A.C. DİNÇ

METU 2010

HLA FOM DEVELOPMENT WITH MODEL TRANSFORMATIONS

ALİ CEM DİNÇ

MAY 2010

HLA FOM DEVELOPMENT WITH MODEL TRANSFORMATIONS

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

BY

ALİ CEM DİNÇ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

MAY 2010

Approval of thesis:

HLA FOM DEVELOPMENT WITH MODEL TRANSFORMATIONS

submitted by **ALİ CEM DİNÇ** in partial fulfillment of requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan ÖZGEN
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** ________________

Assoc. Prof. Halit Oğuztüzün
Supervisor, **Computer Engineering** ________________

Examining Committee Members

Assoc. Prof. Nihan Kesim Çiçekli ________________________
Computer Engineering Dept., METU

Assoc. Prof. Halit Oğuztüzün ________________________
Computer Engineering Dept., METU

Assoc. Prof. Ali H. Doğru ________________________
Computer Engineering Dept., METU

Asst. Prof. Aysu Betin Can ________________________
Informatics Institute, METU

Asst. Prof. Pınar Şenkul ________________________
Computer Engineering Dept., METU

Date : 04 - May - 2010

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Ali Cem Dinç

Signature :

ABSTRACT

HLA FOM DEVELOPMENT
WITH MODEL TRANSFORMATIONS

Dinç, Ali Cem
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Halit Oğuztüzün

May 2010, 90 pages

There has been a recent interest in the model-based development approach in the modeling and simulation community. The Model-Driven Architecture (MDA) of OMG envisions a fully model-based development process where models are created for capturing not only requirements, but also designs and implementations. Domain-specific metamodels and model transformations constitute the cornerstones of this approach. We have developed transformations from the data part of Field Artillery (FA) domain models to High Level Architecture (HLA) Object Model Template (OMT) models, honoring the MDA philosophy. In the MDA terminology, the former corresponds to the CIM (Computation-Independent Model) or, arguably, PIM (Platform-Independent Model), and the latter corresponds to the PSM (Platform-Specific Model), where the platform is HLA. As a case study for the source metamodel, we have developed a metamodel for the data model part of the (observed) fire techniques of the FA domain. All of the entities in the metamodel are derived from the NATO's Command and Control Information Exchange Data Model (C2IEDM) elements.

Keywords: Metamodeling, Domain Specific Modeling, Model Transformations, Field Artillery

ÖZ

MODEL DÖNÜŞÜMLERİ İLE
HLA FOM GELİŞTİRME

Dinç, Ali Cem

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Assoc. Prof. Dr. Halit Oğuztüzün

Mayıs 2010, 90 sayfa

Son yıllarda modelleme ve simülasyon camiasında model tabanlı geliştirim yaklaşımına dair bir ilgi ortaya çıkmıştır. OMG'nin Model Güdümlü Mimarisi (MDA - Model Driven Architecture), modellerin sadece isterlerin değil, bunların yanı sıra tasarım ve gerçekleştirimin de yakalandığı, tamamen model tabanlı bir geliştirme sürecini hedeflemektedir. Alana özel metamodeller ve model dönüşümleri bu yaklaşımın köşe taşlarını teşkil etmektedir. Sahra Topçuluğu (ST) alan modellerinin veri modeli kısımlarından High Level Architecture (HLA) Object Model Template (OMT) modellerine MDA felsefesine uygun dönüşümler geliştirdik. MDA terimler dizgesinde ST Modeli Platformdan Bağımsız Model (PBM)'ye, federasyon mimarisi modeli ise, HLA'nın platform olduğu, Platforma Özel Model (PÖM)'e karşılık gelmektedir. Kaynak metamodel örneği olmak üzere, ST alanının gözetlemeli atış tekniklerinin veri modeli kısmına dair bir metamodel geliştirmiş bulunuyoruz. Metamodeldeki bütün varlıklar NATO'nun Command and Control Information Exchange Data Model (C2IEDM) elemanlarından türetilmiştir.

Anahtar Kelimeler: Metamodelleme, Alan Spesifik Modelleme, Model Dönüşümleri, Sahra Topçuluğu

To My Family

ACKNOWLEDGEMENTS

I would like to thank Assoc. Prof. Halit Oğuztüzün for his direction, assistance, and guidance. In particular, Assoc. Prof. Halit Oğuztüzün's recommendations and suggestions have been invaluable for the project and for software improvement.

I also wish to thank Mr. Gürkan Özhan, who has all taught me techniques of programming and technical writing. He has made available his support in a number of ways with timely assistance although he has lots to do with his doctoral dissertation.

Finally, words alone cannot express the thanks I owe to Meryem Dinç, my wife, for her encouragement and assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
C2IEDM	Command and Control Information Exchange Data Model
CFF	Call For Fire
CIM	Computation Independent Model
CMMS	Conceptual Models of the Mission Space
DCMF	Defense Conceptual Modeling Framework
DoD	The United States Department of Defense
FA	Field Artillery
FADM	Field Artillery Conceptual Data Model
FAMM	Federation Architecture Meta Model
FAT	Field Artillery Team
FCC	Fire Control Center
FDC	Fire Direction Center
FDM	Federation Design Model
FFE	Fire for Effect
FOM	Federation Object Model
FOP	Forward Observing Post
GME	Generic Modeling Environment

GReAT	Graph Rewrite and Transformations
GUI	Graphical User Interface
HLA	High Level Architecture
IEEE	Institute of Electrical and Electronic Engineers
JC3IEDM	Joint Command, Control and Consultation Information Exchange Data Model
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MIC	Model Integrated Computing
MOM	Management Object Model
MTO	Message To Observer
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Model Template
PIM	Platform Independent Model
PSM	Platform Specific Model
RTI	Run Time Infrastructure
SOM	Simulation Object Model
UML	Unified Modeling Language

CHAPTER 1

INTRODUCTION

This chapter provides the motivation and background for this study, represents the scope and assumptions, discusses the related work and yields about model driven technologies and approaches.

1.1. Motivation

The Model-Driven Engineering (MDE) approach is becoming prominent in software and systems engineering, bringing forth a model-centric approach to the development cycle in contrast with today's mostly code-centric practices [1]. A well-known MDE initiative is the Model Driven Architecture (MDA) of Object Management Group (OMG). Model transformations are considered the heart of MDA, where the Platform Independent Model (PIM) of a system to be constructed, is transformed, or refined, into a Platform Specific Model (PSM) [2]. Both PIM and PSM conform to their own metamodels, which act as languages that define these models.

Model Integrated Computing (MIC), an earlier manifestation of MDE, relies on metamodeling to define domain-specific modeling languages and model integrity constraints. The language is then used to automatically compose a domain-specific model-building environment for creating, analyzing, and evolving the system through modeling and generation.

During the last decade, MIC has gained acceptance through several fielded systems [3], and it is recognized in both academia and industry today. In the MIC approach, a crucial point is the generation, where (domain-specific) models are transformed into lower level executable and/or analysis models. Model transformation techniques and tools are essential to MIC in realizing the generation process.

In this thesis, the transformation process of Field Artillery Conceptual Data Model (FADM) [4] into the Federation Design Model (FDM) [5] targeted to the HLA-OMT Model [6]. An OMT model can further be used to generate Java/AspectJ code for execution on an HLA Run-time Infrastructure (RTI)0. Figure 1 depicts a high-level picture of the overall work. Our main focus is in phase 1 where the model transformer converts an FADM which conforms to the Field Artillery (FA) MetaModel into an HLA - OMT federation architecture model, which conforms to the Federation Architecture MetaModel (FAMM).

In this sense this work can be considered as an application of the MIC approach. Both of the source and target models are developed in Generic Modeling Environment (GME) tool. The model transformations are carried out with Graph Rewrite and Transformations (GReAT), and partly hand-coded in C++. Both tools are provided by Institute of Software Integrated Systems at Vanderbilt University.

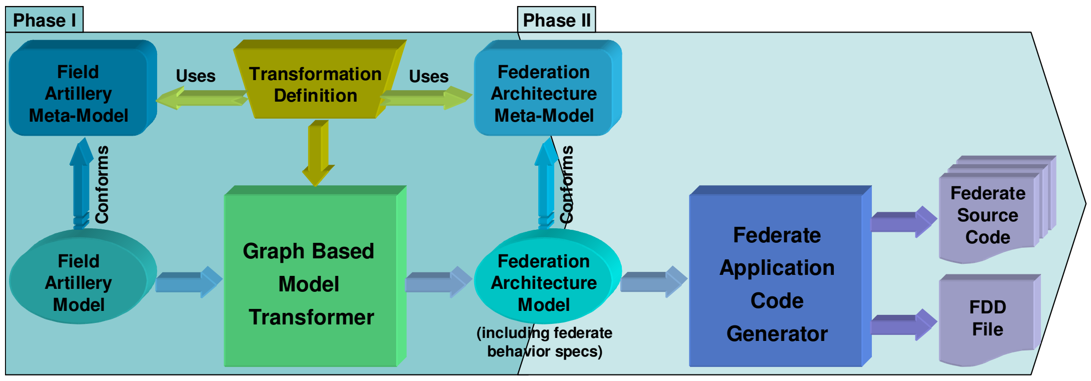

Phase I
Phase II
Field Artillery Meta-Model
Uses
Transformation Definition
Uses
Federation Architecture Meta-Model
Conforms
Conforms
Field Artillery Model
Graph Based Model Transformer
Federation Architecture Model
(including federate behavior specs)
Federate Application Code Generator
Federate Source Code
FDD File

Figure 1. The model transformation of FADM to FAMM to executable code

1.2. Scope and Assumptions

Both the source and target domains are restricted to data models only; that is, any behavior modeling is out of the scope of this work. The source and target domains are modeled in GME [8] and the model transformation is developed in GReAT [9][10]. The FADM includes such elements as domain entities, messages and entity interactions through messaging. This setting only addressed static aspects; that is, there was no notion of dynamism, like missions or tasks. The same situation was valid for the HLA-OMT metamodel side. Although it enabled to define classes, interactions, attributes and parameters resembling the practices of object-oriented programming, it lacked dynamic behavior such as federation executions.

1.3. Organization of the Thesis

The rest of the thesis is organized as follows. Section 2 lays the background; briefly introduce the field artillery domain, HLA, GME and GReAT tools used in modeling and model transformations, respectively. Section 3 is for the source model that sets up the Field Artillery Data Model. The target model, HLA-OMT Model, will be described in the Section 4. Section 5 is devoted to the transformation of FADM to HLA-OMT Model. Finally, Section 6 concludes the thesis.

CHAPTER 2

BACKGROUND

This chapter introduces the key concepts, technologies and tools used, including the modeling tool and the model transformation tool.

2.1. Related work

The Model-Driven Architecture of OMG envisions a development paradigm where designers create a Platform-Independent Model (PIM) of the design, which is then refined into a Platform-Specific Model (PSM). Exposing a better solution to the transformation of a Platform Independent Model (PIM) to a Platform Specific Model (PSM) is still the main problem in the Model Driven Engineering. This problem will certainly continue to occupy the greatest attention in the MDA community as it is the main driving force of the motivation for the MDA vision. On the other hand, we are considering changing the domain of the model transformation problem from HLA-based distributed simulation. The reason for this is that the HLA-based distributed simulation is just far too huge to be a modeling concept. It involves cross-cutting concepts, technologies and issues. One can simulate almost anything with HLA and it is virtually possible to ever devise a generic conceptual model and a generic federation design model for HLA itself. In that respect HLA can be considered as a very abstract and intangible “entity” to develop a PIM and PSM for.

2.1.1. Model Driven Development\Architecture

MDA introduces the concept of the Platform Independent Model (PIM) and the Platform Specific Model (PSM). A PIM is an abstract model of the software design that omits any platform (i.e. implementation-specific details). A PSM, on the other hand, is another model that includes implementation-specific details. The PSM is obviously dependent on the PIM, and arguably one (the PSM) can be derived automatically from the other one (the PIM). However, this derivation process is highly domain-specific: different domains might need different methods for implementing the derivation. This thesis uses a technique and a prototype tool developed at the Institute for Software Integrated Systems of Vanderbilt University for creating highly configurable model transformation tools that can be applied in the MDA context. The technique (and the tool) is based on a well-established theoretical framework based on graph transformations.

The Conceptual Models of the Mission Space (CMMS) effort, initiated by the U.S. Department of Defense (DoD), aims to facilitate the development and reuse of simulation models. CMMS is defined in [11] as first abstractions of the real world that serve as a frame of reference for simulation development by capturing the basic information about important entities involved in any mission and their key actions and interactions. CMMS emphasizes the implementation-independent functional descriptions of the real world processes, entities, environmental factors, and associated relationships and interactions constituting a particular set of missions, operations or tasks. An important part of CMMS includes the domain specific conceptual models, called “Mission Space Models”. They are consistent, structured and functional descriptions of real military operations or processes. Some recent studies, notably Defense Conceptual Modeling Framework (DCMF) [12] and the conceptual modeling tool KAMA 0 have further elaborated the vision promoted by the CMMS.

Joint Command, Control and Consultation Information Exchange Data Model (JC3IEDM) are the core of NATO Reference Model and is also a view model of NATO STANAG 5525 [14]. The data model is focused primarily on the information requirements that support the operations planning and execution activities of a military or civilian headquarters or a command post. JC3IEDM has recently evolved from C2IEDM, or Command and Control Information Exchange Data Model [15] by additionally including and modeling new joint operational concepts.

2.1.2. Model Transformation

Model transformation is the process of converting one or more models – called source models – to one output model – the target model – of the same system [16]. This process takes a model conforming to a given source meta-model as input and produces another model conforming to a target meta-model as output. The transformation process, composed of a set of rules, should itself be considered as a model. As a consequence, it is based on a corresponding meta-model, that is an abstract definition of the used transformation language.

The deprivation for techniques that can be usable for model transformations has been recently anticipated in the UML realms. To illustrate, see [17], [18] and [19]. Model transformation is an essential tool for many applications, including translating abstract design models into concrete implementation models [19] or for specification techniques [18]. The innovations in UML (such as [20]) accentuate the use of meta-models, and yield solid foundation for the accurate specification of domain semantics.

Many languages and tools have been developed to specify and execute transformation process. In 2002 OMG issued the Query/View/Transformation request for proposal [21] to define a standard transformation language. Although the specification was finalized at the end of 2005, there are lots of researches going on about model transformations. Over the last decade, in parallel to the OMG process, a number of model transformation approaches have been introduced both from academia and industry. The paradigms, constructs, modeling approaches, used tools distinguish these model transformation techniques. Each of them has its own suitability for a certain set of problems. For detailed classification of a today’s model transformation approaches please refer to [22], [23] and [24].

In this thesis, GReAT would be used to transform FADM into the FDM targeted to the HLA-OMT Model so as to take the advantages of graph-based model transformation.

2.1.3. Graph-based Model Transformation

Graph transformation, or graph rewriting is known as the technique of creating a new graph out of an original graph using some automatic rules. Graph grammars and graph rewriting [25][26] have been developed during the last 25+ years as techniques for formal modeling and tools for very high-level programming. It has large number of applications, ranging from verification of software to algorithms.

Graph transformations can be used as computational isolation. Basically, the state of a computation can be represented as a graph and the transformation rules on that graph represent the further steps in that. These rules formed by an original graph, which is going to be matched to a subgraph in the complete state, and a replacing graph, which would replace this matched subgraph.

Beyond the ground-laying work in the theory of graph grammars and rewriting, the approach has found several applications as well. Graph rewriting has been used in formalizing the semantics of statecharts [27], as well as various concurrency models [26]. Several tools —including programming environments— have been developed [28][29] that illustrate the practical applicability of the graph rewriting approach. These environments have demonstrated that complex transformations can be expressed in the form of rewriting rules, and graph rewriting rules can be compiled into efficient code. Programming via graph transformations has been applied in some domains [25] with reasonable success.

2.2. Elements of Field Artillery

This section provides basic understanding of artillery firing systems, which will be the case study domain of the thesis. Specifically, the case study concepts are adopted from the work done in [30] where an HLA based simulation is developed to train the officers in an artillery battalion.

Field artillery is a unit composed of artillery weapons. The general mission of field artillery is to destroy, neutralize or suppress the enemy by cannon, rocket, and missile fire and to help integrate all fire support assets into combined arms operations [31]. The field artillery system provides close support to maneuver forces, counter fire and interdiction as required. These fires neutralize, canalize, or destroy enemy attack formations or defenses; obscure the enemy's vision or otherwise inhibit his ability to acquire and attack friendly targets; and destroy targets deep in the enemy rear with long-range rocket or missile fires. Field artillery support can range from conventional fires in a company zone to massive nuclear and chemical fires across a corps front [31].

An artillery battery, being one of the 3 to 6 parts of an artillery battalion has 3 main units in its organizational structure. These are;

- Forward Observing Post (FOP),
- Fire Control Center (FCC), and
- Weapons (6-8 weapons for every battery),

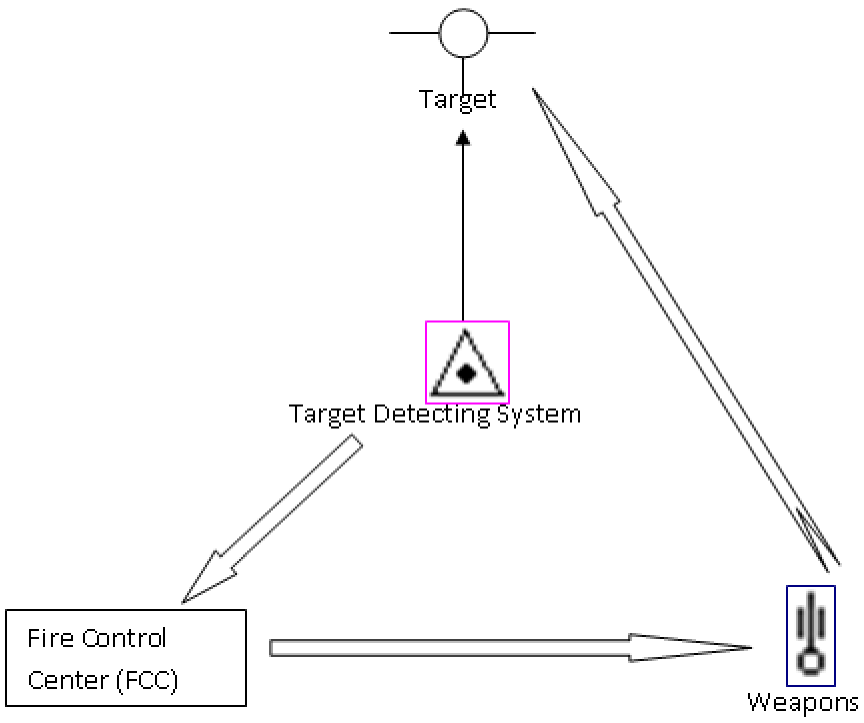


Figure 2. A Typical Artillery Battery System

Figure 2 sketches the flow information and interactions between the units in the battery organization. The rest of this section briefly defines these units and summarizes the roles that they play in a battle environment.

2.2.1. Forward Observing Post (FOP)

There are three FOP teams in an artillery battery organization. FOPs are the eyes of the battery and their basic duty is to detect the targets and make the adjustments after the shots. They usually consist of several soldiers with distance measurement equipment to detect the location of the targets, and various communication equipments to inform the fire control center of the data they acquire.

2.2.1.1. Target Detection

In tactical military operations, military units usually use 1/25000-scaled maps. In these maps, 1cm is equal to 250 m and 4 cm is equal to 1000 m. The grid lines in these maps are located in every 1000 m, in both horizontal and vertical directions. Each 1000x1000 square in the map is called grid square. Figure 3 depicts a simplified example of a military map. The FOP detects targets by usually employing two techniques, namely, grid coordinate technique, and polar coordinate technique:

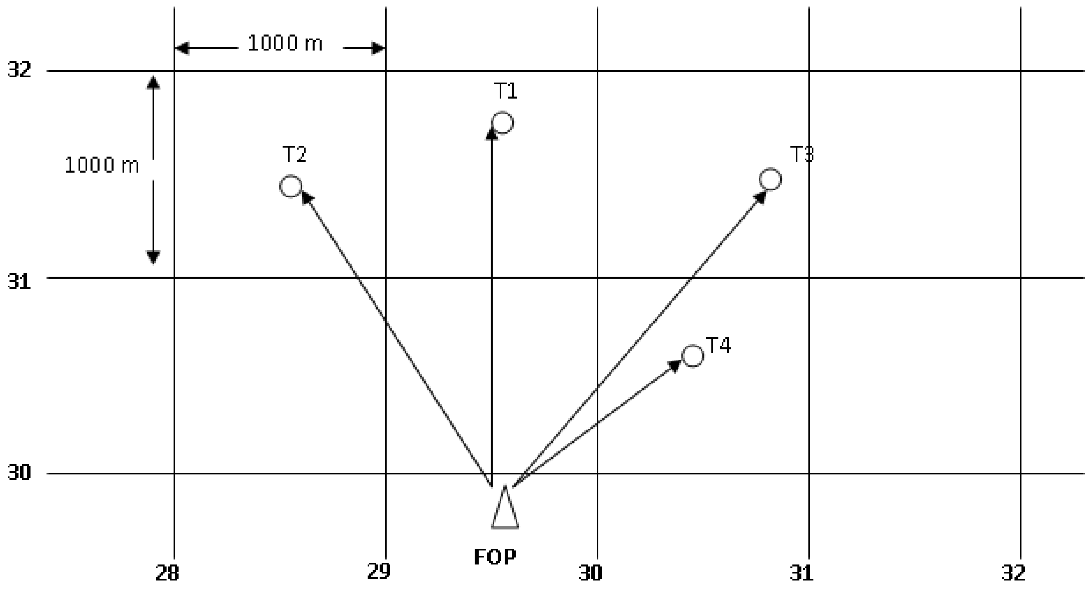


Figure 3. Simplified Military Map Showing Target Detection Techniques

a) Grid Coordinate Method: FOP determines the grid coordinate of a point by map searching. The approximate coordinates of the targets in Figure 3 are determined below. The second column indicates the west-to-east, and the third column indicates the south-to-north coordinates.

Table 1. Coordinates in Grid Coordinate Method

Target	W-to-E	S-to-N
T1	29550	31780
T2	28600	31450
T3	30880	31500
T4	30380	30600

b) Polar Coordinate Method: FOP informs the FCC with the distance and direction angle of the target. Distance is measured by laser-meter in meters, and direction angle is measured by compass in mils. The angles widen in clockwise direction, starting from 0 in the north, and ending at 6400, again in the north. The approximate polar coordinates of the targets in Figure 3 are determined below:

Table 2. Coordinates in Polar Coordinate Method

Target	Distance	Dir. Angle
T1	1850	6400
T2	1700	5740
T3	1950	750
T4	975	930

2.2.1.2. Adjustment

FOP detects the vertical and horizontal deflections of the hit points of the shots from the target. As in the case of target detection, he uses laser-meter and compass to measure the deflections. Later he communicates the results to the FCC.

2.2.2. Fire Control Center (FCC)

There is only one FCC in an artillery battery. The FCC inputs various environmental data such as wind direction and magnitude, the information coming from the FOP, and its own knowledge of the location of the target, the FOP, and the weapons, and computes the necessary parameters for the weapons to make their shoots to the target. Although there are many automated fire control tools, it is unnecessary risk not to calculate it manually. Manual fire control is simply made as follows:

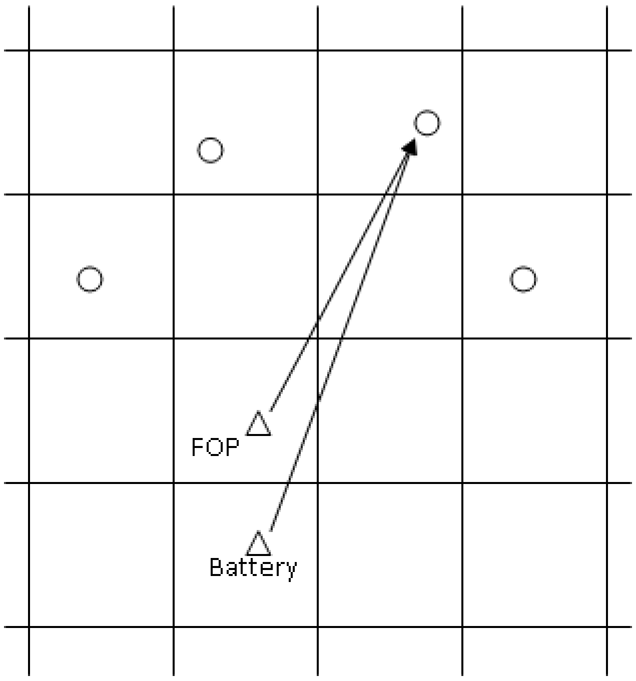


Figure 4. A Horizontal Plan Paper

First of all, the FCC gets the location of the target from the FOP and marks it on a sheet called the horizontal plan paper, which has vertical and horizontal grid lines on it (see Figure 4). Horizontal plan paper represents a 1/25000-scaled map of the battle area. Location of the FOP and the battery are already marked on it.

Then it measures the distance from battery to the target by using a special ruler. Remember that the distance that the FOP detected was from FOP to target. By using the distance and the elevation interval between target and the battery, FCC calculates the powder charge and the vertical angle of the weapons. Powder charge has a direct relationship with the initial velocity, which constitutes one of the two major outputs of the mathematical calculations that the FCC made (the other is the vertical angle). Finally, since the battery and the target are most probably not vertically aligned, the FCC measures the direction angle of the weapons to the target.

Thus, the three shoot parameters that the FCC computes and provides to a weapon are, powder charge, direction angle, and vertical angle. The following computations are done similarly, after receiving the adjustment values regarding the previous shot from the FOP, until the target is destroyed.

2.2.2.1. Field Artillery Observed Fire

The general mission of Field Artillery (FA) is to destroy, neutralize or suppress the enemy by cannon, rocket, and missile fires and to help integrate all fire support assets into combined arms operations. FA weapons are usually located in defiladed areas in order to protect them from enemy detection. This nature of FA gunnery makes it an indirect fire problem. Observed fire, the technique that solves the indirect FA gunnery problem, is carried out by the coordinated efforts of the Forward Observers (FwdObserver), the Fire Direction Center (FDC), and firing sections of the firing unit, all together forming the Field Artillery Team (FAT). Authoritative reference [31] provides a comprehensive explanation on tactics, techniques and procedures for FA fire direction process.

Basic duty of the FO, considered the eyes of the FAT, is to detect and locate suitable indirect fire targets within his zone of observation. (In fact, the FO functionality is realized as several soldiers equipped with sophisticated binoculars, laser range finders and maps among others for accurate target location, and communication equipment to convey observation information to the FDC. We do not attempt to capture such information in our model. Rather, we aim at a functional description of the domain.) In order to start an attack on a target, the FO issues a Call For Fire (CFF) request to the FDC. It contains all information needed by the FDC to determine the method of attack.

As it is unlikely to achieve a target hit in the first round of fire (due to such error factors as improper target location, nonstandard ammunition, distortions in the barrel and meteorological effects), the common practice is firstly to conduct adjustment on the target. Usually the central cannon are selected as the adjusting weapon. The FO provides correction information to the FDC after each shot based on his spotting of the detonation. The correction information includes, but is not limited to deviation, range, height of bust, observer-target direction and distribution corrections. Moreover, the FO may request changes on any of the fire parameters such as method of fire, method of control, ammunition or trajectory. Changes on the target description are also included in the correction information. Once a target hit is achieved, the FO initiates the Fire for Effect (FFE) phase by noting this in his correction. FFE is carried out by cannons firing all together with the same fire parameters as the last adjustment shot. After the designated number of rounds is fired, the FO sends a final correction including surveillance information. Based on the surveillance information, if the desired effect on the target is achieved, mission ends. Otherwise, the FO may request repetitions, or restarts the adjustment phase if deemed necessary.

2.2.2.2. Adjustment Followed By Fire for Effect Mission

The Adjustment followed by Fire for Effect (AdjFFE), which is by far the most common and widely known among the FA observed fire missions. The mission starts by FwdObserver initiating a Call for Fire (CFF) request to the Fire Direction Center (FDC), consisting of three sub-messages, among which the mission type (i.e., AdjFFE) is also specified. Once the FDC receives the CFF, it determines how the target will be attacked. The FwdObserver observes bursts. After a spotting has been made, the observer must send corrections to the FDC to move the bursts onto the adjusting point. To conclude the correction cycle, the observer sends a last surveillance message. At the end of FFE, the FwdObserver announces the effect on the target. According to the result, the mission is completed, aborted, or continued.

