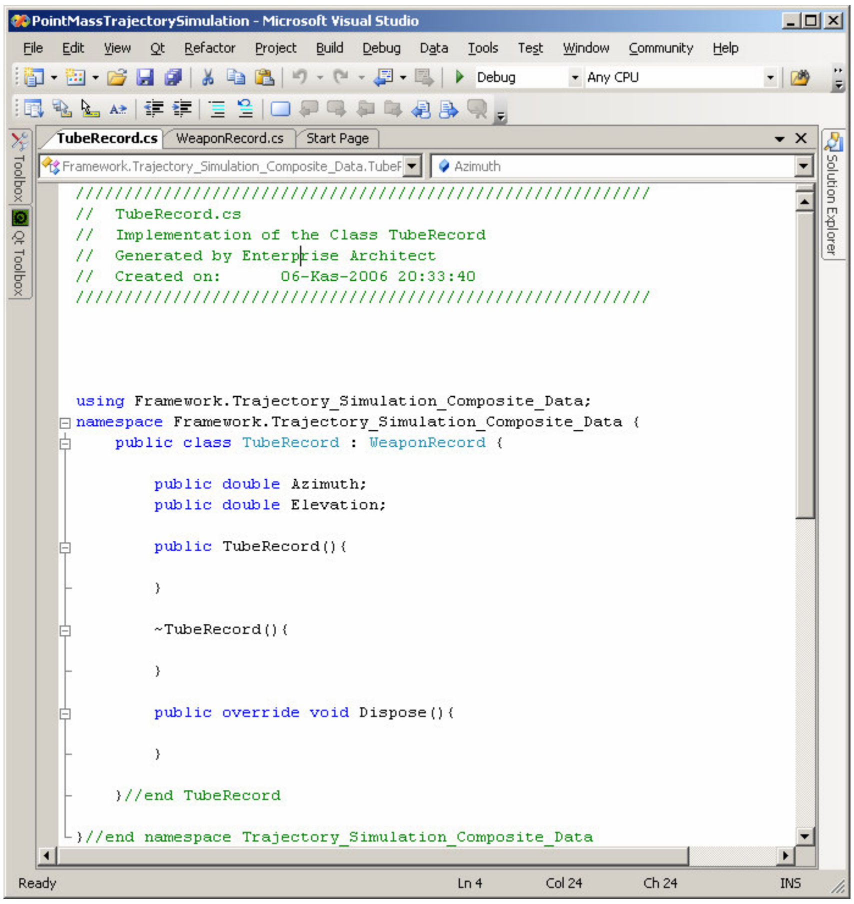

cd Aerodynamics_Model
AerodynamicsModel
# Coordinate_System: CoordinateSystem
+ AerodynamicsModel()
- ~AerodynamicsModel()
+ computeAerodynamicForce(DynamicsModelState, AtmosphereRecord, PhysicalsRecord, AerodynamicsRecord) : Aerodynamic_Force
+ Dispose() : void
PointMassAerodynamicsModel
+ computeAerodynamicForce(DynamicsModelState, AtmosphereRecord, PhysicalsRecord, AerodynamicsRecord) : Aerodynamic_Force
+ Dispose() : void
+ PointMassAerodynamicsModel()
- ~PointMassAerodynamicsModel()

Figure 157 Aerodynamics Model Hierarchy of C# Point Mass Trajectory Simulation Framework


PointMassTrajectorySimulation - Microsoft Visual Studio
File Edit View Qt Refactor Project Build Debug Data Tools Test Window Community Help
Debug
Any CPU
TubeRecord.cs
WeaponRecord.cs
Start Page
Framework.Trajectory_Simulation_Composite_Data.TubeR
Azimuth
Toolbox
Qt Toolbox
Solution Explorer
Ready
Ln 4
Col 24
Ch 24
INS

```
///////////////////////////////////////////////////////////
//  TubeRecord.cs
//  Implementation of the Class TubeRecord
//  Generated by Enterprise Architect
//  Created on:      06-Kas-2006 20:33:40
///////////////////////////////////////////////////////////




using Framework.Trajectory_Simulation_Composite_Data;
namespace Framework.Trajectory_Simulation_Composite_Data {
    public class TubeRecord : WeaponRecord {

        public double Azimuth;
        public double Elevation;

        public TubeRecord(){

        }

        ~TubeRecord(){

        }

        public override void Dispose(){

        }

    }//end TubeRecord

}//end namespace Trajectory_Simulation_Composite_Data
```

