TRANSFORMING MISSION SPACE MODELS TO EXECUTABLE SIMULATION MODELS

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

BY

GÜRKAN ÖZHAN

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

SEPTEMBER 2011

Approval of the thesis:

TRANSFORMING MISSION SPACE MODELS TO EXECUTABLE SIMULATION MODELS

Submitted by **GÜRKAN ÖZHAN** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy 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. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering Dept., METU** _____

Examining Committee Members:

Prof. Dr. Faruk Polat
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Levent Yılmaz
Computer Science and Software Eng., Auburn University _____

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Dept., METU _____

Assist. Prof. Dr. Aysu Betin Can
Informatics Institute, METU _____

Date: _____

I hereby declare that the 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: Gürkan Özhan

Signature:

ABSTRACT

TRANSFORMING MISSION SPACE MODELS
TO EXECUTABLE SIMULATION MODELS

Özhan, Gürkan
Ph.D., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Halit Oğuztüzün

September 2011, 232 pages

This thesis presents a two step automatic transformation of Field Artillery Mission Space Conceptual Models (ACMs) into High Level Architecture (HLA) Federation Architecture Models (FAMs) into executable distributed simulation code. The approach followed in the course of this thesis adheres to the Model-Driven Engineering (MDE) philosophy. Both ACMs and FAMs are formally defined conforming to their metamodels, ACMM and FAMM, respectively. ACMM is comprised of a behavioral component, based on Live Sequence Charts (LSCs), and a data component based on UML class diagrams. Using ACMM, the Adjustment Followed by Fire For Effect (AdjFFE) mission, which serves as the source model for the model transformation case study, is constructed. The ACM to FAM transformation, which is defined over metamodel-level graph patterns, is carried out with the Graph Rewriting and Transformation (GReAT) tool. Code generation from a FAM is accomplished by employing a model interpreter that produces Java/AspectJ code. The resulting code can then be executed on an HLA Run-Time Infrastructure (RTI). Bringing a fully fledged transformation approach to conceptual modeling is a distinguishing feature of this thesis. This thesis also aims to bring the chart notations to the attention of the mission space modeling community regarding the description of military tasks, particularly their communication aspect. With the experience gained, a set of guidelines for a domain-independent transformer from any metamodel-based conceptual model to FAM is offered.

Keywords: Domain Specific Modeling, Graph-Based Model Transformation, Field Artillery, High Level Architecture, Code Generation

ÖZ

GÖREV UZAYI MODELLERİNİ
KOŞTURULABİLİR SİMÜLASYON MODELLERİNE DÖNÜŞTÜRME

Özhan, Gürkan
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Halit Oğuztüzün

Eylül 2011, 232 sayfa

Bu tez Sahra Topçuluğu Görev Uzayı Kavramsal Modelleri'nin (ACM) Yüksek Seviye Mimarisi (HLA) Federasyon Mimari Modelleri'ne (FAM), onun da koşturulabilir dağıtık simülasyon koduna iki kademeli otomatik dönüşümünü sunmaktadır. Bu tezin seyrinde izlenen yaklaşım Model Güdümlü Mühendislik (MDE) felsefesiyle örtüşmektedir. ACM ve FAM'ların her ikisi de, sırası ile, metamodelleri olan ACMM ve FAMM ile uyumludurlar. ACMM, Canlı Sıralama Çizelgelerine (LSCs) dayanan bir davranış bileşeni ile, UML sınıf diyagramlarına dayanan bir veri bileşeninden oluşmaktadır. ACMM kullanılarak, model dönüşüm örnek çalışması için kaynak model teşkil eden, Tanzim Sonrası Tesir Atışı (AdjFFE) görevi de kurgulanmıştır. Metamodel seviyesi çizge örüntüleri üzerinden tanımlanan ACM'den FAM'a dönüşüm, GReAT adı verilen araç ile gerçekleştirilmiştir. FAM'dan kod üretilmesi, Java/AspectJ kodu üreten bir model yorumlayıcısı kullanılarak başarılmıştır. Üretilen kod daha sonra bir HLA Koşma-Zamanı Altyapısı (RTI) üzerinde çalıştırılabilmektedir. Kavramsal modellemeye olgunlaşmış bir dönüşüm yaklaşımı getirmek bu çalışmanın ayırt edici bir özelliğidir. Bu tez, askeri görevlerin, iletişim yönü ön plana alınarak betimlenmesiyle ilgili olarak, çizelge notasyonlarını görev uzayı modelleme camiasının dikkatine sunmayı da hedeflemektedir. Elde edilen tecrübe ile, metamodel tabanlı herhangi bir kavramsal modelden FAM'a alandan bağımsız bir dönüştürücü için bir takım kılavuzlar ortaya konulmuştur.

Anahtar Kelimeler: Alana Özgü Modelleme, Çizge Tabanlı Model Dönüşümü, Sahra Topçuluğu, Yüksek Seviye Mimarisi, Kod Üretimi

To my family

ACKNOWLEDGEMENTS

I express sincere gratitude and appreciation to my supervisor Assoc. Prof. Dr. Halit Oğuztüzün for his guidance, continuous support and insightful comments throughout this research.

Many thanks to Prof. Dr. Faruk Polat, Prof. Dr. Hakkı Toroslu, Assoc. Prof. Dr. Ali Doğru, Assoc. Prof. Dr. Levent Yilmaz and Assist. Prof. Dr. Aysu Betin Can for their contributions in my thesis committee.

I also would like to thank to fellow PhD students Okan Topçu and Mehmet Adak for their fruitful discussions and comments.

I cannot say enough to express my gratitude to my wife Didem for her endless patience during my thesis study and for her support and assistance in every aspect of my life. I would like to mention my beloved ones here, Ece and Gülce, whom were born and grown into their early childhoods along with this marathon. Finally, my love and respect goes my father and especially my mother; without her motivating advices this thesis might have not been completed on time.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ACM	Artillery Conceptual Model
ACMM	Artillery Conceptual Meta-Model
Adj	Adjustment
AdjFFE	Adjustment Followed by Fire For Effect
AGG	Attributed Graph Grammar
Alt	Alternative
AOP	Aspect Oriented Programming
API	Application Programming Interface
ATL	ATLAS Transformation Language
AToM3	A Tool for Multi-formalism and Meta Modelling
BML	Battle Management Language
BMM	Behavioral Metamodel
BOM	Base Object Model
BOTL	Bidirectional Object oriented Transformation Language
C2	Command and Control
CF	Check Firing
CFF	Call For Fire
CIM	Computation Independent Model
CL	Cease Loading
CM	Conceptual Model
CMMS	Conceptual Models of the Mission Space
CodeGen	Code Generator
DCMF	Defense Conceptual Modeling Framework
DDM	Data Distribution Management
DIHT	Domain-Independent HLA Transformer
DoD	Department of Defense
DSAM	Distributed Simulation Architecture Model
DSML	Domain Specific Modeling Language
Exc	Exception

FA	Field Artillery
FAM	Federation Architecture Model
FAMM	Federation Architecture Meta-Model
FAT	Field Artillery Team
FC	Fire Command
Fd	Federate
FDC	Fire Direction Center
FDD	FOM Document Data
FDO	Fire Direction Officer
FDSS	Facility for Distributed Simulation Systems
Fed	Federation
FFE	Fire For Effect
FO	Fire Order
FOM	Federation Object Model
FSMM	Federation Structure Meta-Model
FUJABA	From UML to Java and Back Again
FwdObserver	Forward Observer
GME	Generic Modeling Environment
GReAT	Graph Rewriting And Transformation
GXL	Graph eXchange Language
HFMM	HLA Federation Metamodel
HLA	High Level Architecture
HMSC	High-Level MSC
HOMM	HLA Object Meta-Model
HSMM	HLA Services Meta-Model
IEEE	Institute of Electrical and Electronic Engineers
ISupp	Immediate Suppression
ITU-T	International Telecommunication Union
JC3IEDM	Joint C3 Information Exchange Data Model
KAMA	KAvramsal Modelleme Aracı (in Turkish)
LHS	Left Hand Side
LSC	Live Sequence Chart
LscRTILib	LSC RTI Library
MDA	Model Driven Architecture
MDE	Model Driven Engineering

MetaGME	GME Metamodel
MIC	Model Integrated Computing
METU	Middle East Technical University
MOF	Meta Object Facility
MOM	Management Object Model
MSC	Message Sequence Chart
MTO	Message To Observer
OCL	Object Constraint Language
Oid_W_Msg	Observer Identification and Warning Message
OMG	Object Modeling Group
OMT	Object Model Template
Opt	Option
Par	Parallel
PIM	Platform Independent Model
PSM	Platform Specific Model
pRTI	Pitch RTI
RBAC	Role-Based Access Control
RHS	Right Hand Side
RTI	Runtime Infrastructure
RTILib	RTI Library
Seq	Sequence
SIW	Simulation Interoperability Workshop
SOA	Service Oriented Architecture
SOM	Simulation Object Model
SOP	Standing Operating Procedures
STANAG	STANdardization AGreement
Supp	Suppression
SVBR	Semantics of Business Vocabulary and Business Rules
UDM	Universal Data Model
UML	Unified Modeling Language
UMT	Universal Model Transformer
UWE	UML-based Web Engineering
VIATRA	VIsual Automated model TRAnsformations
XML	Extendible Markup Language
XMSF	Extendible Modeling and Simulation Framework

CHAPTER I

INTRODUCTION

The Model-Driven Engineering (MDE) approach [1] is becoming prominent in software and systems engineering, bringing forth a model-centric approach to the development cycle in contrast to today's mostly code-centric practices. 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 into a Platform Specific Model (PSM), which can be readily translated to executable code [2].

Model Integrated Computing (MIC) [3], an earlier manifestation of MDE, relies on metamodeling to define domain-specific modeling languages and model integrity constraints. The metamodel (also called a paradigm) is then used to automatically compose a domain-specific model building environment for creating, analyzing, and evolving the system through modeling and generation. In the MIC approach, a crucial point is generation, where (domain-specific) models are transformed into lower level executable or analysis models. Model transformation techniques and tools are essential to MIC for enabling the generation process.

1.1 Motivation and Scope

There has been a considerable proliferation of literature on model transformations, and specifically on graph-based transformations during the last two decades [4] and a rapid dissemination of the MDE approach in the last decade [5][6][7][8]. As such, a recent interest has been shown by the modeling and simulation community [9][10]. More importantly, it is seen that the approach is perceived as a key ingredient in various major defense modeling and simulation program of works and researches [88][89][90][91][92].

Up to our knowledge, in most of the related works model transformations are used as a facilitating step in achieving a major objective in various application areas such as semantic web, data mining, knowledge engineering and military. These transformations are usually applied among very specific, narrow domains, compromising realistic concerns and

restrictions [67][68][69][70][71][72]. Many of the efforts are single step source model to target model transformations [67][68][69][70][71][78][83][80]. Some are either done within the same domain (mostly operating on a single model), or between two highly similar, tightly coupled domains [67][80]. Finally, although model transformation works emphasizing either data [76][77][81][82][83][91][94], or behavior transformation [71][72][73][74] [79] are abundant, works equipoising both aspects in an integrated fashion are rare.

In most of the cited works, the employment of a formal metamodel for the subject model or metamodel usage in transformations is not a primary concern [68][70][71][81][83]. The usual approach is to analyze the model to obtain an abstracted form (could be comparable to a kind of metamodel) as a preliminary step and consult to it during the transformation process. Moreover, many of these transformations are not formally defined, but rather presented barely as algorithms or pseudo codes and usually implemented in a high level programming language. This nature causes these transformation efforts to be hampered by procedural details and lack of comprehensibility.

Among many of the works that exhibit a more MDA centric characteristic, it is seen that although apparently well thought out rules and guidelines on mapping PIM elements to PSM elements are present, no tool support is provided for automating model transformations [88][89][93][94]. The target model is built manually or guided by a GUI-based tool from scratch based on the source model and the rules. (A comparison of related work on model transformations appears in Section 5.7).

Bringing a fully-fledged transformation approach to conceptual modeling is a distinguishing feature of our work. In this thesis, we put forth a formal, declarative and visual transformation process from Field Artillery Conceptual Model (ACM) [11] to Federation Architecture Model (FAM) [12]. The produced FAM is then fed into a code generator, packaged as a "model interpreter" in MIC parlance, to generate Java/AspectJ code that can be executed on a High Level Architecture (HLA) Run-Time Infrastructure (RTI) [13]. In this sense our work can be considered as a sequence of applications of the MIC approach. It is intended as an MDE-based end-to-end systems development endeavor from the conceptual model to executable simulation code, promoting model transformation usage. We treat both data and behavior on equal grounds in our transformation perspective. Furthermore, we assess our work in the view of a set of model transformation properties that are published in the literature. From the experience gained and lessons learned, we also offer a number of suggestions for tailoring the conceptual model of any source domain for the pursuit of achieving domain-independent FAM transformations.

Within the scope of this thesis, the development of ACM is realized as a preliminary step before the ACM to FAM transformation definition work. ACM is comprised of a behavioral component and a data component. Both sub-metamodels are separately developed and seamlessly integrated with each other. Using ACM, the well known *Adjustment Followed by Fire For Effect* (AdjFFE) mission is also modeled, which is the source model for the model transformation case study. To the best of our knowledge, this work is unique in applying the LSC language for the modeling of military tasks.

We consider this work as a pioneering step towards introducing the overarching vision of model driven development advocated by the MDE into the modeling and simulation domain. The kind of MDE work accomplished in this thesis has been cited as a challenge in various publications [88][89][90][93].

Using our implementation a field artillery domain expert competent in modeling can develop his ACM, run the ACM to FAM transformer on it and obtain the corresponding FAM. Then applying a second transformation (i.e., running the code generator), he would produce the base and default aspect codes for federation execution on an HLA RTI. The code becomes ready for execution with a hand from a programmer after organizing it into an Eclipse project and weaving the hand-written computation aspect that sets the run-time values for the data structures.

1.2 The Context of the Transformations

In order to clarify the purpose and provide a referential overview of the process, Figure 1.1 illustrates the two-phased transformation approach in a nutshell. The first phase is a model-to-model transformation whereas the second is a model-to-code transformation, executed in sequence. We envision an HLA-based distributed simulation development process consisting of conceptual modeling (ACM), federation architecture modeling (FAM) and federate code generation, in that order. The ACM is a PIM (or a Computation Independent Model (CIM) [20] from a more abstract perspective) of the real world system (i.e., field artillery domain) with which the simulation is concerned. The FAM is a PSM, where the platform is the RTI in our case. It constitutes a major portion of the federation design documentation. The graph-based model transformer produces a FAM from an ACM and the code generator produces executable code from that FAM.

ACMM and FAMM are the metamodels of ACMs and FAMs, respectively. Both metamodels (and consequently, the models) have data and behavior parts. The metamodel of Live Sequence Charts (LSC) and Message Sequence Charts (MSC) [14][15][12] are used for behavioral representation in both metamodels (referred to as BMM (Behavioral MetaModel) in the figure). Both data models are integrated with the behavioral model in

that the top level data model elements are extended from a set of designated LSC and MSC elements. The transformation definitions are structured accordingly, so that firstly the data model transformation is conducted, followed by the behavioral model transformation. In the second phase, the federate application code generator produces executable federate source codes and useful artifacts such as Federation Object Model (FOM) Document Data (FDD) from the FAM.