2.2.2.3. Tactical and Technical Fire Direction

Fire direction is the employment of firepower. The objectives of fire direction are to provide continuous, accurate, and responsive fire support under all conditions [31]. The FDC, considered the brain of the FAT, receives the CFF from the observer, determines firing data, and converts them to fire commands to be executed by the firing sections. The FDC abstraction can be separated into two distinguished functionalities, namely, tactical and technical fire direction.

The primary concern of tactical fire direction is to determine how the target will be attacked. This is specified as a fire order in which information concerning the units to fire, and the type and amount of ammunition to be fired are included. Technical fire direction is conducted by issuing fire commands where the necessary information for orienting, loading and firing a howitzer is included.

Battalion directed and autonomous modes are the two alternatives under which fire direction can be conducted. In the present study we focus on battalion directed organization for several reasons: First, in this setting there is a clear assignment of tactical and technical fire direction functionalities to the battalion and battery FDCs, respectively. This separation of roles to two different actors leads to more straightforward domain modeling. Interested reader may refer to [31] for details.

In battalion-directed setting, the information flow between the FO and the rest of the FAT is carried through the battalion FDC as illustrated in Figure 5. In this respect the battalion FDC is the sole contact point of the FO. The battalion FDC receives the CFF from the FO, prepares the fire order and Message To Observer (MTO) and send them to the battery FDC and the FO, respectively. After the FO receives the MTO, it waits for a detonation to spot. The MTO conveys the necessary information for the FO to assist in pursuing and synchronizing with projectile detonations. Note that the battalion FDC is a higher post than the FO and may override his decisions made in the CFF with the MTO.

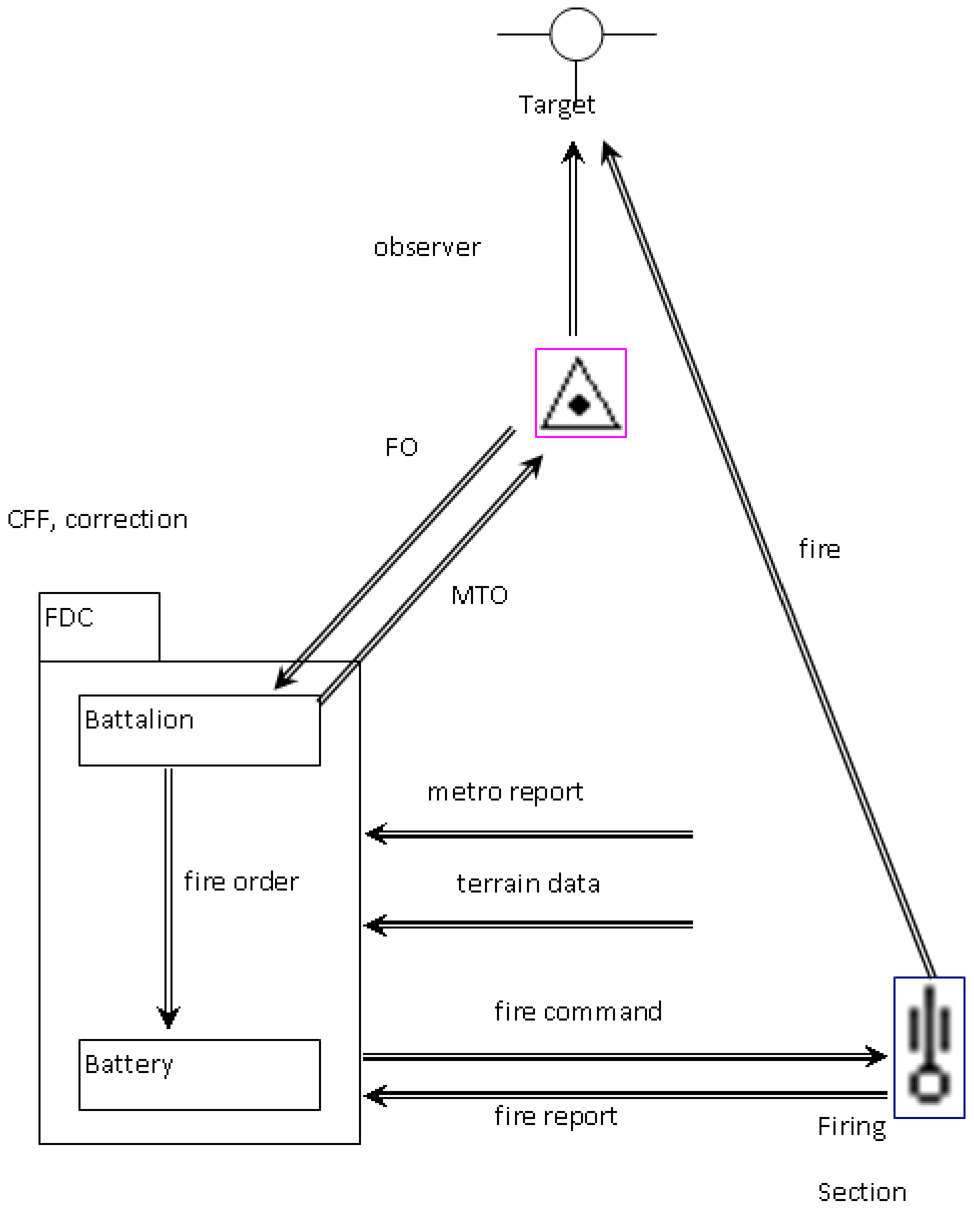


Figure 5. Typical Field Artillery Team Setting

When the fire order is received by the battery FDC, it starts to prepare the fire command to be sent to the firing sections. The fire command contains detailed technical data enabling the firing section personnel to load and orient howitzers to convey fire to the target. In order to compute the fire parameters in the fire command, the battery takes into consideration the situation map where all the information pertaining to the terrain as well as the FO and target locations are present, the metro report issued by the meteorology station, and current ammunition reserve. (Computation of the firing parameters, however, is out of our modeling scope.)

If the fire command tells so, the firing section sends a volley to the target according to the parameters in the fire command. The section chief reports to the battery FDC about the action that the firing section has just performed. A firing report generally covers synchronization data, ammunition or task status.

In most cases, certain parts of the fire order and fire command does not show any difference from one mission to the next, and based on the tactical situation, personnel and weapon status, type and amount of ammunition available, and the commander's intuition, some parts of the fire order and fire command can be announced as standard. If some element of the fire order or fire command is not provided, then the standard (default) value for that element is accounted as valid. Standards stay in effect until cancelled, changed with another standard or overridden inside a fire order or fire command.

2.2.3. Weapons

Weapons (more specifically, cannons) apply the orders coming from the FCC. One of the cannons, typically the one located in the middle of the battery, is called as the base cannon. The base cannon continue to apply the orders at each loop, until the target is hit. Once the target is hit, all the cannons in the battery make their shots to the target with the same shoot parameters as the base cannon. This is called a group shot.

Of course there are many issues that must be taken into account in a real world scenario regarding the cannons themselves, such as barrel deformations due to heating, variation of the minimum time between two shots, internal ballistics and external ballistics of the bullets, and so on. For the sake simplicity, these factors will be ignored in our case study. However, once the proof of concept is achieved, they may definitely be included.

2.3. High Level Architecture

The High Level Architecture (HLA) is general purpose architecture for distributed computer simulation systems. The HLA is not software, it provides a common framework. Using HLA, computer simulations can communicate to other computer simulations regardless of the computing platforms. Communication between simulations is managed by a Run-Time Infrastructure (simulation) (RTI).

The HLA was approved as US Department of Defense (DoD)'s technical architecture for modeling and simulation in September 1996. In HLA terminology [32] a federate is one simulation (e.g. could represent one platform, like a ship, a cockpit simulator) while a federation is a named set of interacting federates with the support of RTI to form an integrated simulation (e.g. could represent an aggregate, like a naval task group simulation). A federate is a member of a federation. Federation Execution means a session of a federation executing together.

The High Level Architecture (HLA) consists of the following components:

- Interface Specification. The interface specification document defines how HLA compliant simulators interact with the Run-Time Infrastructure (RTI). The RTI provides a programming library and an application programming interface (API) compliant to the interface specification.
- Object Model Template (OMT). The OMT specifies what information is communicated between simulations and how it is documented.
- HLA Rules. Rules that simulations must obey to be compliant to the standard.

2.3.1. Interface specification

The HLA Interface Specification defines the interface between the simulation and the software that will provide the network and simulation management services. RTI is the software that provides these services [33][34]. The interface specification is object oriented. Many RTIs provide APIs in C++ and the Java programming languages.

The interface specification is divided into service groups:

- Federation Management: Creating, modifying, deleting and dynamic control of a federation execution are provided by federation management services.

- Declaration Management: Joined federates use the services in this group to declare their interest to an object class attribute or an interaction class.
- Object Management: The services in this group deal with the registration, modification, and deletion of object instances and the sending and receiving of interactions.
- Ownership Management: The services in this group are used to transfer ownership of instance attributes among joined federates.
- Time Management: Messages sent by different joined federates are delivered in a consistent order throughout the federation execution by the time management services and associated mechanisms.
- Data Distribution Management: The services in this group provide information on data relevance at different levels and allow refining the data requirements.
- Support Services: This group includes miscellaneous services for performing such actions as setting advisory switches, manipulating regions, or RTI start-up and shutdown.

2.3.2. HLA rules

The HLA rules describe the responsibilities of federations and the federates that join. These are a set of rules, which must be followed to achieve proper interaction of federates in a federation. These following rules describe the responsibilities of simulations and RTI in HLA federations [33].

- Federations shall have an HLA Federation Object Model (FOM), documented in accordance with the HLA Object Model Template (OMT).
- In a federation, all representation of objects in the FOM shall be in the federates, not in the run-time infrastructure (RTI).
- During a federation execution, all exchange of FOM data among federates shall occur via the RTI.
- During a federation execution, federates shall interact with the run-time infrastructure (RTI) in accordance with the HLA interface specification.

- During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.
- Federates shall have an HLA Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT).
- Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.
- Federates shall be able to transfer and/or accept ownership of an attribute dynamically during a federation execution, as specified in their SOM.
- Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM.
- Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

2.3.3. OMT (Object Model Template)

The OMT prescribes a common method for recording the information that will be produced and communicated by each simulation participating in the distributed exercise [33]. An HLA object model is a composition of a set of interrelated components encapsulating information on classes of objects and their attributes and interactions and their parameters. HLA object models identify the data exchanged at runtime to achieve federation objectives. The OMT Specification defines the format and syntax of HLA object models. Furthermore it provides a common framework for the communication between HLA simulations.

All objects and interactions managed by a federate, and visible outside the federate, are described according to the standard OMT. This common template facilitates understanding and comparisons of different federates and federations, and provides a contract between members of a federation on the types of objects and interactions that will be supported across multiple interoperating simulations [33]. The primary objective of the HLA OMT is to facilitate interoperability among simulations and reuse of simulation components.

HLA object models are documented using OMT components, which represent information about classes of objects, their attributes and their interactions in tabular form. The template for the core of an HLA object model shall consist of the following components:

- Object model identification table: To associate important identifying information with the HLA object model.
- Object class structure table: To record the namespace of all federate or federation object classes and to describe their class-subclass relationships.
- Interaction class structure table: To record the namespace of all federate or federation interaction classes and to describe their class–subclass relationships.
- Attribute table: To specify features of object attributes in a federate or federation.
- Parameter table: To specify features of interaction parameters in a federate or federation.
- Dimension table: To specify dimensions for filtering instance attributes and interactions.
- Time representation table: To specify the representation of time values.
- User-supplied tag table: To specify the representation of tags used in HLA services.
- Synchronization table: To specify representation and data types used in HLA synchronization services.
- Transportation type table: To describe the transportation mechanisms used.
- Switches table: To specify initial settings for parameters used by the RTI.
- Data type tables: To specify details of data representation in the object model.
- Notes table: To expand explanations of any OMT table item.
- Routing space table: To specify routing spaces for object attributes and interactions in a federation.
- FOM/SOM lexicon: To define the terms used in the tables.

OMT specifies three types of object models:

- Federation Object Model (FOM): The FOM describes the all shared object, attributes and interactions and associations for the whole federation among federates essential to a particular federation.
- Simulation Object Model (SOM): A SOM describes the shared object, attributes and interactions used for a single federate.
- Management Object Model (MOM): MOM identifies classes and interactions related to federation management.

All of the OMT components shall be completed when specifying an HLA object model for both federations and individual federates. However, certain tables may be empty or devoid of domain-specific content. For instance, although federations typically support interactions among their federate's, some federates (such as a stealth viewer) might not be involved in interactions. In this situation, the interaction class structure table would contain only the single interaction class required by the HLA and the parameter table would be empty in that federate's SOM. It is also expected that federates commonly have objects with attributes of interest across the federation; in such cases, these objects and attributes shall be documented. However, a federate or an entire federation may exchange information solely via interactions; in which case, its object class structure table and attribute table would contain only HLA-required data.

The final HLA OMT component, the FOM/SOM lexicon, is essential to ensure that the semantics of the terms used in an HLA object model are understood and documented. The HLA MOM specifies a designated set of information elements that are associated with federation executions. Implementation of the MOM information elements as specified in IEEE STD 1516.1-2000 ([34]) provides a mechanism for management of federation executions using existing HLA services. Inclusion of the MOM is required for all FOMs. Any FOM or SOM that fully conforms to all of the rules and constraints stated in this specification is a compliant object model.

2.4. GME (Generic Modeling Environment)

The Generic Modeling Environment (GME) is a configurable toolkit for creating domain-specific modeling and program synthesis environments [8]. The configuration is accomplished through metamodels specifying the modeling paradigm (i.e., modeling language) of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain. It defines the family of models that can be created using the resultant modeling environment.

The metamodels are used to automatically generate the target domain-specific environment. An interesting aspect of this approach is that the environment itself is used to build the metamodels. The generated domain-specific environment is then used to build and manipulate domain models that are stored in a model database.

The metamodels specifying the modeling paradigm are used to automatically generate the target domain-specific environment. The generated domain-specific environment is then used to build domain models that are stored in a model database or in XML format. These models are used to automatically generate the applications or to synthesize input to different COTS analysis tools.

GME has a modular, extensible architecture that uses MS COM for integration. GME is easily extensible; external components can be written in any language that supports COM (C++, Visual Basic, C#, Python etc.). GME has many advanced features. A built-in constraint manager enforces all domain constraints during model building. GME supports multiple aspect modeling. It provides metamodel composition for reusing and combining existing modeling languages and language concepts. It supports model libraries for reuse at the model level. All GME modeling languages provide type inheritance. Model visualization is customizable through decorator interfaces.

2.4.1. Technical Overview

The GME includes several other relevant features:

- It is used primarily for model-building. The models take the form of graphical, multi-aspect, attributed entity-relationship diagrams. The dynamic semantics of a model is not the concern of GME – that is determined later during the model interpretation process.

- It supports various techniques for building large-scale, complex models. The techniques include: hierarchy, multiple aspects, sets, references, and explicit constraints. These concepts are discussed later.
- It contains one or more integrated model interpreters that perform translation and analysis of models currently under development.

GME has a modular, component-based architecture depicted in Figure 6 below.

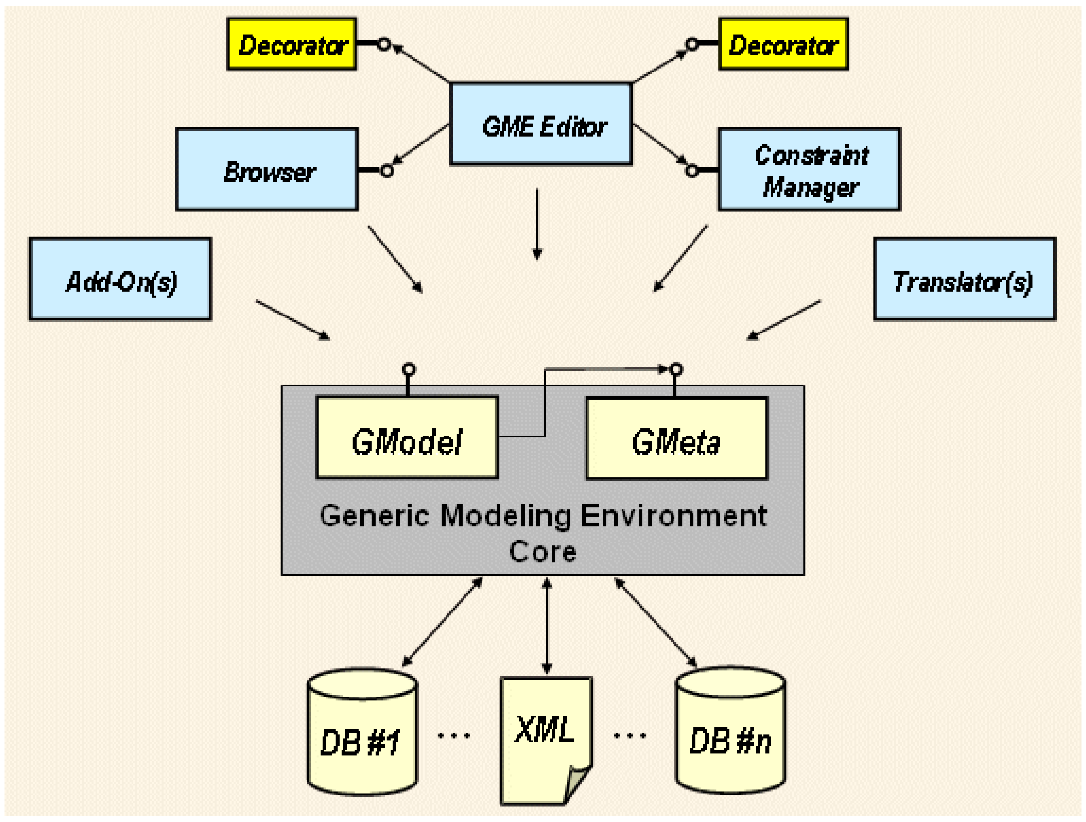


Figure 6. Generic Modeling Environment Architecture

The Core component implements the two fundamental building blocks of a modeling environment: objects and relations. Among its services are distributed access (i.e. locking) and undo/redo [8].

On top of the architecture, the user interacts with these components: the GME User Interface, the Model Browser, the Constraint Manager, Interpreters and Add-ons.

Add-ons are event-driven model interpreters. The GModel component exposes a set of events, such as "Object Deleted," "Set Member Added," "Attribute Changed," etc. External components can register to receive some or all of these events. They are automatically invoked by the GModel when the events occur. When a particular domain calls for some special operations, these can be supported without modifying the GME itself [8].

The Constraint Manager behaves as an interpreter and an add-on at the same time. It can be invoked when event-driven constraints are present in the given paradigm and it is also invoked explicitly by the user. Depending on the priority of a constraint, the operation that caused a constraint violation is aborted. For less serious constraint violations, the Constraint Manager only issues a warning message.

There is no special privilege for the GME User Interface component in this architecture. Any operation that can be done through the Graphical User Interface (GUI) can also be done programmatically through the interfaces. By this way, the architecture is very flexible and supports extensibility of the whole environment.

2.4.2. Modeling Concepts

The vocabulary of the domain-specific languages implemented by different GME configurations is based on a set of generic concepts built into GME itself. GME supports various concepts for building large-scale, complex models as depicted in Figure 7.

A Project contains a set of Folders. Folders are containers that help organize Models, just like folders on a disk help organize files. Folders contain Models. Models, Atoms, References, Connections and Sets are all first class objects, or FCOs for short. FCO is used as the abstract base class for these elements in modeling.

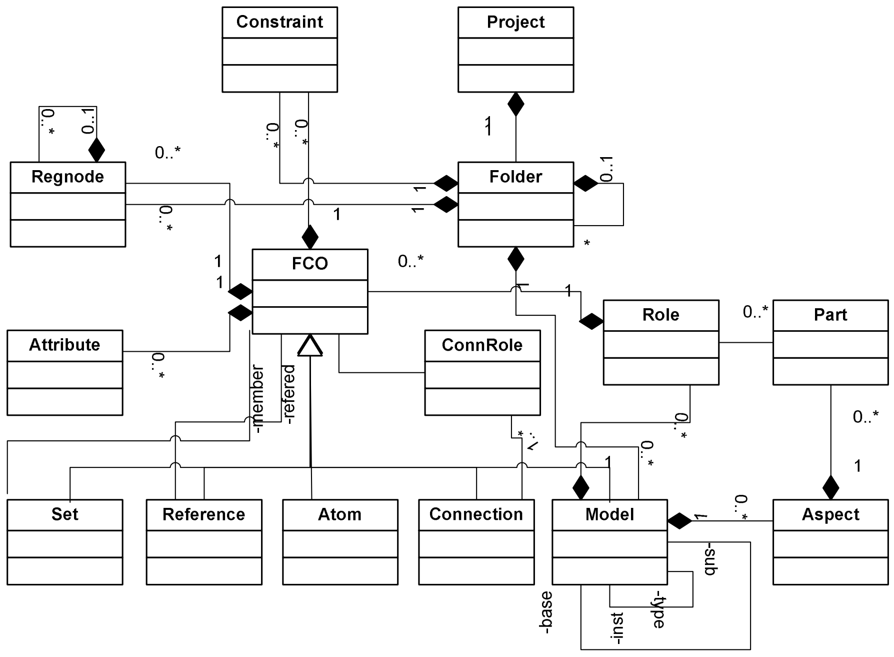


Figure 7. GME modeling concepts [8]

Atoms are the elementary objects; that is, they cannot contain parts. Each kind of Atom is associated with an icon and can have a predefined set of attributes, whose values are user changeable.

Models are the compound objects that can have parts and inner structure. A part in a container Model always has a Role. The modeling paradigm determines what kind of parts are allowed in Models acting in which Roles, but the modeler determines the specific instances and number of parts a given model contains (of course, explicit constraints can always restrict the design space). Any element must have at most one parent, which must be a Model. At least one Model does not have a parent and is called a root Model.

A common way of expressing a relationship between two model elements in GME is with a Connection. Connections can be directed or undirected, and have Attributes. In order to make a Connection between two modeling elements they must have the same parent in the containment hierarchy. It is specified what kind of objects can participate in a given kind of Connection. Connections can further be restricted by explicit Constraints, such as their multiplicity.

In GME, a Reference must appear as a part in a Model. This establishes a relationship between the Model that contains the Reference and the referred-to object. Any FCO, except for a Connection, can be referred to (even References themselves). A Reference always refers to exactly one FCO, while a single FCO can be referred to by multiple References.

Some information does not lend itself well to graphical representation. GME provides the facility to augment the graphical objects with textual attributes. All FCOs can have different sets of Attributes among the kinds text, integer, double, boolean and enumerated.

The GME is made up of instances of Folders, FCOs (Models, Atoms, Sets, References, Connections), Roles, Constraints and Aspects. These are the main concepts that are used to define a modeling paradigm. As soon as a particular model is created in GME, it becomes a type (class). It can be sub typed and instantiated as many times as the user wishes. The general rules that govern the behavior of this inheritance hierarchy are:

- Only attribute values of model instances can be modified. No parts can be added or deleted.
- Parts cannot be deleted but new parts can be added to subtypes.

This concept supports the reuse and maintenance of models because any change in a type automatically propagates down the type hierarchy. Also, this makes it possible to create libraries of type models that can be used in multiple applications in the given domain.

2.4.3. Extensibility

GME identifies data and tool integration as one of its primary application areas, so data access and standards-compliant extensibility is one of its primary design goals. Hence, GME is completely component-based with public interfaces among its components. Most notably, the GME editor, i.e. the visualization component, the model storage and logic, and the meta-modeling module is separated by interfaces which are accessible to user-written components as well, thus giving them access level identical to that of the GME editor. Since the component model is COM, the primary languages for integration are C++ and Visual Basic, while Java, Python, etc. access is also available. Access is bi-directional, and fully transactional, which makes different 'on-line modeling' scenarios feasible. For example, the GME user interface itself can be used as the user interface of a generated application to provide feedback to the user in terms of the models. Furthermore, the bi-directional access makes it possible to convert legacy data into models in an automated fashion [8].

Programming at the component level is somewhat challenging in the sense that it requires advanced transaction control and event handling. Several alternatives provide easier access through simpler interfaces (albeit with limited functionality). First, the GME pattern-based report language provides simple reporting capabilities by interpreting macro definitions in a simple text input file. A more complex interface is layered on top of the COM interfaces providing an easy-to-use extensible C++ API. GME also provides bi-directional XML access for both model and meta-model information [8].

2.4.4. Metamodeling with GME

Defining a modeling paradigm can be considered just another modeling problem. It is quite natural then that GME itself is used to solve this problem. There is a metamodeling paradigm defined that configures GME for creating metamodels [35]. These models are then automatically translated into GME configuration information through model interpretation. Originally, the metamodeling paradigm was hand-crafted. Once the metamodeling interpreter was operational, a meta-metamodel was created and the metamodeling paradigm was regenerated automatically. This is similar to writing C compilers in C.

The metamodeling paradigm is based on UML [36]. The syntactic definitions are modeled using pure UML class diagrams and the static semantics are specified with constraints using the Object Constraint Language (OCL) [40]. Only the specifications of presentation/visualization information are extensions to UML, mainly in the form of predefined object attributes.

Just as the reusability of domain models from application to application is essential, the reusability of meta-models from domain to domain is also an important consideration. In GME a library of meta-models of important sub-domains is made available to the meta modeler, who then can pick and choose from them, extend and compose them together to specify new domain languages. The extension and composition mechanisms must not modify the original metamodels for two reasons. First, changes in the meta-model libraries, reflecting a better understanding of the given domain, for example, should propagate to the meta-models that utilize them. Second, by precisely specifying the extension and composition rules, using inheritance and equivalence operators, for instance, models specified in the original domain language can be automatically translated to comply with the new, extended and composed, modeling language. This is a simple and elegant solution to the well-known model migration problem. For more detail on metamodel composition please see [37].

2.4.5. User Interface

The native graphical user interface of GME is shown in the Figure 8 below. The picture shows a model of a signal flow graph loaded. In this simple model, only Models, Atoms and Connections are used. The window on the right hand side shows the Model Browser that displays the whole project in a tree-like fashion. The Aggregate tab displays the containment hierarchy, while the Inheritance tab shows the type inheritance hierarchy. The Meta tab provides an overview of the modeling paradigm specifications. The bottom window is the Part Browser where all the parts that are available in the current aspect of the current model are shown. Notice that two tabs indicate the aspects of the signal flow model: SignalFlow and Parameter.

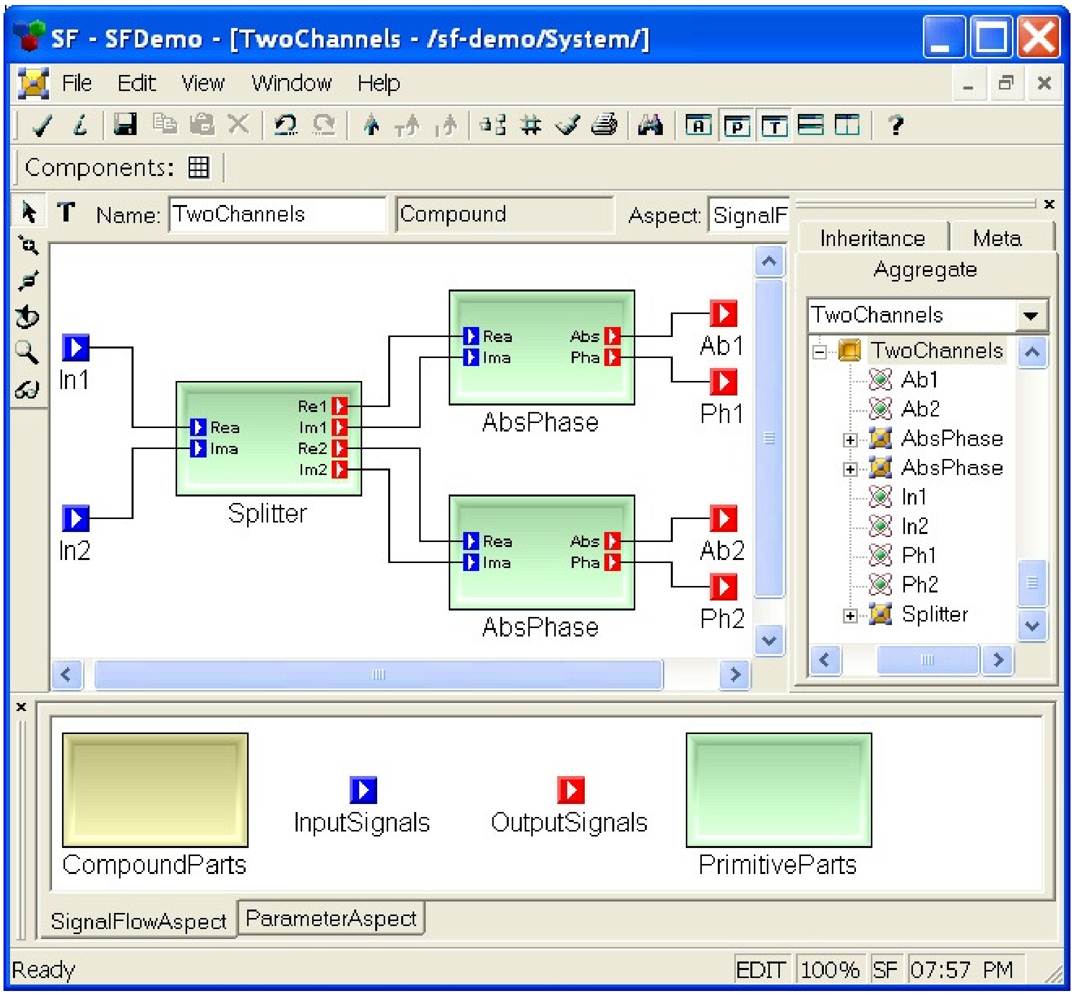


Figure 8. Graphical User Interface of GME

2.5. Graph Transformations with GreAT

Graph Rewriting and Transformation (GReAT) [9][10] is a transformation language developed for model-to-model transformations/rewriting. In other words, GReAT is a tool for building model transformation tools using graph transformation techniques. This section provides an overview of GReAT, while [38] provides a more detailed description. The operational semantics of GReAT is formally defined in [10]. GReAT is based on the theoretical work of graph grammars and transformations [26] and belongs to the set of practical graph transformations systems.