Figure 158 Automatically Generated Tube Record Code of C# Point Mass Trajectory Simulation Framework

APPENDIX L

DATA FLOW DIAGRAMS

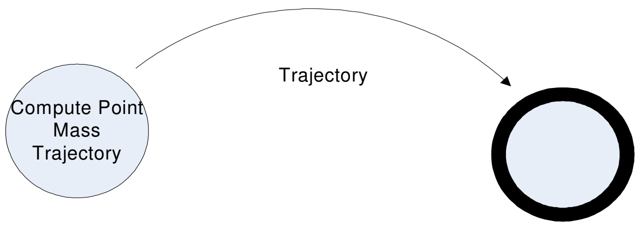
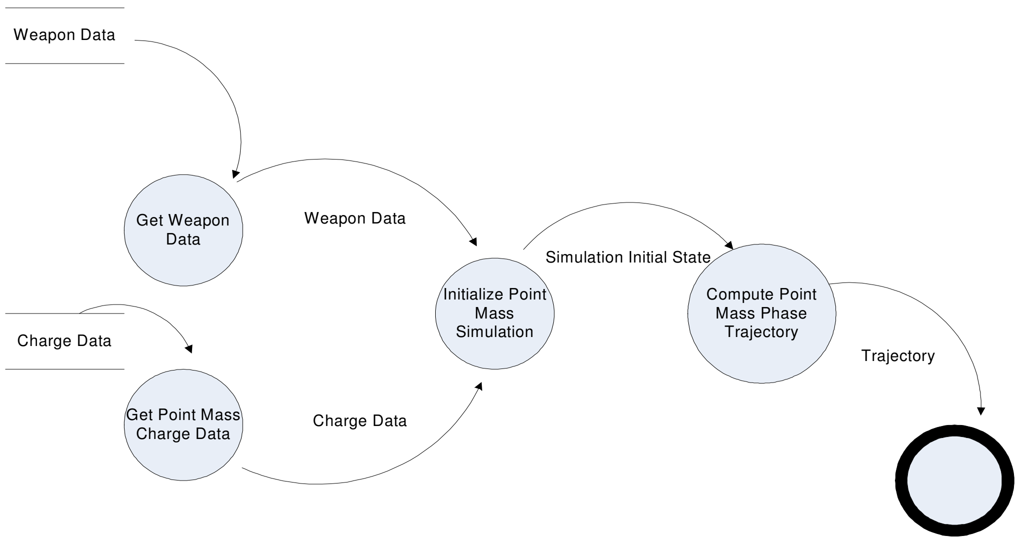


Figure 159 Top Level Data Flow Diagram for Function Oriented Point Mass Trajectory Simulation Abstract Design



Weapon Data
Get Weapon Data
Weapon Data
Charge Data
Get Point Mass Charge Data
Charge Data
Initialize Point Mass Simulation
Simulation Initial State
Compute Point Mass Phase Trajectory
Trajectory