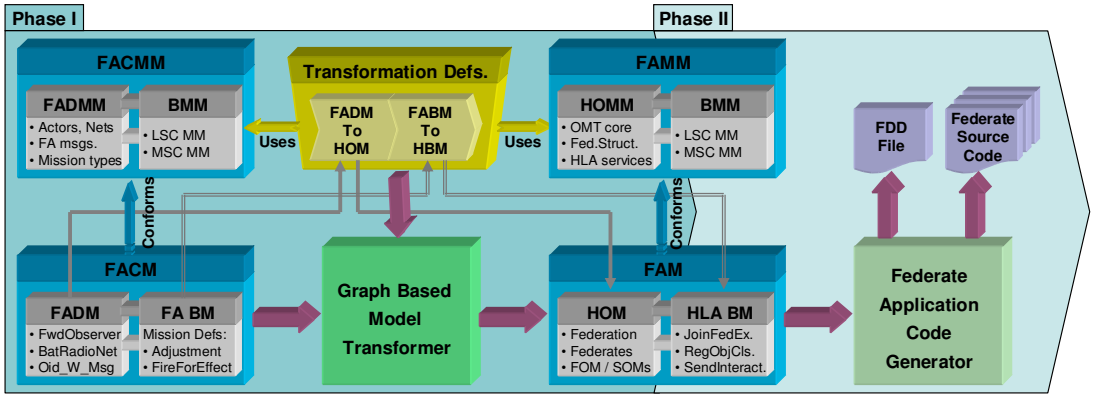


Figure 1.1 The view of FACM to FAM to executable code transformation

It may look plausible to directly produce HLA federate codes from the conceptual model, instead of going through two steps of model transformations. Our approach is more appealing in at least two ways. First, ACM rests at a higher conceptual level (corresponding to PIM), while federation code is at a lower, much detailed level. FAM on the other hand, is at an intermediary level (corresponding to PSM, where HLA defines the platform), serving as a bridge between the two levels. It has a clearer mapping from ACM, and to federation code. This makes the transformations more modular and maintainable. Second, the components of a FAM, that is, the HLA Object Model Template (OMT) model (a FOM or a Simulation Object Model (SOM)) and intra-federation behavioral model are useful artifacts in their own right. Furthermore, once a FAM, which is machine processible, is available it can be used as an input to further processing, such as optimization, tuning, debugging, verification and validation.

The tasks of modeling and metamodeling are both carried out using the Generic Modeling Environment (GME) [3], an open source toolkit for creating domain-specific modeling and program synthesis environments. GME initially serves as a metamodel development environment for domain analysts. Metamodels developed in GME conform to

MetaGME, the metamodel (in fact, meta-metamodel) provided by GME. Once a metamodel is registered in GME, it provides a domain-specific model building environment for model developers, characterized by the registered metamodel.

The graph-based model transformer is developed with Graph Rewriting and Transformation (GReAT) [6], a graph-based model transformation specification language, and partly hand-coded in C++. GReAT models conform to the pre-registered UMLModelTransformer (UMT) metamodel that comes bundled with the GME installation. Hence, model transformations are also defined as models developed in the GME environment. The transformations are defined over the metamodels of the source and target domains, expressed in a Unified Modeling Language (UML)-based notation. The metamodels are exported into the transformation model by invoking a special interpreter embedded in the visual editor of GME. The relationships between the models and metamodels mentioned above are summarized in Figure 1.2. The modeling and model transformation activities and products in this thesis are formally defined due to the conformance associations between the models and their metamodels.

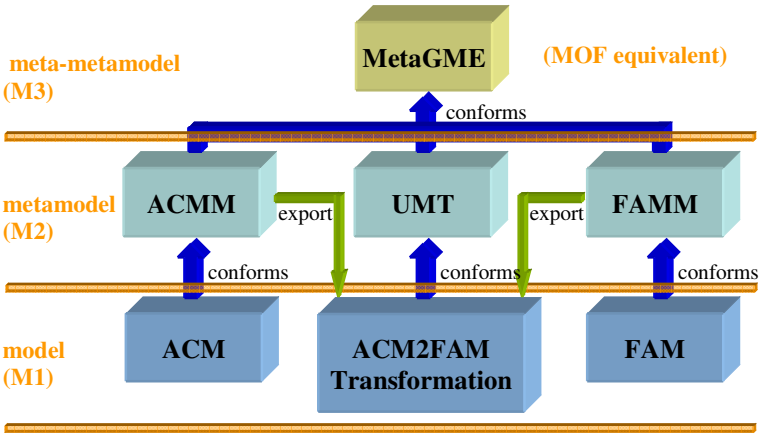


Figure 1.2 The three layers of modeling used in the transformations

Within the context of this thesis, the ACM metamodel and ACM2FAM transformation are developed. The FAM metamodel [85] and code generation from FAM [86] were developed in previous theses. We present the details of ACM, ACM2FAM transformation and demonstrate an end-to-end transformation series for the ACM model of the conventional *Adjustment Followed by Fire For Effect* (*AdjFFE*) mission to simulation execution in subsequent chapters. We also analyze the presented transformation work with

respect to a set of principles published in literature for analyzing model transformation approaches. Finally we draw lessons learned and propose directions for further research.

1.3 Rationale for Using Graph Transformations and GReAT

Graph grammars and graph transformations have been recognized as powerful devices for specifying and performing complex model-to-model transformations. From a mathematical viewpoint, models in MIC are graphs, to be more precise, vertex and edge labeled multi-graphs (i.e., graphs that are permitted to have edges that have the same end vertices), where the labels are denoting the corresponding entities in the metamodel. It has been proved useful to formulate the model transformation problem as a graph transformation problem.

Graph transformation offers a set of techniques and associated formalisms that are directly applicable to model transformation [18][19]. It is powerful and appealing in many ways. First, it is visual, in that the source, the target and the transformation itself can be expressed in a visual way. Second, it is formally founded, in that it is possible to prove certain properties of the transformation by resorting to graph theory. Third, it offers a clean semantic model to understand and specify model transformations. For example, the order of rule application is implicit, and the traversal of source models and creation of target models is implicit. This allows one to hide the procedural details of the transformation, making the transformations more compact and maintainable. Last, but not least, it offers mechanism for transformation composition. The major bottleneck associated with graph transformation is poor runtime performance.

GReAT [6] is a tool that allows users to specify graph transformations in a graphical form with precise formal and executable semantics. GReAT has a high-level control flow language built on top of the graph transformation language with sequencing, non-determinism, hierarchy, recursion and branching constructs. GReAT is based on the use of UML class diagrams (and Object Constraint Language (OCL)) for representing the domains of the transformations, including structural integrity constraints over those domains. Transformations over multiple domains are supported, and cross-links among domains are defined at the metamodeling level. Another advantage of selecting GReAT is its integration with the source and target model development environment, GME [3]. A transformation definition is yet another model defined in GME (see sections 2.3 and 2.4).

1.4 Organization of the Thesis

This dissertation is organized as follows: Chapter I points to the theoretical foundations that this work builds on, introduces the motivation and scope, presents the context of the

model transformations, explains the rationale behind the adopted approach and finally outlines the organization of the thesis.

Chapter II provides a literature overview of the concepts, techniques and tools used in the thesis. Specifically, the model driven approach to software development, graph transformations concepts, a set of prominent graph transformation tools, the field artillery observed fire domain and HLA domain are summarized. Additionally, GME and GReAT, which are respectively the modeling and graph transformation tools used, are introduced. Then MSC and LSC, the formalisms that ACM and FAM use in behavioral modeling are presented. Since FAM, the target domain of the transformations, is developed as part of another thesis, a brief overview of it is also provided.

Chapter III summarizes the entire model driven development work put forth in this thesis in a concise, tool and technology independent, and abstract conceptual framework.

Chapter IV presents the ACM metamodel in detail. First, a high level overview of ACM is presented, and then its realization in the GME environment is explained. The chapter also demonstrates the development of the source model of the transformation case study, namely, the AdjFFE mission model, as an ACM instance in GME. The complete AdjFFE mission model in graphical LSC notation is provided in Appendix A. The chapter further discusses the challenges encountered and provides an informal assessment of ACM and usage of LSC in modeling military tasks. Finally it concludes with related works on conceptual modeling.

Chapter V presents the two step automatic transformation of ACM to FAM and FAM to executable distributed simulation code. The ACM to FAM transformation is explained in two sections as data and behavior transformation. Then, the code generation mechanism from FAM is briefly introduced, followed by some excerpts from of the generated AdjFFE code and its execution on an HLA RTI. It also explains a preprocessing step required by the code generator, where the FAM is further refined so that every LSC containing multiple instances is stripped into several LSCs containing only one federate and the federation instances. The details of the transformation rules are delegated to Appendix B. The chapter further provides an analysis of ACM to FAM transformation in terms of modularity, internal transformation composition, staging, scope and direction. Finally, the chapter concludes with related works on model transformations.

Chapter VI is a discussions and future work chapter where we first discuss on the issues and lessons learned from modeling ACMM and defining ACM2FAM transformation. Then we discuss the formerly developed FAMM and simulation code generator within the context of ACM transformations. After that, a comparison of the artifacts of this thesis with

MDA standards is made. The chapter concludes with pointing to future research directions. Specifically it aims to draw the attention of the reader to a domain-independent CM transformer for HLA, higher order transformations and BOM usage for intra-federate modeling. The requirements and outline of the CM transformer, drawn from the experience gained in this work, is discussed in a broader context.

Chapter VII concludes the thesis by highlighting the major accomplishments and the novelties of this dissertation. It also points to the way ahead for further research efforts.

Appendix A presents all of the LSCs for the Adjustment Followed by Fire For Effect (AdjFFE) mission model in graphical notation. Each LSC is provided with a brief description of its purpose, execution conditions and logic.

Appendix B outlines the set of most prominent ACM to FAM model transformation blocks and rules as implemented in GReAT-configured GME.

Appendix C summarizes the changes made in the metamodels and the simulation code generator in the course of developing ACM2FAM transformation. The details of the change logs are documented in the accompanying thesis CD.

Appendix D provides hints and recommendations derived from our experience in realizing the ACM2FAM transformation for future model transformation developers of GReAT.

CHAPTER II

BACKGROUND

This chapter provides a literature overview of the concepts, techniques and tools used in this dissertation. Specifically, the model driven approach to software development, graph transformations, Field Artillery (FA) domain and High Level Architecture (HLA) domain are summarized. The Model Driven Architecture (MDA) is OMG's manifestation of model-driven software development for the future, which envisions systematic refinements, or technically speaking, transformations of high level domain models into platform specific models and finally down to executable code. Model transformation through graph transformation is currently one of the commonly used techniques in putting model-driven development into practice. There is an extensive set of graph transformation-based tools and environments developed in the literature, of which we present some. GReAT [6] is the graph transformation tool used in this thesis, which runs on top of GME, the tool that we have used in modeling the source and target domains, as well as defining the GReAT transformation model. Hence, GME and GReAT are also introduced in their own sections. The FA observed fire techniques and HLA constitute the domains of the source and target models used in this thesis. Artillery Conceptual Model (ACM) and Federation Architecture Model (FAM) are the metamodels of the source and target models formally developed in GME. Message Sequence Chart (MSC) and Live Sequence Chart (LSC), which are the formalisms used for behavioral modeling in both ACM and FAM are also introduced here. Since FAM is developed as part of another work, a brief summary of it is also provided. ACM is developed as part of this thesis and is the subject of Chapter 4.

2.1 Model Driven Architecture/Engineering

Kleppe et al. [20] state in their MDA book that, "The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform". MDA provides an approach for, and enables tools to be provided for:

- specifying a system independently of the platform that supports it,

- specifying platforms,
- choosing a particular platform for the system, and transforming the system specification into one for a particular platform

The primary goals of MDA are portability, interoperability and reusability in the course of architectural separation of concerns. The Model Driven Architecture (MDA) [21] is a framework for software development put forth by the Object Management Group (OMG). The MDA development life cycle, which is shown in Figure 2.1, does not look very different from the traditional life cycle in that the same phases are identified. A remarkable difference is the artifacts that are created during the development process. The artifacts are formal models that can be processed by the computers. The following three models are at the core of the MDA.

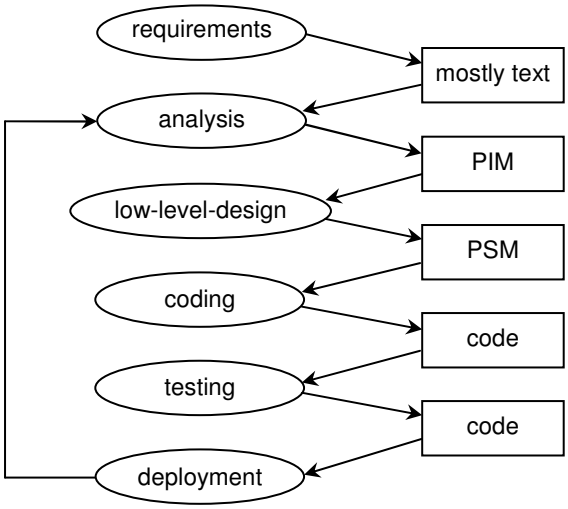


Figure 2.1 MDA software development life cycle [20]

Platform Independent Model (PIM): It is a model with a high level of abstraction so that it is independent of any implementation technology. The base PIM expresses only business functionality and behavior.

Platform Specific Model (PSM): A PSM is customized to specify a system in terms of implementation constructs that are in one specific implementation technology. MDA proposes that a PIM be transformed into one or more PSMs. It is clear that a PSM will only seem comprehensible to a developer who has detailed knowledge about the specific platform.

Code: The final step in the development is the transformation of each PSM to code.

MDA promises productivity, interoperability and maintainability improvements in the software development lifecycle.

Kent [22] remarks that MDA focuses on architecture, on artifacts, on models. Although MDA declares there might be a richer modeling space, it chooses to focus on just one dimension, the transformation between platform independent and platform specific models.

The OMG MDA strategy envisions a world where models play a more direct role in software production, being amenable to manipulation and transformation by machine. Model Driven Engineering (MDE) is wider in scope than MDA. MDE combines process and analysis with architecture.

Schmidt [1] states that MDE technology is a promising approach to address platform complexity. Domain-Specific Modeling Languages (DSMLs) formalize the application structure, behavior, and requirements within particular domains. DSMLs are described using metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively.

Generators and transformation engines analyze certain aspects of models and then produce various types of artifacts, such as source code, simulation inputs, test cases or alternative model representations. The ability to produce artifacts from models helps ensure the consistency between application implementations and analysis information associated with functional and quality requirements captured by models. This automated transformation process is often referred to as "correct-by-construction," in place of conventional handcrafted "construct-by-correction" software development processes.

MDE tools force domain-specific constraints and perform model checking that can detect and prevent many errors early in the life cycle. In addition, MDE tool generators need not be as complicated since they can produce artifacts that map onto higher-level, often standardized, middleware platform APIs and frameworks, rather than lower-level operating system APIs. As a result, it is often much easier to develop, debug, and evolve MDE tools and applications created with these tools.

Model Integrated Computing (MIC) [3], an earlier manifestation of MDE, relies on metamodeling to define DSMLs and model integrity constraints. The metamodel (also called a paradigm) is then used to automatically compose a domain-specific model building environment for creating, analyzing, and evolving the system through modeling and generation. In the MIC approach, a crucial point is 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.

2.2 Graph Transformations

Graph grammars and graph transformations have been recognized as a powerful technique for specifying complex transformations. Graph grammars are an extension of textual grammars and they give rise to node replacement grammars [5][23] and hyperedge replacement grammars [24][25]. Graph transformation research is associated with various mathematical fields such as category theory, set theory and algebra, and applies it to graphs. The prominent techniques in this area are double pushout [26], single pushout [27] and programmed structure replacement systems [28]. A brief introduction to graph transformation concepts is provided in Section 2.2.1 and some of the prominent graph transformation tools are shortly mentioned in section 2.2.2.

2.2.1 Graph Transformation Concepts

Graph transformations can be used as a computation abstraction. The basic idea is that the state of a computation can be represented as a graph, further steps in that computation can then be represented as transformation rules on that graph. Such rules consist of an original graph, which is to be matched to a subgraph in the complete state, and a replacing graph, which will replace the matched subgraph. Formally, a graph rewriting system consists of a set of graph rewrite rules of the form L→R , with L being called pattern graph (or Left-Hand Side (LHS)) and R being called replacement graph (or Right-Hand Side (RHS) of the rule). A graph rewrite rule is applied to the host graph by searching for an occurrence of the pattern graph and by replacing the found occurrence by an instance of the replacement graph.