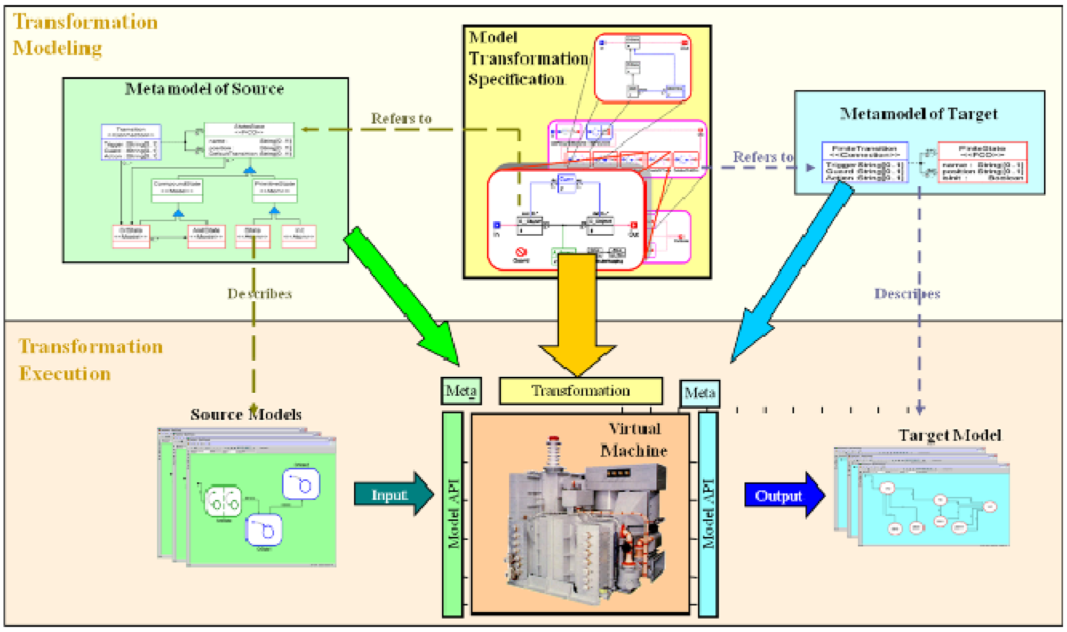


Figure 9. Transformation Modeling and Execution in GReAT [9]

GReAT uses metamodels to specify the abstract syntax of the input and the target models (i.e. the modeling languages), and sequenced graph rewriting rules for specifying the transformation itself. It takes the input graph, applies the transformations to it, and generates the output graph. Inputs to the GRE are the UML class diagrams for the input and output graphs (also known as meta-models), the transformation specification and the input graph. The GRE executes the rules according to the sequencing and produces an output graph based upon the actions of the rules. The approach used in GReAT is illustrated in Figure 9 above.

For building a model transformation tool using GReAT, one first has to specify the metamodels of the input and the target models. This is done using the UML-style class diagrams, using GME and UDM [39].

The GReAT transformation rules are graph rewriting rules that transform a part of the input model (a typed, attributed graph of model elements) into part of the target model (a typed, attributed graph of model elements). The rule has a pattern (to be matched against input model), a guard (an expression to be evaluated on the result of the match), and a set of actions (that create of delete model elements, or modify attributes). The match is always computed starting from specific nodes, called the pivot nodes in the graph to limit the search, and pivot nodes could be passed down to subsequent rules. The rule execution is explicitly sequenced, and various control structures (including conditional and looping structures are available). GReAT includes a number of tools, as illustrated on the Figure 10 below.

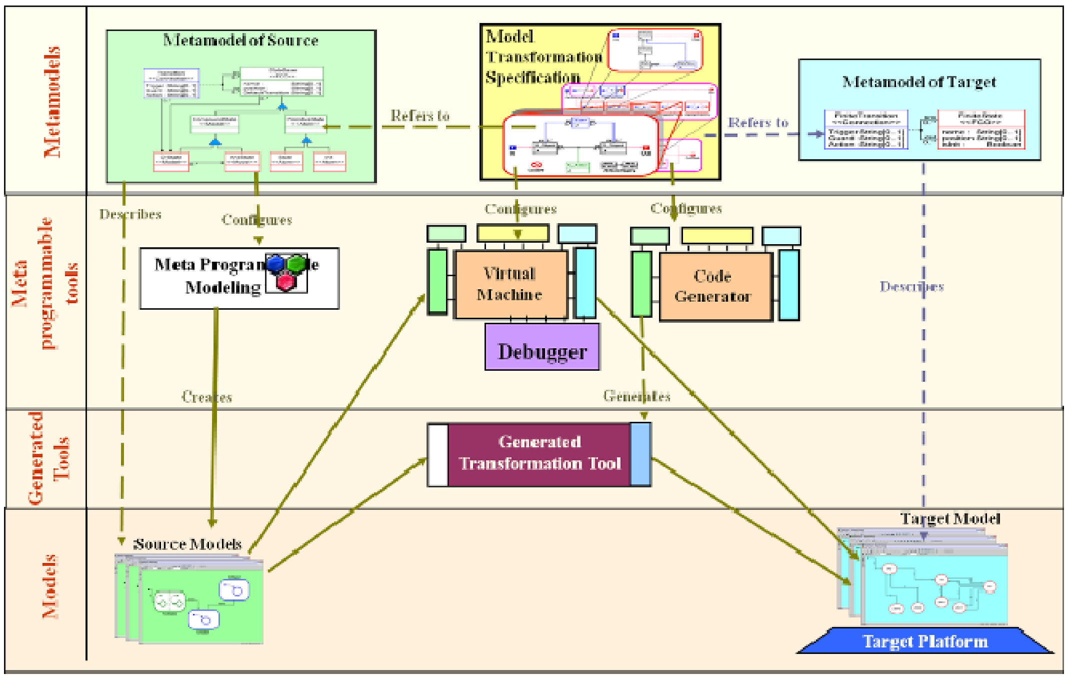


Figure 10. Transformation Tools in GReAT [9]

GReAT programs (transformation specs) are typically executed using a 'virtual machine', called the GR Engine. This is an interpreter that interprets the rewriting program, and allows the use of the debugger. Once the transformation is corrected, one can use the code generator the produce executable (C++) code from the transformation program, which is then linked with other libraries to form an executable that runs with much better performance. The GReAT transformation rules could include executable code written in C++, in the form of ' attribute mapping' code -- this is for computations that need to be performed upon each transformation step, best written as imperative code.

The GReAT tool suite has been designed for the rapid specification and implementation of model-to-model transformations. These transformations are required in many domains. A few use case scenarios of this tools suite are:

- Developing model interpreters that convert GME models (conforming to a meta-model) to XML files conforming to a particular dtd.
- Developing model interpreters that convert GME models (conforming to a metamodel) to a set of secondary data structures. A visitor can then be written to convert the secondary models to text.
- Developing model interpreters that convert GME models (conforming to a metamodel) to GME models conforming to another metamodel.
- Developing transformers that convert xml files belonging to one dtd to xml files belonging to another dtd.
- Developing transformers that convert xml files belonging to a dtd to GME models.

GReAT is the model transformation language that we have employed in FACM to FAM transformations. *UMLModelTransformer* paradigm, which is the metamodel of the GReAT language, comes out of the box as registered in GME. By creating models conforming to this paradigm in GME, we are able to define our model transformations. GReAT can be divided into 3 distinct parts: (1) Pattern specification language, (2) Graph transformation language, and (3) Control flow language, which we briefly summarize below. The details on all the three parts can be found in [9].

2.5.1. The Pattern Specification Language

The heart of a graph transformation language is the pattern specification language and the related pattern matching algorithms. Graph patterns allow selecting portions of the input (host) graph, and thus specify the scope of individual transformation steps. In broadest terms, the goal of the pattern language is to specify patterns over graphs (of objects and links), where the vertices and edges belong to specific classes and associations. GReAT assumes that a UML class diagram is available for the objects. The UML class diagram can be considered as the "graph grammar," which specifies all legal constructs formed over the objects that are instances of classes introduced in the class diagram. Please refer to [9] for details.

2.5.2. Graph Rewriting Transformation Language

Another important concern besides pattern specification is the specification of static structural constraints in graphs and ensuring that these are maintained throughout the transformations [25]. Model-to-model transformations usually transform models from one domain to models that conform to another domain making the problem two-fold. The first problem is to specify and maintain two different models conforming to two different metamodels. An even more important problem to address involves maintaining references between the two models.

GReAT's solution to these problems is to use the source and destination metamodels to explicitly specify the temporary vertices and edges. This approach creates a unified metamodel along with the temporary objects. Then the source model, destination model, and temporary objects can be treated as a single graph. In GReAT, each pattern object's type conforms to the unified metamodel and only transformations that do not violate the metamodel are allowed. At the end of the transformation, the temporary objects are removed and the two models conform exactly to their respective metamodels.

The graph transformation language of GReAT defines a production (also referred to as rule) as the basic transformation entity. A production contains a pattern graph that consists of pattern vertices and edges. These pattern objects conform to a type from the metamodel. Each pattern has another attribute that specifies the role it plays in the transformation. A pattern can play the following three different roles:

- Bind – used to match objects in the graph.
- Delete – also used to match objects in the graph, but after these objects are matched they are deleted from the graph.
- New – used to create objects after the pattern is matched.

The execution of a rule involves matching every pattern object marked either bind or delete. If the pattern matcher is successful in finding matches for the pattern, then for each match the pattern objects marked delete are deleted from the match and objects marked new are created.

Sometimes the patterns by themselves are not enough to specify the exact graph parts to match and we need other, non-structural constraints on the pattern. These constraints or pre-conditions are expressed in a guard and are described using Object Constraint Language (OCL) [40]. There is also a need to provide values to attributes of newly created objects and/or modify attributes of existing object. Attribute Mapping is another ingredient of the production: it describes how the attributes of the “new” objects should be computed from the attributes of the objects participating in the match. Attribute mapping is applied to each match after the structural changes are completed.

A production is thus a 4-tuple, containing a pattern graph, mapping function that maps pattern objects to actions, a guard expression (in OCL), and an attribute mapping.

2.5.3. Controlled Graph Rewriting and Transformation

To increase the efficiency and effectiveness of GReAT, it is essential to have efficient implementations for the productions. Since the pattern matcher is the most time consuming operation, it needs to be optimized. One solution is to reduce the search space (and thus time) by starting the pattern-matching algorithm with an initial context. An initial context is a partial binding of pattern objects to input (host) graph objects. In order to provide initial bindings, the production definition is expanded to include the concept of ports. Ports are elements of a production that are visible at a higher-level and can then be used to supply initial bindings. Ports are also used to retrieve output objects from the production.

The next concern is the application order of the productions. The control flow language of GReAT supports the following features:

- Sequencing – rules (i.e., productions) can be sequenced to fire one after another. This is achieved by attaching the output port of the first rule to the input port of the next.
- Non-Determinism – when required parallel execution of a set of rules can be specified. The order of execution of these rules is non-deterministic. This construct is achieved by attaching the output of one rule to the input of more than one rule.
- Hierarchy – High-level rules have been introduced in the language. These are used for encapsulation and data abstraction. Compound

rules can contain other compound rules or primitive transformation rules.

- Recursion – A high level rule can “call” itself.
- Test/Case – A conditional branching construct that can be use to choose between different control flow paths.

2.5.4. The GReAT Engine

The model transformation language described above is supported through a Graph Rewriting and Transformation Execution Engine (GReAT-E), whose architecture is shown in Figure 11. The engine works as an interpreter: it takes the model transformation “program” in the form of a data structure, and it “executes” it on an input graph to produce an output graph. The engine uses generic API-s (using the model-driven reflection package called UDM [39]), and is thus suitable for executing any model transformation.

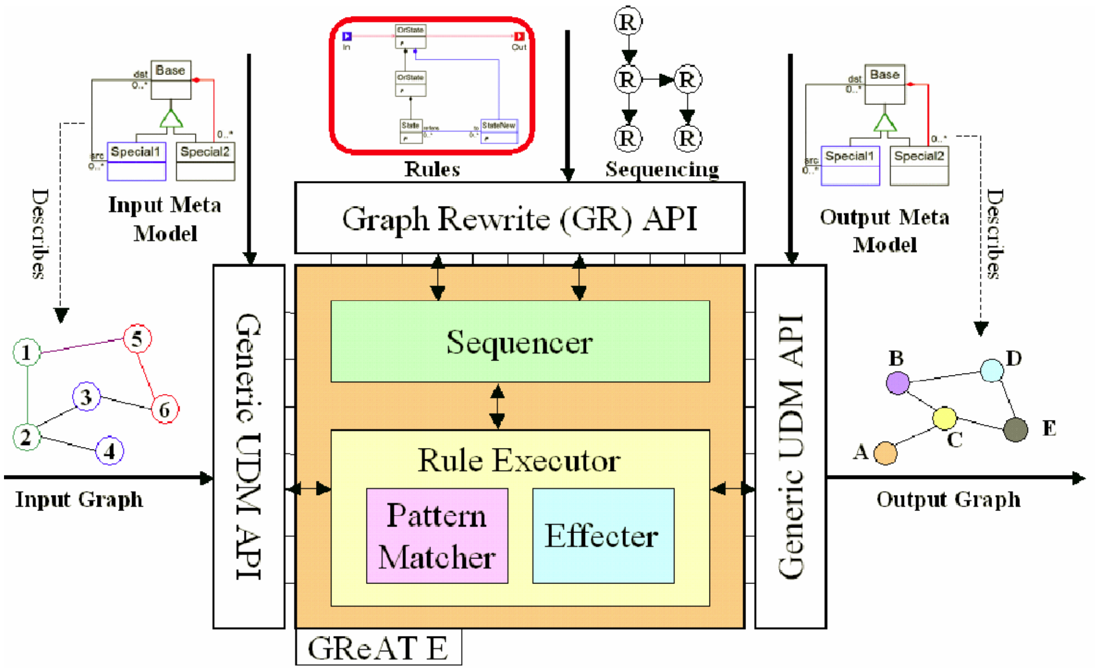


Figure 11. Run time architecture of the GReAT Interpreter [9]

The interpreter accesses the input and output graph with the help of a generic UDM API that allows the traversal of input and output graph. The rewrite rules are stored in their own language format and can be accessed using the language specific UDM API. The GReAT is composed of two major components, (1) Sequencer, (2) Rule Executor (RE). The Rule Executor is further broken down into (1) Pattern Matcher (PM) and (2) Effector (or 'output generator'). The Sequencer determines the order of execution for the rules using the '*Execute'* function described above and it calls the *ExecuteRule* for each rule. The rule executor internally calls the PM with the pattern of the rule. The matches found by the PM are used by the Effector to manipulate the output graph by performing the actions specified in the rules.

The Pattern Matcher finds the subgraph(s) in the input graph that are isomorphic to the pattern specification. When a pattern vertex/edge matches a vertex/edge in the input graph, the pattern vertex/edge will be bound to that vertex/edge. The matcher starts with an initial binding supplied to it by the Sequencer. Then it incrementally extends the bindings till there are no unbound edges/vertices in the pattern. At each step it first checks every unbound edge that has both its vertices bound and tries to bind these. After it succeeds to bind all such edges it then finds an edge with one vertex bound and then binds the edge and its unbound vertex. This process is repeated until all the vertices and edges are bound.

CHAPTER 3

FIELD ARTILLERY METAMODEL

This section is about the formalization of the conceptual model. The data model identifies the entities in the FA domain along with their properties and associations. The conceptual data model is constructed as a metamodel by using the GME. Once the FA metamodel is registered with GME, it automatically provides a customized environment to model particular FA data models.

Figure 12 depicts an excerpt of the metamodel as shown in the GME Browser window. The DurableData, EntityFADomain and Message folders and the Utility sheet contain the data model part. Finally, FADMMRoot sheet gathers the top level elements in these folders under the root element, DataModel.

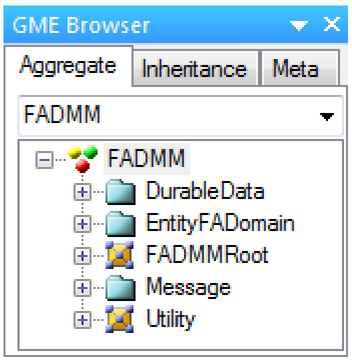


Figure 12. An overview of FADMM in GME Browser

Before moving any further, a clarification on the levels of modeling would be worthwhile. Object Management Group (OMG) introduces a four-layer metamodel hierarchy for defining modeling, metamodeling, and meta-metamodeling languages and activities in [36].

Table 3 relates the FADM metamodel to OMG's four-layer modeling hierarchy.

Table 3. FA Metamodel correlated with OMG's model hierarchy

OMG's Metamodel Hierarchy	**Related Model**
Meta-metamodel (M3 layer)	GME metamodel (metaGME)
Metamodel (M2 layer)	FADM metamodel (referred to as a "paradigm" in GME vernacular)
Model (M1 layer)	A particular FA mission description, e.g., The Adjustment followed by Fire For Effect (AdjFFE).
Run-time instance (M0 layer)	A particular execution of a FA mission (e.g., exercising an AdjFFE scenario)

3.1. Data Model

Messages are an important part of the FA domain information. Typically, they are highly structured and they have with many optional or conditional fields of various data types. There are further syntactic, semantic or cardinality constraints on the message structures, both on a single field and inter-field basis.

Our analysis has revealed two kinds of message usage in the domain. The first kind includes those messages that are sent as single chunks of information independent of any previous ones. Every such message supplants its immediate predecessor of the same type. The second kind of usage is practically an accumulation of a series of communications of the same message type. Specifically, the current interpretation of a message at a particular destination is a function of all previous receptions of that message kind. In such a usage, the first reception of a message creates an initial copy at the destination. Subsequent message receptions result in updates on the original copy. The message is removed from the scope of the actor with the arrival of a special deletion message. In the FA domain model the majority of the message usages are of the first kind.

The mission hierarchy builds up the set of FA observed fire missions that are modeled in this work. The mission model elements themselves do not possess mission related information; rather they are simple atomic elements merely used as markers of mission types. Mission specific information is conveyed within message structures. Mission model elements exist as parts of some of those message structures. Figure 13 and Figure 14 show the synopsis of the data model hierarchy as manifested in GME Browser.

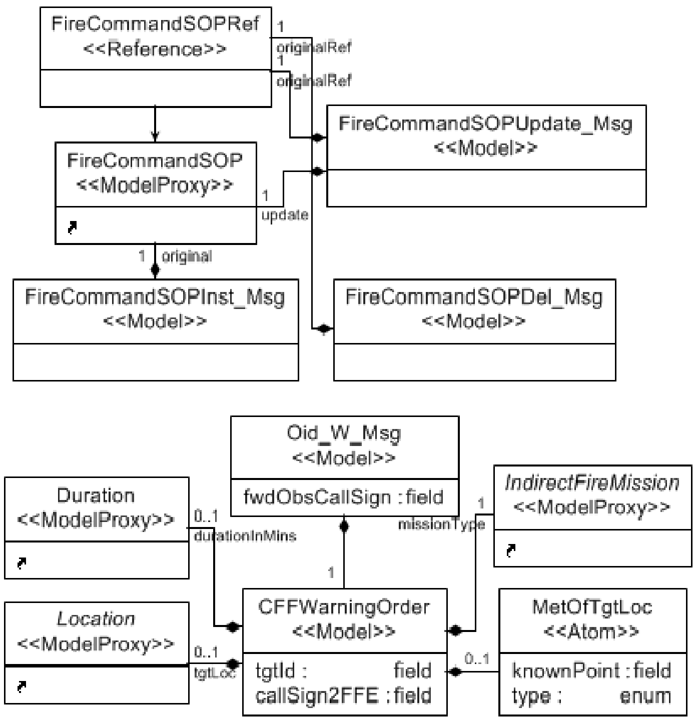


Figure 13. A sample of data model in GME

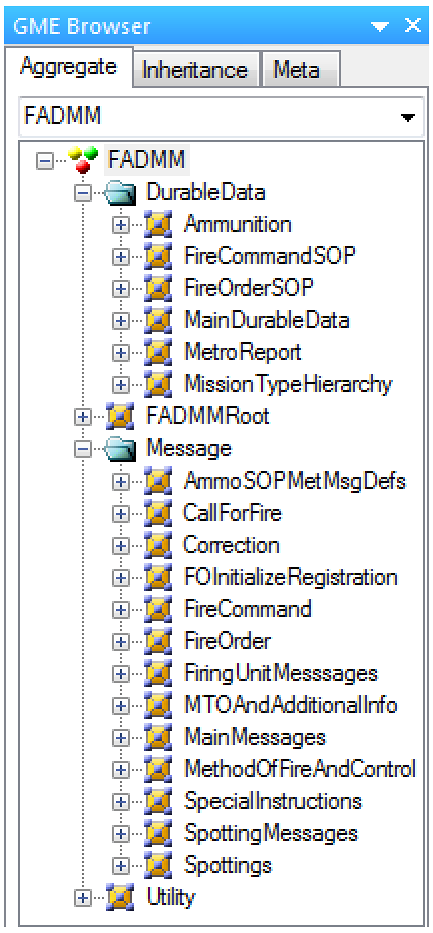


Figure 14. An overview of data model in GME Browser

3.2. Metamodeling of Observed Fire in GME

An examination of the narrative FA model outline of 2.2. Elements of Field Artillery suggests a set of messages, entities and relationships between these entities.

The observed (indirect) fire techniques of the Field Artillery (FA) domain constitute the source model for the transformations. This section first gives a conceptual overview on the elements and fire direction processes of the FA domain, and then presents the metamodeling effort of the domain implemented in GME environment.

The entities in the FA domain are FO, Battalion FDC, Battery FDC, Firing Section, FAT and Target. Note that, when we examine the field manuals [31], it is seen that there are many more sub-divisions and personnel working in coordination in an FA unit. In order to reduce complexity and provide a better understanding of the responsibilities, we worked out to gather all these real-life participants under following functional entities.

From a military perspective, firing sections are organized under batteries and batteries are organized under battalions. However, from modeling perspective, the kinds of interactions that can occur among them are more important and organizational hierarchy is not a serious concern. Thus, we gathered the Battalion FDC, Battery FDC and Firing Section under the aggregate FAT (Field Artillery Team). In fact, this non-hierarchic organization schema is relevant in order to sketch message communication diagrams straightforwardly in GME.

Target is rather a passive actor that most of the time is a primary concern of the others. Its most distinguishing features are that it provides a location to issue fire orders for, shapes the method of engagement, fire and control according to its kind, and provide feedback for mission termination. Last, but not least, a field artillery operation cannot be envisioned at all without a target concept.

All the entities mentioned are modeled as GME models and are considered specializations of an abstract Entity model element. Figure 15 depicts the entities as defined in GME.

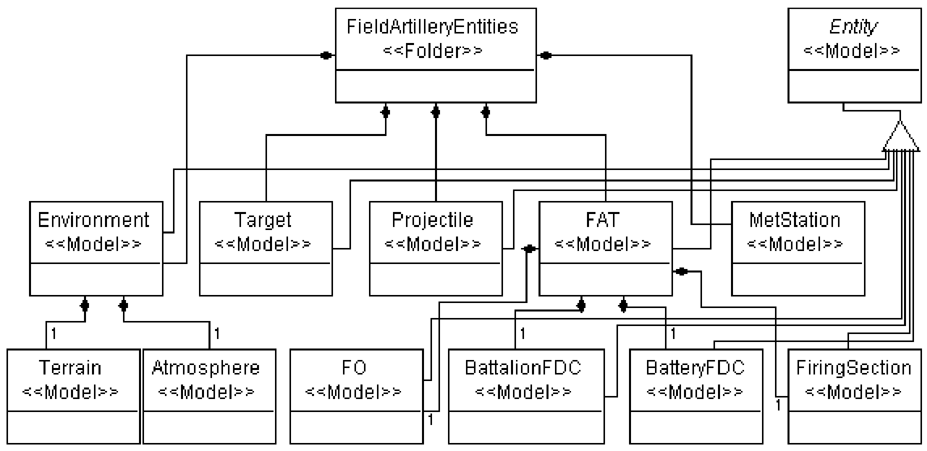
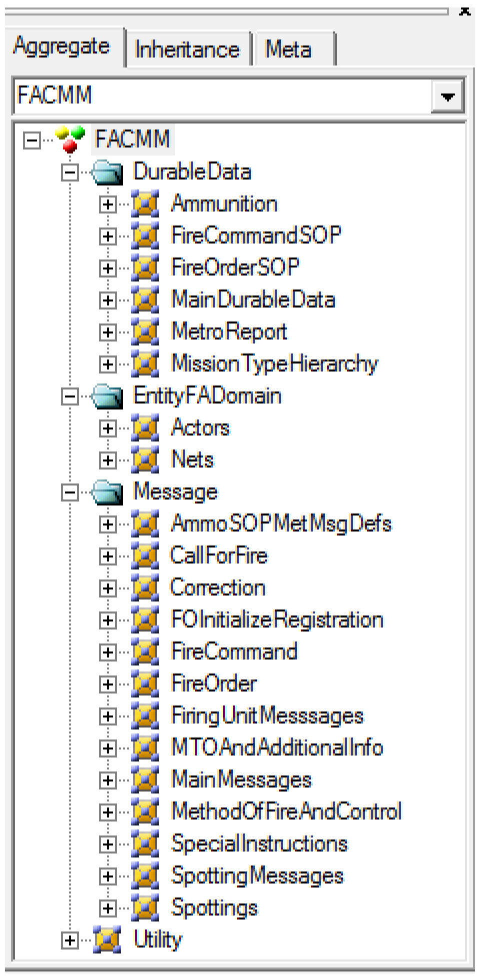


Figure 15. Field Artillery Entities



- FACMM
  - DurableData
    - Ammunition
    - FireCommandSOP
    - FireOrderSOP
    - MainDurableData
    - MetroReport
    - MissionTypeHierarchy
  - EntityFADomain
    - Actors
    - Nets
  - Message
    - AmmoSOPMetMsgDefs
    - CallForFire
    - Correction
    - FOInitializeRegistration
    - FireCommand
    - FireOrder
    - FiringUnitMesssages
    - MTOAndAdditionalInfo
    - MainMessages
    - MethodOfFireAndControl
    - SpecialInstructions
    - SpottingMessages
    - Spottings
  - Utility

Figure 16. An overview of the FACM metamodel

An overview of the FACM metamodel, as sown in GME model browser, is given in Figure 16. The components making up the model are organized under four major folders.

The actors are the active driving forces of mission executions. In modeling actors we assumed a functional perspective rather than a one to one mapping of real life entities. Therefore they represent personnel, equipment or an aggregation of these in FA batteries. Actors are the producers and comsumers of the message traffic.

Nets are similar to actors in terms for producing and consuming messages. However they are an abstraction of a set of actors in the domain. Any message entering a net is distributed to the actors in the net. Hence they resemble the real life nets in the mission space.