Figure 160 Compute Point Mass Trajectory Data Flow Diagram

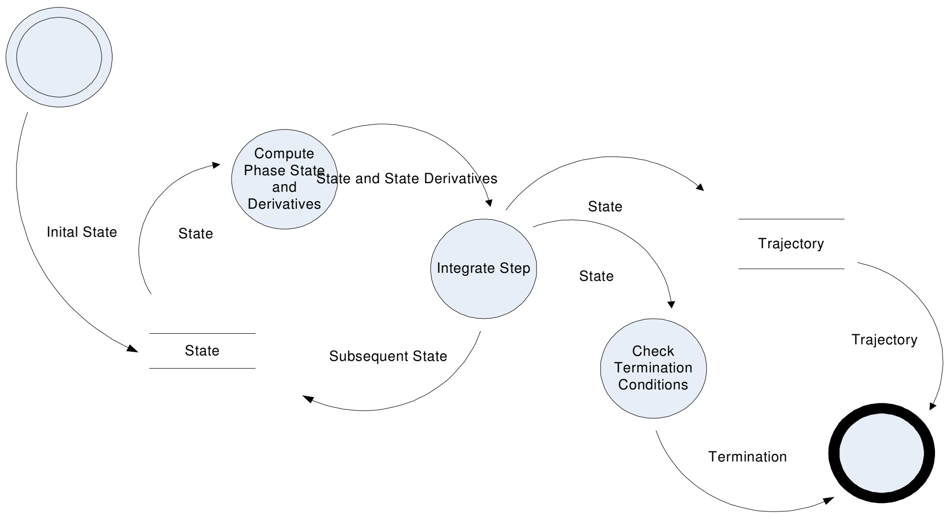


Figure 161 Compute Point Mass Phase Trajectory Data Flow Diagram

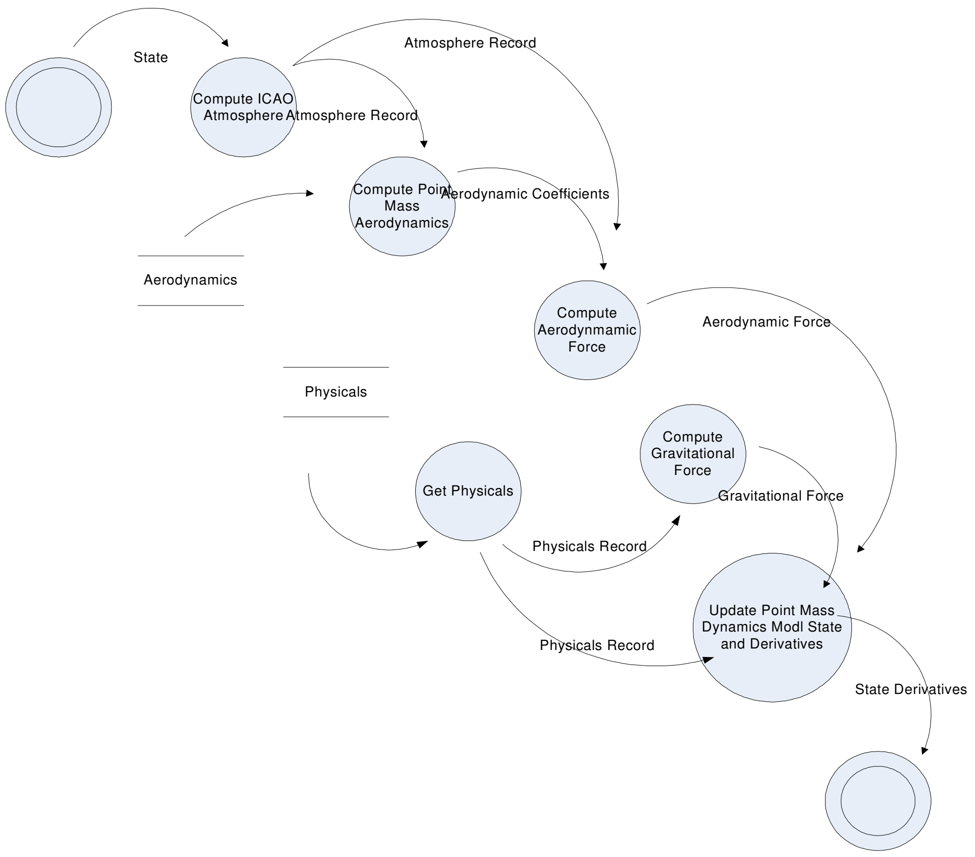


Figure 162 Phase State and Derivatives Data Flow Diagram

APPENDIX M

TIGER AND JAGUAR DATA

Table 11 TIGER and JAGUAR Data

PHYSICALS	
Reference Area	0.00515 m2
Reference Mass	4.7 kg
AERODYNAMICS	
Mach Points	[0 0.7 0.85 0.87 0.9 0.93 0.95 1 1.09]
Cd_0	[0 -0.119 -0.12 -0.122 -0.126 -0.148 -0.182 -0.3 -0.5]
CHARGE DATA	
Muzzle Velocity	220m/s

APPENDIX N

COMPLETE REUSE INFRASTUCTURE

APPENDIX N is appended to this dissertation in an optical media. Content supplied in the optical media is as follows:

01. TSONT

Trajectory Simulation Ontology is given in this folder. There are three sub folders as follows:

01. TSONT\01.1 TSONT Protege OWL

This folder contains a Protégé project for TSONT and TSONT in OWL format.

01. TSONT\01.2 TSONT HTML

This folder contains OWLDoc of TSONT. It is a set of browsable web pages that represent the ontology. One should start browsing from *index.html*.

01. TSONT\01.3 Sample DAVE-ML

This folder contains a set of sample DAVE-ML files.

02. Platform Independent Framework Architecture

This folder contains the platform independent trajectory simulation framework architecture. There are two sub folders as follows:

02. Platform Independent Framework Architecture\02.1 Platform Independent Framework Architecture EA

This folder contains platform independent trajectory simulation framework architecture as Sparx Systems Enterprise Architect 6.0 project.