The graph patterns can be rendered in the concrete syntax of their respective source or target language or in the (Meta Object Facility –MOF [16]) abstract syntax. The LHS often contains conditions in addition to the LHS pattern. Some additional logic (e.g., in string and numeric domains) is needed in order to compute target attribute values (such as element names). An extended form of patterns with multiplicities on edges and nodes is also common. In most approaches, scheduling has an external form and the scheduling mechanisms include non-deterministic selection, explicit condition, and iteration (including fixpoint iterations). Fixpoint iterations are particularly useful for computing transitive closures.

From a mathematical viewpoint models in MIC are graphs, to be more precise: vertex and edge labeled multi-graphs (i.e., graphs that are permitted to have edges that have the

same end vertices), where the labels are denoting the corresponding entities in the metamodel. It is plausible to formulate the model transformation problem as a graph transformation problem. We can then use the mathematical concepts of graph transformations to formally specify the intended behavior of a model transformer.

Many tasks in software development have been formulated using the graph transformation approach, including weaving of aspect-oriented programs, application of design patterns, and the transformation of platform-independent models into platform specific models (Please refer to Section 5.7 for a selective list of related works on model transformations).

2.2.2 Some Prominent Graph Transformation Tools

AToM3 [29] is a visual Meta-Modeling tool written in Python, which supports modeling of complex systems, characterized by possibly large numbers of components and aspects whose structure as well as behavior cannot be described in a single formalism. Using the metamodels, AToM can automatically generate a tool to process models. Manipulations of models can be expressed as graph grammars, at the meta-level. Some of these manipulations are the behavior-preserving transformations of models between formalisms, optimization, code generation and simulation.

AGG (Attributed Graph Grammars) [30] is a rule based visual language supporting an algebraic approach to graph transformation. It aims at the specification and prototypical implementation of applications with complex graph-structured data. AGG may be used (implicitly in "code") as a general purpose graph transformation engine in high-level Java applications employing graph transformation methods. The tool environment provides graphical editors for graphs and rules and an integrated textual editor for Java expressions. Moreover, visual interpretation and validation is supported.

BOTL (Bidirectional Object oriented Transformation Language) [31] allows to specify transformations among object oriented models and to verify the desired properties of applicability and metamodel conformance at specification time. BOTL is proposed as a language for the specification of mappings between the different model layers of the MDA. However, BOTL can be easily extended to specify transformations on a single model.

VIATRA2 (VIsual Automated model TRAnsformations) [32] is framework that provides a general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains. It provides a transformation language with both declarative and imperative features, based upon popular formal mathematical techniques of graph transformation (GT) and abstract

state machines (ASM). It has a high performance transformation engine supporting incremental model transformations, trigger-driven live transformations, and handling huge models (e.g. of 100,000 elements). Generic and meta-transformations (type parameters, rules manipulating other rules) for providing reuse of transformations are amongst its other salient features.

The Atlas Transformation Language (ATL) [7] is a hybrid language (a mix of declarative and imperative constructions) designed to express model transformations as required by the MDA approach to answer the QVT RFP issued by OMG. It is described by an abstract syntax (a MOF meta-model), a textual concrete syntax and an additional graphical notation allowing modelers to represent partial views of transformation models. A transformation model in ATL is expressed as a set of transformation rules. ATL is supported by a set of development tools built on top of the Eclipse environment: a compiler, a virtual machine, an editor, and a debugger. There is an initial library of ATL transformations and number of documentation available in open source from the GMT Eclipse project.

The FUJABA (From UML to Java and Back Again) Tool Suite [8] is an open source tool providing developers with support for model-based software engineering and re-engineering. It is a formal, graphical, object-oriented software system specification language, employing UML class diagrams and specialized activity diagrams, so called Story Diagrams based on graph transformations. It is capable of generating Java code based on the formal specification of a systems' structure and behavior which results in an executable system prototype. In Fujaba metamodelling is done with MOF [16] and transformations specified by triple graph grammars. Finally, Fujaba's easy plug-in mechanism makes it a celebrated and extensible toolkit.

2.3 Generic Modeling Environment (GME)

Generic Modeling Environment (GME) [3][33] is a configurable toolkit for creating domain-specific modeling and program synthesis environments. GME puts the MIC [3] vision into practice. The configuration is achieved through metamodels specifying the modeling language (i.e., "paradigm" in the GME vernacular) of the application domain, which contains the syntactic, semantic, and presentation information regarding the domain. The paradigm defines the family of models that can be created using the resultant modeling environment.

The metamodel for each domain-specific modeling language is defined using the UML-based metamodeling language named MetaGME, which plays exactly the same role MOF [16] plays in UML2 [17].When a metamodel is registered in GME, GME provides a

domain-specific model building environment. The generated environment is then used to build and manipulate domain models. These models can serve as input to various model-driven development activities, including model transformation and code generation. This is called model interpretation in GME parlance.

Apart from the visual model editor, GME provides a generic API, called `BON2`, to access the models by paradigm-specific interpreters. This API exposes the internal representation of the models, which is a network of object instances and links (associations). Using the API, developers are able to programmatically traverse and manipulate a GME model with the same set of capabilities provided by the visual GME environment. The API supports both C++ and Java programming languages. The federate code generator of the second phase transformation shown in Figure 1.1 is implemented using the Java interface.

2.3.1 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 2.2.

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. An `FCO` is used as the abstract base class for these elements in modeling.

`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.

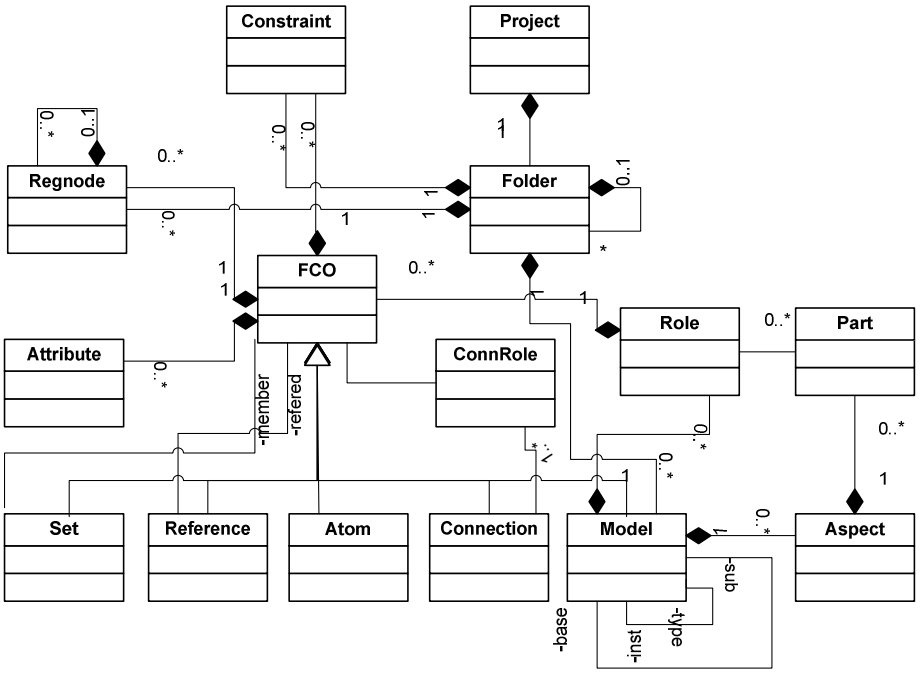


Figure 2.2 GME modeling concepts [33]

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`.

2.4 Graph Rewrite and Transformations (GReAT)

Graph Rewriting and Transformation (GReAT) [6] is a transformation language developed for model-to-model transformations and rewriting. GReAT is based on the theoretical work on graph grammars and transformations [4]. GReAT's metamodel, the `UMLModelTransformer` (`UMT`) paradigm, comes bundled with the GME installation. By creating models conforming to this paradigm in GME, it is possible to define model transformations.

GReAT defines a production (i.e. Rule in UMT terms) as the basic transformation entity. A production contains a pattern graph that consists of pattern vertices and edges. The pattern graph consists of elements from the source and target metamodels and elements that are newly introduced inside the transformation model (such as cross links or globals) Each pattern object has a bind, delete or new designation that specifies the role it plays in the transformation. Bind is used to match objects in the graph. Delete is also used to match objects in the graph, but afterwards they are deleted from the graph. New is 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 other, non-structural constraints on the pattern are needed. These constraints or pre-conditions are expressed in a `Guard` and are described using OCL. `AttributeMapping` elements provide values to attributes of newly created objects and/or modify attributes of existing object. Attribute mapping is applied to each match after the structural changes are completed.

`Rules` are the basic production units, specifying graph patterns in terms of the source and target metamodels. `Rule`s are explicitly sequenced. `Test/Case` is used to specify the conditional execution of a transformation. Compound rules, consisting of `Block` and `ForBlock`, help to modularize transformation sequences and to control traversal schemes. They provide the means to organize rules into higher-level hierarchies. Within a `Block`, rules are chained (and thus sequenced) by passing previously matched elements from rule to rule. Compound rules can contain other compound rules, `Rules` and `Tests`; however, they have slightly different semantics inside. If we have n incoming packets in a `Block` then the all of the packets will be pushed through the first internal rule and then the next internal rule starts. On the other hand, with `ForBlock`, the first packet will be pushed through all its internal rules to produce output packets and then the next packet will be taken. `ExpressionRef` is a reference to a previously defined test or (compound) rule. It opens up the possibility for recursion and rule reuse.

In GReAT, parallel execution of a set of rules can be specified. The order of execution of these rules is non-deterministic. This is achieved by connecting the output of a rule to the input of more than one rule.

GReAT transformations can also specify objects and associations not explicitly present in the input or output metamodels, including cross-metamodel associations. These entities are called `CrossLinks` and their instances exist only as the transformation is being performed.

Defining a GReAT transformation consists of, first importing the source and target GME metamodels, second specifying the graph rewriting rules using the imported metamodel objects, third defining sequencing for the rules by grouping them into rule blocks, and forth configuring the transformation by specifying the source and target models (files) and the starting rule (or rule block).

The model transformation language is supported through the GReAT execution engine as shown in Figure 2.3. The engine basically inputs the transformation definition (i.e. rules and sequencing) and a source model to automatically produce a corresponding target model. The engine uses a generic API using the model-driven reflection package called `Universal Data Model` (`UDM`) [34], and is thus suitable for executing any model transformation that is realized using GReAT. GreAT's rule executor consists of a pattern matcher and an effecter that work in tandem to execute a transformation rule. The graph-based model transformer presented in this thesis employs a user code library written using the `UDM` API for the fast execution of some complex transformation rules.

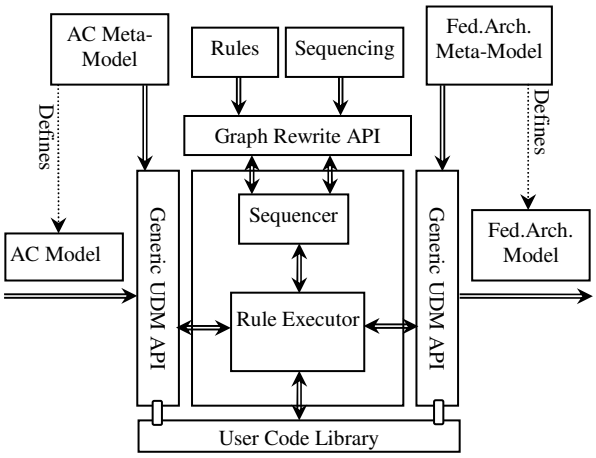


Figure 2.3 GReAT execution engine (adapted from [6]).

2.5 Field Artillery Observed Fire Techniques

This section presents a conceptual overview on the elements and fire direction processes of the observed (i.e., indirect) fire techniques of the Field Artillery (FA) domain. It also

introduces a narration model for the *adjustment followed by fire for effect* mission, which is in the subject of the transformation case study. The content provided in this section is based on the public domain US Army field manuals [35][36][37], which provide comprehensive explanations on tactics, techniques and procedures for FA fire direction process.

2.5.1 Elements of the Field Artillery Team

The Army field manual FM-50 [37] states, "The general mission of 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), as related in Figure 2.4.

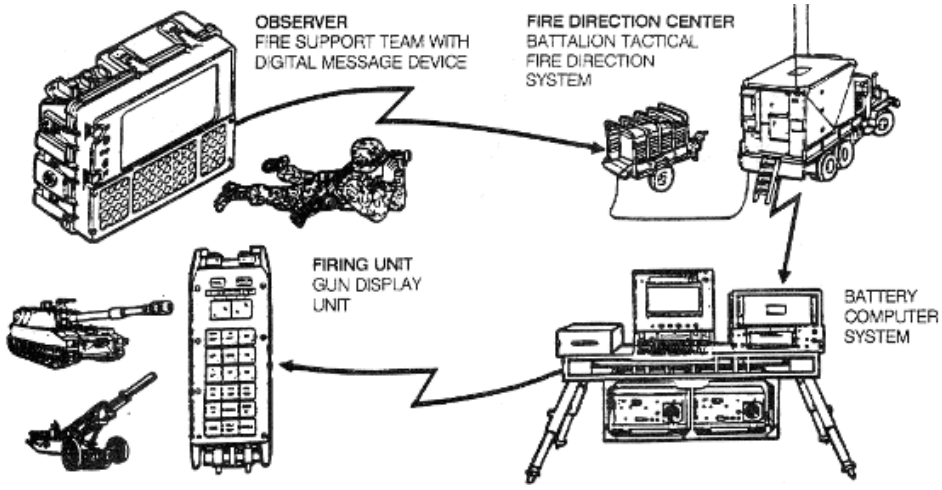


Figure 2.4 The field artillery team [35]

Forward Observer

For artillery and mortar support, fire support team personnel act as the observers, or "eyes", of the FAT. Since we opt for a functional point of view and avoid tackling with domain details, we regard all of the personnel and equipment of the fire support team under the general title FwdObserver. The FwdObserver detects, locates and describes suitable targets and transmits this information to the FDC to request for observed fires. He strives to adjust fires onto targets by providing surveillance data pertaining to the fires. In this study,

the FwdObserver operates under the pre-designated control option, in that he is assigned to a particular battery from which he may request fire support.

Fire Direction Center

In combat, the FA battalion provides indirect fire support to maneuver forces on the battlefield. Among the key components of the battalion, only the battalion FDC functionality is associated with our modeling concerns. The main duty of the battalion FDC is to provide tactical fire planning and fire control. It may also give technical fire direction assistance to battery FDCs as required.

The FA cannon battery is the firing unit within the cannon. The battery FDC is the control center, or "brain", as it were, of the gunnery team. The FDC receives fire orders from the battalion FDC or calls for fire from observers and process that information by using tactical and technical fire direction procedures. Two notable key personnel within the battery FDC are the Fire Direction Officer (FDO) and the FDC computer. The FDO is responsible for all FDC operations including supervising the operation of the FDC, establishing Standing Operating Procedure (SOP), checking target location, announcing fire order, and ensuring accuracy of firing data sent to the guns. The FDC computer operates the primary means of computing firing data. He determines and announces fire commands.

Fire direction is the employment of firepower. Basically there are two types of fire direction methods, called 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 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 [36]. In battalion-directed mode, the battalion FDC is the focal point that carries out tactical fire direction. Technical fire direction is left to the battery FDC. In autonomous mode, the battery FDC is the most prominent actor, being responsible for both tactical and technical fire direction. In this setting, the battalion FDC monitors the radio net and may override battery FDC's commands, take the control over, or abort the mission. The presented case study assumes the autonomous mode.

Firing Unit

The firing unit serves as the "brawn" of the gunnery team. It consists of the firing unit headquarters, firing sections and several other parts. The duty of the firing section is to deliver fires as directed by the FDC. Its composite organization is treated as a single entire unit in our modeling.