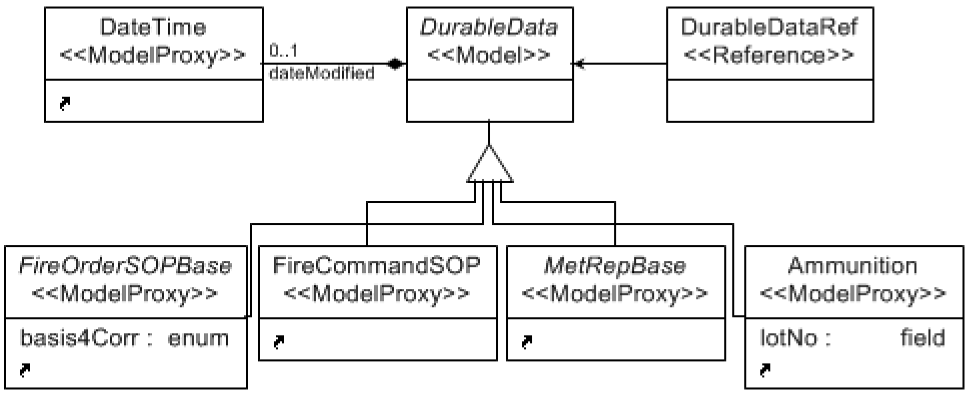


Figure 17. An example of DurableData in FACM

The DurableData folder contains the model elements that represent the long lasting domain information such as ammunition, SOPs, and metro reports. Durable data are usually expected to exist and keep their states until a mission is either completed or aborted. Their states can be (partly) modified and these modifications are transmitted among actors. The sketch of DurableData is shown in Figure 17 as an example. The details can be found in Appendix B.

The messages are communicated among the actors while executing FA missions. These are generally instantaneously generated by its sender and are void. Once they are consumed by the receiver(s). In short they are short-lived, volatile information chunks that do not have lives spanning multiple message communications. The messages are best candidates for transforming into interaction classes in OMT.

The utility sheet consists of a variety of lower level components that are commonly used by the higher level messages. These encompass such components as date and time, location, measurement, etc. These utility components are specially treated during transformations in that they have their own transformation rules and these rules are called from the higher level rules that transform messages. Consequently the utilities facilitate modularization and reuse.

Please refer to Appendix B for the details of the entities.

CHAPTER 4

HLA OMT METAMODEL

4.1. Federation Design Model

The HLA-OMT metamodel has been developed in a previous study [6]. Here we provide only refreshment that would be necessary for the reader to understand our target model.

The Federation Design Model (FDM) provides an interface to define a federation and the federates and to connect them to the related FOM and SOMs. In FDM, FOM and SOMs are referenced object models. Each design can include one federation and one FOM reference, while there may be any number of federates and SOMs.

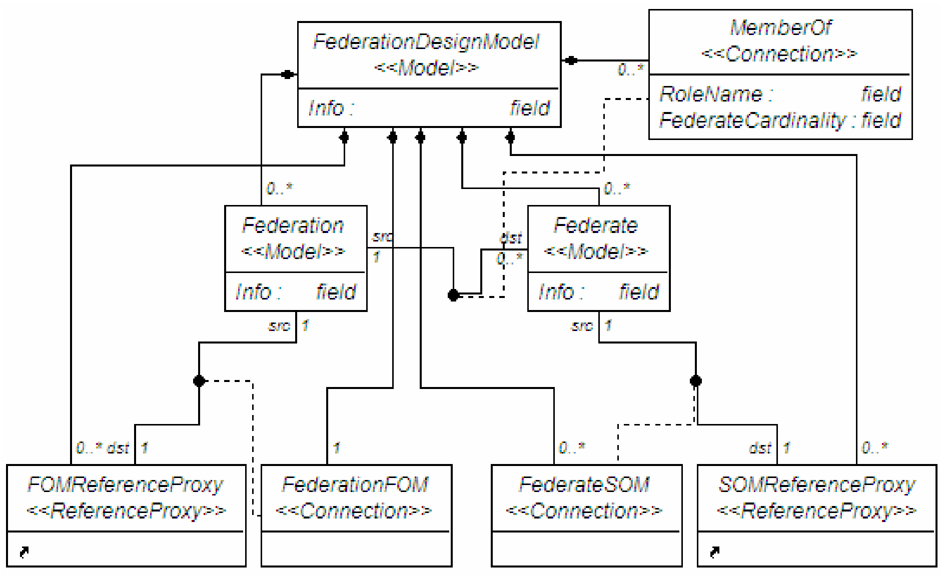


Figure 18. Federation design model

Figure 18 shows the GME class diagram of federation design model. There is a “MemberOf” connection between federation and federates. This connection presents the federation execution capabilities. Federation Design Model is not intended to complete; it is constructed to show the usage of object models in context.

4.2. Object Model

The Object Model paradigm sheet includes the main diagram for object models. As seen in Figure 19, there are three types of object models, namely, FOM, SOM and Other. FOM and SOM are HLA object models which are defined in HLA OMT specification. The “Other” type provides a template for “temporary” object models.

Object Model, which is the parent of FOM, SOM and Other, is an abstract class. The inheritance operator in this figure presents a parent-child relation that is analogous to the inheritance in usual OO approach. Object models have some attributes [6]:

- Name: The name assigned to the object model.
- Version: The version identification assigned to the object model.
- Modification Date: The latest date on which this version of the object model was created or modified.
- Purpose: The purpose for which the federate or federation was developed.
- Application Domain: The type or class of application to which the federate or federation applies.
- Sponsor: The organization that sponsored the development of the federate or federation.
- Point of Contact: The name of the point of contact (POC) for information.
- POC Organization: The organization with which the POC is affiliated.
- POC Telephone: The telephone number for the POC.
- POC E-mail: The e-mail address of the POC.
- References: Additional sources of information. The default value is “NA”.

- Other: Other information related to the object model. The default value is “NA”.
- MOM version: The version of the included management object model. The default value is “IEEE 1516”. Selecting a MOM name is required for all FOMs. In SOMs the MOM shall be included if needed.
- Notes: Note labels added to the object model.

Object models have five aspects, namely Classes, User-Supplied Tags, Synchronization, Switches and Time Representation. In each aspect the model definitions are taken from the related paradigm sheets with proxy elements. The Classes aspect includes the definition of object classes, attributes, interaction classes and parameters. The User-Supplied Tags aspect includes the user-supplied tag elements. The Synchronization aspect includes the definition of synchronization point models. The Switches aspect includes the switches in order to change the initial settings and time representation aspect includes the definition of look ahead and timestamp models.

4.3. OMT Core

Both FOM’s and SOM’s structure make use of the same set of model constructs defined inside the OMTCore package. Therefore the OMT elements can be considered as the core of the object model, as its name implies. In Figure 19, we see the object model structure. The OMT core contains any number of federates and HLA classes. Federates have publish and subscribe relationships to attribute lists, which in turn have relationship to the attributes of HLA classes.

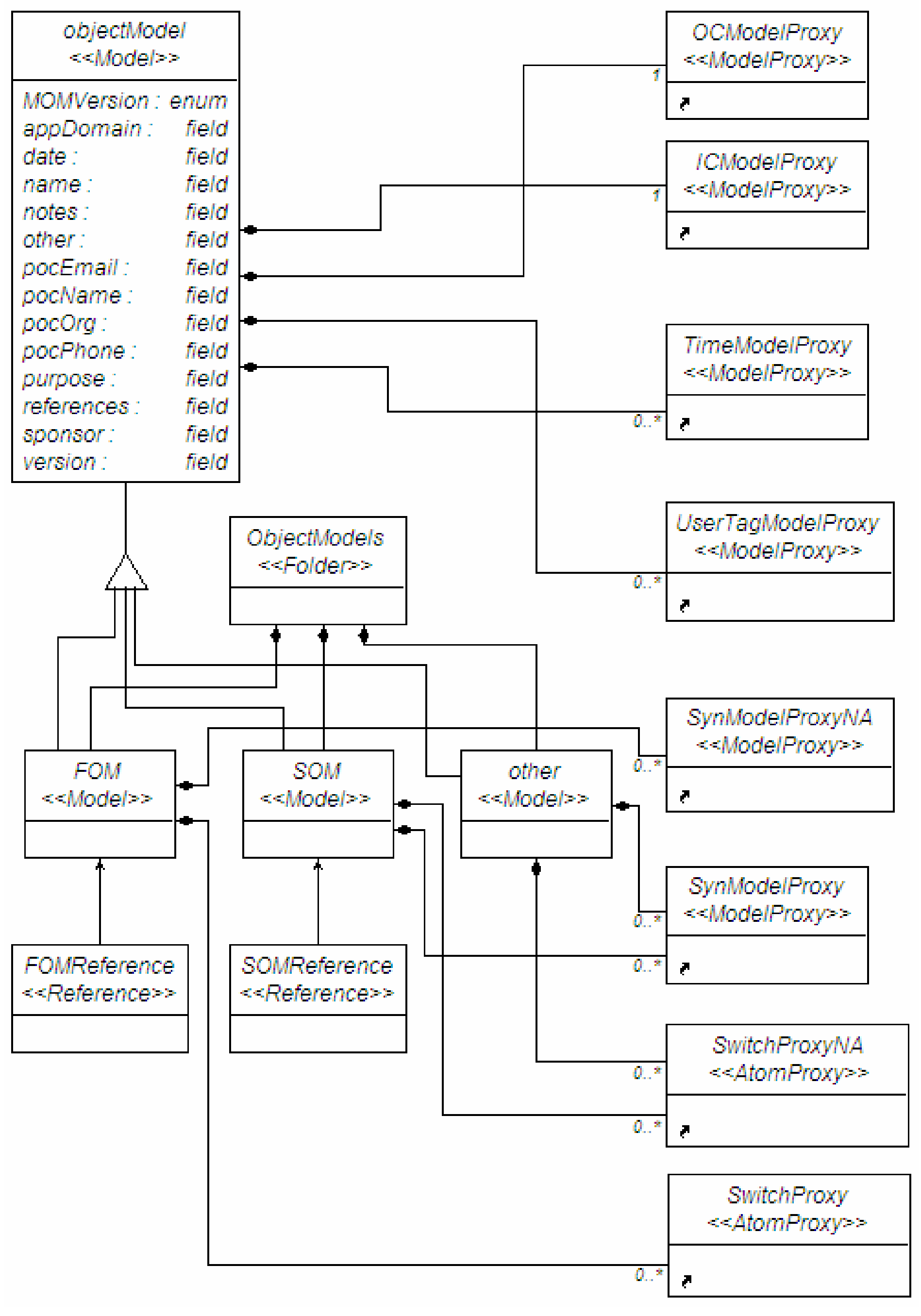


Figure 19. OMT Object Models diagram

There are two types of HLA classes and HLA attributes. Each class has its own type of attributes, namely, Object attribute and Interaction parameter. The value that an HLA attribute carries is represented with a separate HLAAttribValue FCO construct, which has a recursively defined tree structure having primitive GME-typed atomic values at the leaves. This structure is displayed in Figure 20.

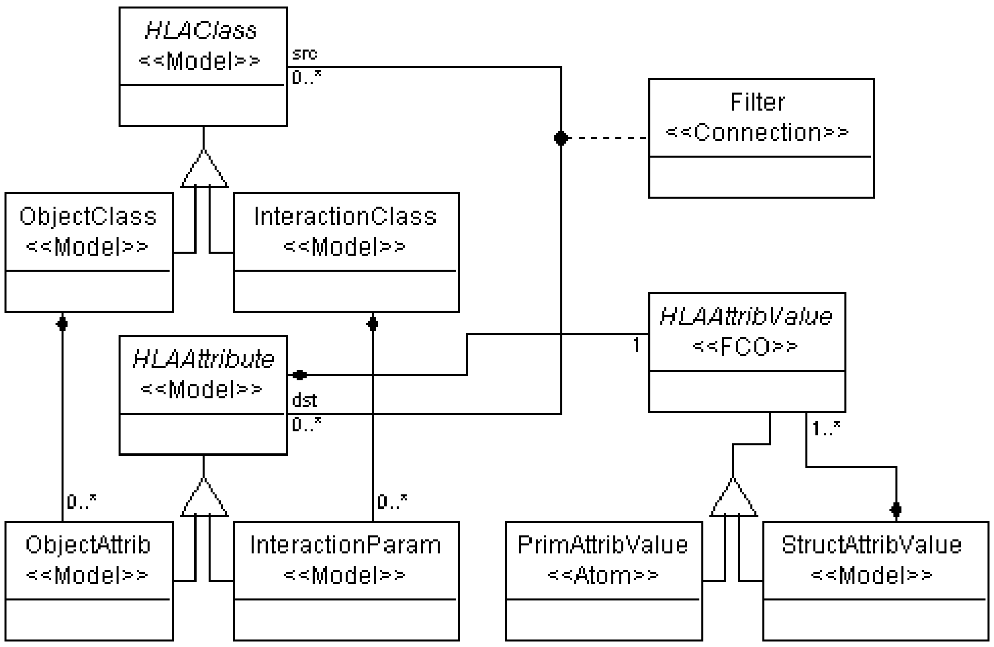
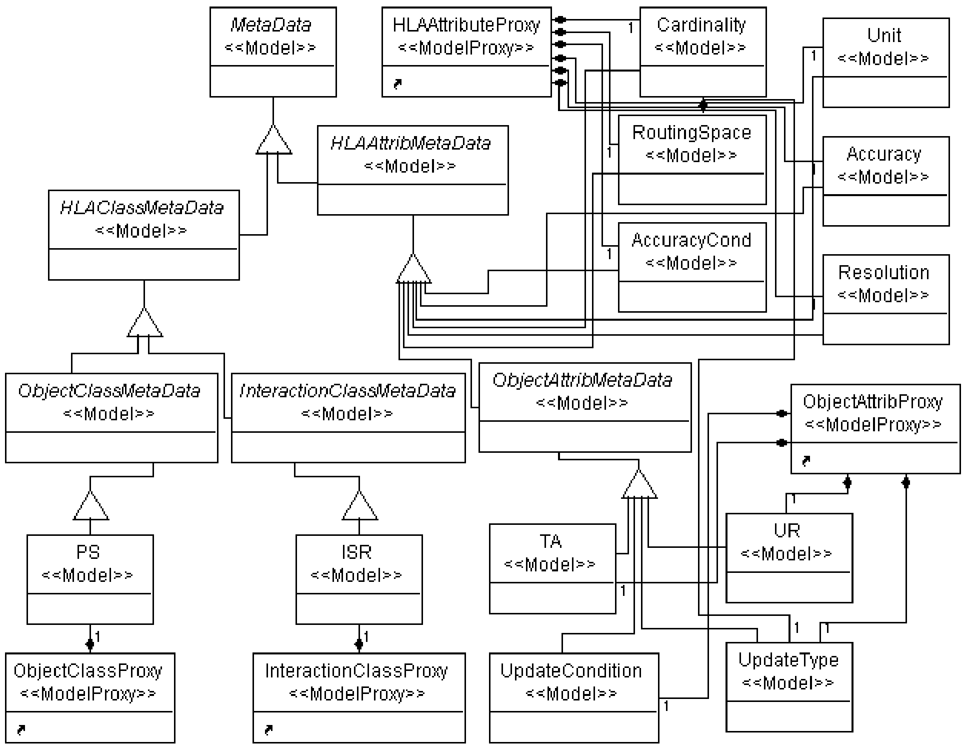


Figure 20. HLA Class and Attribute Structure

4.4. The Meta Data

Meta data are used in declaring various HLA borne properties of HLA classes and attributes. The MetaData model element is specified into HLAClassMetaData and HLAAttribMetaData submodels. HLAClassMetaData is further categorized into ObjectClassMetaData and InteractionClassMetaData, whose single submodels being PS and ISR, respectively. HLAAttribMetaData has six immediate leaf submodels, and an ObjectAttributeMetaData submodel that has four further submodels. Finally, all these leaf metadata elements are contained in HLA classes and attributes with cardinality 1 (see Figure 21). Note that the metadata hierarchy naturally follows the HLA class and attribute hierarchy. These metadata have distinguished data types provided by HLA.



MetaData <<Model>>
HLAAttributeProxy <<ModelProxy>>
Cardinality <<Model>>
Unit <<Model>>
RoutingSpace <<Model>>
Accuracy <<Model>>
HLAAttribMetaData <<Model>>
HLAClassMetaData <<Model>>
AccuracyCond <<Model>>
Resolution <<Model>>
ObjectClassMetaData <<Model>>
InteractionClassMetaData <<Model>>
ObjectAttribMetaData <<Model>>
ObjectAttribProxy <<ModelProxy>>
PS <<Model>>
ISR <<Model>>
TA <<Model>>
UR <<Model>>
ObjectClassProxy <<ModelProxy>>
InteractionClassProxy <<ModelProxy>>
UpdateCondition <<Model>>
UpdateType <<Model>>
1

Figure 21. HLA Meta Data Hierarchy

4.5. The Meta Data Types

This section defines the types of the meta data elements. As known from the programming language theory, every datum must have a type associated with it that resembles the set of all of the similar other data elements. This rule surely applies in our case. Figure 22 shows the top level meta data type hierarchy for HLA meta data. Each meta data type consists of a set of literals. Primitive literals are string, double, integer, and boolean literals, and there are infinitely many of them, except for the Boolean literals, which are true and false.

Please refer to [6] and Appendix C for the other details of the HLA-OMT Meta Model.

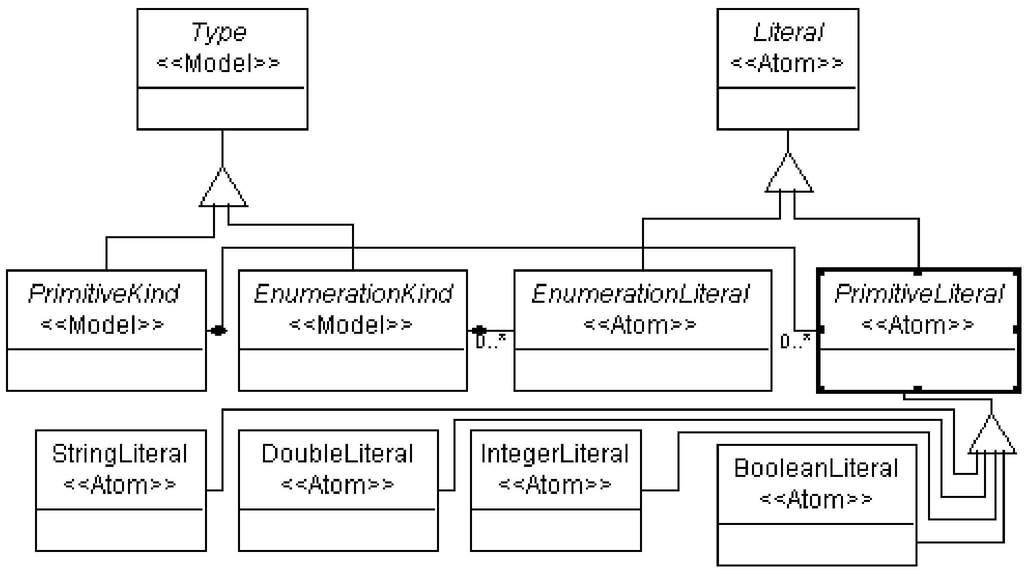


Figure 22. Top Level HLA Meta Data Type Hierarchy

CHAPTER 5

FA DATA MODEL TO HLA-OMT TRANSFORMATIONS

The data model transformation of FADMM2HOMM has little in graph-based pattern matching, but has much done in user code library. The DataTypes block contains rules for creating any non-standard basic, simple, enumerated, array and fixed record data types that the following rules will refer to. Figure 23 shows the top-level data model transformation block.

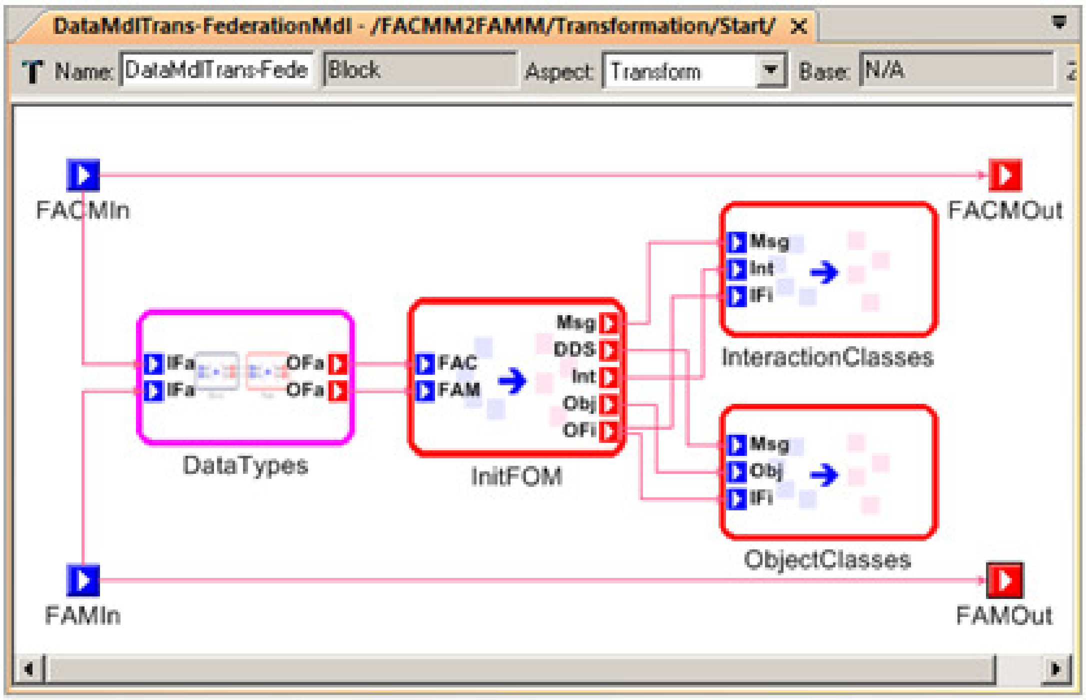


Figure 23. Top-level data model transformation block

The InitFOM rule is shown in Figure 24. The pattern matches if the FACM data model has a Messages and DurableDataStore folders (The match on FAM side about data types is trivial since those elements were hand-prepared in a previous rule. Once the input and output patterns match, an ObjectModels folder is created under HLAObjectModel parent folder. Moreover a FOM element is created with all of its child objects under ObjectModels. HLAInteractionRoot and HLAPrivilegeTo-DeleteObject are the two constant top level objects from which all of the interaction classes and object classes are extended [41]. A blue color of an association or a model entity indicates “CreateNew”, whereas black indicates “Bind” semantic. Input packets arrive at a rule trough blue input ports and leave out via the red output ports. AttributeMapping provides the user an opportunity to specify custom code (in C++) for applying more complicated and/or flexible operations on the bound objects using the UDM API [39]. These operations range from simple ones such as setting a new object’s name or position on the screen to sophisticated graph traversals, object creations or deletions.

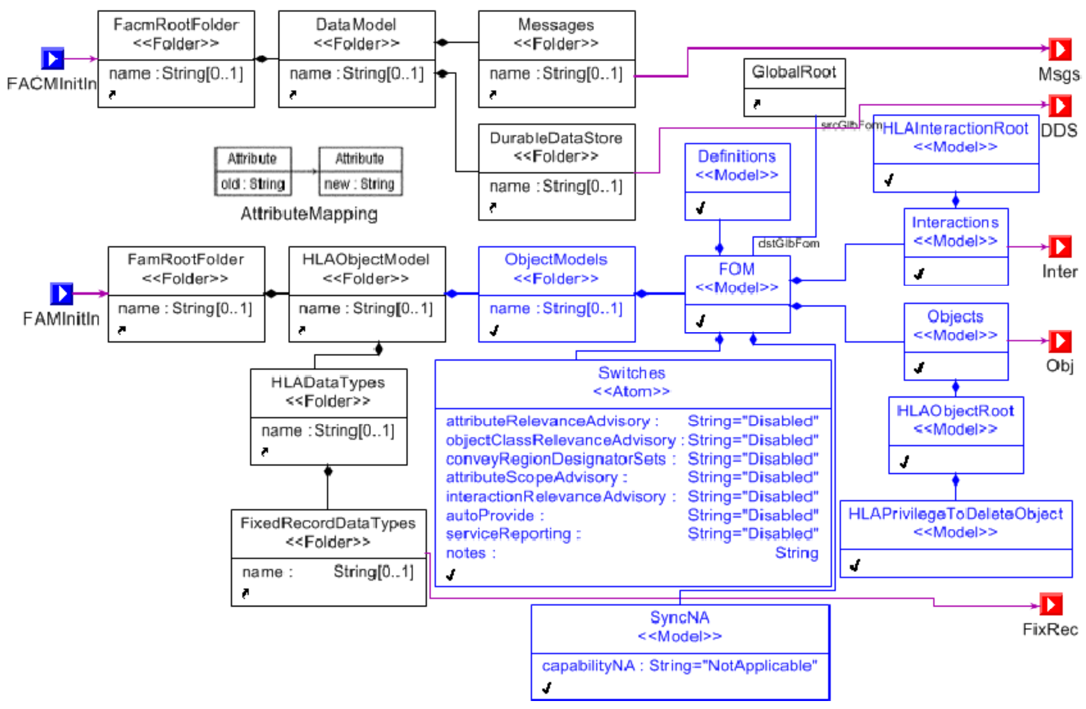

FacmRootFolder <<Folder>> name : String[0..1]
DataModel <<Folder>> name : String[0..1]
Messages <<Folder>> name : String[0..1]
DurableDataStore <<Folder>> name : String[0..1]
FACMInitIn
Attribute old : String
Attribute new : String
AttributeMapping
GlobalRoot
Msgs
DDS
Inter
Obj
FixRec
srcGlbFom
dstGlbFom
HLAInteractionRoot <<Model>>
Definitions <<Model>>
Interactions <<Model>>
Objects <<Model>>
HLAObjectRoot <<Model>>
HLAPrivilegeToDeleteObject <<Model>>
FOM <<Model>>
FAMInitIn
FamRootFolder <<Folder>> name : String[0..1]
HLAObjectModel <<Folder>> name : String[0..1]
ObjectModels <<Folder>> name : String[0..1]
HLADataTypes <<Folder>> name : String[0..1]
FixedRecordDataTypes <<Folder>> name : String[0..1]
Switches <<Atom>>
attributeRelevanceAdvisory : String="Disabled"
objectClassRelevanceAdvisory : String="Disabled"
conveyRegionDesignatorSets : String="Disabled"
attributeScopeAdvisory : String="Disabled"
interactionRelevanceAdvisory : String="Disabled"
autoProvide : String="Disabled"
serviceReporting : String="Disabled"
notes : String
SyncNA <<Model>>
capabilityNA : String="NotApplicable"

Figure 24. InitFOM rule

InteractionClasses rule, as seen in Figure 25, simply creates an interaction class per NonDurableMsg matched from the FACM. The sister rule ObjectClasses does a similar job as creating an object class per DurebleData. Remember that the non-durable messages and durable data define families of objects with varying size and complexity underneath them. The whole convolution of converting field artillery messages and durable data into HLA classes and objects is hindered under the AttributeMapping code, which makes a call to the user code library to perform a programmatic transformation using the UDM API [39]. We have developed the user code library as Microsoft Visual C++ project. It consists of hundreds of lines of code with a handful of methods. The source code of the library is presented in Appendix A. In addition to that, all other transformation rules defined for FADM to HLA-OMT transformation is presented in Appendix D.