02. Platform Independent Framework Architecture\02.2 Platform Independent Framework Architecture HTML

This folder contains platform independent trajectory simulation framework architecture as a set of browsable web pages. One should start browsing from *index.htm.*

03. MATSIX Architecture

This folder contains the platform MATSIX architecture. There are two sub folders as follows:

03. MATSIX Architecture\03.1 MATSIX Architecture EA

This folder contains MATSIX architecture as Sparx Systems Enterprise Architect 6.0 project.

03. MATSIX Architecture\03.2 MATSIX Architecture HTML

This folder contains MATSIX architecture as a set of browsable web pages. One should start browsing from *index.htm.*

04. MATSIX Code

This folder contains implementation of MATSIX. We would like to remind you that this framework is implemented by using MATLAB 7.1.

05. LYNX Code

This folder contains the implementation of LYNX simulation. It is one of the trajectory simulations built upon MATSIX using framework completion approach so should be interpreted with MATSIX. Folder also contains a sample tester script *tester.m* that runs LYNX for a sample case and plots a trajectory.

06. PUMA Code

This folder contains the implementation of PUMA simulation. It is one of the trajectory simulations built upon MATSIX using framework completion approach so should be interpreted with MATSIX. Folder also contains a sample tester script *tester.m* that runs PUMA for a sample case and plots a trajectory.

07. C Sharp Example

This folder contains the C# point mass trajectory framework case study. There are two sub folders as follows:

07. C Sharp Example\07.1 C Sharp Example Model

This folder contains the C# point mass trajectory framework architecture. There are two sub folders as follows:

07. C Sharp Example\07.1 C Sharp Example Mode\07.1.1 C Sharp Example Model EA

This folder contains C# point mass trajectory framework architecture as Sparx Systems Enterprise Architect 6.0 project.

07. C Sharp Example\07.1 C Sharp Example Mode\07.1.2 C Sharp Example Model HTML

This folder contains C# point mass trajectory framework architecture as a set of browsable web pages. One should start browsing from *index.htm.*

07. C Sharp Example\07.2 C Sharp Example Code

This folder contains the C# point mass trajectory framework code in a Microsoft Visual Studio 2005 project.

08. PANTHERA Abstract Design

This folder contains the abstract software design for PANTHERA. There are two sub folders as follows:

08. PANTHERA Abstract Design\08.1 PANTHERA Abstract Design Visio

This folder contains the abstract software design for PANTHERA as Microsoft Visio 2003 document.

08. PANTHERA Abstract Design\08.2 PANTHERA Abstract Design HTML

This folder contains the abstract software design for PANTHERA as a set of browsable web pages. One should start browsing from *index.htm*

09. PANTHERA

This folder contains implementation of PANTHERA. We would like to remind you that this framework is implemented by using MATLAB 7.1 Simulink.

10. TIGER

This folder contains implementation of TIGER. We would like to remind you that this framework is implemented by using MATLAB 7.1 Simulink.

11. JAGUAR

This folder contains implementation of JAGUAR. We would like to remind you that this framework is implemented by using MATLAB 7.1 Simulink.

CURRICULUM VITAE

PERSONAL INFORMATION :

Surname :	DURAK
Name :	Umut
Date of birth :	16 February1976
Place of birth :	Ankara - TURKEY
Nationality :	Turkish
Permanent address :	35.CAD. 7/38 100.YIL Ankara ,TURKEY Tel: +90 (312) 285 44 07 Mobile Tel: +90 (532) 622 04 12
Work address :	TÜBİTAK-SAGE Turkish Scientific & Technical Research Council Defense Industries Research & Development Institute P.K. 16, 06261 Mamak – ANKARA / TURKEY Tel: +90 (312) 590 91 76 Fax: +90 (312) 590 91 48-49 E-mail: udurak@sage.tubitak.gov.tr umut@durakailesi.com
Marital status :	Single
Profession :	Mechanical Engineer

EDUCATION :

School	*Years*	*Diploma, Grade and Subject*
Primary School		
Cumhuriyet Primary School Sındırgı, Balıkesir, TÜRKİYE	1982 – 1986	
Atatürk Primary School Sındırgı, Balıkesir, TÜRKİYE	1986 – 1987	Primary School Diploma
Secondary & High School		
Sırrı Yırcalı Anatolian High School Balıkesir,TÜRKİYE	1987 - 1991	Secondary School Diploma
Eskişehir Fatih Science High School Eskişehir, TÜRKİYE	1991 - 1994	High School Diploma
University		
METU Department of Mechanical Engineering İnönü Blv., Ankara, TÜRKİYE	1994 - 1999	B.Sc.
METU Department of Mechanical Engineering İnönü Blv., Ankara, TÜRKİYE	1999 - 2001	M.Sc. , "Tool Management in CIM"
METU Department of Mechanical Engineering İnönü Blv., Ankara, TÜRKİYE	2001 - 2007	Ph.D. , "Ontology Based Reuse Infrastructure for Trajectory Simulation"