2.5.2 Adjustment Followed By Fire For Effect Mission

Observed fire is carried out by the coordinated efforts of the field artillery team, which is composed of the forward observer, the Fire Direction Center(s) (FDC), and several firing sections of the firing unit. The basic duty of the forward observer is to detect and locate suitable indirect fire targets within his zone of observation. In order to start an attack on a target, the forward observer 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, the common practice is first to conduct adjustment on the target. Usually the central gun is selected as the adjusting weapon. The observer provides correction information to the battery FDC after each shot based on his spotting of the detonation. Once a target hit is achieved, the observer initiates the Fire For Effect (FFE) phase by noting this in his correction message. 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 observer 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 observer may request repetitions, or restarts the adjustment phase if deemed necessary. Figure 2.5 presents a simplified sketch of a typical FAT setting as well as the most common communication sequence among the team members.

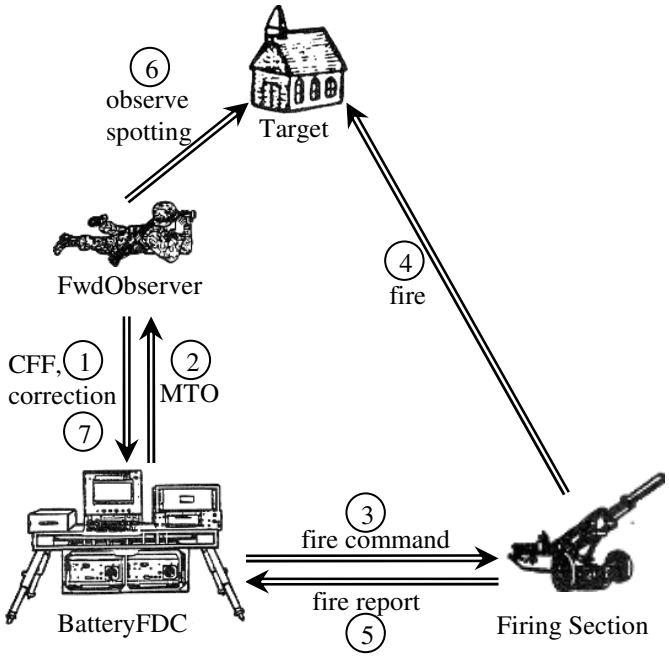


Figure 2.5 Typical Field Artillery Team mission setting

2.6 High Level Architecture

The HLA related background material of this section is based on IEEE standards [13][38][39][40]. HLA is the common architecture that combines simulations (also called federates) into a larger simulation (also called a federation). It is based on the publish/subscribe paradigm. A federation execution is a session of a federation executing together. A federation has a name, and involves:

- supporting middleware called Runtime Infrastructure (RTI)
- a common object model for the data exchanged between federates, called FOM
- member federates

A federate is a member of a federation, one point of attachment to the RTI. A federate may correspond to one platform, such as a cockpit simulator, or a combined simulation, such as an entire national air traffic control simulation.

Federates and the RTI are software. The Federation Object Model (FOM) is the data created by the federation developer typically by using a tool. The FOM states an agreement on the data exchanged among the participating federates.

The relationship between the software components is presented in Figure 2.6. Federates are shown in the figure as either simulations, surrogates for live players, or tools for distributed simulation such as data collectors and passive viewers. A federate might consist of several processes, possibly running on different computers. A federate might model a single entity, like a vehicle, or many entities, like all the vehicles in a city.

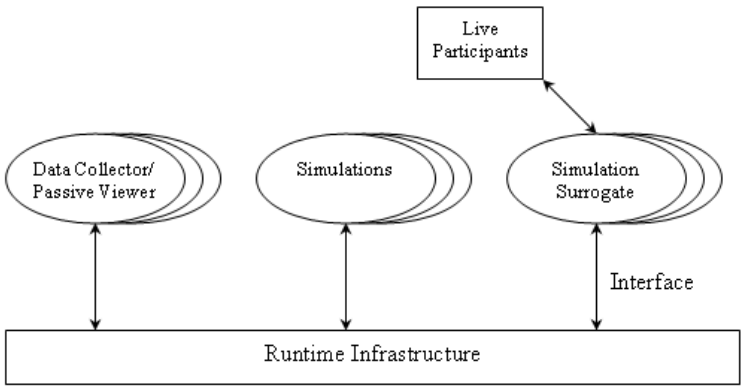


Figure 2.6 Software Components in the HLA [13]

HLA is foremost a software architecture, rather than a particular implementation of an infrastructure or tools designed to work with it. The HLA standard supports a variety of

implementations. Therefore, it is defined not by software, but by a set of documents. The HLA standard has three parts:

- Object Model Template (OMT)
- HLA Rules
- Interface Specification

At the time of this writing, there are two parallel efforts in progress for the adoption of HLA by standards bodies. One is through the Object Management Group (OMG), which has adopted version 1.3 of the HLA interface specification as "Facility for Distributed Simulation Systems (FDSS)". The other is through IEEE, of whose standards are HLA Framework and Rules [13], Federate Interface Specification [38], and OMT [39].

2.6.1 The Object Model Template (OMT)

The OMT advises the structure of all FOMs. The FOM is the vocabulary of data exchanged through the RTI for an execution of the federation. Hence, the FOM does not describe data internal to a single federate, but data that are shared with other federates. The main components of the OMT are interaction classes and object classes.

An interaction is a collection of data sent by a federate at one time through the RTI to other federates. An interaction may represent an occurrence or event in the simulation model of interest to more than one federate. An interaction may be defined to occur at a point in simulation time. A federate sends an interaction; other (interested) federates receive the interaction. The interaction is transitory in that it has no continued existence after it has been received. Each interaction carries with it a series of named data called parameters.

Objects in the RTI refer to simulated entities that are of interest to more than one federate. They persist or endure for some interval of simulated time. Object classes are comprised data fields called attributes.

The OMT describes the instances of the classes. Each class has a name, and defines a set of named data called attributes. Federates create instances of these classes, and change the state of an object instance in simulation time by supplying new values for its attributes. Federates indirectly communicate with each other in terms of interactions and objects through the RTI. Each federate must make some conversion from its internal representation of simulated entities to HLA objects as specified in the FOM. If the federate is HLA-compliant, the translation may be straightforward; otherwise it may be more complicated. In short FOM represents the common, agreed vocabulary between members of a federation.

2.6.2 HLA Rules

The HLA rules express design goals and constraints on HLA-compliant federates and federations. The first five rules deal with federations, the latter five with federates.

2.6.3 The Management Object Model (MOM)

HLA federations are typically distributed systems. Federates often run on many computers. Thus federations are subject to the peculiarities associated with distributed systems. The RTI offers facilities to maintain and manage a shared view of the federation as a distributed system. Management data can be described and distributed just like simulation data. It allows the RTI to describe and manage the state of a federation.

The RTI itself creates the instances and updates attribute values associated with the MOM. System management can be accomplished through the use of federates designed for this purpose. Because the MOM is the same for all federations (since it is RTI managed), management federates can be reused.

The MOM also defines a set of interactions that can be used to affect the state of other federates. The RTI is required to respond correctly to MOM interactions. These interactions are used to regulate the federation's operation, request information, and report on federate activities.

2.6.4 The HLA Services

HLA services fall into six groups that are defined by the commonality of interest.

(i) Federation Management

Federation services manage a federation in two ways:

- By defining a federation execution in terms of existence and membership
- By accomplishing federation-wide operations.

To define a federation, there are services to create a federation execution and to allow a federate to join the execution or resign from it. Every federate must join a federation execution.

Federation-wide operations include the coordination of federation saves and restores. There are also services to allow a federation to define and meet a federation-wide synchronization point.

(ii) Declaration Management

The declaration management services are the way for federates to declare their intent to produce (publish) or consume (subscribe to) data. The RTI uses these declarations for routing data, transforming data, and interest management. On the subject of routing, the RTI uses subscriptions to decide what federates should be informed of the creation or update of entities. Received data go through reduction and re-labeling in accordance with the federate's subscriptions before being delivered. Finally, the RTI uses declarations to indicate interest to publishing federates. The RTI can tell a federate whether any other

federate is subscribed to data it intends to produce, so that it can stop producing when no other federate needs the information.

(iii) Object Management

Object management services are used for the actual exchange of data. A federate uses services from this group to send and receive interactions. These services are also used to register new instances of an object class and to update its attributes. Other federates will have services from this group invoked on them to receive interactions, discover new instances, and receive updates of instance attributes. Other services of this group are used to control how data are transported, to ask for new updates of attribute values, and to inform a federate whether it should expect data.

(iv) Ownership Management

The ownership management services in the RTI implement the HLA's notion of responsibility for simulating an entity. The RTI ensures that at most one federate at a time owns a given instance attribute. Responsibility for simulating an entity can be shared between federates in two ways.

- First, the complete modeling of an entity may be shared among federates.
- Second, the modeling of entities may pass from one federate to another in the course of a federation execution.

Ownership management can be ignored if a federation does not need it.

(v) Time Management

While federates are executing in their own threads of control, the proper ordering of events between federates is an important problem to be solved. In HLA, ordering of events is expressed in "logical time". Logical time is an abstract concept; it is not necessarily fixed to any representation or unit of time. The RTI's time management services do two things:

- They allow each federate to advance its logical time in coordination with other federates.
- They control the delivery of time-stamped events so that a federate never has to receive events from other federates in its past.

(vi) Data Distribution Management

Data distribution management (DDM) services control the producer-consumer relationships among federates. Whereas the declaration management services manage those relationships in terms of interaction and object classes, DDM manages in terms of instances and abstract routing spaces.

(vii) Support Services

Support services utilized by joined federates for performing name-to-handle and handle-to-name transformation, setting advisory switches, manipulating regions and RTI startup and shutdown.

2.7 Message Sequence Chart and Live Sequence Chart

LSC is the formalism used for behavior representation in both source and target models of this model transformation work. Since LSC is derived from MSC and share many similarities with it, MSC is introduced before LSC. Illustrative examples of the graphical MCS/LSC notations are provided in Section 4.4 along with the AdjFFE model demonstration. For a clearer understanding the reader is encouraged to refer to these examples while reading each paragraph of this section. Note that the MSC/LSC features that are not used in this thesis are omitted. For a more extensive coverage, see [15][14].

2.7.1 Message Sequence Chart

An MSC consists of a collection of instances. An instance represents an abstract entity on which events can be specified. Events are message inputs, message outputs, actions, conditions, timers and co-regions. An instance is denoted by a hollow box with a vertical line extending from the bottom. The vertical line represents a time axis where time runs from top to bottom. Each instance thus has its own time axis and time may progress independently and at different speeds on two axes.

An MSC can be referenced from within another MSC. This nesting and referencing mechanisms facilitate encapsulation and modular design principles. MSC references may have actual parameters that must match the corresponding parameter declarations of the MSC definition. MSC references must not directly or indirectly refer to their enclosing MSC. References are represented by rounded rectangles.

The gates represent the interface between the MSC and its environment. Any message or order relation attached to the MSC frame constitutes a gate. The message gates are used when references to the MSC are put in a wider context in another MSC. The actual gates on the MSC reference are then connected to other gates or instances.

Events specified on an instance are totally ordered in time, except in coregions (see below). An event executes instantaneously, and two events cannot take place at the same time. Events on different instances are ordered due to the requirement that message input by one instance must be preceded by the corresponding message output in another instance. All events in a chart form a partially ordered set. (Recall that a partial order on a set is a binary relation that is reflexive, anti-symmetric and transitive.)

Actions are events that are local to an instance. Actions are represented by a box on the instance axis with an action label inside. Actions are used to specify some computation performed by the instance.

A message output/input represents the sending/reception of a message to/from another instance or the environment. A message exchange is represented as an arrow from the instance axis of the sender to the instance axis of the receiver. The arrow is labeled with a message identifier. Message exchange is, by default, asynchronous; that is, the message input is not necessarily simultaneous with the message output.

There are two types of conditions, namely, setting and guarding conditions. Setting conditions are intended to describe a current global system state, or some non-global, possibly shared, state. Guarding conditions restrict the behavior of an MSC by only allowing the execution of events in a certain part of the MSC. A condition is represented by a hexagon extending across the instance axes for which it holds.

Timers are local to an instance. The setting of a timer is represented by an hourglass symbol placed next to the instance time line and labeled with a timer identifier. Timer reset is represented by a cross linked by a horizontal line to the time line. Timer timeout is represented by an arrow from the hourglass symbol to the time line.

Coregions are parts of instance axes where the usual requirement of total ordering is lifted. A coregion is shown as replacing a part of the instance axis with a dashed line.

Inline expressions are used to compose event structures inside an MSC. The inline operators refer to alternative, parallel and sequential composition, iteration, exception and optional regions. A frame encloses the operands and the dashed lines denote operand separators. Extra-global inline expressions are those crossing the MSC frame and covering all of the instances. They are associated with the corresponding inline expressions on the enclosing (see below) instance.

For enhancing the modularity of MSCs, there is a form of hierarchical decomposition of complex diagrams into a collection of simpler diagrams, known as instance decomposition. For each decomposed instance there is a sub-MSC. The single instance that is decomposed is represented by more than one instance in the sub-MSC.

High-level MSC (HMSC) provides a means to graphically define how a set of MSCs can be combined together. The HMSC incorporates sequencing, conditioning and inline expressions that are interpreted much similar to the ones found in MSC.

2.7.2 Live Sequence Chart

The most prominent feature of LSC on top of MSC is the ability to make a distinction between optional and mandatory behavior. This applies to several elements in charts.

Universal charts specify behavior that must be satisfied by every possible run of a system, whereas for existential charts this restriction is relaxed to at least one run. Universal charts are denoted by a solid box around the chart and existential charts are denoted by a dashed box.

LSC introduces the notion of a prechart to restrict the applicability of a chart. The prechart is like a precondition that when satisfied activates the main chart. The prechart is denoted by a dashed hexagon containing zero or more events.

LSC allows messages to be "hot" or "cold". A "hot" message is mandatory; that is, if it is sent then it must be received eventually. This is denoted by a fully drawn arrow. For a "cold" message reception is not required, hence it may be "lost". This is denoted by a dashed arrow. A distinction is also made between a "hot" (i.e., mandatory) and a "cold" (i.e., optional) condition. A "hot" condition causes an illegal termination of the chart if evaluated to false, and the opposite (i.e., exit from the condition scope) holds for a "cold" condition. "Hot" and "cold" notions are further applied to the instance axes. Any point where an event is specified on the instance axis is called a location. A location may be "hot" indicating that the corresponding event must eventually take place, or "cold" indicating that event may never occur. A "hot" and a "cold" location is represented by the instance axis being fully drawn and dashed, respectively.

LSC further brings enhancements in the semantics of conditions and event occurrence. A shared condition forces synchronization among the sharing instances; that is, condition will not be evaluated before all instances have reached it and no instance will progress beyond the condition until it has been evaluated. Simultaneous regions allow grouping several elements, which should be observed at the same time.

Chronologically, the last set of enhancements to LSCs are the notion of time (and a sort of real time), and a notion of genericity via variables and symbolic instances [43].

2.8 Overview of Federation Architecture Metamodel

FAMM is a proposed metamodel for specifying the architecture of an HLA-compliant federation [12][85]. FAMM formalizes the standard Object Model and Federate Interface Specification. Beyond formalizing the existing HLA standard, FAMM allows the behavioral description of federates based on LSCs. Having the behavioral models of the participating federates gives us the ability to test the federation architecture by executing the federation.

Federation Architecture is a major portion of the federation design documentation in HLA based distributed simulations. Federation design includes the activities for:

- Forming HLA Object Model (federation and simulation object models):

- Specifying the behaviors of participating federates so that they can fulfill their responsibilities within the federation

The Federation Architecture Model (FAM) for a particular federation conforms to FAMM. It involves the Federation Model (Federation Structure, Federation Object Model and related HLA Services) and the Behavior Models for each participating federate.

As the composition diagram in Figure 2.7 indicates, FAMM involves two main sub-metamodels: One for specifying the observable behaviors, and the other for defining the HLA FOM and the HLA service interface.