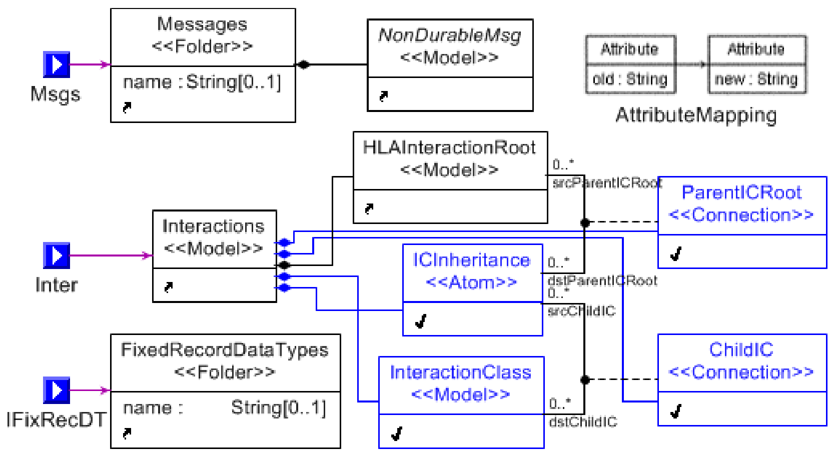


Figure 25. InteractionClasses rule

We have identified and implemented three distinct approaches, the third one having two varieties, to transforming a FACM message into an OMT class with attributes. The problem is that, a FA message is usually highly structured, with the possibility of having child objects being bound to their parents in varying cardinalities. This makes the situation even worse, because in order to represent such combination possibilities we would need many patterns, hence rules. For example if a structure can have n direct children each with a 0..1 cardinality, then we would need at least n parallel rules to cope with the source model. This is one of the reasons why we called for user code library support. In this case we only need one rule no matter how many children with whatever cardinality a message structure may have. The other reason is course performance gain; direct C++ executes much faster than first matching a graph and then calling the effector (Note that solving graph isomorphism problem is NP-hard, which is done in every pattern matching). In all of them an OMT class is created for a FA message or durable data (from now on, FA message and durable data will be used interchangeably when not stated explicitly otherwise). The difference lies in the construction of the attributes of the class. Whereas a FA message is highly structured, an OMT class has a fairly simple structure; it consists of only OMT attributes (which correspond to message parts) as well as some HLA-specific elements. The data model view of FADM Input model that will be transformed into HLA-OMT output model for the following examples is shown in Figure 26. Note that the Oid_W_Msg message which is a Call for Fire message in FA has been expanded in the view so as to be used in the following transformation cases as an example.

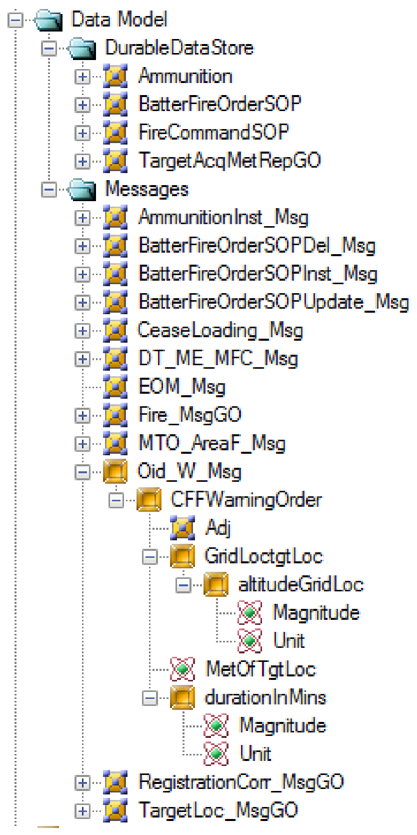


Figure 26. Example FADM Input Model

5.1. Message Structure Flattened into a Set of Plain Attributes

In this strategy, a field artillery message structure is transformed into a set of fully flattened OMT attributes within the OMT class. All of the attributes have primitive/simple data types and their names consist of a string of concatenated message structure names, separated by '_', from the leaf to the root node. This is best explained with a visual example. Figure 27 shows a typical FA message which is part of a call for fire request and Figure 28 shows the transformation result of it using this strategy.

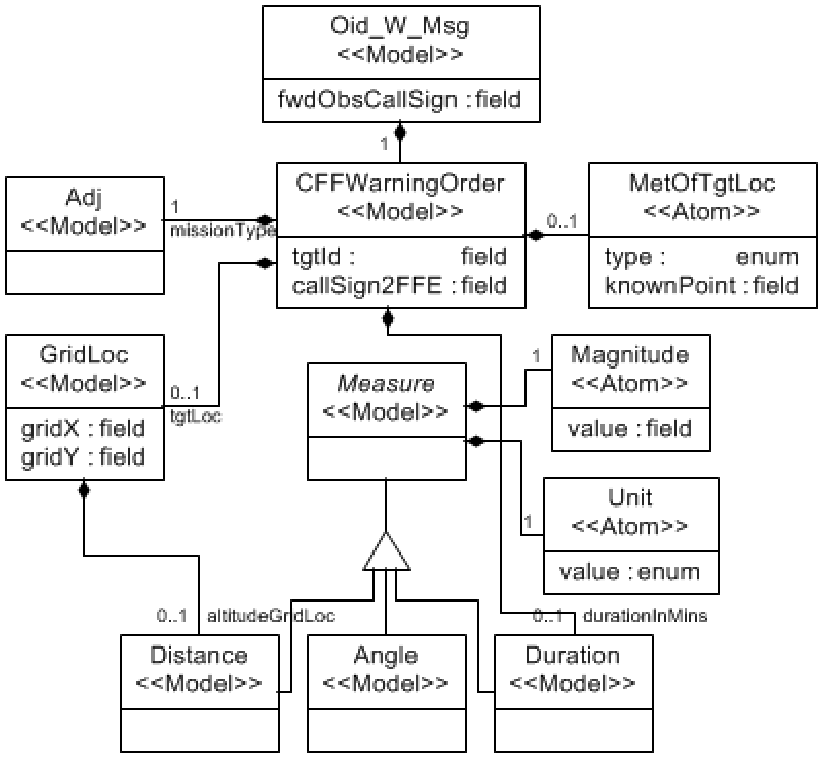
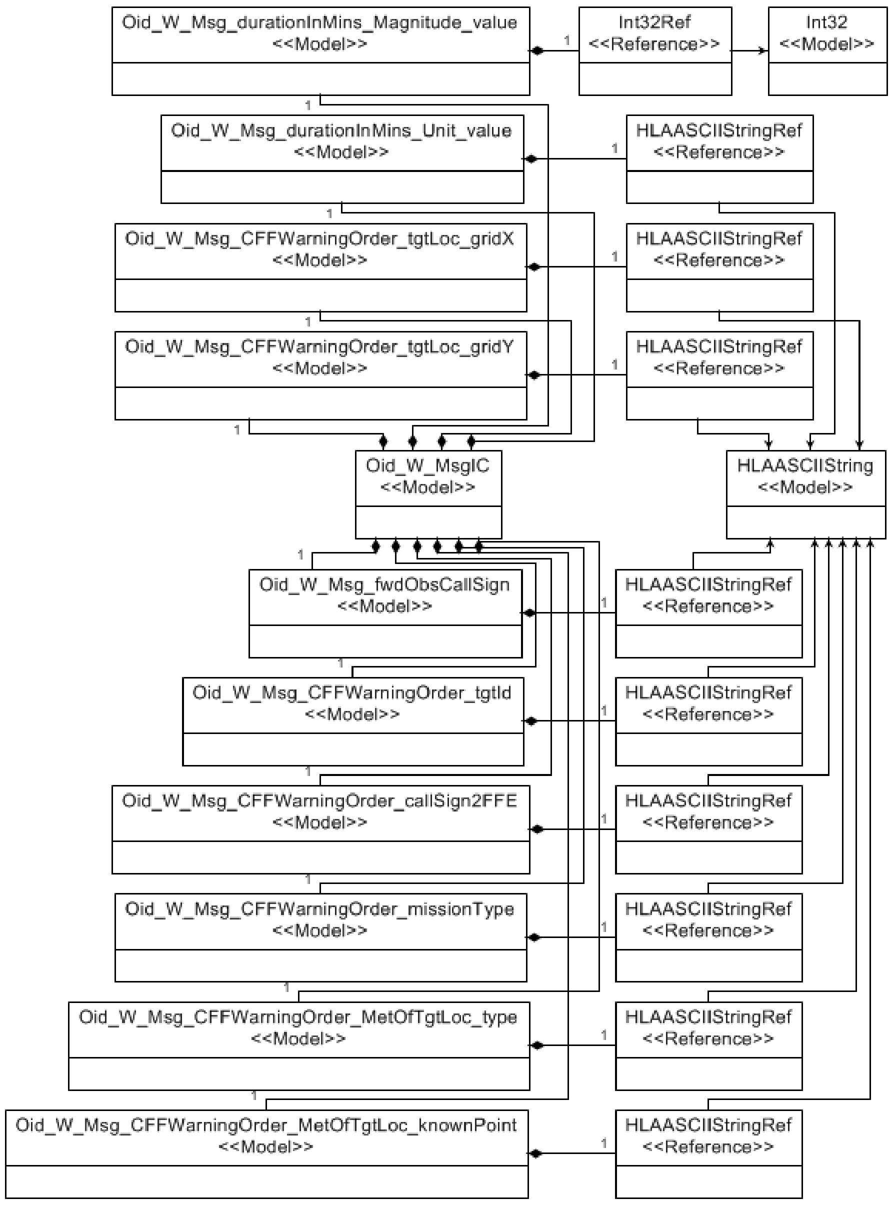


Figure 27. FA message of a call for fire request



Oid_W_Msg_durationInMins_Magnitude_value <<Model>>
Int32Ref <<Reference>>
Int32 <<Model>>
Oid_W_Msg_durationInMins_Unit_value <<Model>>
HLAASCIIStringRef <<Reference>>
Oid_W_Msg_CFFWarningOrder_tgtLoc_gridX <<Model>>
Oid_W_Msg_CFFWarningOrder_tgtLoc_gridY <<Model>>
Oid_W_MsgIC <<Model>>
HLAASCIIString <<Model>>
Oid_W_Msg_fwdObsCallSign <<Model>>
Oid_W_Msg_CFFWarningOrder_tgtId <<Model>>
Oid_W_Msg_CFFWarningOrder_callSign2FFE <<Model>>
Oid_W_Msg_CFFWarningOrder_missionType <<Model>>
Oid_W_Msg_CFFWarningOrder_MetOfTgtLoc_type <<Model>>
Oid_W_Msg_CFFWarningOrder_MetOfTgtLoc_knownPoint <<Model>>
1

Figure 28. Transformation result of call for fire by flattenning

5.2. Message Structure Mapped into one Attribute of FixedRecord

This schema is at the other extreme compared to the previous one. This time only one super attribute is created under the class having an HLA fixed record data type. The data type directly mimics the FA message structure. Members of the fixed record type are also made fix record types until primitive/simple types of the leaves are reached. In other words, the class and the attribute together form a fairly simple structure, but the whole complexity of the message structure is pushed inside the attribute's data type. The above example's transformation with this schema is shown in Figure 29.

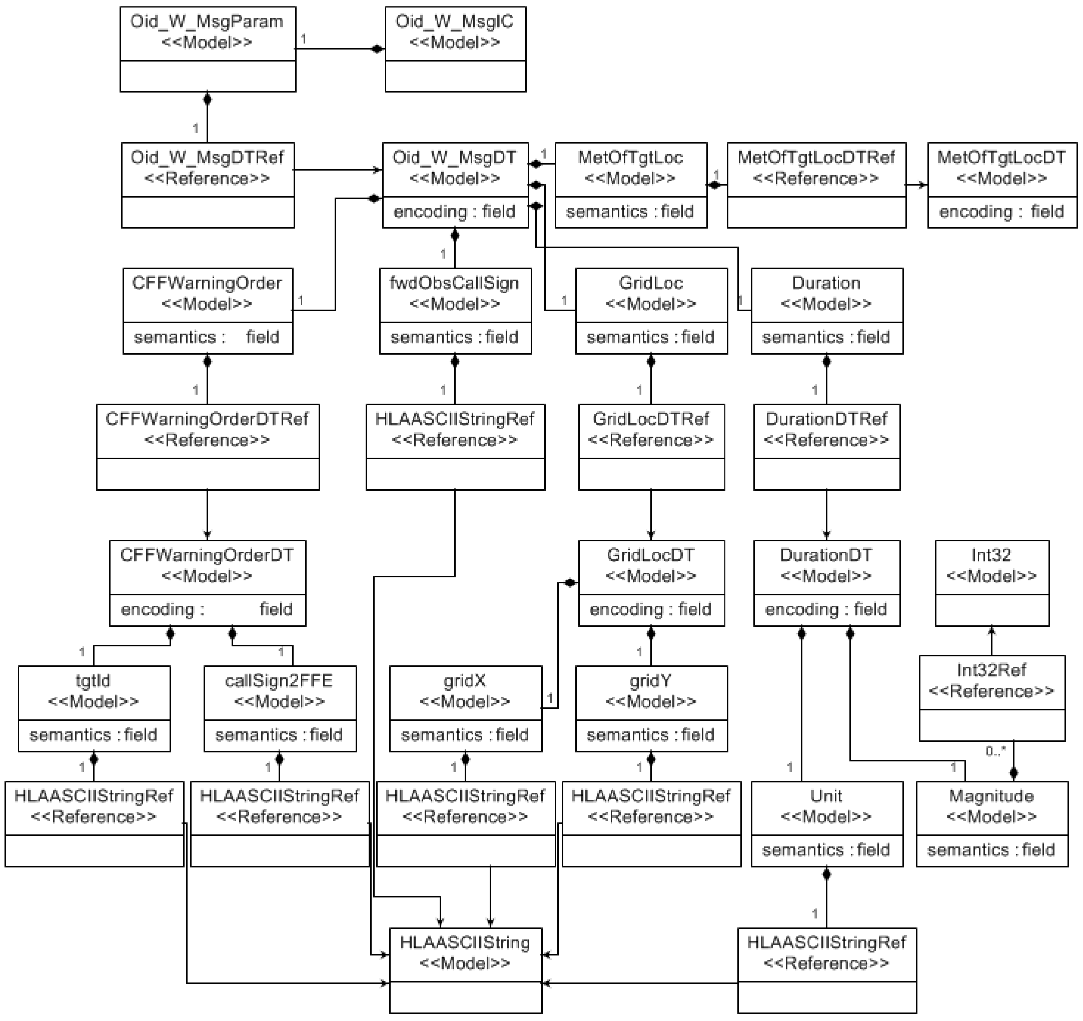

Oid_W_MsgParam <<Model>>
Oid_W_MsgIC <<Model>>
Oid_W_MsgDTRef <<Reference>>
Oid_W_MsgDT <<Model>> encoding : field
MetOfTgtLoc <<Model>> semantics : field
MetOfTgtLocDTRef <<Reference>>
MetOfTgtLocDT <<Model>> encoding : field
CFFWarningOrder <<Model>> semantics : field
fwdObsCallSign <<Model>> semantics : field
GridLoc <<Model>> semantics : field
Duration <<Model>> semantics : field
CFFWarningOrderDTRef <<Reference>>
HLAASCIIStringRef <<Reference>>
GridLocDTRef <<Reference>>
DurationDTRef <<Reference>>
CFFWarningOrderDT <<Model>> encoding : field
GridLocDT <<Model>> encoding : field
DurationDT <<Model>> encoding : field
Int32 <<Model>>
Int32Ref <<Reference>>
tgtId <<Model>> semantics : field
callSign2FFE <<Model>> semantics : field
gridX <<Model>> semantics : field
gridY <<Model>> semantics : field
Unit <<Model>> semantics : field
Magnitude <<Model>> semantics : field
0..*
HLAASCIIString <<Model>>
1

Figure 29. The FA message transformed by fixed record

5.3. A Hybrid Solution of Flattening and Fixed Recording

The third approach is actually a mixture of the two previous strategies. The field artillery message structure is transformed into an OMT attribute having a fixed record data type, within the OMT class. Each common message part that is reused (among multiple messages) is transformed into a fixed record type's field, having further a fixed record data type, mimicking the common message part. All of the other non-common parts of the message structure are transformed into fields of the fixed record type having appropriate primitive/simple types, with the field name mimicking the message structure hierarchy. The field name consists of a string of concatenated message structure types, separated by _, from the leaf to the root node as in the first scheme (i.e., message structure flattening).

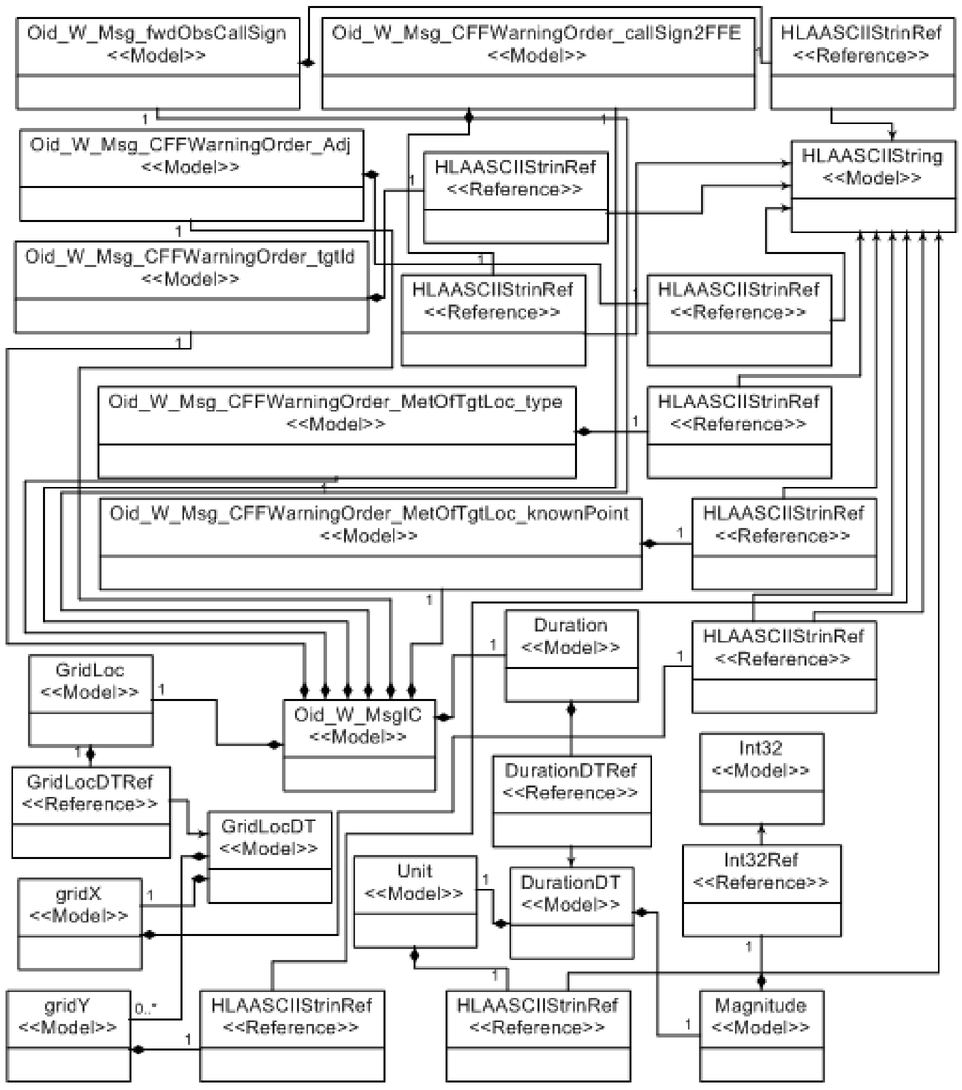

Oid_W_Msg_fwdObsCallSign <<Model>>
Oid_W_Msg_CFFWarningOrder_callSign2FFE <<Model>>
HLAASCIIStrinRef <<Reference>>
Oid_W_Msg_CFFWarningOrder_Adj <<Model>>
HLAASCIIString <<Model>>
Oid_W_Msg_CFFWarningOrder_tgtId <<Model>>
Oid_W_Msg_CFFWarningOrder_MetOfTgtLoc_type <<Model>>
Oid_W_Msg_CFFWarningOrder_MetOfTgtLoc_knownPoint <<Model>>
Duration <<Model>>
GridLoc <<Model>>
Oid_W_MsgIC <<Model>>
GridLocDTRef <<Reference>>
GridLocDT <<Model>>
DurationDTRef <<Reference>>
Int32 <<Model>>
Int32Ref <<Reference>>
Unit <<Model>>
DurationDT <<Model>>
gridX <<Model>>
gridY <<Model>>
Magnitude <<Model>>
1
0..*

Figure 30. The FA message transformed by hybrid solution

This approach can exhibit different levels of commitment to the two schemas that it combines. We identified two variations of this scheme. In the first variant, the low level fixed record type corresponding to the common message structure has a further continuing fixed record type inside, whereas in the second variant, the low level fixed record type corresponding to the common message structure has a set of flattened primitive/simple typed fields. In the beginning there was only the first variant, however, experience showed that it was not ideal in that if there were similar sub-fixed record types, these would quickly clutter the FixedRecordDataTypes folder. Later we naturally devised the second variant, which is free of this flaw, more compact and looks more legitimate. Many tests in transformations have revealed that the third schema, variant two is the best one among the proposed approaches to transforming FA messages to HLA classes.

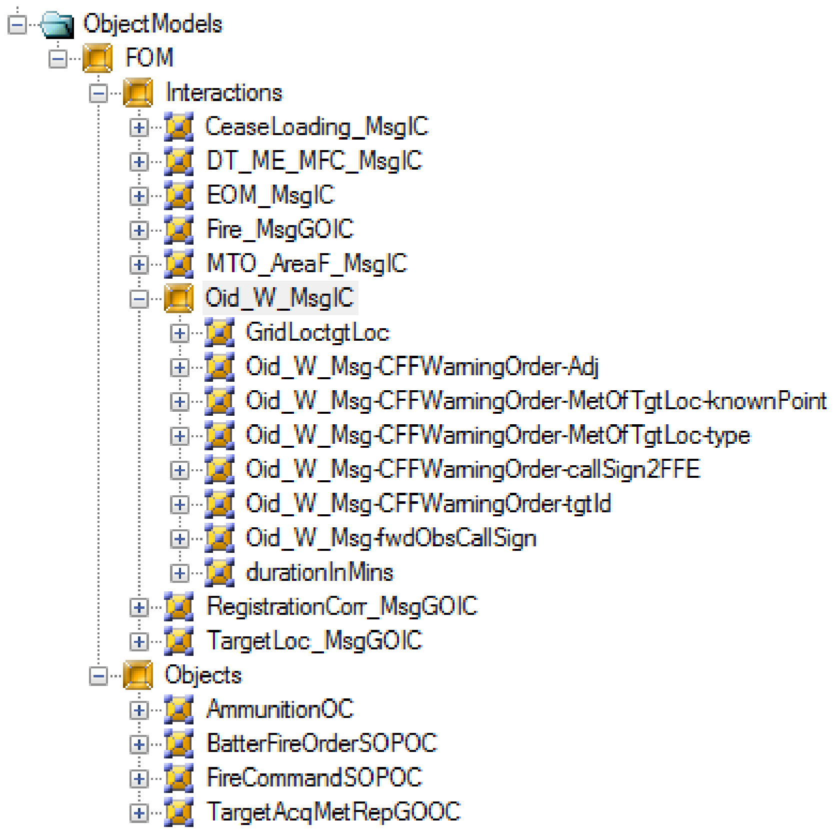

```
ObjectModels
  FOM
    Interactions
      CeaseLoading_MsgIC
      DT_ME_MFC_MsgIC
      EOM_MsgIC
      Fire_MsgGOIC
      MTO_AreaF_MsgIC
      Oid_W_MsgIC
        GridLoctgtLoc
        Oid_W_Msg-CFFWarningOrder-Adj
        Oid_W_Msg-CFFWarningOrder-MetOfTgtLoc-knownPoint
        Oid_W_Msg-CFFWarningOrder-MetOfTgtLoc-type
        Oid_W_Msg-CFFWarningOrder-callSign2FFE
        Oid_W_Msg-CFFWarningOrder-tgtId
        Oid_W_Msg-fwdObsCallSign
        durationInMins
      RegistrationCorr_MsgGOIC
      TargetLoc_MsgGOIC
    Objects
      AmmunitionOC
      BatterFireOrderSOPOC
      FireCommandSOPOC
      TargetAcqMetRepGOOC
```

Figure 31. Example HLA-OMT Output Model

To provide more liberty to the user, we established a configuration mechanism to select among the four available alternatives. The choice is made inside the source model. Then the selected strategy is extracted at run time and the flow is controlled accordingly. The above example Call for Fire Request message transformation with this schema using second variant is shown in Figure 30. In addition to that, the output HLA_OMT model of Figure 26 is presented in Figure 31.

CHAPTER 6

CONCLUSION

In this thesis we have developed a data model and a model transformer for Field Artillery (FA) observed fire domain. The most striking promise of the FA metamodel is the opening up the path to MDE. The model transformer converts a FA data model, which conforms to the FA metamodel into the object model part (i.e., OMT) of an HLA federation architecture model. The model transformer tool used is Graph Rewriting and Transformation (GReAT), a model transformation generator based on graph transformations, where the transformations are a set of explicitly sequenced elementary rewriting operations [9]. A preliminary version of the model transformation study is presented in [42].

We proposed three approaches in transforming an FA model into an HLA OMT model. In the first one is a field artillery message structure is transformed into a set of fully flattened OMT attributes within the OMT class. The advantage of this schema is its simplicity, where all of the attributes have primitive/simple data types and their names consist of a string of concatenated message structure names, separated by '_', from the leaf to the root node. On the other hand, there is no possibility of representing composite data types that could exhibit a one-to-one correspondence to the source model element (data) structure here.

The second approach, on the contrary to the first one, only creates one super attribute under the class having an HLA fixed record data type, whose parts are also fixed record types until primitive/simple types of the leaves are reached. In other words, the class and the attribute together form a fairly simple structure, but the whole complexity of the message structure is pushed inside the attribute's data type, which is structurally equivalent to the source model (data) structure. This structural resemblance could both be an advantage or disadvantage. If a prospective model executor that uses the target model supports complex data types, then a direct data type mapping could be possible. However, if does not, then there is even no change to make use of the produced target model at all.

The third proposal is a well-balanced compromise between the previous two extreme alternatives. This time the FA message structure is transformed into an OMT attribute having a fixed record data type, within the OMT class, where each reusable common message part is transformed into a fixed record type's field, having further a fixed record data type, just as in the second approach. Apart from that, all of the non-common parts are transformed into fields of the fixed record type having simple types, similar to the first approach. We proposed identified two variants of this third method. In one of them, the low level fixed record type corresponding to the common message structure has a further continuing fixed record type inside. In the other one, the low level fixed record type corresponding to the common message structure has a set of flattened simple typed fields.

The third approach combines the merits of the previous two approaches, providing a flexible, yet expressive enough model representation capability. On the other hand, exercising with the two variants of the third approach revealed that the first variant would result in redundant sub fixed-records for common, repeating model parts. The second variant eliminates this duplication problem, yielding a compact and legitimate target model for a source model.

REFERENCES

[1] Schmidt, D.C., "Model-Driven Engineering", IEEE Computer, vol.39 no.2, pp. 25-32, 2006.

[2] Agrawal A., Levendovszky T., Sprinkle J., Shi F., Karsai G., "Generative Programming via Graph Transformations in the Model Driven Architecture", Workshop on Generative Techniques in the Context of Model Driven Architecture, OOPSLA , Nov. 5, 2002, Seattle, WA.

[3] J. Greenfield, K. Short, S. Cook, S. Kent: "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools", Wiley, 2004.

[4] Gürkan Özhan, Halit Oguztüzün, and Pinar Evrensel, Modeling of Field Artillery Tasks with Live Sequence Charts, The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology 2008 5: 219-252.

[5] Topçu, O., Adak, M. and Oğuztüzün, H., "A Metamodel for Federation Architectures", ACM Transactions on Modeling and Computer Simulation (TOMACS), vol.18 no.3, art.10, July 2008

[6] D. Çetinkaya and H. Oğuztüzün, "A Metamodel for the HLA Object Model", 20th European Conference on Modeling and Simulation (Track on Modeling and Simulation Methodologies), Bonn, Germany, May 2006.

[7] Adak, M., Topçu, O. and Oğuztüzün, H., "Model-based code generation for HLA federates", Software—Practice & Experience, vol.40, issue 2 (February 2010)

[8] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle J. and Volgyesi, P., "The Generic Modeling Environment", In Proceedings of WISP'2001, Budapest, Hungary, May 2001.

[9] A. Agrawal, G. Karsai and A. Ledeczi., "An End-to-End Domain-Driven Software Development Framework", 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, California, October 2003.

[10] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, "On the Use of Graph Transformations for the Formal Specification of Model Interpreters", JUCS, November 2003.

[11] Jack Sheehan, Terry Prosser, Harry Conley, George Stone, Kevin Yentz and Janet Morrow, "Conceptual Models of the Mission Space (CMMS): Basic Concepts, Advanced Techniques, and Pragmatic Examples" 98 Spring Simulation Interoperability Workshop Papers

[12] Antonio De Nicola, Vandana Kabilan, Michele Missikoff and Vahid Mojtahed, "Practical Issues in Ontology Modeling: The Case of Defence Conceptual Modeling Framework-Ontology" 6th European Union Framework Programme.