<u>LANGUAGE SKILLS</u> :

Native Language	: Turkish
Other Languages	: English

<u>PUBLICATIONS</u>:

1. Durak, U., Ünver, H.Ö., Anlağan, Ö. and Kılıç, S.E, *Conceptual Design of a Gage and Fixture Tracking System Using a Distributed Industrial Framework*, 9th International Machine Design and Production Conference, Ankara, 2000.

2. Durak, U., Ünver, H.Ö., Anlağan, Ö., Kılıç, S.E. , *Atölye Kontrol Sistemleri*, Mühendis ve Makina, Şubat, 2001

3. Ünver, H.Ö., Anlağan, Ö., Kılıç, S.E. , Durak, U., *Savunma Sanayiinde İmalat Yönetim Sistemleri ve Bir Uygulama*, Savunma Sanayii Sempozyumu, Ankara, 2000.

4. Durak,U., Anlağan, O. and Demirors,O., *Agent Based Shop Floor Control System Development and Software Reuse*, 12th DAAAM International Symposium: Intelligent Manufacturing & Automation: Focus on Precission Engineering, Jena, Germany, 2001.

5. Durak, U., Ünver, H.Ö., Anlağan,Ö., Kılıç,S.E., *Tümleşik İmalat Ortamında Takım Yönetimi*, TMMOB Makine Mühendisleri Odası Konya Şubesi Makina Tasarım ve İmalat Teknolojileri Kongresi, Konya, 2001.

6. Durak,U., Anlağan, O., *ISO 9000 Applied to Software Processes in Defense R&D Industry*, Journal of Naval Science and Engineering, (1)1 ,2003

7. Durak, U., Anlağan, Ö., Oğuztüzün, H., *Yeniden Kullanılabilir Uçuş Benzetimi Mimarisi İçin Yol Haritası*, SAVTEK 2004, 2004, Ankara

8. Durak,U., Dayanç, K., Elaldı, F., Anlağan, Ö., *Topçu Roketlerinin Atış Kontrol Sistemleri için Yeni Nesil Balistik Çözücü* USMOS 2005, Ankara, 2005.

9. Durak, U., Oğuztüzün, H. and Mahmutyazıcıoğlu, G., *Domain Analysis for Reusable Trajectory Simulation*, Euro-SIW 2005, Toulouse, France, 2005.

10. Aytar Ortaç, S., Durak, U., Kutluay, Ü.,Küçük, K., Candan, C., *Yeni Nesil Balistik Çözücü'nün Farklı Uygulamalarda Kullanımı*, SAVTEK 2006, Ankara, 2006

11. Durak, U., Oğuztüzün, H. and İder, K., *An Ontology for Trajectory Simulation*, WinterSim06, Monterey, CA, 2006.

12. Durak, U., Oğuztüzün, H. and İder, K., *Ontology Based Trajectory Simulation Framework*, Journal of Computing and Information Science in Engineering, -ACCEPTED-

13. Aytar Ortaç, S., Durak, U., Kutluay, Ü.,Küçük, K. and Candan, C., *NABK Based Next Generation Ballistic Table Toolkit*, 23rd International Symposium on Ballistics, Tarragona, Spain, 2007.

14. Durak, U., Güler, S., Oğuztüzün, H., and İder, K., *An Exercise In Ontology Driven Trajectory Simulation with MATLAB Simulink*, 21st EUROPEAN Conference on Modelling and Simulation, Prague, Czech Republic, 2007.

15. Özdikiş, Ö, Oğuztüzün, H, and Durak, U., *OWL to UML : Transforming Domain Models to Framework Architectures*, Manuscript.

WORK EXPERIENCE :

Organization	Years	***Job***
TUSAŞ ENGINE INDUSTRIES Eskişehir, TÜRKİYE	1999 – 2001	Research Engineer

Organization	Years	Job
TÜBİTAK-SAGE	2001-2003	Research Engineer
	2003-2007	Senior Research Engineer
Ankara, TÜRKİYE	2007-	Head Research Engineer