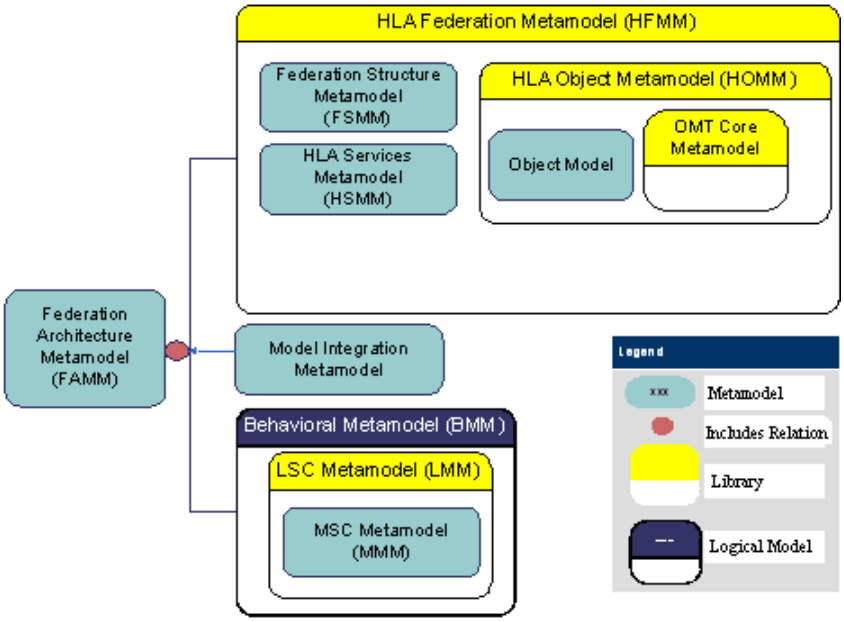


Figure 2.7 Federation Architecture Metamodel structure ([12])

Figure 2.8 depicts the relationship between FAMM and Federation Architecture. Each participating federate's behavior is modeled using the behavioral metamodel while the FOM is described by using the HLA Object Metamodel. HLA Object Metamodel (HOMM) is a formalization of HLA Object Model Template (OMT) [39]. The OMT Core folder includes the table contents specified in HLA OMT.

Federation Structure Metamodel (FSMM) represents the structural aspects of the federation. This metamodel allows the developer to define a federation and its participating federate applications, and to readily connect them to their respective FOM and SOMs. In this sub-metamodel, the participating federate applications are emphasized and their

corresponding SOMs can be specified in addition to the FOM. The FOM and SOMs that are referred by FSMM are prepared with HOMM.

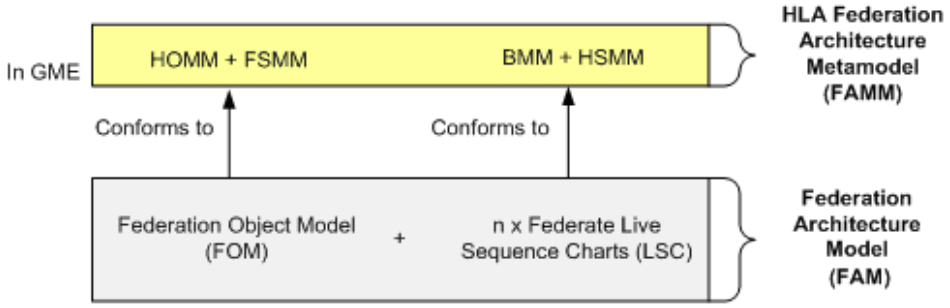


Figure 2.8 Relationship between a FAM and its metamodel ([12])

The HLA Services Metamodel (HSMM) defines the interface of the standard services of Runtime Infrastructure (RTI). These management services provide a functional interface between federates and the RTI. These interfaces arranged into seven basic groups are as follows: Federation management, declaration management, object management, ownership management, time management, data distribution management, and support services [38].

Behavioral Metamodel (BMM) provides an abstract syntax for specifying the dynamic and the observable behaviors of a federate. Modeling the behavior of a federate can involve not only the HLA-specific behavior such as creating regions, but also the interactions between the components of the federate and the live entities (e.g., the user) in the environment. The observable behaviors of a federate are represented using Message Sequence Charts (MSCs) and Live Sequence Charts (LSCs) in the metamodel.

LSC is a graphical language introduced by Harel and his colleagues [14][42], as an extension of MSC, for specifying the patterns of interactions between components in a concurrent system. MSCs are widely used in the specification of telecommunication systems. The MSC language is standardized by ITU [41], the most recent standard being Recommendation Z.120 [15]. Many features of MSCs are adopted in the UML sequence diagrams. LSC extends MSC by providing notations for distinguishing mandatory and optional behavior and by promoting conditions to first class elements.

LSC metamodel defines basic LSC concerns such as instance, event, message, parallel, alternative, loop and interconnection between these concerns in the meta-level. These concerns are matched to the first class objects such as folder, atom, model, reference and connection, which are defined in the Generic Modeling Environment (GME).

LSC instances can represent federation executions, federates (possibly, with their constituent modules), live entities such as interactive users and environments. An LSC document which includes one or more LSC diagrams represents a federate's behavior. Federate application code is generated for the given LSC document. A federate may have some constituent modules whose behavior we might prefer to model explicitly. Each such module is represented by an instance in the LSC model, and code is generated specifically for it (Please refer to Section 5.5 for code generation per LSC instance).

CHAPTER III

THE CONCEPTUAL FRAMEWORK

This chapter aims to provide a conceptual framework for the model-driven engineering work, including metamodeling and model transformation, presented in this thesis, before delving into the nuts and bolts of the particular application presented at length in the subsequent chapters. The content is abstracted away as much as possible from the details and jargon of the specific domains, tools and technologies used in an effort to facilitate comprehensibility and appeal to a broader range of readers and potential adopters.

We present a formal, multi-stage model transformation endeavor from a domain Conceptual Model (CM) to a Distributed Simulation Architecture Model (DSAM), and from that, to executable simulation codes and supporting artifacts. Referring to the MDA terminology, CM and DSAM constitute the Platform-Independent Model (PIM) and Platform-Specific Model (PSM) of the model transformation work, respectively. The end-to-end transformation process is depicted in Figure 3.1 to be elaborated on in subsequent sections. CMs and DSAMs are formally built due to compliance with their metamodels CMM and DSAMM, respectively. The transformation is defined over these metamodels.

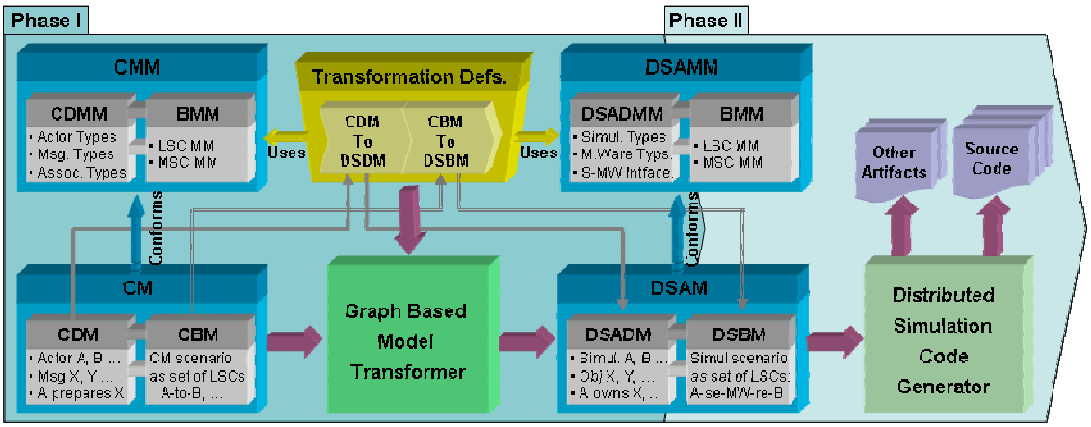


Figure 3.1 The overall model transformation process

3.1 The Models

This section introduces the source and target models, particularly the conceptual data model, the distributed simulation architecture data model and the behavioral model, which is employed by both of the source and target models.

3.1.1 The Conceptual Data Model

The CMM's data model, CDMM, consists of a set of domain entities called actors, which are able to perform computations and receive/send messages (from/to other actors and the environment) on a one-to-one or multi-cast basis. The multi-cast communication media are called nets, which are represented simply as sets of references to actors. The communicated messages are collections of domain information, extracted from authoritative sources and composed in different granularities.

The messages can be categorized as being durable or non-durable. This durability distinction facilitates the transformation definitions for target PSMs of distributed simulation domain, such as HLA because there, this distinction between message communications matters. Durable type of messages represent information that is intended to be kept and maintained for a duration by the receiver. Non-durable type of messages represent information that is meant to be immediately used and then forgotten by the receiver (barring, of course, logging).

The upper level elements of the CDMM and their associations are sketched in the UML diagram of Figure 3.2. In the figure, the `Model` elements are the primary building blocks of the communicated data and can be organized recursively to accommodate for composite structures. The `Folder` elements are containers that are similar to folders found in computer file systems and are used to maintain model components organized. The data model is buildup of `Messages`, `Actors` and `DurableDataStore` folders. The messages, consisting of durable and non-durable types, are stored in the `Messages` folder. The durable data messages are further specialized into *instantiation*, *update* and *delete* types. Since the objects corresponding to durable data messages need to be maintained throughout system life time, they are kept in the `DurableDataStore` folder. An *instantiation* type of durable message contains the original copy of the durable data (i.e., persistent object) to be placed in the store for the first time. Subsequent *update* messages contain template objects that are used to update the effective copy residing in the store. The message indicates the corresponding persistent object to be deleted from the store. The `Actors` folder keeps the domain elements of type `Actor` and `Net`. `Net` is a special kind of `Actor` and is treated the

same as a source or destination for message communications throughout the mission scenarios realized in CMM.

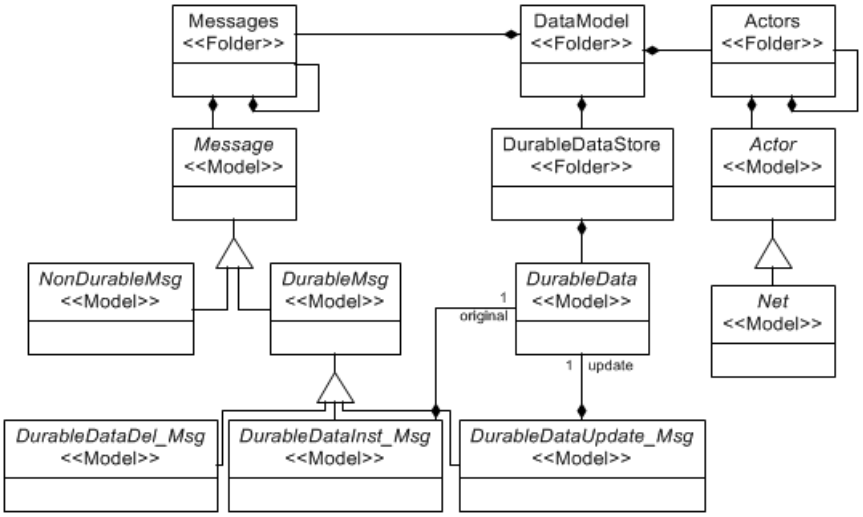


Figure 3.2 The upper level CDMM elements

3.1.2 The Distributed Simulation Architecture Data Model

The DSAMM's data model, DSADMM, consists of elements that collectively define the static view of a set of autonomous and loosely coupled interoperating simulations. The interactions are mediated via the simulation infrastructure, or middleware. The middleware functions as the overarching manager, knows about the identities and data exchange interests of the participating simulations and orchestrates all of the communication traffic, whether being one-to-one or one-to-many. To be more concrete, the individual simulations in an HLA-based distributed simulation [13][38][39] are called *federates*, the middleware is called the *Run-Time Infrastructure* (*RTI*) and all of this simulation environment, along with a common simulation data exchange model, are collectively called the *federation* The DSADMM defines the structure and organization of the communicated data as classes of simulation objects in a simulation data exchange model, categorized by having lifetimes of single interactions, or the whole simulation.

The prominent elements of the DSADMM and their associations are depicted in Figure 3.3. In addition to `Folder` and `Model` types, the DSADMM introduces `Connection` types, which are association classes between two model elements. The simulation data model consists of the simulation environment, a number of simulation members, which are

"members of" the simulation environment and a simulation data exchange model. The data exchange model houses instances of simulation classes, which represent the data structures communicated within the overall simulation environment. The simulation class is specialized into simulation object and simulation interaction types, of which the former is intended to model persistent information and the latter is intended to model instantaneous events. In a similar vein, objects are associated with durable data messages and interactions are associated with non-durable data messages defined in CMM. The simulation classes contain attributes having data types defined in the specific distributed simulation domain. For instance, HLA has a default set of simple, enumeration, array and record data types. The simulation environment "manages" and has an overview of the overall communication taking place among the simulation members. The simulation members "use" a set of simulation classes, which they produce or consume in order to share data with each other.

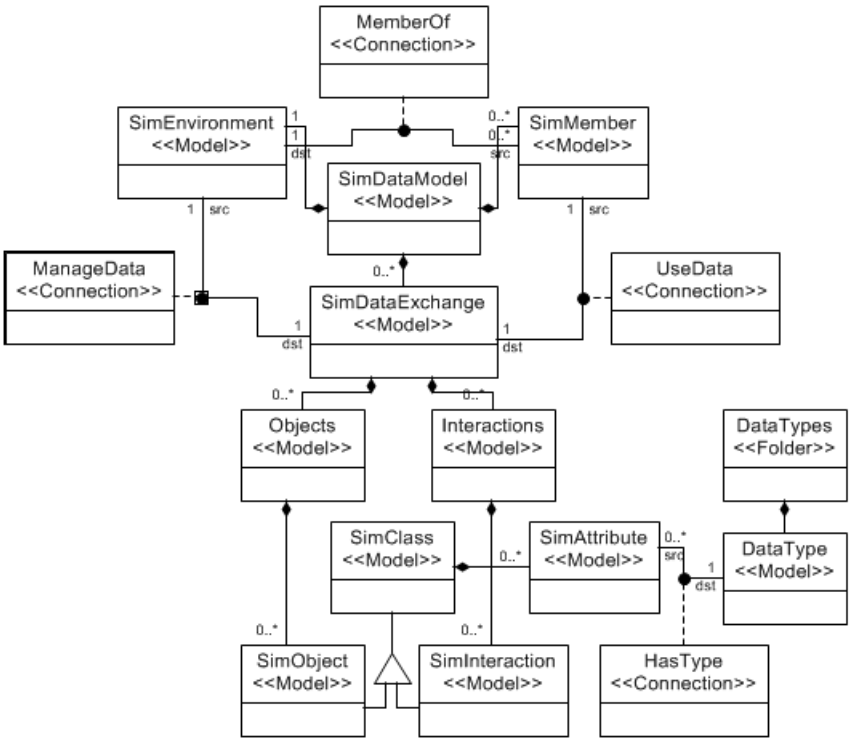


Figure 3.3 Prominent DSADMM elements

3.1.3 The Behavioral Model

The behavioral metamodel, BMM, is used in both of the source and target models. BMM is a representation of the LSC/MSC formalism, which is comparable and shares

many constructs with UML sequence charts. LSC is based upon MSC [41] and extends it with various means for distinguishing between possible, necessary and forbidden behavior [42][43]. The behavioral metamodel is capable of representing the discrete communication behavior of many practical systems, consisting of components exchanging messages, independently of the domain. This communication aspect of the system behavior is particularly emphasized from the LSC modeling perspective. A simplified illustration of the upper-level and outstanding elements of the BMM and their associations are provided in the UML class diagram of Figure 3.4.