[13] KAMA "A conceptual modeling tool for the mission space", Progress Report II, General Staff Presidency, Turkish Armed Forces, 2006.

[14] MIP. "JC3IEDM ratified as NATO STANAG 5255". Multilateral Interoperability Programme. http://www.mip-site.org/010_Public_Home_News.htm. Retrieved 2009-03-11.

[15] MIP. "Overview of the C2 Information Exchange Data Model (C2IEDM)". Multilateral Interoperability Programme, Greding, Germany, 20 November 2003.

[16] OMG. MDA Guide Version 1.0.1. omg. Object Management Group, 2003-06-01

[17] Milicev, D., "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments," IEEE Transaction on Software Engineering, Vol. 28, No. 4, April 2002,

[18] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h.: UMLAUT: an extendible UML transformation framework, in Proc. Automated Software Engineering, ASE'99, Florida, October 1999.

[19] Tony Clark, Andy Evans, Stuart Kent: The Metamodelling Language Calculus: Foundation Semantics for UML. FASE 2001.

[20] Tony Clark, Andy Evans, Stuart Kent: Engineering Modelling Languages: A Precise Meta-Modelling Approach. FASE 2002.

[21] OMG. MOF 2.0 Query/Views/Transformation RFP, 2002. Object Management Group document ad/2002-04-10

[22] Krzysztof Czarnecki and Simon Helsen , 'Classification of Model Transformation Approaches', OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture.

[23] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 'A Taxonomy of Model Transformation' , ENTCS2006

[24] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. IBM Systems J., 45(3), June 2006.

[25] Blostein D., Schürr A., "Computing with Graphs and Graph Rewriting", Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany.

[26] Rozenberg,G., "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations", Vol.1-2. World Scientific, Singapore, 1997

[27] Maggiolo-Schettini, A., Peron, A.: Semantics of Full Statecharts Based on Graph Rewriting, Springer LNCS 776, 1994, pp. 265--279.

[28] A. Schürr, "PROGRES for Beginners", Technical Report, Lehrstuhl für Informatik III, RWTH Aachen, Germany.

[29] Taentzer, G.: AGG: A Tool Enviroment for Algebraic Graph Transformation, in Proc. of Applications of Graph Transformation with Industrial Relevance, Kerkrade, The Netherlands, LNCS,Springer, 2000.

[30] Aydemir H., Term Project, MS 531 Course, METU Informatics Institute, 2004.

[31] Global Security Organization, http://www.globalsecurity.org/military/library/policy/army/fm/6-20/index.html, last accessed 30.06.09.

[32] Defense Modeling and Simulation Office-DMSO, "HLA Glossary", 1998.

[33] U.S. Department of Defense, "Draft Standard for Modeling and Simulation High Level Architecture", IEEE P1516, April 1998

[34] IEEE Std. 1516, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules, IEEE, 2000.

[35] G. Nordstrom, J. Sztipanovits, G. Karsai, A. Ledeczi: Metamodeling - Rapid Design and Evolution of Domain- Specific Modeling Environments, Proceedings of the IEEE ECBS'99 Conference, pp. 68-74, Nashville, TN, April, 1999.

[36] OMG, “UML 2 Unified Modeling Language: Infrastructure”, Object Management Group, February 2007.

[37] Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P. & Maroti, M. (2001). On Metamodel Composition. IEEE Conference on Control Applications 2001, Mexico City, Mexico.

[38] Agrawal A., Karsai G., Shi F., “A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations”, Technical report, (ISIS), Vanderbilt University, Nashville, TN, 2003

[39] A. Bakay, “The UDM Framework,” http://www.isis.vanderbilt.edu/Projects/mobies/. Last accessed 23.10.2009.

[40] Object Management Group, Object Constraint Language Specification, OMG Document formal/01-9-77. September 2001.

[41] IEEE 2000c. Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template Specification (IEEE 1516.2).

[42] Özhan, G. and Oguztüzün, H., “Model-Integrated Development of HLA-Based Field Artillery Simulation”, In Proceedings of the European Simulation Interoperability Workshop, (06E-SIW-027), Stockholm, Sweden, June19-22, 2006.

APPENDIX A

TRANSFORMATION SOURCE CODE

Source code for transforming field artillery messages into HLA classes.

```
#include <UdmBase.h>
#include <UdmUtil.h>
#include <cint_string.h>
#include "stack"
#include "..\Udm\HOMM.h"
#include "..\Udm\FADMM.h"

class DataTypeRetriever
{
public:
	//Given any matched node, dataTypaClassName and dataTypeName, this method
first tracks upwards along the node tree to the root and
	//then tracks down into to HLADataTypes node and then scans inside all of the
HLA data type folders under that, to find the data type
	//identified by the dataTypaClassName and dataTypeName.
	static HOMM::HLADataTypeRef_RefersTo_Base GetDataType(Udm::Object matchNode,
string dataTypeClassName, string dataTypeName){
		//Go up through the node tree to the root node.
		Udm::Object fadmRoot = matchNode;
		int depth = matchNode.depth_level() ;
		for (int i = depth ; i>0 ; i--){
			fadmRoot = fadmRoot.GetParent();
		}
		//Locate the specific DataType under one of the child folders (i.e.,
ArrayDataTypes, SimpleDataTypes, etc.) of the HLADataTypes folder
		HOMM::HLADataTypeRef_RefersTo_Base DataType;
		std::set<Udm::Object> fadmChilds = fadmRoot.GetChildObjects();
		//bool found = false;
		if (!fadmChilds.empty()){
			for( std::set<Udm::Object>::iterator iter= fadmChilds.begin();
iter!= fadmChilds.end() && (DataType.operator ==(NULL)) ; ++iter){
				Udm::Object childNode = (Udm::Object)(*iter);
				std::string childName = "Name";
				childNode.GetStrValue("name",childName);

				Uml::Class cls =  childNode.type();
				Udm::StringAttr clsNameStrAttr = cls.name();
				string clsNameStr = clsNameStrAttr.operator
std::string();

				if (clsNameStr == "HLADataTypes"){
					std::set<Udm::Object> fadmDTChilds =
childNode.GetChildObjects();
					for( std::set<Udm::Object>::iterator iter2=
fadmDTChilds.begin(); iter2!= fadmDTChilds.end() && (DataType.operator ==(NULL));
++iter2){
						Udm::Object childNode2 =
(Udm::Object)(*iter2);
						std::string childName2 = "Name";

	childNode2.GetStrValue("name",childName2);
```

```
                                                Uml::Class cls2 =  childNode2.type();
                                                Udm::StringAttr clsNameStrAttr2 =
cls2.name();
                                                string clsNameStr2 =
clsNameStrAttr2.operator std::string();

                                                if (clsNameStr2 == "ArrayDataTypes" ||
clsNameStr2 == "SimpleDataTypes" || clsNameStr2 == "FixedRecordDataTypes"){
                                                        std::set<Udm::Object>
fadmADTChilds = childNode2.GetChildObjects();
                                                        for(
std::set<Udm::Object>::iterator iter3= fadmADTChilds.begin(); iter3!=
fadmADTChilds.end() && DataType.operator ==(NULL); ++iter3){
                                                                Udm::Object childNode3 =
(Udm::Object)(*iter3);
                                                                std::string childName3 =
"Name";

        childNode3.GetStrValue("name",childName3);


                                                                Uml::Class cls3 =
childNode3.type();
                                                                Udm::StringAttr
clsNameStrAttr3 = cls3.name();
                                                                string clsNameStr3 =
clsNameStrAttr3.operator std::string();

                                                                if
(clsNameStr3==dataTypeClassName && childName3 == dataTypeName){
                                                                        DataType =
HOMM::HLADataTypeRef_RefersTo_Base::Cast(childNode3);

                                                                        HOMM::MgaObject
mgaObj = HOMM::MgaObject::Cast(childNode3);
                                                                        string mgaName =
mgaObj.name().operator std::string();
                                                                        //found = true;
                                                                        //break;
                                                                        return DataType;
                                                                }

                                                        }
                                                }

                                        }
                                }
                        }
                }
                return DataType;
        }
};
enum HLAObjectType {
        OBJECT,
        INTERACTION
};


class ModelTransUtils
{
protected:
        static HOMM::HLADataTypeRef_RefersTo_Base GetHLAASCIIStringDT(Udm::Object
matchNode){
                return
DataTypeRetriever::GetDataType(matchNode,"HLAArrayData","HLAASCIIString");;
        }
        static HOMM::HLADataTypeRef_RefersTo_Base GetInt32DT(Udm::Object matchNode){

                return
DataTypeRetriever::GetDataType(matchNode,"HLASimpleData","Int32");
        }
        static HOMM::HLADataTypeRef_RefersTo_Base GetReal32DT(Udm::Object matchNode){
                return
DataTypeRetriever::GetDataType(matchNode,"HLASimpleData","Real32");
        }



        //Compose the node (i.e., ICParameter) name by concatenating all ancestor
names up until the root node, delimited by -
        static string GetAbsoluteNameForNode(Udm::Object node, Udm::Object root){

                return GetAbsoluteNameForNode(node,root,false);
        }
        //Compose the node (i.e., ICParameter) name by concatenating all ancestor
names up until the root node, delimited by -
        static string GetAbsoluteNameForNode(Udm::Object node, Udm::Object root, bool
useNodeTypeInsteadOfName){
                std::string nodeName = "";
                std::string absoluteName = "";
                if (useNodeTypeInsteadOfName){
                        Uml::Class cls1st =  node.type();
                        Udm::StringAttr clsNameStrAttr1st = cls1st.name();
                        absoluteName = clsNameStrAttr1st.operator std::string();
                        while (node.operator != (root)){
                                node = node.GetParent();
                                //NOTE!: If we use the existing variables
above(cls1st,clsNameStrAttr1st), then node's parent type names get mixed up!!!
                                //Even the same var names for above and below would be
OK, but new declaration is a MUST. This costed me 6 hours!!!
                                Uml::Class cls =  node.type();
                                Udm::StringAttr clsNameStrAttr = cls.name();
                                nodeName = clsNameStrAttr.operator std::string();
                                absoluteName = nodeName+"-"+absoluteName;
                        }
                }else{
                        node.GetStrValue("name",absoluteName);//Initially assign the
"name" attribute to absoluteName variable
                        while (node.operator != (root)){
                                node = node.GetParent();
                                node.GetStrValue("name",nodeName);
                                absoluteName = nodeName+"-"+absoluteName;
                        }
                }
                return absoluteName;
        }

        //Push the child nodes of a node to the given stack
        static void PushNodeChildrenToStack(std::stack<Udm::Object>&
nodeStack,Udm::Object node){
                if (/*nodeStack.operator ==(NULL) || */node.operator ==(NULL))
return;
                std::set<Udm::Object> childs = node.GetChildObjects();
                if (!childs.empty()){
                        for( std::set<Udm::Object>::iterator iter= childs.begin();
iter!= childs.end(); ++iter){
                                Udm::Object child = (Udm::Object)(*iter);
                                if (child.operator !=(NULL)){
                                        nodeStack.push(child);
                                }
                        }
                }
        }

        static void CreateAttributesForNode(Udm::Object node, Udm::Object root,
/*HOMM::InteractionClass&*/ Udm::Object& InteractionClass, HLAObjectType
hlaObjType){
                //First get the HLAAsciiString and Int32 data type objects at hand to
use later in Parameter data type ref constructions
                HOMM::HLADataTypeRef_RefersTo_Base HLAAsciiString =
GetHLAASCIIStringDT(InteractionClass);
```

```
		HOMM::HLADataTypeRef_RefersTo_Base Int32 =
GetInt32DT(InteractionClass);
		HOMM::HLADataTypeRef_RefersTo_Base Real32 =
GetReal32DT(InteractionClass);

		//Compose the node (i.e., ICParameter) name by concatenating all
ancestor names up until the root node, delimited by -
		std::string absoluteName = GetAbsoluteNameForNode(node,root);

		Uml::Class cls =  node.type();
		Udm::StringAttr clsNameStrAttr = cls.name();
		string clsNameStr = clsNameStrAttr.operator std::string();
		Udm::ChildrenAttr<::Uml::Attribute> childrenAttr = cls.attributes();
		std::vector<::Uml::Attribute> attrs = childrenAttr.operator
std::vector<::Uml::Attribute>();
		//Create a parameter for every attribute of a node. Node being branch
or leaf does not matter. Note that if the node has attributes,
		// then no parameter is created excessively for the node itself, but
only for its attributes.
		for( std::vector<::Uml::Attribute>::iterator iter= attrs.begin();
iter!= attrs.end(); ++iter){
			::Uml::Attribute childAttr = (::Uml::Attribute)(*iter);
			Udm::StringAttr udmStrAttrName = childAttr.name();
			Udm::StringAttr udmStrAttrType = childAttr.type();

			string attrNameStr = udmStrAttrName.operator std::string();
			string attrTypeStr = udmStrAttrType.operator std::string();

			HOMM::OMTAttribute omtAttribute;

			//Code repetition of below:
			if (hlaObjType==INTERACTION){
				//HOMM::Parameter ICParameter=
HOMM::Parameter::Create(InteractionClass);
				omtAttribute =
HOMM::Parameter::Create(InteractionClass);
			}else if (hlaObjType==OBJECT){
				omtAttribute =
HOMM::Attribute::Create(InteractionClass);
			}
			omtAttribute.name()=absoluteName+"-"+attrNameStr;
			//Set the dataType for ICParameter.
			//IMPORTANT NOTE/FACT: enumerated attribute types in FADMM are
treated as String by the Udm API!!! (e.g. CFF tgt type comes as String typed in C++
code here!) This simplifies our job :-)
			HOMM::HLADataTypeRef dataTypeRef =
HOMM::HLADataTypeRef::Create(omtAttribute);

			std::string dtRefName = "DataType";
			if (attrTypeStr=="String"){
				HLAAsciiString.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
				dataTypeRef.ref() = HLAAsciiString;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
			}
			else if (attrTypeStr=="Integer"){
				Int32.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
				dataTypeRef.ref() = Int32;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
			}else if (attrTypeStr=="Double"||attrTypeStr=="Real"){
				Real32.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
				dataTypeRef.ref() = Real32;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
			}
		}
		//Create a parameter for nodes that have NO attributes and that are 
leaf nodes! (such as marker nodes like Shot, Fire, etc.)
		if (attrs.size()==0 && node.GetChildObjects().empty()){
			//HOMM::Parameter ICParameter= 
HOMM::Parameter::Create(InteractionClass);
			HOMM::OMTAttribute omtAttribute;
			if (hlaObjType==INTERACTION){
				//HOMM::Parameter ICParameter= 
HOMM::Parameter::Create(InteractionClass);
				omtAttribute = 
HOMM::Parameter::Create(InteractionClass);
			}else if (hlaObjType==OBJECT){
				omtAttribute = 
HOMM::Attribute::Create(InteractionClass);
			}

			omtAttribute.name()=absoluteName;//
(std::string)NonDurableMsg.name()+ "_" +  objName;
			HOMM::HLADataTypeRef dataTypeRef = 
HOMM::HLADataTypeRef::Create(omtAttribute);
			//Set the dataType for ICParameter. Give it a default datatype 
of HLAAsciiString
			std::string dtRefName = "DataType";
			HLAAsciiString.GetStrValue("name",dtRefName);
			dataTypeRef.name()=dtRefName+"Ref";
			dataTypeRef.ref() = HLAAsciiString;//new 
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr< 
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
		}
	}


	//Note: on durable data part, we do not transform DurableData_Msg, as similar 
to NonDurableMsg, but DurableData, which is part of DurableData_Msg
	static void TransformFAMessage2OMTClass(Udm::Object& faMsg, Udm::Object& 
omtClass, HLAObjectType hlaObjectType){
		std::stack<Udm::Object>& nodeStack = std::stack<Udm::Object>();//This 
is an initialization stmt in C++. If
		//Initially push the root (i.e., faMsg) node.
		nodeStack.push(faMsg);
		Udm::Object currentNode;
		while (!nodeStack.empty()){
			currentNode = nodeStack.top();
			nodeStack.pop();
			CreateAttributesForNode(currentNode, faMsg, 
omtClass,hlaObjectType);
			PushNodeChildrenToStack(nodeStack, currentNode);
		}
	}
	//-------------------------------3rd approach type: Convert Msgs into Full 
HLAFixedRecord Types

	static void TransformFAMessage2OMTClass_FullFixRec(Udm::Object& FAMsg, 
Udm::Object node, HOMM::HLAFixedRecordData& HLAFixedRecordData, 
HOMM::FixedRecordDataTypes& DataTypes){
		//By giving UseOnlyTypeInNaming false, we use 
absolutePathToRoot+attrName in new DataType names. By giving true we only use 
attrType in new DataType names
		TransformFAMessage2OMTClass_FullFixRec(FAMsg, node, 
HLAFixedRecordData, DataTypes,false);
	}
	static void TransformFAMessage2OMTClass_FullFixRec(Udm::Object& FAMsg, 
Udm::Object node, HOMM::HLAFixedRecordData& HLAFixedRecordData, 
HOMM::FixedRecordDataTypes& DataTypes, bool UseOnlyTypeInNaming){
		//CreateAttributesForNode_FullFixRec(currentNode, faMsg, 
omtClass,hlaObjectType);
		//First get the HLAAsciiString and Int32 data type objects at hand to 
use later in Parameter data type ref constructions
		HOMM::HLADataTypeRef_RefersTo_Base HLAAsciiString = 
GetHLAASCIIStringDT(DataTypes);
		HOMM::HLADataTypeRef_RefersTo_Base Int32 = GetInt32DT(DataTypes);
		HOMM::HLADataTypeRef_RefersTo_Base Real32 = GetReal32DT(DataTypes);

		//Compose the node (i.e., ICParameter) name by concatenating all
ancestor names up until the root node, delimited by -
		std::string absoluteName = GetAbsoluteNameForNode(node,FAMsg);

		Uml::Class cls =  node.type();
		Udm::StringAttr clsNameStrAttr = cls.name();
		string clsNameStr = clsNameStrAttr.operator std::string();
		Udm::ChildrenAttr<::Uml::Attribute> childrenAttr = cls.attributes();
		std::vector<::Uml::Attribute> attrs = childrenAttr.operator
std::vector<::Uml::Attribute>();
		//Create a Field on the HLAFixedRecordData for every attribute of the
node (that corresponds to the FixedRecord). These fields would have
SimpleDataTypereferences.

		//Create a parameter for every attribute of a node. Node being branch
or leaf does not matter. Note that if the node has attributes,
		// then no parameter is created excessively for the node itself, but
only for its attributes.
		for( std::vector<::Uml::Attribute>::iterator iter= attrs.begin();
iter!= attrs.end(); ++iter){
			::Uml::Attribute childAttr = (::Uml::Attribute)(*iter);
			Udm::StringAttr udmStrAttrName = childAttr.name();
			Udm::StringAttr udmStrAttrType = childAttr.type();

			string attrNameStr = udmStrAttrName.operator std::string();
			string attrTypeStr = udmStrAttrType.operator std::string();

			HOMM::Field fixedRecField =
HOMM::Field::Create(HLAFixedRecordData);
			if (UseOnlyTypeInNaming){
				//NOTE: we used attrNameStr instead of attrTypeStr
when UseOnlyTypeInNaming==true, as opposed to below, because for primitive types
attrTypeStr would result in String, Integer, etc, which is not desired.
				fixedRecField.name()=attrNameStr;
			}else{
				fixedRecField.name()=absoluteName+"-"+attrNameStr;
			}
			//IMPORTANT NOTE/FACT: enumerated attribute types in FADMM are
treated as String by the Udm API!!! (e.g. CFF tgt type comes as String typed in C++
code here!) This simplifies our job :-)
			HOMM::HLADataTypeRef dataTypeRef =
HOMM::HLADataTypeRef::Create(fixedRecField);

			std::string dtRefName = "DataType";
			if (attrTypeStr=="String"){
				HLAAsciiString.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
				dataTypeRef.ref() = HLAAsciiString;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
			}
			else if (attrTypeStr=="Integer"){
				Int32.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
				dataTypeRef.ref() = Int32;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
			}else if (attrTypeStr=="Double"){
				Real32.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
				dataTypeRef.ref() = Real32;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
			}

		}

		//		//Create a parameter for nodes that have NO attributes
and that are leaf nodes! (such as marker nodes like Shot, Fire, etc.)
```

```
		//if (attrs.size()==0 && node.GetChildObjects().empty()){
		//	//HOMM::Parameter ICParameter=
HOMM::Parameter::Create(InteractionClass);
		//	HOMM::OMTAttribute omtAttribute;
		//	if (hlaObjType==INTERACTION){
		//		//HOMM::Parameter ICParameter=
HOMM::Parameter::Create(InteractionClass);
		//		omtAttribute =
HOMM::Parameter::Create(InteractionClass);
		//	}else if (hlaObjType==OBJECT){
		//		omtAttribute =
HOMM::Attribute::Create(InteractionClass);
		//	}

		//	omtAttribute.name()=absoluteName;//
(std::string)NonDurableMsg.name()+ "_" +  objName;
		//	HOMM::HLADataTypeRef dataTypeRef =
HOMM::HLADataTypeRef::Create(omtAttribute);
		//	//Set the dataType for ICParameter. Give it a default datatype
of HLAAsciiString
		//	std::string dtRefName = "DataType";
		//	HLAAsciiString.GetStrValue("name",dtRefName);
		//	dataTypeRef.name()=dtRefName+"Ref";
		//	dataTypeRef.ref() = HLAAsciiString;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
		//}

		//If the node has further children (i.e., structure), then create a
FixedRecordType for every child, push those children on the stack, and let the loop
repeat the same process of creating simple types for that every child node's
attributes and fixedrec types for further children of the child node
		std::set<Udm::Object> childs = node.GetChildObjects();
		if (!childs.empty()){
			for( std::set<Udm::Object>::iterator iter= childs.begin();
iter!= childs.end(); ++iter){
				Udm::Object child = (Udm::Object)(*iter);
				if (child.operator !=(NULL)){
					FADMM::MgaObject mgaChild =
FADMM::MgaObject::Cast(child);
					Udm::StringAttr udmStrAttrMgaChild =
mgaChild.name();
					Uml::Class umlClsMgaChild = mgaChild.type();
					string mgaChildNameStr =
udmStrAttrMgaChild.operator std::string();
					string attrTypeStr =
(umlClsMgaChild.name()).operator std::string();

					string namePrefix;
					if (UseOnlyTypeInNaming){
						namePrefix=attrTypeStr;
					}else{
						namePrefix=absoluteName+"-
"+mgaChildNameStr;
					}

					//Create a Field for the child object under the
HLAFixedRecordData of the parent object, that will contain a HLADataTypeRef to a
further child HLAFixedRecordData
					HOMM::Field fixedRecField =
HOMM::Field::Create(HLAFixedRecordData);
					fixedRecField.name()=namePrefix;
					//Create a separate HLAFixedRecordData for the
child, to be passed into the recursive call for further building up the data
structure for the child's descendants.
					HOMM::HLAFixedRecordData fixedRecordData =
HOMM::HLAFixedRecordData::Create(DataTypes);
					fixedRecordData.name()=namePrefix+"-
FixRecType";
```

```
                                    //Create a HLADataTypeRef under the Field of
the child object and establish the reference association to the would be later
constructed HLAFixedRecordData of the child object
                                    HOMM::HLADataTypeRef fixedRecordDataRef =
HOMM::HLADataTypeRef::Create(fixedRecField);
                                    fixedRecordDataRef.name()=namePrefix+"-
FixRecTypeRef";
                                    fixedRecordDataRef.ref() = fixedRecordData;
                                    //Call the method recursively with the child
object together with its HLAFixedRecordData
                                    TransformFAMessage2OMTClass_FullFixRec(FAMsg,
child, fixedRecordData, DataTypes,UseOnlyTypeInNaming);
                              }
                        }
                  }

            }



            //-------------------------------------------------------------------------
------------
            // --------------------------- P U B L I C   M E T H O D S -------------------
------------
public:

            static void TransformNonDurableMsg2InteractionCls(FADMM::NonDurableMsg&
NonDurableMsg, HOMM::InteractionClass& InteractionClass){
                  InteractionClass.name()=(std::string)NonDurableMsg.name()+"IC";

            TransformFAMessage2OMTClass(NonDurableMsg,InteractionClass,INTERACTION);
            }

            static void TransformDurableData2ObjectCls(FADMM::DurableData& DurableData,
HOMM::ObjectClass& ObjectClass){
                  ObjectClass.name()=(std::string)DurableData.name()+"OC";
                  TransformFAMessage2OMTClass(DurableData,ObjectClass,OBJECT);
            }


            static void
TransformNonDurableMsg2InteractionCls_FullFixRec(FADMM::NonDurableMsg&
NonDurableMsg, HOMM::HLAFixedRecordData& HLAFixedRecordData,
HOMM::FixedRecordDataTypes& DataTypes){

            TransformFAMessage2OMTClass_FullFixRec(NonDurableMsg,NonDurableMsg,HLAFixedRe
cordData,DataTypes);
            }

            static void TransformDurableData2ObjectCls_FullFixRec(FADMM::DurableData&
DurableData, HOMM::HLAFixedRecordData& HLAFixedRecordData,
HOMM::FixedRecordDataTypes& DataTypes){

            TransformFAMessage2OMTClass_FullFixRec(DurableData,DurableData,HLAFixedRecord
Data,DataTypes);
            }


            static void TransformDurableData2ObjectCls_Hybrid(FADMM::DurableData&
DurableData, HOMM::ObjectClass& ObjectClass,HOMM::FixedRecordDataTypes& DataTypes){
//                InteractionClass.name()=(std::string)NonDurableMsg.name()+"IC";

            TransformFAMessage2OMTClass_Hybrid(DurableData,ObjectClass,OBJECT,DataTypes);
            }

            static void
TransformNonDurableMsg2InteractionCls_Hybrid(FADMM::NonDurableMsg& NonDurableMsg,
HOMM::InteractionClass& InteractionClass,HOMM::FixedRecordDataTypes& DataTypes){
            //      InteractionClass.name()=(std::string)NonDurableMsg.name()+"IC";
```

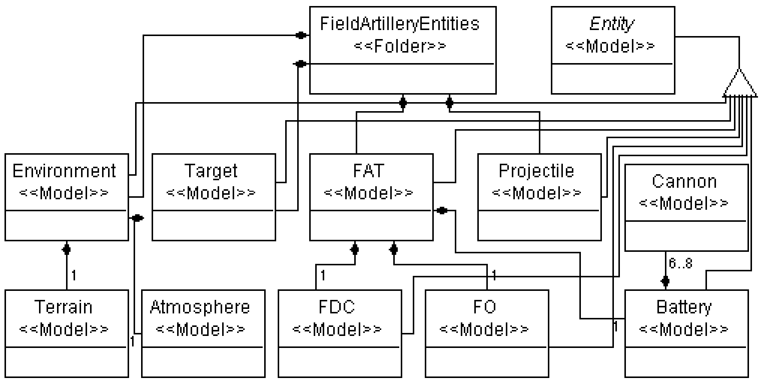
```
        TransformFAMessage2OMTClass_Hybrid(NonDurableMsg,InteractionClass,INTERACTION
,DataTypes);
        }

        //static std::set<std::string>& commonFadmElements;

        static bool IsCommonUtilityElement(std::string fadmNodeType,
std::set<std::string>& commonFadmElements){
                if (!commonFadmElements.empty()){
                        for( std::set<std::string>::iterator iter=
commonFadmElements.begin(); iter!= commonFadmElements.end() /*&& (DataType.operator
==(NULL))*/ ; ++iter){
                                if (fadmNodeType == (*iter)){
                                        return true;
                                }
                        }
                }
                return false;
        }

        static void InitializeCommonElements(std::set<std::string>&
commonFadmElements){
                ///*std::set<std::string>& */commonFadmElements =
std::set<std::string>();
                //std::stack<Udm::Object>& nodeStack =
std::stack<Udm::Object>();//This is an initialization stmt in C++. If
                commonFadmElements.insert("PolarLoc");
                commonFadmElements.insert("GridLoc");
                commonFadmElements.insert("ShiftKPLoc");
                commonFadmElements.insert("Duration");
                commonFadmElements.insert("Angle");
                commonFadmElements.insert("Distance");
                commonFadmElements.insert("Speed");
                commonFadmElements.insert("Pressure");
                commonFadmElements.insert("Temperature");
                commonFadmElements.insert("DateTime");
                commonFadmElements.insert("LateralShiftDist");
                commonFadmElements.insert("VerticalShiftDist");
                commonFadmElements.insert("RangeShiftDist");
                commonFadmElements.insert("HorizontalDir");
                commonFadmElements.insert("VerticalDir");
                commonFadmElements.insert("RangeDir");
                commonFadmElements.insert("SheafDir");
                commonFadmElements.insert("CardinalDir");
                commonFadmElements.insert("LateralShiftAng");
                commonFadmElements.insert("VerticalShiftAng");
                commonFadmElements.insert("SheafShiftAng");
        }

        //Note: on durable data part, we do not transform DurableData_Msg, as similar
to NonDurableMsg, but DurableData, which is part of DurableData_Msg
        static void TransformFAMessage2OMTClass_Hybrid(Udm::Object& faMsg,
Udm::Object& omtClass, HLAObjectType hlaObjectType,HOMM::FixedRecordDataTypes&
DataTypes){

                std::set<std::string>& commonFadmElements=std::set<std::string>();

                InitializeCommonElements(commonFadmElements);

                std::stack<Udm::Object>& nodeStack = std::stack<Udm::Object>();//This
is an initialization stmt in C++. If
                //Initially push the root (i.e., faMsg) node.
                nodeStack.push(faMsg);
                Udm::Object currentNode;
                while (!nodeStack.empty()){
                        currentNode = nodeStack.top();
                        nodeStack.pop();

                        Uml::Class cls = currentNode.type();
                        Udm::StringAttr clsNameStrAttr = cls.name();
                        string clsNameStr = clsNameStrAttr.operator std::string();

                        //Else part is the same as in TransformFAMessage2OMTClass. The
if part is new; it creates FixedRecTypes for common elements (only once per type!)
                        if (IsCommonUtilityElement(clsNameStr,commonFadmElements)){

                                FADMM::MgaObject currentNodeMga =
FADMM::MgaObject::Cast(currentNode);

                                HOMM::OMTAttribute omtAttribute;
                                if (hlaObjectType==INTERACTION){
                                        //HOMM::Parameter ICParameter=
HOMM::Parameter::Create(InteractionClass);
                                        omtAttribute =
HOMM::Parameter::Create(omtClass);
                                }else if (hlaObjectType==OBJECT){
                                        omtAttribute =
HOMM::Attribute::Create(omtClass);
                                }

        omtAttribute.name()=(std::string)currentNodeMga.name();//absoluteName+"-
"+attrNameStr;

                                HOMM::HLADataTypeRef& HLADataTypeRef=
HOMM::HLADataTypeRef::Create( omtAttribute);

        HLADataTypeRef.name()=(std::string)currentNodeMga.name()+"DTRef";


                                //HOMM::HLAFixedRecordData& HLAFixedRecordData=
HOMM::HLAFixedRecordData::Create(DataTypes);

                                //Check if the FixedRecord data type already exists,
and if so, get it
                                string fixRecDTName = clsNameStr;
                                HOMM::HLADataTypeRef_RefersTo_Base baseDT =
DataTypeRetriever::GetDataType(DataTypes,"HLAFixedRecordData",fixRecDTName);
                                HOMM::HLAFixedRecordData HLAFixedRecordData =
HOMM::HLAFixedRecordData::Cast(baseDT);
                                // If the FixedRecord data type does not exist, then
create it for the first time
                                if (HLAFixedRecordData.operator ==(NULL)){
                                        HLAFixedRecordData=
HOMM::HLAFixedRecordData::Create(DataTypes);
                                        HLAFixedRecordData.name()=
fixRecDTName;//(std::string)currentNodeMga.name()+"DT";

                                        //THESE BELOW ARE TWO ALTERNATIVES TO SELECT
ONE FROM

        //TransformFAMessage2OMTClass_FullFixRec(currentNode,currentNode,HLAFixedReco
rdData,DataTypes,true);

        TransformFAMessagePart2HLAFixedRecord(currentNode,currentNode,HLAFixedRecordD
ata,DataTypes,false);
                                }
                                //Bind the DataType Ref to the FixedRecord data
                                HLADataTypeRef.ref() = HLAFixedRecordData;

                        }else{
                                CreateAttributesForNode(currentNode, faMsg,
omtClass,hlaObjectType);
                                PushNodeChildrenToStack(nodeStack, currentNode);
                        }
                }
        }

        static void TransformFAMessagePart2HLAFixedRecord(Udm::Object root,
Udm::Object node, HOMM::HLAFixedRecordData& HLAFixedRecordData,
HOMM::FixedRecordDataTypes& DataTypes, bool UseOnlyTypeInNaming){
		//CreateAttributesForNode_FullFixRec(currentNode, faMsg,
omtClass,hlaObjectType);
		//First get the HLAAsciiString and Int32 data type objects at hand to
use later in Parameter data type ref constructions
		HOMM::HLADataTypeRef_RefersTo_Base HLAAsciiString =
GetHLAASCIIStringDT(DataTypes);
		HOMM::HLADataTypeRef_RefersTo_Base Int32 = GetInt32DT(DataTypes);
		HOMM::HLADataTypeRef_RefersTo_Base Real32 = GetReal32DT(DataTypes);

		//Compose the node (i.e., ICParameter) name by concatenating all
ancestor names up until the root node, delimited by -
		std::string absoluteName = GetAbsoluteNameForNode(node,root,true);

		Uml::Class cls =  node.type();
		Udm::StringAttr clsNameStrAttr = cls.name();
		string clsNameStr = clsNameStrAttr.operator std::string();
		Udm::ChildrenAttr<::Uml::Attribute> childrenAttr = cls.attributes();
		std::vector<::Uml::Attribute> attrs = childrenAttr.operator
std::vector<::Uml::Attribute>();
		//Create a Field on the HLAFixedRecordData for every attribute of the
node (that corresponds to the FixedRecord). These fields would have
SimpleDataTypereferences.

		//Create a parameter for every attribute of a node. Node being branch
or leaf does not matter. Note that if the node has attributes,
		// then no parameter is created excessively for the node itself, but
only for its attributes.
		for( std::vector<::Uml::Attribute>::iterator iter= attrs.begin();
iter!= attrs.end(); ++iter){
			::Uml::Attribute childAttr = (::Uml::Attribute)(*iter);
			Udm::StringAttr udmStrAttrName = childAttr.name();
			Udm::StringAttr udmStrAttrType = childAttr.type();

			string attrNameStr = udmStrAttrName.operator std::string();
			string attrTypeStr = udmStrAttrType.operator std::string();

			HOMM::Field fixedRecField =
HOMM::Field::Create(HLAFixedRecordData);
			if (UseOnlyTypeInNaming){
				//NOTE: we used attrNameStr instead of attrTypeStr
when UseOnlyTypeInNaming==true, as opposed to below, because for primitive types
attrTypeStr would result in String, Integer, etc, which is not desired.
				fixedRecField.name()=attrNameStr;
			}else{
				fixedRecField.name()=absoluteName+"-"+attrNameStr;
			}
			//IMPORTANT NOTE/FACT: enumerated attribute types in FADMM are
treated as String by the Udm API!!! (e.g. CFF tgt type comes as String typed in C++
code here!) This simplifies our job :-)
			HOMM::HLADataTypeRef dataTypeRef =
HOMM::HLADataTypeRef::Create(fixedRecField);

			std::string dtRefName = "DataType";
			if (attrTypeStr=="String"){
				HLAAsciiString.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
				dataTypeRef.ref() = HLAAsciiString;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
			}
			else if (attrTypeStr=="Integer"){
				Int32.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
				dataTypeRef.ref() = Int32;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
			}else if (attrTypeStr=="Double"){
				Real32.GetStrValue("name",dtRefName);
				dataTypeRef.name()=dtRefName+"Ref";
```

```
                              dataTypeRef.ref() = Real32;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
                       }

               }

               std::set<Udm::Object> childs = node.GetChildObjects();
               //Create a fixed record field for nodes that have NO attributes and
that are leaf nodes! (such as marker nodes like Shot, Fire, etc.)
               if (attrs.size()==0 && childs.empty()){
                       HOMM::Field fixedRecField =
HOMM::Field::Create(HLAFixedRecordData);
                       if (UseOnlyTypeInNaming){
                              //Use only the node's type name in field name
                              Uml::Class cls =  node.type();
                              Udm::StringAttr clsNameStrAttr = cls.name();
                              string nodeTypeStr = clsNameStrAttr.operator
std::string();
                              fixedRecField.name()=nodeTypeStr;
                       }else{
                              //Use the canonical path from the node to the root in
field name
                              fixedRecField.name()=absoluteName;
                       }
                       //IMPORTANT NOTE/FACT: enumerated attribute types in FADMM are
treated as String by the Udm API!!! (e.g. CFF tgt type comes as String typed in C++
code here!) This simplifies our job :-)
                       HOMM::HLADataTypeRef dataTypeRef =
HOMM::HLADataTypeRef::Create(fixedRecField);
                       std::string dtRefName = "DataType";
                       HLAAsciiString.GetStrValue("name",dtRefName);
                       dataTypeRef.name()=dtRefName+"Ref";
                       dataTypeRef.ref() = HLAAsciiString;//new
Udm::PointerAttr<::HOMM::HLADataTypeRef_RefersTo_Base>();  //Udm::PointerAttr<
::HOMM::HLADataTypeRef_RefersTo_Base> ref()
               }
               //If the node has further children (i.e., structure), then create a
FixedRecordType for every child, push those children on the stack, and let the loop
repeat the same process of creating simple types for that every child node's
attributes and fixedrec types for further children of the child node
               if (!childs.empty()){
                       for( std::set<Udm::Object>::iterator iter= childs.begin();
iter!= childs.end(); ++iter){
                              Udm::Object child = (Udm::Object)(*iter);
                              if (child.operator !=(NULL)){
                                     TransformFAMessagePart2HLAFixedRecord(root,
child, HLAFixedRecordData, DataTypes,UseOnlyTypeInNaming);
                              }
                       }
               }
        }
```

APPENDIX B

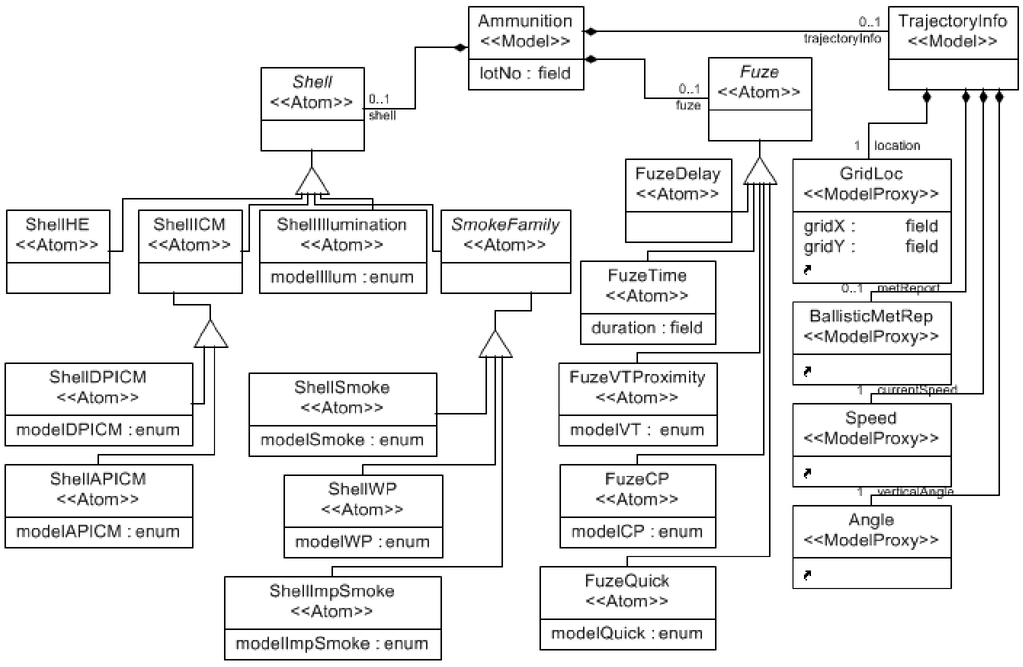
GME SCREENSHOTS OF FADMM

Entities

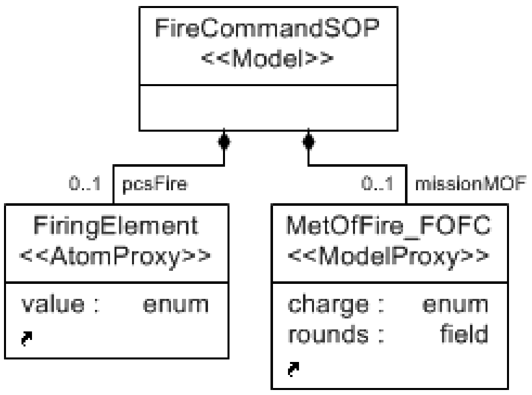


Durable Data

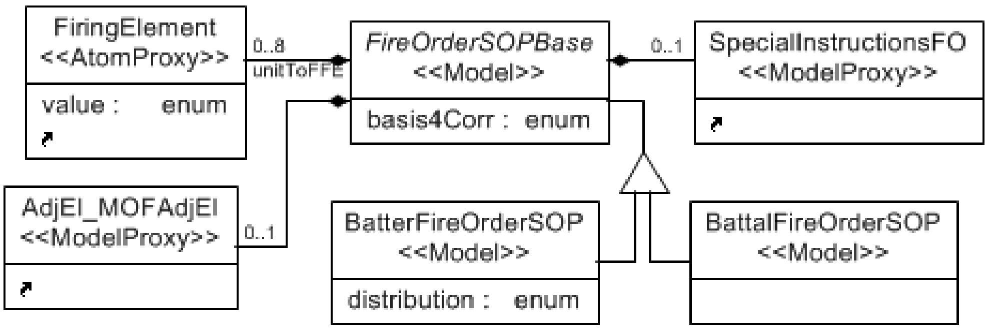
a. Ammunition



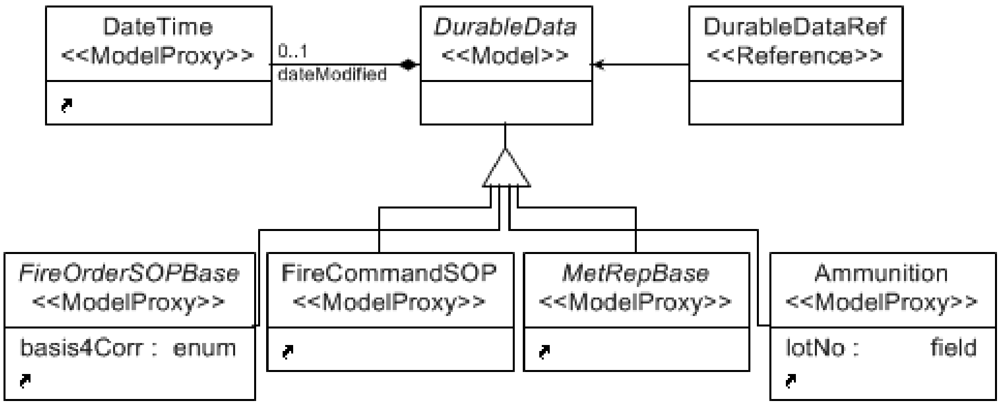
b. FireCommandSOP



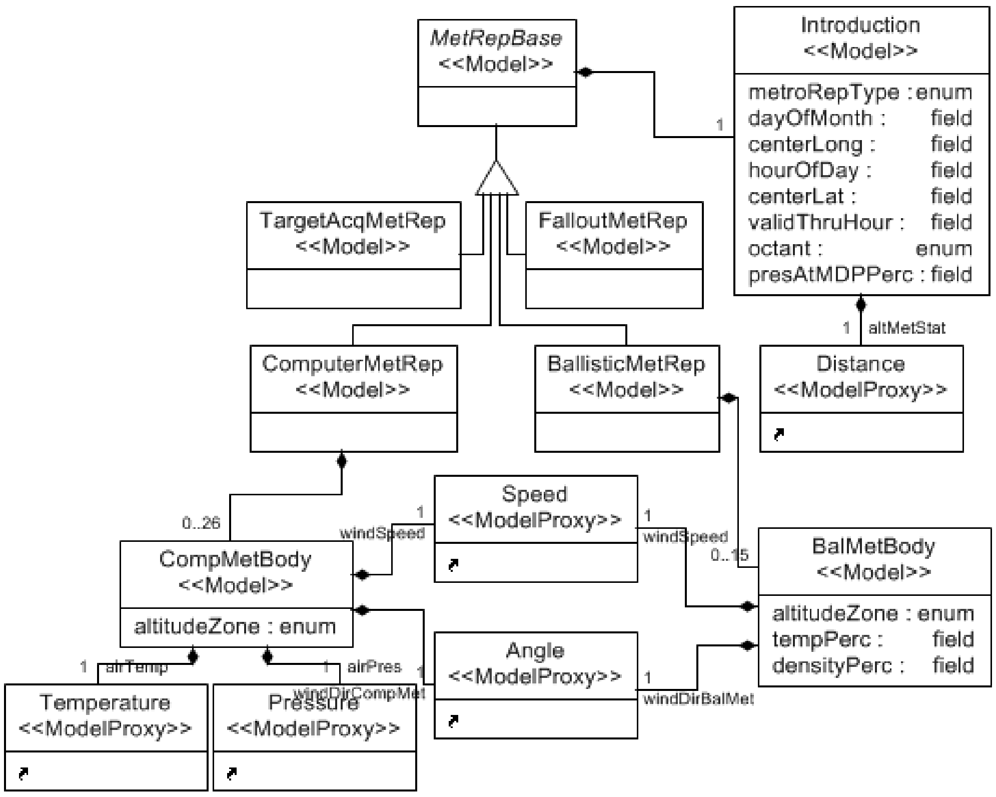
c. FireOrderSOP



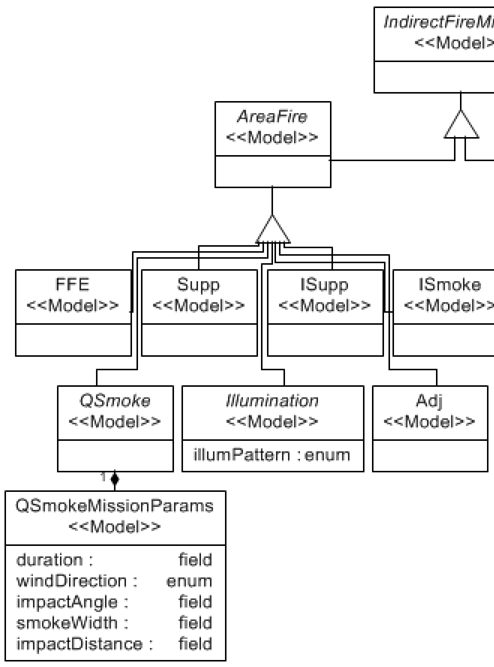
d. MainDurableData



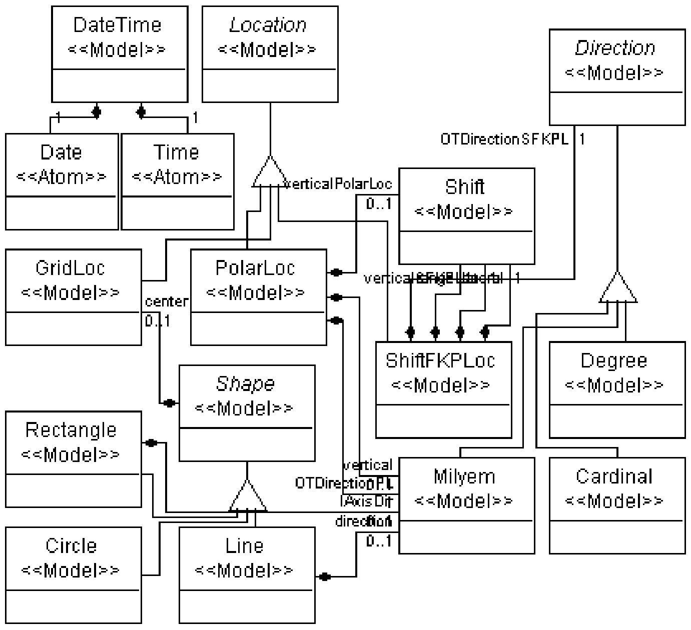
e. MetroReport



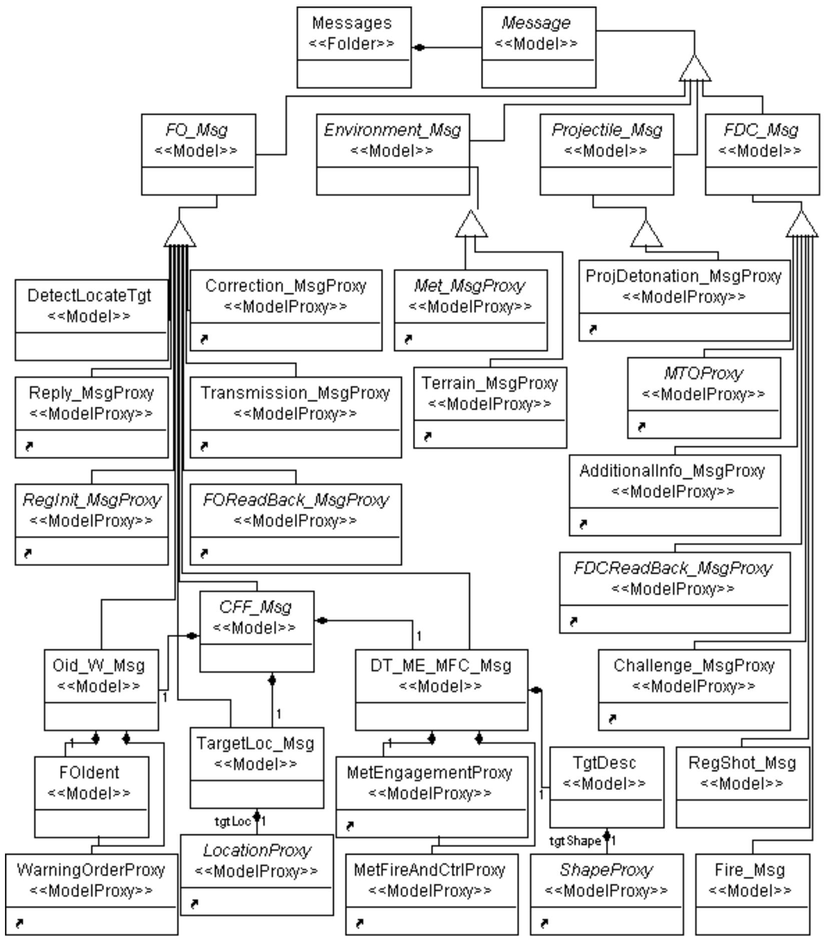
f. MissionTypeHierarchy



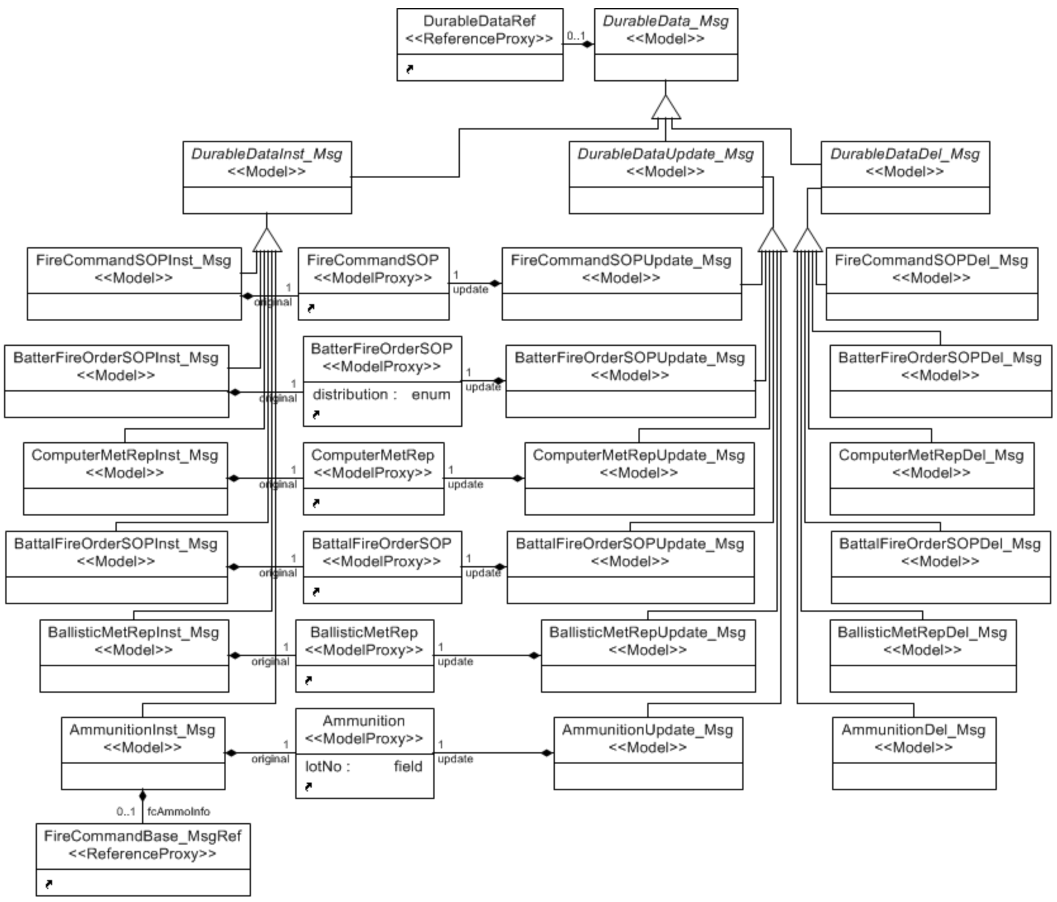
Utilities



Messages

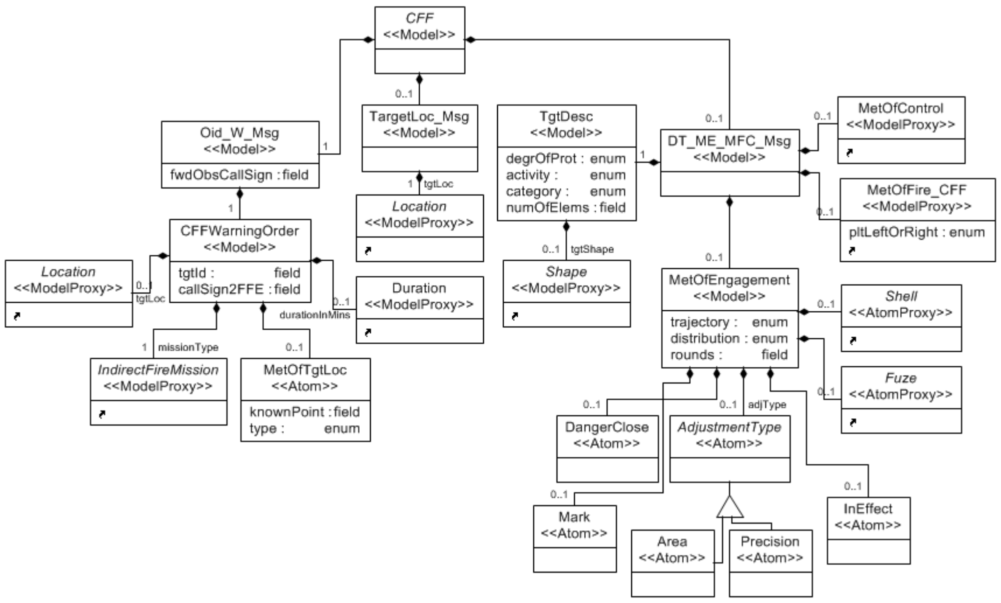

Messages <<Folder>>
Message <<Model>>
FO_Msg <<Model>>
Environment_Msg <<Model>>
Projectile_Msg <<Model>>
FDC_Msg <<Model>>
DetectLocateTgt <<Model>>
Correction_MsgProxy <<ModelProxy>>
Met_MsgProxy <<ModelProxy>>
ProjDetonation_MsgProxy <<ModelProxy>>
Reply_MsgProxy <<ModelProxy>>
Transmission_MsgProxy <<ModelProxy>>
Terrain_MsgProxy <<ModelProxy>>
MTOProxy <<ModelProxy>>
RegInit_MsgProxy <<ModelProxy>>
FOReadBack_MsgProxy <<ModelProxy>>
AdditionalInfo_MsgProxy <<ModelProxy>>
FDCReadBack_MsgProxy <<ModelProxy>>
CFF_Msg <<Model>>
Oid_W_Msg <<Model>>
DT_ME_MFC_Msg <<Model>>
Challenge_MsgProxy <<ModelProxy>>
FOIdent <<Model>>
TargetLoc_Msg <<Model>>
MetEngagementProxy <<ModelProxy>>
TgtDesc <<Model>>
RegShot_Msg <<Model>>
WarningOrderProxy <<ModelProxy>>
LocationProxy <<ModelProxy>>
MetFireAndCtrlProxy <<ModelProxy>>
ShapeProxy <<ModelProxy>>
Fire_Msg <<Model>>
tgtLoc
tgtShape
1