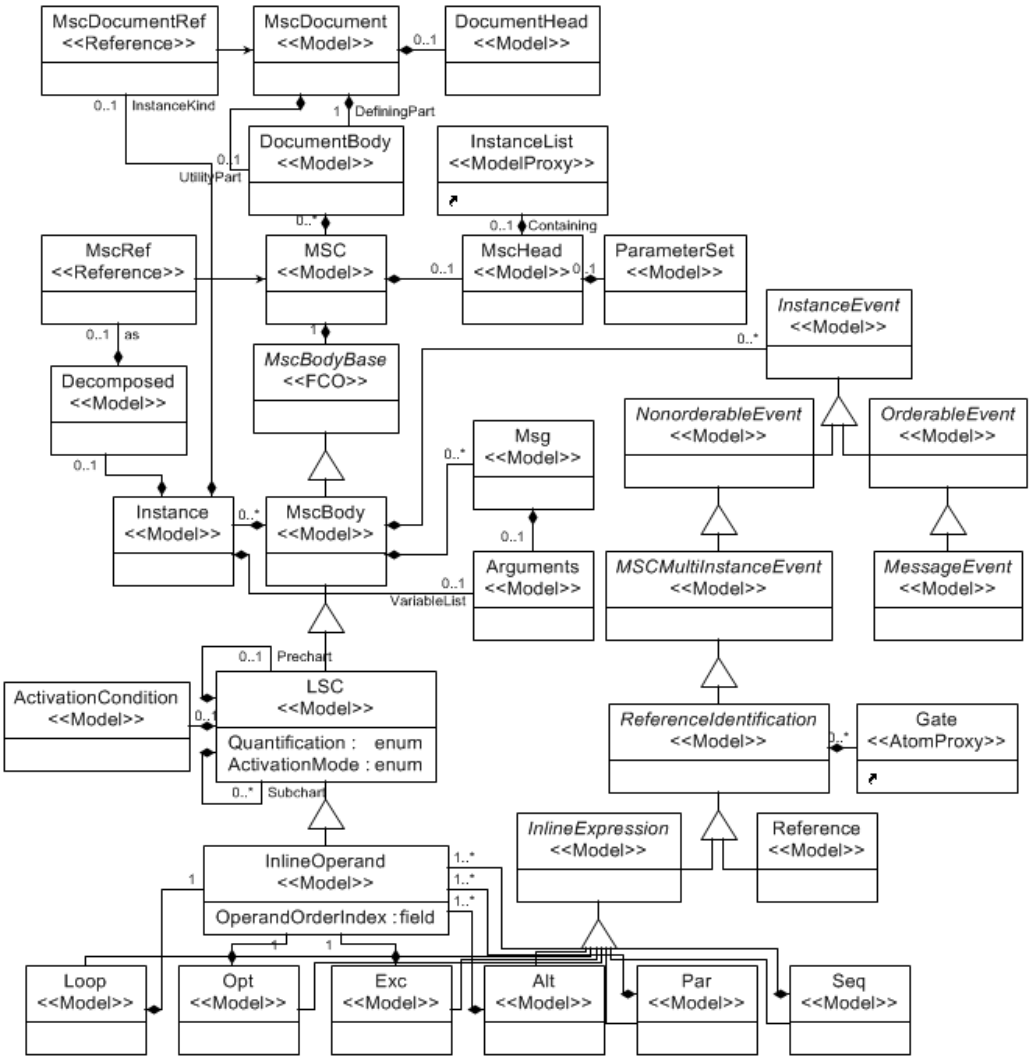


Figure 3.4 Simplified illustration of upper-level BMM elements

The complete behavioral specification of a system can be viewed as a global description of its components from the communication/interaction viewpoint. This system specification is captured in a single MSC document, which consists of a document head and one or two document bodies. The head part includes declaration lists for the instances, messages and timers used in the document and optionally a reference to another document that it "inherits" from (not shown in the figure). The body part of the document is modularized into a set of MSCs. Each MSC, similar to the MSC document, but pertaining to only its own scope, has a head and a body. LSC is the most commonly used MSC body type and is the primary means for representing the behavioral specification of the system being modeled. The LSC contains, besides others, a set of references to the instances that interact with each other using a rich variety of instance events. An important and relevant event group from a model transformation perspective is the message event, which provides the mechanism to exchange data between the instances in the form of LSC messages. LSC is recursively defined and is allowed to refer to other MSCs in order to favor better modularizing and componentizing big behavioral descriptions. Inline operand, which is defined to be specialized from LSC, is the main building block of the non-orderable, multi-instance type of events called inline expressions. Inline expressions include constructs for defining loop, optional, exceptional, alternative, parallel and sequential flows in a behavioral specification. The language of LSC (or MSC or UML sequence diagrams, for that matter) is, to a great extent, expressive enough for comprehensive specification of systems, although in practice they are often used to represent particular use-cases, scenarios or traces of systems.

3.1.4 Model Integration

The data and behavior models of both CMM and DSAMM are stand alone, separately built sub-models. LSC provides a generic infrastructure for modeling the discrete communication behavior of a system as a partially ordered set of events (mainly as message passing) between a group of instances. In the context of a specific domain, these generic behavioral elements need to be specialized as the domain's entities. The specializations are naturally derived from instance, message and other elements of the LSC metamodel. The integration of the behavioral and data models is thus achieved by extending the relevant data model elements from the behavioral model elements in the sense of UML inheritance.

The integration points of the behavioral and data models of CMM and DSAMM are shown in Figure 3.5. Specifically, on the CMM side, `Actor` and `Message` of CDMM inherit from `Instance` and `Msg` of BMM, respectively. On the DSMM side, `SimEnvironment` and `SimMember` of DSADMM inherit from `Instance` of BMM,

`SimClass` of DSADMM inherits from `Msg` of BMM, and `SimAttribute` of DSADMM inherits from `Argument` of BMM.

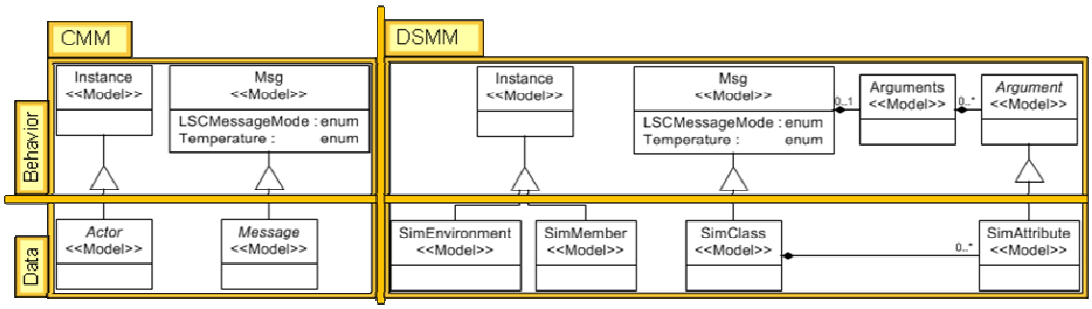


Figure 3.5 Integration of data and behavior in conceptual and simulation models

3.2 The Model Transformations

3.2.1 Overview of the Model Transformer

The graph-based model transformer, which is based on the theoretical work on graph grammars and transformations [4], produces a DSAM from a CM. An overview of the architecture of the transformer is illustrated in Figure 3.6.

The model transformer interprets both the CM and the DSAM as vertex and edge labeled multi-graphs (i.e., graphs that are permitted to have edges that have the same end vertices), where the labels denote the corresponding entities in the metamodels. Then the model transformation work is formulated as a graph transformation problem defined over the source and target metamodels. The model transformer defines a production (i.e., transformation rule) as the basic transformation entity. A production contains a pattern graph that consists of pattern vertices and edges, which are elements from the source and target metamodels (called LHS and RHS patterns in graph transformation vernacular). Each pattern object has a *bind*, *delete* or *new* designation that specifies the role it plays in the transformation. Bind is used to match objects in the graph. Delete is also used to match objects in the graph, but afterwards they are deleted from the graph. New is used to create objects after the pattern is matched. Sequencing is accomplished by grouping transformation rules into recursively defined blocks and connecting these rules and blocks in sequential, parallel or conditional branching organizations.

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 other, non-structural constraints on the pattern are needed. These constraints or pre-conditions are expressed in special *guard* expressions.

The transformer also provides access to a programming API, that can be used further to manipulate and fine tune the generation, after the structural changes are completed in a rule execution. This extra mechanism is incorporated by invoking user code library methods from within transformation rules. The user code library is written to facilitate model transformations in terms of improved execution performance and saving from the tedium of graphically defining many uninteresting transformation rules.

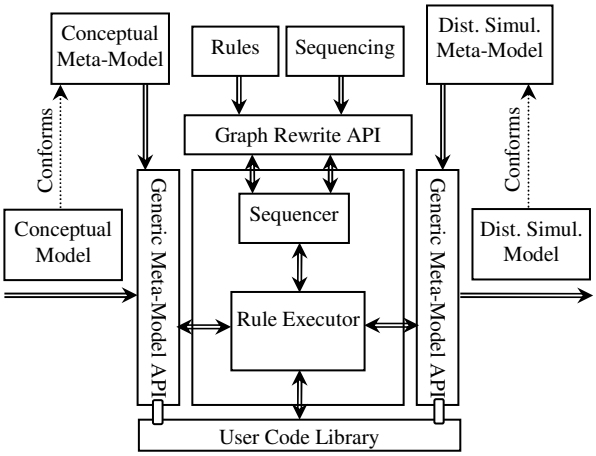


Figure 3.6 Overview of the architecture of the model transformer

3.2.2 Key Elements of the CM to DSAM Transformation

Adopting a parallel design principle, the CM to DSAM transformation is essentially formulated around the core of data and behavior model transformations, executed in sequence. Before and after these core blocks, come the smaller sets of pre and post rules that set up and tear down the stage for the more platform specific distributed simulation environment. There are also preliminary transformation steps using both data and behavioral models that produce temporary structures to be utilized in subsequent transformation rules. This approach to CM-to-DSAM transformation is illustrated in Figure 3.7. The behavioral transformation generally traverses the top-down LSC structure, starting from the MSC document and going down to individual LSCs and the events inside the LSCs (please refer to Figure 3.4 for LSC/MSC structure). Since the top-level data model

elements are extended from LSC elements, the LSC transformation implicitly covers the data model elements as well.

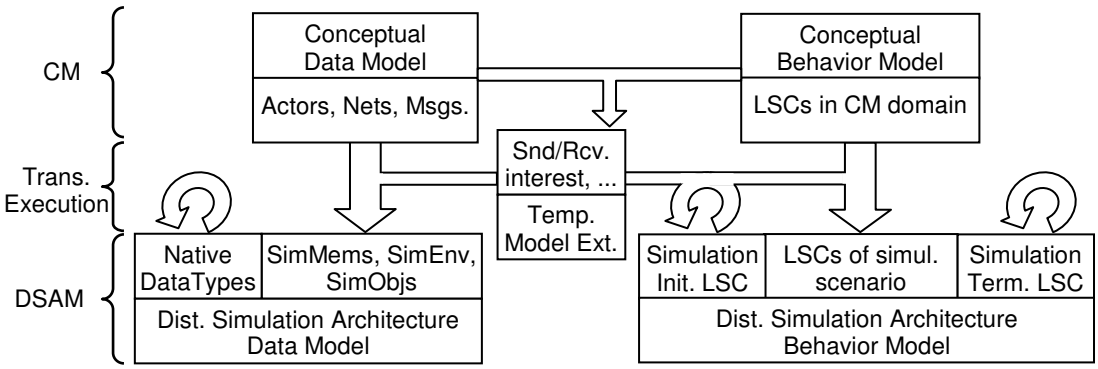


Figure 3.7 An overview of CM to DSAM transformation

The set of key transformation steps are enumerated in the list below and the key mappings done from the CM to DSAM during the transformation process are summarized in Table 3.1.

- every actor is mapped to a simulation member;
- every non-durable message is mapped to a simulation interaction;
- every durable data element is mapped to a simulation object;
- the simulation environment element is brought in as a collection of communicating simulation members, every actor to actor non-durable message communication is mapped to a simulation member to simulation member communication via the simulation environment (running the middleware), using a pair of send/receive interaction messages;
- every actor to actor instantiation type of durable message communication is mapped to a simulation member to simulation member communication via the simulation environment, using three pairs of register/discover object, request/provide attribute update and update/reflect attributes messages;
- every actor to actor update type of durable message communication is mapped to a simulation member to simulation member communication via the simulation environment, using a pair of update/reflect attributes messages;
- every actor to actor delete type of durable message communication is mapped to a simulation member to simulation member communication via the simulation environment, using a pair of delete/remove object messages;

- the default distributed simulation types (that serve simulation classes) are brought in; simulation environment initialization is introduced in a preliminary LSC by creating the environment, joining the simulation members to the environment, declaring simulation member data exchange interests and other sorts of simulation-specific initializations;
- simulation environment shut down is brought in to the final LSC by resigning the registered simulation members from the simulation middleware and destroying the simulation environment;
- finally, the rest of the CM LSC parts are directly (i.e., one-to-one) mapped to equivalent DSAM LSC parts.

Since the data model elements are mostly composed of hierarchically organized optional and mandatory parts, it is more convenient to perform the details of data transformations using a programming API, rather than capturing all of the possible pattern combinations in separate rules, if possible.

Table 3.1 Summary of mappings from Conceptual Model to Distribted Simulation Architecture Model

CM Component	**DSAM Component**
Actor/Net	Simulation member
Non-durable message	Simulation interaction
Durable message	Simulation object
<NA>	Simulation environment
Actor-actor non-durable comm.	Sm-sEnv-sm send/receive interaction
Actor-actor durable comm. (inst. type)	Sm-sEnv-sm register/discover object + request/provide attribute update + update/reflect attributes
Actor-actor durable comm. (upd. type)	Sm-sEnv-sm update/reflect attributes
Actor-actor durable comm. (del. type)	Sm-sEnv-sm delete/remove object
<NA>	Default distributed simulation types
<NA>	Sim. env. init. LSC (create env., join sim. mems., declare data exchange interests, init. others)
<NA>	Sim. env. destruction LSC (resign sim. mems., destruct sim. env.)
Other CM LSC components	Other DSAM LSC components

It is important to note that this mapping is one of many possibilities. It can be used, for example, to create a first-cut simulator for the modeled domain. Different design decisions can be effected by defining different transformation rules. We argue that for any domain specific conceptual model which can integrate with the presented CMM as an upper level model, the model transformation approach presented in this thesis can be used to automatically generate a corresponding distributed simulation model and code. The supported distributed simulation model is the Federation Architecture Meta-Model (FAMM) [12] which formalizes HLA.

3.2.3 Transforming Message Communications

The crux of the model transformation work presented in this thesis is the transformation of a typical one-to-one direct communication between the actors of a CM. A simplified and abstracted schematic of this transformation involving a non-durable message event communication is illustrated in Figure 3.8. The transformation also demonstrates the mappings of the CM actors and messages onto DSAM counterparts.

The loosely coupled communication architecture of DSAMM would normally necessitate an actor `A` to `B` out-event transmission in a CM to be represented as (simulation) member `A` sending an out-event to the (simulation) environment first and the environment sending another out-event to member `B`. However, instead of having these two explicit outs (and two implicit ins), we have decided to implement one explicit out-event between member `A` and the simulation environment and an explicit in-event between member `B` and the simulation environment, employing both in and out-event types. In this setting, if the out-event has execution order *n*, the in-event is given a higher order, say *n+1*. This member centric event mapping better supports the code generator's code generation strategy which considers each LSC instance (i.e. member) and its associated events individually while producing the member base code and computation aspect code [61].