a. *AmmoSOPMetMsgDefs*

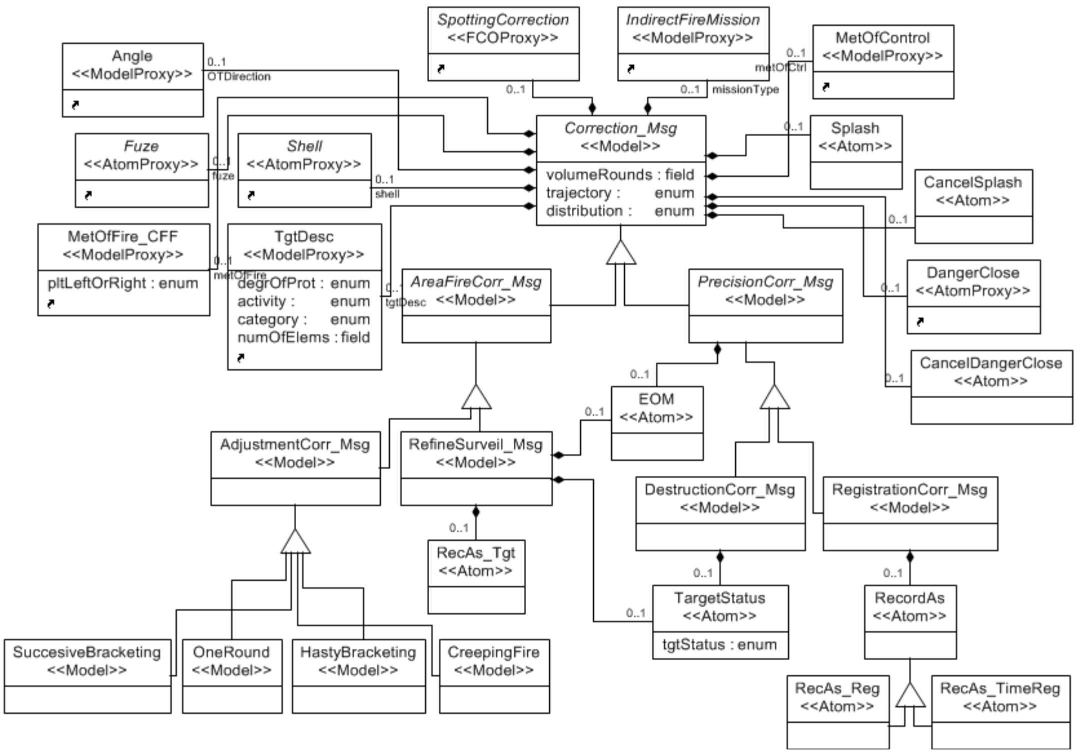


DurableDataRef <<ReferenceProxy>>
0..1
DurableData_Msg <<Model>>
DurableDataInst_Msg <<Model>>
DurableDataUpdate_Msg <<Model>>
DurableDataDel_Msg <<Model>>
FireCommandSOPInst_Msg <<Model>>
FireCommandSOP <<ModelProxy>>
FireCommandSOPUpdate_Msg <<Model>>
FireCommandSOPDel_Msg <<Model>>
BatterFireOrderSOPInst_Msg <<Model>>
BatterFireOrderSOP <<ModelProxy>>
distribution : enum
BatterFireOrderSOPUpdate_Msg <<Model>>
BatterFireOrderSOPDel_Msg <<Model>>
ComputerMetRepInst_Msg <<Model>>
ComputerMetRep <<ModelProxy>>
ComputerMetRepUpdate_Msg <<Model>>
ComputerMetRepDel_Msg <<Model>>
BattalFireOrderSOPInst_Msg <<Model>>
BattalFireOrderSOP <<ModelProxy>>
BattalFireOrderSOPUpdate_Msg <<Model>>
BattalFireOrderSOPDel_Msg <<Model>>
BallisticMetRepInst_Msg <<Model>>
BallisticMetRep <<ModelProxy>>
BallisticMetRepUpdate_Msg <<Model>>
BallisticMetRepDel_Msg <<Model>>
AmmunitionInst_Msg <<Model>>
Ammunition <<ModelProxy>>
lotNo : field
AmmunitionUpdate_Msg <<Model>>
AmmunitionDel_Msg <<Model>>
1 original
1 update
0..1 fcAmmoInfo
FireCommandBase_MsgRef <<ReferenceProxy>>

b. *CallForFire*



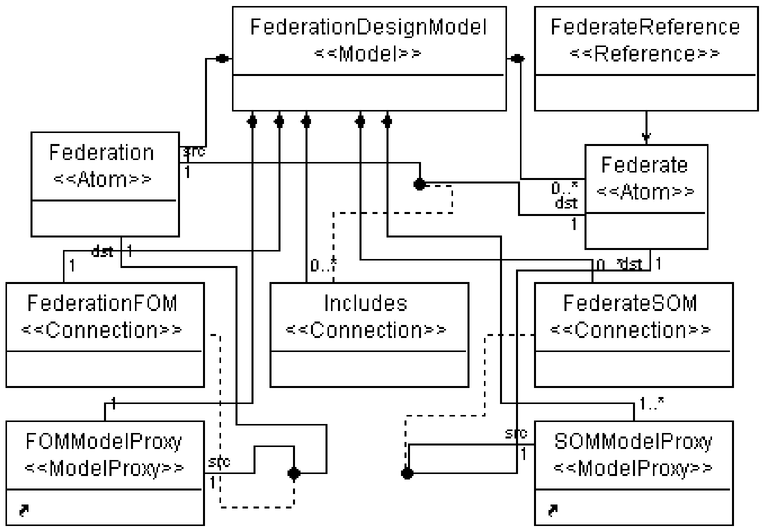
c. Correction



APPENDIX C

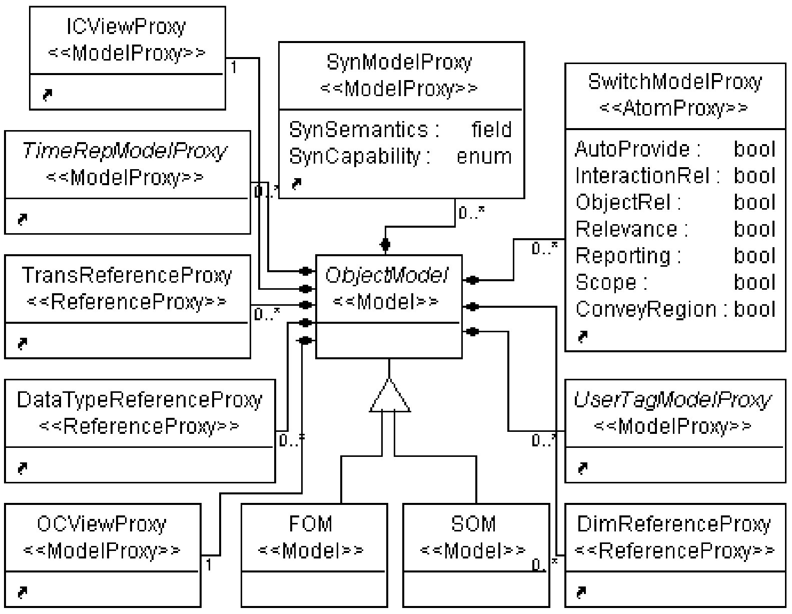
GME SCREENSHOTS OF HLA-OMT MODEL

Main Diagram



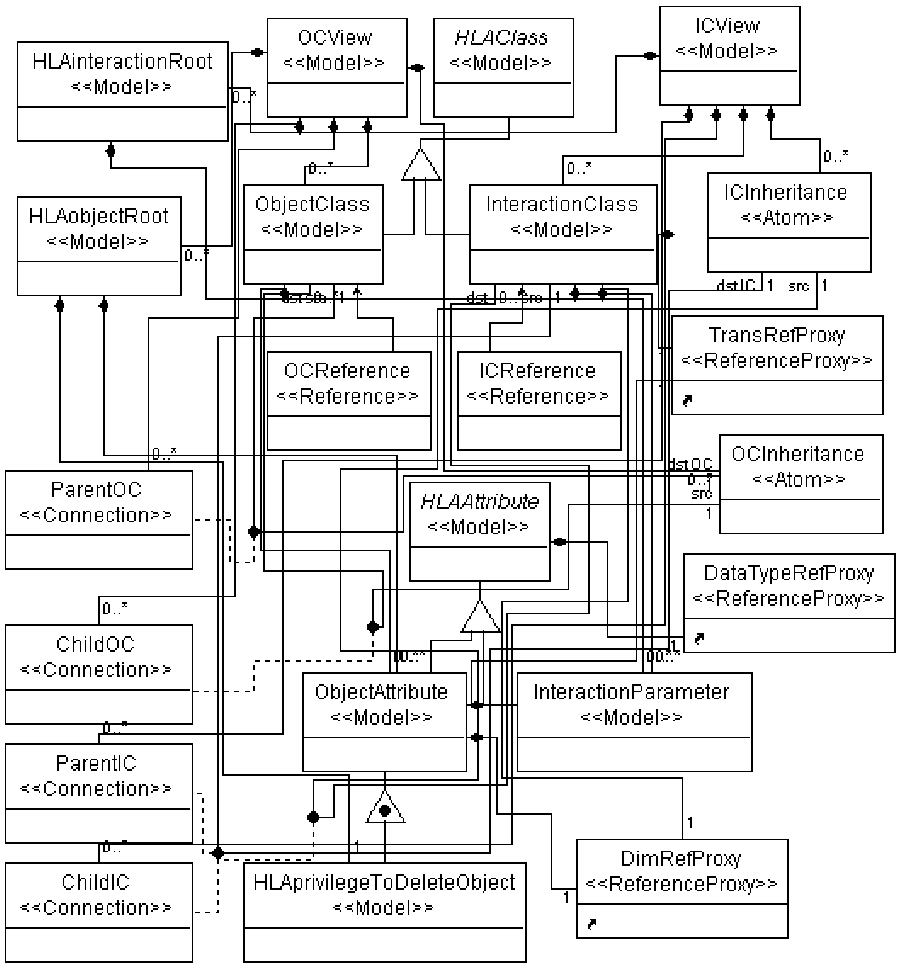
FederationDesignModel <<Model>>
FederateReference <<Reference>>
Federation <<Atom>>
Federate <<Atom>>
FederationFOM <<Connection>>
Includes <<Connection>>
FederateSOM <<Connection>>
FOMModelProxy <<ModelProxy>>
SOMModelProxy <<ModelProxy>>
src
dst
1
0..*
1..*

Object Model

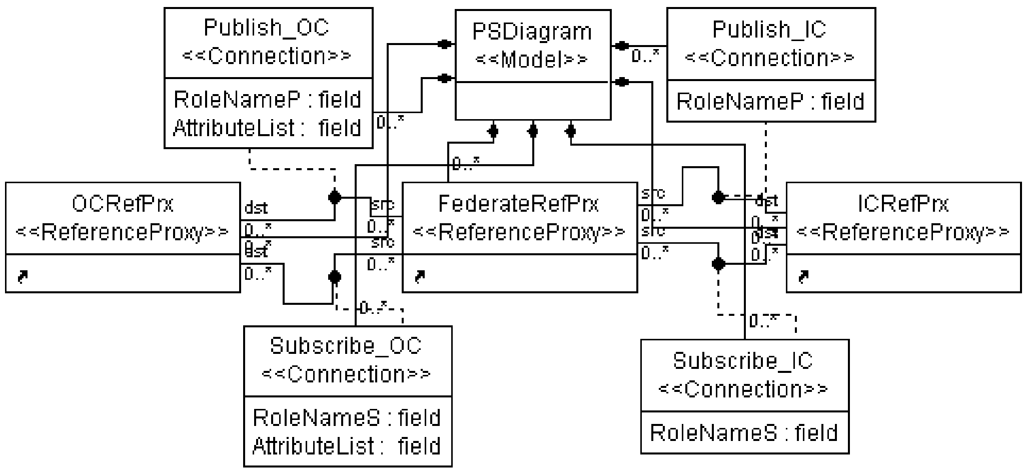


ICViewProxy <<ModelProxy>>
SynModelProxy <<ModelProxy>>
SynSemantics : field
SynCapability : enum
SwitchModelProxy <<AtomProxy>>
AutoProvide : bool
InteractionRel : bool
ObjectRel : bool
Relevance : bool
Reporting : bool
Scope : bool
ConveyRegion : bool
TimeRepModelProxy <<ModelProxy>>
TransReferenceProxy <<ReferenceProxy>>
ObjectModel <<Model>>
DataTypeReferenceProxy <<ReferenceProxy>>
UserTagModelProxy <<ModelProxy>>
OCViewProxy <<ModelProxy>>
FOM <<Model>>
SOM <<Model>>
DimReferenceProxy <<ReferenceProxy>>
1
0..*

HLA Classes


HLAinteractionRoot <<Model>>
OCView <<Model>>
HLAClass <<Model>>
ICView <<Model>>
HLAobjectRoot <<Model>>
ObjectClass <<Model>>
InteractionClass <<Model>>
ICInheritance <<Atom>>
TransRefProxy <<ReferenceProxy>>
OCReference <<Reference>>
ICReference <<Reference>>
OCInheritance <<Atom>>
ParentOC <<Connection>>
HLAAttribute <<Model>>
DataTypeRefProxy <<ReferenceProxy>>
ChildOC <<Connection>>
ObjectAttribute <<Model>>
InteractionParameter <<Model>>
ParentIC <<Connection>>
ChildIC <<Connection>>
HLAprivilegeToDeleteObject <<Model>>
DimRefProxy <<ReferenceProxy>>
0..*
dst
src
1

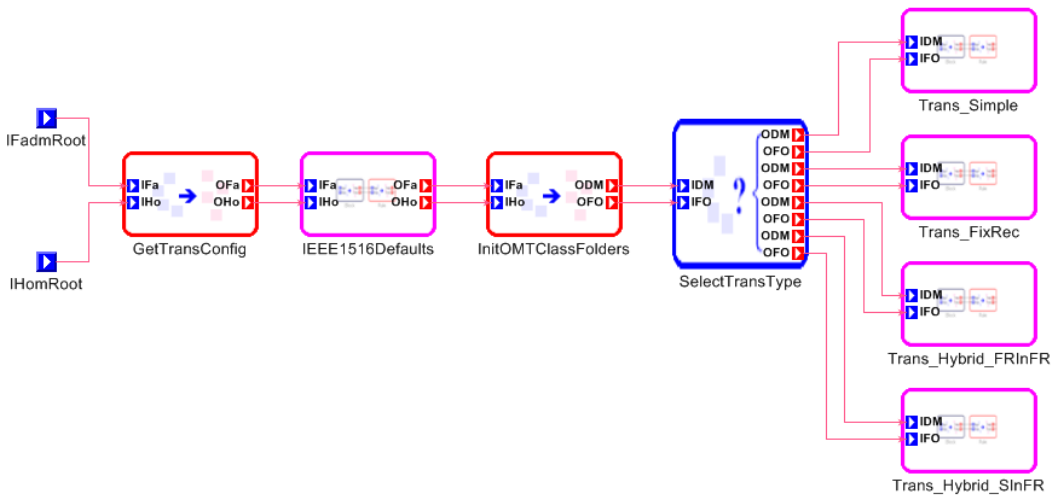
Publish-Subscribe Diagram


Publish_OC <<Connection>>
RoleNameP : field
AttributeList : field
PSDiagram <<Model>>
Publish_IC <<Connection>>
RoleNameP : field
OCRefPrx <<ReferenceProxy>>
FederateRefPrx <<ReferenceProxy>>
ICRefPrx <<ReferenceProxy>>
Subscribe_OC <<Connection>>
RoleNameS : field
AttributeList : field
Subscribe_IC <<Connection>>
RoleNameS : field
0..*
dst
src

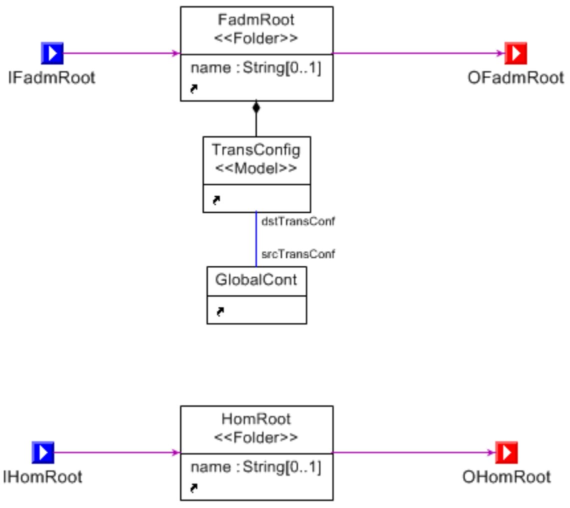
APPENDIX D

FADM TO HLA-OMT TRANSFORMATIONS SCREENSHOTS

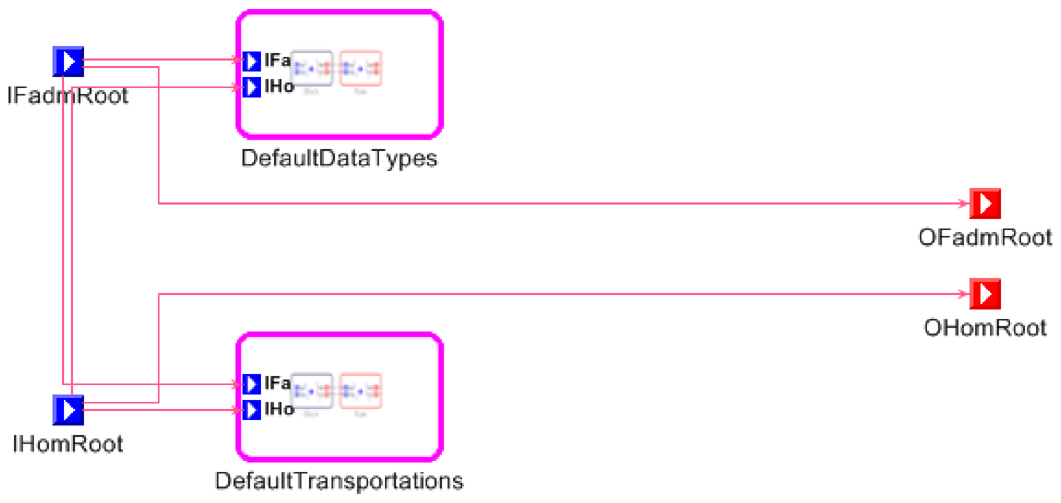
Overview



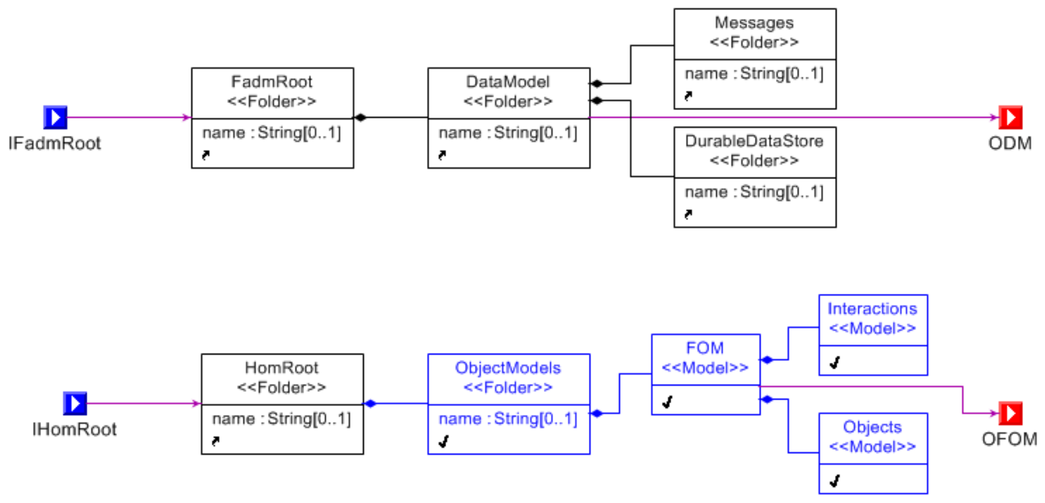
GetTransConfig



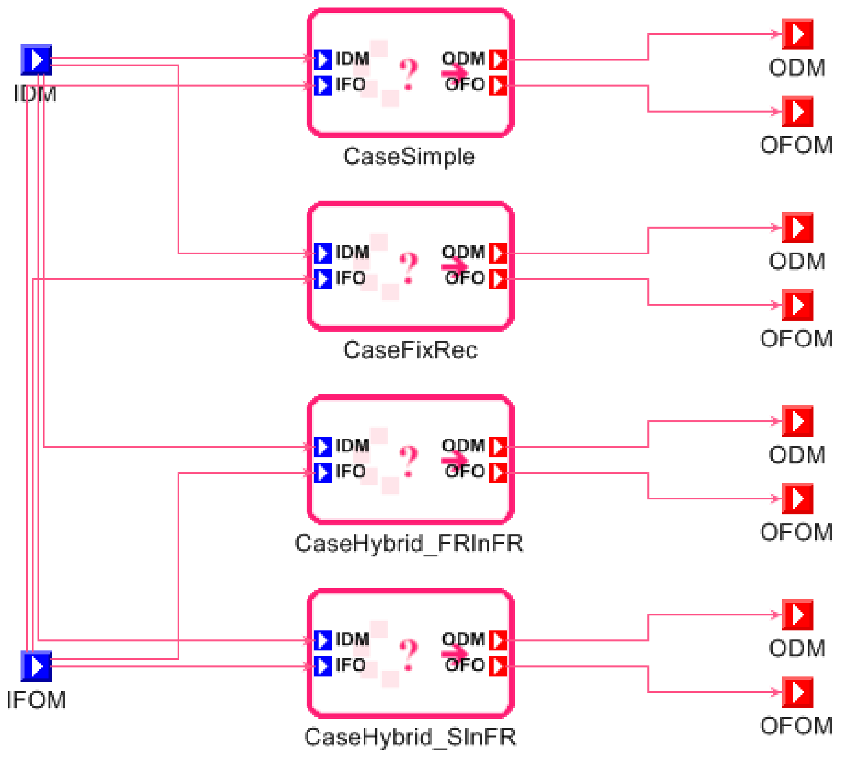
IEEE1516Defaults



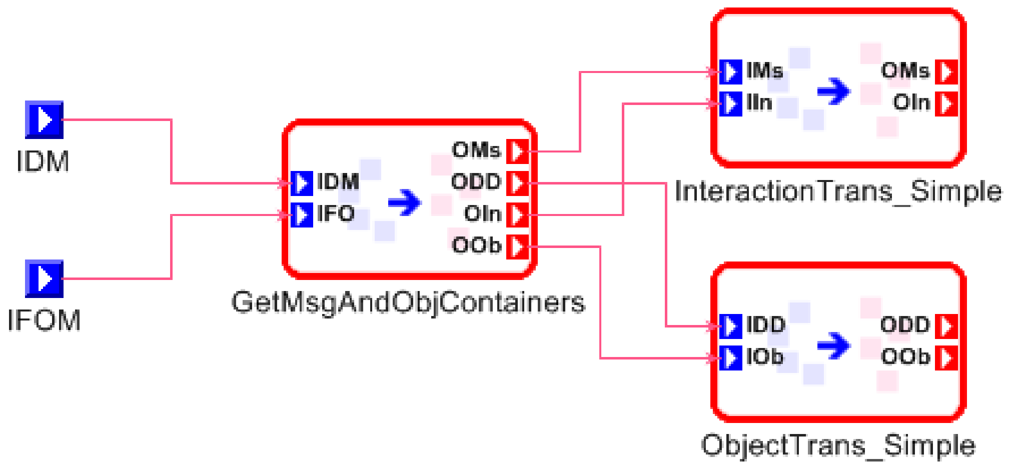
InitOMTClassFolders



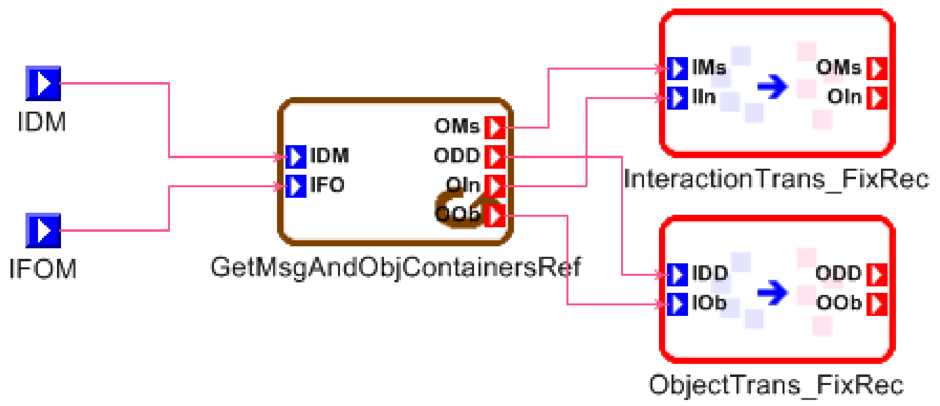
SelectTransType



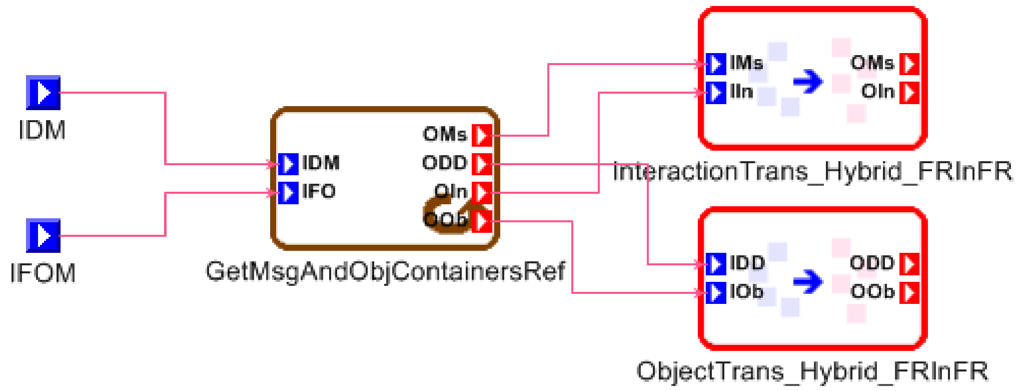
Trans_Simple



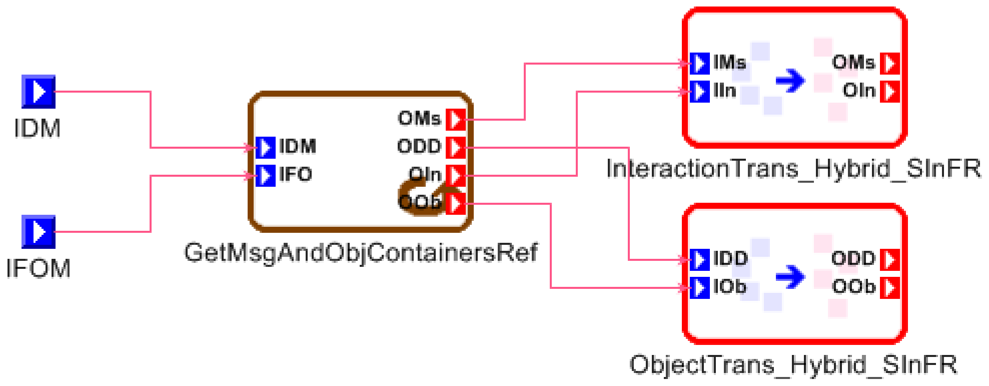
Trans_FixRec



Trans_Hybrid_FRInFR


IDM
IFOM
IDM
IFO
OMs
ODD
OIn
OOb
GetMsgAndObjContainersRef
IMs
IIn
OMs
OIn
InteractionTrans_Hybrid_FRInFR
IDD
IOb
ODD
OOb
ObjectTrans_Hybrid_FRInFR

Trans_Hybrid_SInFR


IDM
IFOM
IDM
IFO
OMs
ODD
OIn
OOb
GetMsgAndObjContainersRef
IMs
IIn
OMs
OIn
InteractionTrans_Hybrid_SInFR
IDD
IOb
ODD
OOb
ObjectTrans_Hybrid_SInFR