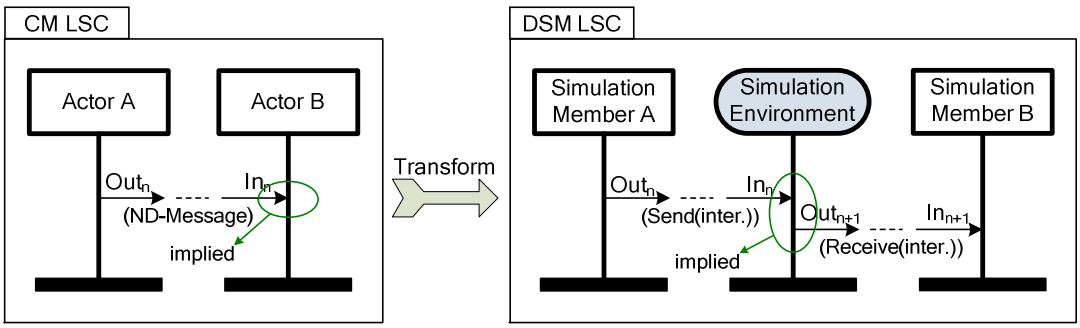


Figure 3.8 Abstracted mapping of a CM message communication to DSAM

Having explained the crucial message communication transformation, it is worthwhile to complement the topic with the higher level and more straightforward simulation scenario generation. The main flow of the transformation follows the organizational LSC/MSC hierarchy of the source model and creates corresponding LSC/MSC components on the DSAM side as progressing along the path. Indeed it would not be completely wrong to call the CM to DSAM transformation generally a LSC transformation. At the end of the transformation, the behavior exhibited in the CM is fully reflected in the produced DSAM. Of course, as visualized in Figure 3.7, there are transformation rules involving DSAMM only patterns that setup and tear down the distributed simulation environment.

3.3 Refining the Simulation Model

The behavioral transformation is a one to one LSC/MSC transformation from CM to DSAM; that is, a corresponding element of the same type is created on the DSAM side for each MSC document, MSC and LSC of the CM. Furthermore, the content of an LSC is transformed as summarized in Sections 3.2.1 and 3.2.2. At the end of the transformation, an equal number of simulation members to the number of actors in a CM LSC plus one simulation environment instance are created in the corresponding DSAM LSC.

A DSAM with this structure does not fully comply with the input requirements of the code generator. As explained in [61], the code generator by design expects and generates code only for one instance (i.e., simulation member) in an LSC. The LSC instance is the focal element in the code generation process, and ultimately code for each LSC instance is generated in separate source files. (Note that the set of LSCs for the same instance type in an MSC document collectively describes the behavior specification of a simulation member corresponding to the instance type in question.) This necessitates a refinement on the generated DSAM, achieved through another DSAM to DSAM transformation named *Multi2BinaryLSC*. The transformation refactors every LSC that contains multiple simulation members and the simulation environment into as many *binary* LSCs as the number of simulation members, each containing one simulation member and the simulation environment. Intrinsically Multi2BinaryLSC accomplishes transformation from a global view of the simulation environment to the collection of local views of the simulation members.

The stripping of multi-instance LSC into binary-instance LSCs of a DSAM is depicted in Figure 3.9. Eventually, every binary LSC only contains its simulation member's mutual communication with the simulation environment – an organization that facilitates per simulation member code generation. Note that the stripping process may end-up in loss of

event orderings in binary-LSCs that were implicitly known in their multi-LSC forms due to transitive chaining of events among the instances.

3.4 Code Generation from the Simulation Model

A produced and refined DSAM is input to the code generator to produce simulation member source codes, simulation environment source code and other artifacts such as simulation configuration. The code generator, which is defined over DSAMM, first traverses a given DSAM using the programming API to generate an intermediate form that facilitates code generation. Then this internal representation is further processed to generate executable code and other products. The heart of the code generator is the generic LSC code generator component, which purely deals with behavior specifications from a communication perspective, independent of the domain concepts they describe. The code generator is specialized into a code generator for the specific simulation domain by way of integrating the underlying domain's object model (e.g., OMT in the case of HLA).

An important feature worth mentioning is the multi-threaded approach taken in code generation. The behaviors of LSC instances that occur in multiple diagrams are handled through parallel threads in the generated code. The behavioral specification of a simulation member can be scattered in multiple LSCs within an MSC document; thus, there are multiple threads of code, each in a separate source file, that describe the execution of a simulation member.

Aspect Oriented Programming (AOP) [62] paradigm is adopted in generating distributed simulation code. The AOP approach provides the separation of cross-cutting concerns. In our case, this allows us to generate code so as to exercise LSCs in a computation-free manner. Then application-specific computational (and other non-communication) aspect advices are to be crafted by the simulation developer; these advices are then woven onto the generated base code by the aspect-oriented programming environment, such as AspectJ. The LSC instance is the focal element in code generation. All LSC instance codes are generated in individual class files and are referenced from the diagram code generated from the LSC itself.

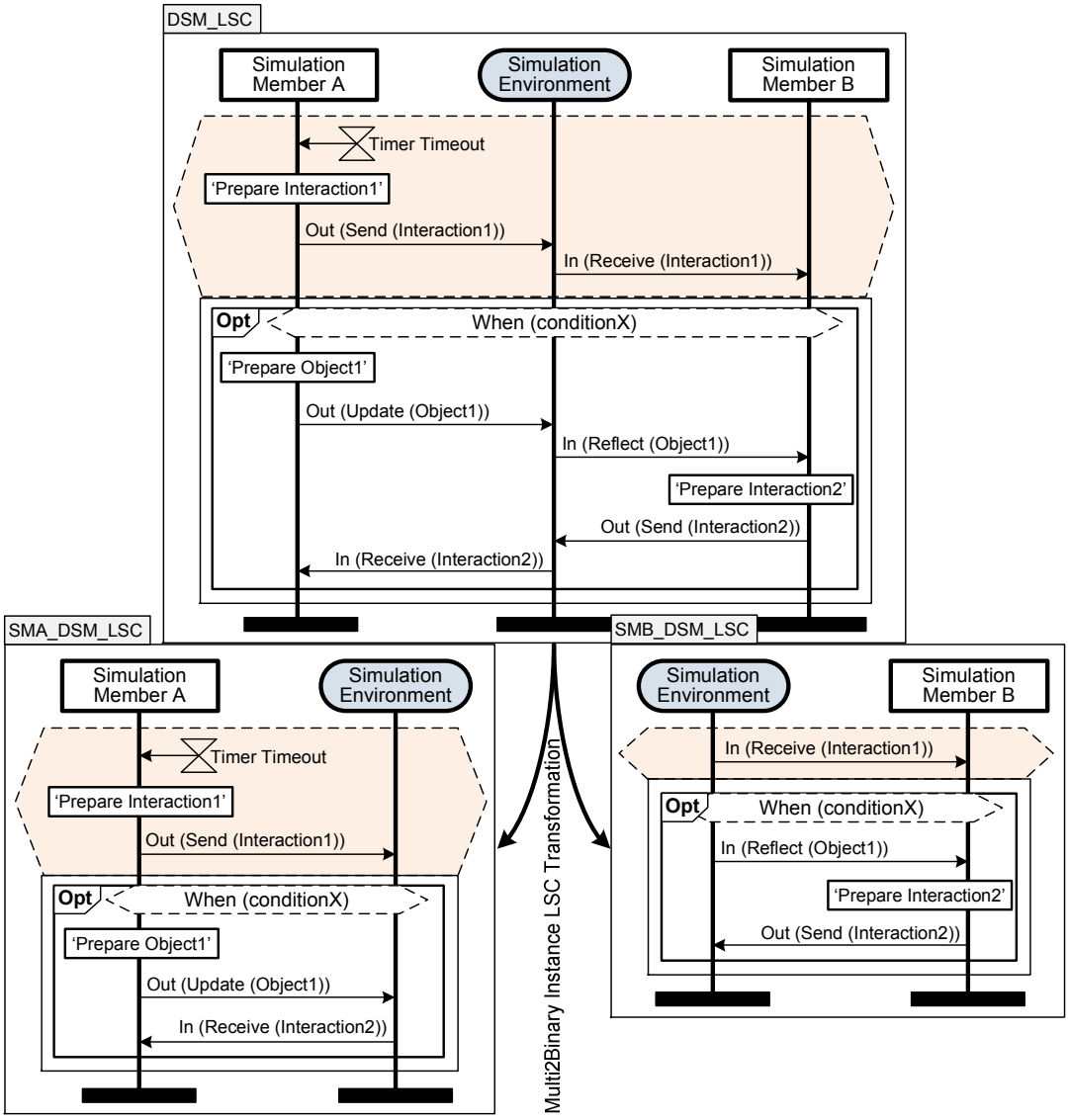


Figure 3.9 Refining a multi-instance LSC into binary-instance LSCs

Figure 3.10 shows the relationship between the generated simulation code files per binary-instance LSC. For every LSC message out-event, a simulation middleware (e.g., RTI in the case of HLA) interface method call is made, and for every LSC message input event, a simulation member interface method callback is generated. The LSC instance aspect code intercepts the middleware interface method calls. It executes developer written computation code and then redirects the call to the middleware with the computation code in effect. On the middleware side, in addition to LSC, an aspect code (middleware instance aspect) is generated for the overall simulation environment. This aspect code catches the middleware callback methods and forwards them to the LSC instance (simulation member)

code. Then in the LSC instance aspect code, the result of the callback (with all arguments) is made available to the developer. The details of the code generator and the code generation process are presented in [61].

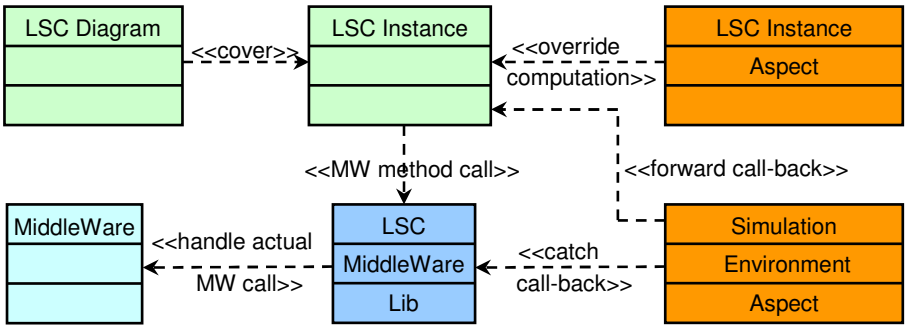


Figure 3.10 Relationship between generated source codes of a binary-instance LSC

3.5 Summary

This thesis presents a comprehensive graph-based model transformation work from a Conceptual Model (CM) to an executable Distributed Simulation Architecture Model (DSAM). The work is undertaken to clearly understand the requirements and challenges of defining transformations from CM-to-DSAM, eventually executing scenarios of conceptual models. Both CMs and DSAMs are formally defined conforming to their metamodels, CMM and DSAMM, respectively. CMM and DSAMM consist of their own separate data models and a common behavioral model. The data components are based on UML class diagram and the behavioral component is based on Live Sequence Chart (LSC). The scenario of a conceptual model is represented by LSC diagrams and forms the kernel of the scenario of the corresponding simulation model generated through model transformation.

In CM-to-DSAM transformation, which is defined over metamodel-level graph patterns, data and behavior is preserved. In fact the result of the execution of the transformation rules is an increase in the “information content” of the models from source to target. The extra platform specific information required for DSAM is provided through the transformation rules, and a user code library. Another transformation named Multi2BinaryLSC, to be applied as a pre-processing step on a produced DSAM before feeding it to the code generator, is also developed. In essence, Multi2BinaryLSC accomplishes transformation

from a global view of the overall simulations to the collection of local views of the individual simulation members.

A second phase transformation is applied by a code generator to produce executable simulation code and other useful artifacts from a DSAM. The code generator consists of an intermediate form generator front-end pipelined to a source code generator back-end. The front-end walks through the input DSAM using the programming API and constructs an internal representation of the model, which is fed to the back-end module to generate source code files for the LSC diagram, instance, computation aspect and simulation execution aspect. Computation logic has to be woven onto the generated aspect codes in order to provide legitimate values for the data structures at runtime.

The presented graph-based model transformation work is powerful and appealing in that it is visual, formally founded (both because it is based on metamodeling and it is possible to state and prove certain properties of the transformations by resorting to the theory of graph grammars and graph transformations) and offers a mechanism for transformation composition. A notable downside of the transformation is its poor performance especially when source models get bigger. This is accountable for every rule execution boiling down to solving the sub-graph isomorphism problem on the input model and the match pattern. This burden is partially relieved by breaking rules into reasonably small chunks and providing as much initial binding on the match pattern as possible. Another facilitator is the employment of the user code library which executes faster than pattern matching and saves from tediously defining many similar transformation rules.

LSCs are particularly powerful for event-based, rather than state-based, descriptions, which supports the trace-view of the system behavior. This could be particularly suitable for trace-based applications, such as scenario specification, and course-of-action analysis. LSCs may not be suitable for representing the execution of tasks that require continuous interactions among entities. Last, current LSC variants lack some well-known control flow constructs such as nested exception handling, jumping the flow to another point and global suspension.

The experience gained in this thesis is a step forward in designing a domain-independent model transformer for DSAM from any conceptual model that is based on LSC for behavioral representation and UML-like class diagrams for data modeling.

CHAPTER IV

FIELD ARTILLERY CONCEPTUAL MODEL

This chapter presents a formalized conceptual model for the Field Artillery (FA) observed (i.e. indirect) fire domain. The structural part of the model identifies the entities in the FA domain along with their properties and associations. The behavioral part of the model is used to describe FA missions in the language of Live Sequence Charts (LSCs). The conceptual model is constructed as a metamodel with the Generic Modeling Environment (GME) toolkit. Once the FA metamodel is registered, GME automatically provides a customized environment to model particular FA missions. The intended application is to use the FA metamodel as the source for defining model transformations targeting FA federation architectures. Another intent is to help evaluate the power and limitations of chart notations for describing military tasks visually yet precisely.

Section 4.1 is an introduction to the chapter, clarifying what is meant by a conceptual model, drawing the general outline of the field Artillery Conceptual Model (ACM), identifying the potential benefits of it and stating where it fits in the overall study. It also describes the reasoning behind selecting LSC for behavioral modeling. Section 4.2 explains ACM's scope, its implementation approach and its two user perspectives corresponding to a model builder and a software developer. Section 4.3 is an in-depth presentation of ACM's implementation in GME. The data model and the integration of the data and behavioral models are demonstrated. The behavioral model is actually the LSC metamodel and since it was developed as part of another study [12][85], its modeling is not covered. Section 4.4 presents the LSCs of the *Adjustment Followed by Fire For Effect* (AdjFFE) mission model in graphical notation. The section also shows the instance decomposition of the BatteryFDC model element into a lower level MSC document and the HSMC that shows how other mission definitions can be accessed. Section 4.5 discusses the challenges encountered and provides an informal assessment of ACM and usage of LSC in modeling military tasks. Finally Section 4.6 presents a selective set of related work of conceptual

modeling from the literature. Note that in the context of this thesis, the distinction between the terms task and mission is not important; thus, these are used interchangeably.

4.1 Introduction

A Conceptual Model (CM) represents the relevant entities of a domain and the relationships between them, independently of implementation details. CMs are essential artifacts both in operational systems and simulation systems lifecycle. In this thesis CM is to be understood in the context of modeling and simulation. Formalization of a CM is achieved when we construct it in a formal language, for example, as a model conforming to some metamodel. A metamodel essentially defines the language in which models are expressed. A formal representation serves as a basis for machine processing, and supports automated generation of useful artifacts, such as other (specialized) models and executable code.

4.1.1 Motivation

Robinson [44] defines a CM as "non-software-specific description of the simulation model that is to be developed, describing the objectives, inputs, outputs, content, assumptions, and simplifications of the model." He also points out that there is a significant need to agree on how to develop CMs and capture information formally. The need for formalizing task representations in military domains has been further emphasized in several other studies [45][46]. In another study, “Mission Space Models” are defined to be domain specific models that are consistent, structured and functional descriptions of real military operations or processes [50].

This chapter presents a tool supported formal model for the FA observed fire domain, verified and validated with a subject matter expert. Considering the definitions given in [44] and [50], it is necessary to underline that the modeling of military tasks by LSCs in this thesis constitute a part of a CM (or a mission space model), emphasizing inter-entity communications, rather than a complete CM (or a mission space model).

Formal modeling of the FA missions has many potential benefits. In the course of modeling one has to fill in the gaps found in the informal descriptions and clarify ambiguities. This helps with the clear understanding of the domain by an individual and shared understanding by a group of people, and facilitates processing with a computer.

The ACM is developed with the intention for use within the context of a model transformation work that aspires to produce executable distributed simulation code from FA mission models through a series of transformations. The purpose of the ACM is to lay the groundwork for a Platform Independent Model (PIM) to be utilized in (semi)automatic

model transformations to a Platform Specific Model (PSM), e.g. a Federation Architecture Model (FAM), where the platform is the High Level Architecture (HLA). A secondary objective is to assess the use of LSCs in FA mission modeling.

4.1.2 Rationale for Using Live Sequence Charts

Message Sequence Chart (MSC) [15], upon which LSC is built, is a well-established visual formalism for the description of inter-working of processes or entities. Both the graphical and textual syntax as well as the formal semantics (in terms of process algebra) are defined for MSC [41]. The sequence diagram notation [17] of Unified Modeling Language (UML) 2.0 is very similar to MSC. LSC is introduced by Damm and Harel [14] as an extension to MSC primarily to provide the distinction between mandatory and optional elements.

The play-in/play-out mechanism proposed by Harel and Marelly [43] support what they call scenario-based programming. The basic idea is to play-in the desired interactions and use LSCs to record them. Later these records are used as behavior specifications, which monitor a user-guided simulation (play-out). The mechanism is realized by the Play Engine, developed by the authors [43]. This operational view put into practice by the play-in/out mechanism looks attractive for the early validation of mission models.

Recently a linking tool called “InterPlay” has been developed [47], which can be used to mix inter-object behavior given in LSCs with separate behavior given for some of the objects in an intra-object language, such as conventional code or statecharts [48]. Note that in this thesis, we are concerned with the observable behavior of a system where the system state is implicit, whereas statecharts emphasize the state-transition view, which may include unobservable behaviors (e.g., data management and computation) as well, and the state is represented explicitly. Enriching the models with intra-entity behavior representation, promised by the InterPlay tool, seems to be an appealing future study and hence, another reason to leverage LSCs

4.2 Metamodel Scope, Methodology and User Perspective

4.2.1 Scope and Assumptions

The ACM addresses certain aspects of technical, rather than tactical, fire direction. Accordingly, the focus is on the autonomous fire direction mode instead of the battalion directed mode. Consequently, the battery FDC becomes the most outstanding actor while the battalion FDC’s role diminishes to merely monitoring the mission activities and interceding in exceptional situations. Even if such an intervention occurs, the flow still keeps the autonomous mode after the battalion fire order (i.e., a new mission assignment) is

received. This makes sure that the missions are always executed within the context of the battery's perspective. Owing to this viewpoint, massing of fires, which requires coordination of multiple batteries under the same battalion FDC, is omitted.

Ammunition preparation, fire parameter computations, ballistic conditions under which a projectile flies, and trajectory calculations all require computational and domain expertise. Such issues are considered far too technical and left out of the scope of the study. These processes are assumed to be transparently performed and their outcomes are readily provided by the infrastructure, if need arises. Moreover, the modelings of the environment such as geographic and man-made features are also omitted. Detailed modeling of possible targets, guns and ammunitions is avoided. What remains inside the scope of the model is the description of the firing missions from the viewpoint of message exchanges among the participants.

The metamodel is built in accordance to the relevant Army field manuals [35][36][37]. On the other hand, there were some routine military procedures that we judged as irrelevant for our modeling purposes. For example, the callee reads back whatever the caller has read within combat radio net conversations for verification purposes. Another example is that at the end of a conversation the parties may enter an authentication session. Such general issues are left out of scope to keep the model less cluttered.

There are seven kinds of FA mission types represented in the model under area and precision fire categories. There are many more special and ammunition specific missions mentioned in the field manuals. Since these seven types are probably the most widely used ones and they adequately serve the purpose of testing the use of LSC in the description of military tasks, no other mission types are modeled. Finally, the entire top level domain entities in the data model are specialized from NATO's JC3IEDM (refer to Section 4.3.1).

4.2.2 Methodology

The ACM is an integration of two separate sub-metamodels, namely, the behavioral and data models, as shown in the sample model of Figure 4.11, where the former relies upon the latter for the definition of domain-specific data types. The term domain is used in the sense of an area of interest, FA being an example.

The FA observed fire mission descriptions are represented by means of LSCs, specialized for the FA domain. Specialization is achieved by formulating FA mission messages as LSC messages and integrating the FA message structures as the data language of LSC. The LSCs in the behavioral model use the data model elements via referencing. Note that the behavioral metamodel is capable of representing the discrete communication behavior of many practical systems, consisting of components exchanging messages,

independently of the domain. This communication aspect of the system behavior is particularly emphasized from the LSC modeling perspective.

The data model consists of domain specific information, including actors, nets, mission messages, message communications and the mission hierarchy. The structures of these entities and relationships among them, as well as constraints are explicitly modeled. The structural constraints such as association, containment, attribute names and types, interface (via the port mechanism) and cardinality are readily defined thanks to the UML based notation of GME. Moreover, logical or semantic constraints are also defined either directly on model elements or globally (i.e., metamodel wide) in OCL.

The AdjFFE mission model is presented in Section 4.4. AdjFFE is one of the most prominent mission types of the FA observed fire and also serves well to reveal the use of the behavioral and data model elements together. Both ACM (metamodel) and AdjFFE (model) are realized using GME.

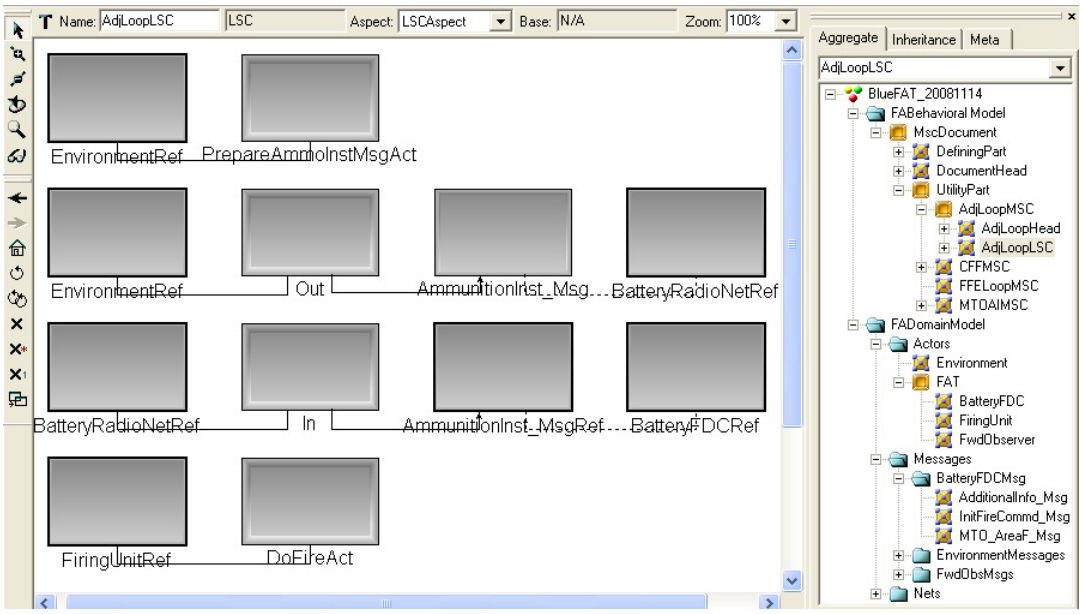


Figure 4.11 A simplified sample ACM model as shown in GME model browser

4.2.3 User Perspective

The MDA brings its own paradigm to software development as compared to the common state of the art approaches. In the MDA paradigm the most outstanding roles of development are the model and transformation rule developers. In our context, the potential users of the FA models are expected to be the FA domain experts who are building the

conceptual models and software developers who need both the source and target metamodels at hand in order to write model transformations.

A conceptual model builder has to develop the FA behavioral and data models as similar to Figure 4.11. Since this thesis's focus is on inter-entity communications, the user is relieved from defining intra-entity processes (e.g. computations, state management). The user is also free of concerning with the specifics of the data types of the data model, in the sense of programming languages.

A software developer, who is a model transforming user, must have a thorough understanding of the source and target metamodels to write transformation rules. He has to deal with the lower level data type mappings between the source and target models, hence requiring programming language knowledge. This user should further incorporate all sorts of HLA specific context that cannot readily be inferred from the FA model into the transformation rules.

4.3 Field Artillery Metamodel

This section illustrates and explains the prominent parts of the metamodel as realized in GME. 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 [17]. Table 4.2 relates the FA metamodel (field Artillery Meta-Model (ACMM)) to OMG's four-layer modeling hierarchy. Please refer to List of Abbreviations section at the beginning of the thesis for the model element name acronyms and abbreviations.

Table 4.2 FA metamodel (ACMM) correlated with OMG's four-layer model hierarchy

OMG's Metamodel Hierarchy	**Related Model**
Meta-metamodel (M3 layer)	GME metamodel (metaGME)
Metamodel (M2 layer)	FA metamodel (referred to as a "paradigm" in GME vernacular) - the ACMM
Model (M1 layer)	A particular FA mission description, e.g., AdjFFE (provided as the case study) - an ACM
Run-time instance (M0 layer)	A particular execution of a FA mission (e.g., exercising an AdjFFE scenario)

4.3.1 Data Model

This section elaborates on the constitution and organization of the structural part of the domain model. First, a brief but informative introduction to the entities is provided. Then, they are explained in detail, along with GME excerpts of the most important ones.

Actors correspond to the real world FA members, such as the personnel, the units or the environment. As they all take part in mission execution, they are considered the producers and consumers of domain information captured in messages.

Messages are an important part of the FA domain information. Typically, they are highly structured and they have 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. These constraints are captured as OCL statements throughout the model.

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.

A single message communication typically involves the triple of a source actor, a destination actor and a message to be sent from the source to the destination. Often there is an extra “net” acting as a means to deliver the message to secondary receivers. In some communications, there is no particular destination actor, but only a net.

The net concept, which is a set abstraction of actor members, is used to serve multicast communication needs within the model. A net transparently relays any message that it receives to its members. Nets are formed according to the dictations of domain specific requirements, such as intra- and inter-battery and meteorological communications.

The mission hierarchy builds up the set of FA observed fire missions that can be modeled. 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. For each kind of mission there

is a corresponding mission definition as part of the behavioral model. In this respect, the mission hierarchy bridges the data and behavioral models together, establishing traceability from the data model to the behavioral model in that, given a mission LSC, there must be a message transmitted within the LSC that indicates the mission's type by including a corresponding mission model element in the message.

Actors

Throughout the modeling work, a functional point of view is adhered to. Hence the organizational structure of the military domain is not of major interest in identifying the actors. Accordingly, the FAT trio is modeled as the FwdObserver, the BatteryFDC and the FiringUnit. FwdObserver is identified as an actor due to his central role in observed fire missions. BatteryFDC and FiringUnit could be organized under the firing battery part of a cannon battery, but since they directly play important roles in missions, they are treated as two actors on their own.

BatteryFDC is further decomposed into BatteryFDO and BatteryFDCComputer. This layered modeling of the BatteryFDC is primarily a consequence of focusing on autonomous fire direction mode. In this setting, as indicated in Section 2.5.1, the BatteryFDO is responsible for producing fire order and fire order Standing Operating Procedures (SOP), and the battery computers (abstracted as the BatteryFDCComputer) participate in producing the fire command. As these messages lie at the heart of FA missions, their producers and consumers deserve to be treated explicitly.

In the Army field manuals [35][36], it is indicated that many important messages are addressed to the BatteryFDC by the other actors, such as FwdObserver, FiringUnit and BattalionFDC. No specific component inside the BatteryFDC is mentioned as this is not a concern to these exterior parties. Of course, when the focus is on the BatteryFDC itself, then intra BatteryFDC actors and their interactions are explicitly described.

The BattalionFDC is an actor outside FAT, supervising the FAT's activities, occasionally intervening or taking over the control on its own accord.

Meteorological data in the form of a metro report is an input for technical fire direction, especially for the computation of fire commands. Usually a meteorology station at the army corps produces and distributes metro reports [36]. ACM accounts for this fact with the MetStation actor which distributes metro reports to the related FAT members.

Being the object of firing missions, Target is an obvious model entity. It is referred in messages such as call for fire, and refinement and surveillance. Target, however, is not elaborated in the model.

All the entities and processes of FA missions are deployed in accordance to an order of battle. This environment is modeled using the FeatureType entity in the sense of JC3IEDM (see below). FeatureType specifically covers geographical and meteorological features. The model, however does not address environmental concerns.

Nets

Within this thesis scope, the radio-net and metro-net concepts corresponding to the real-world battlefield radio nets are modeled. An actor can join or listen to a conversation taking place within a net that it is a member of. By this way multicast or broadcast messages such as meteorology messages are handled uniformly. The BatteryRadioNet consists of references to the FwdObserver, BatteryFDC and BattalionFDC. The MetroNet consists of references to the MetStation, BatteryFDC and BattalionFDC. In both of the nets, the sole inclusion of the BatteryFDC implicitly assumes that the BatteryFDO and BatteryFDCComputer are also informed of the conversations as they are included in the BatteryFDC. Finally, it is worth noting that although the BatteryRadioNet is intended as a usual radio net, the MetroNet is merely an abstraction; it may not be an actual radio-based network.

Figure 4.12 sketches the organization of the FA actor and net entities. Note that there is a Reference to every actor (not shown in the figure) as only the references to actors are used in message communications and nets. All the actors and nets are modeled as GME Model elements and are collected in their respective folders, except the two BatteryFDC members.

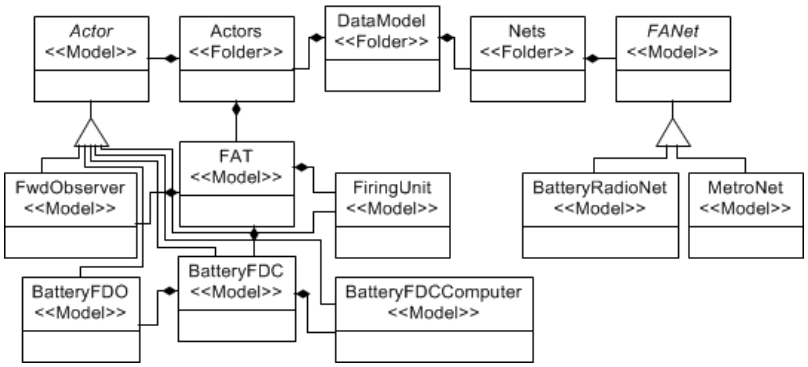


Figure 4.12 The actors and nets of the field artillery data model

Messages

The set of messages comprises the bulk of the FA data model. There are more than 70 messages defined, ranging from simple, single piece of command-type forms to complex, highly structured union forms. In order to utilize powerful modeling principles such as

modularity, reuse and polymorphism, families and hierarchies of messages are defined. There are two major abstraction layers established in the message model, namely the utility and the conceptual layers. The utility layer gathers most of the commonly used parts found in the messages, such as ammunition, measurement, date and time, location, and direction. The conceptual layer contains the higher level messages employed in mission descriptions. These elements include the relevant utility components in addition to the message-specific parts.

The family of messages defined in the metamodel, all extracted from the Army field manuals [35][36], has two kinds of usages as mentioned in in the beginning of this section. The messages of the first category include CFF, MTO, FO (of both BatteryFDC and BattalionFDC), FC, FiringReport, Spotting and Correction types. The messages of the second category are each a trio of instantiation, update and deletion messages for Ammunition, SOPs for FO and FC, and MetroReports. In Figure 4.13, the parts a and b present a sample for each category.

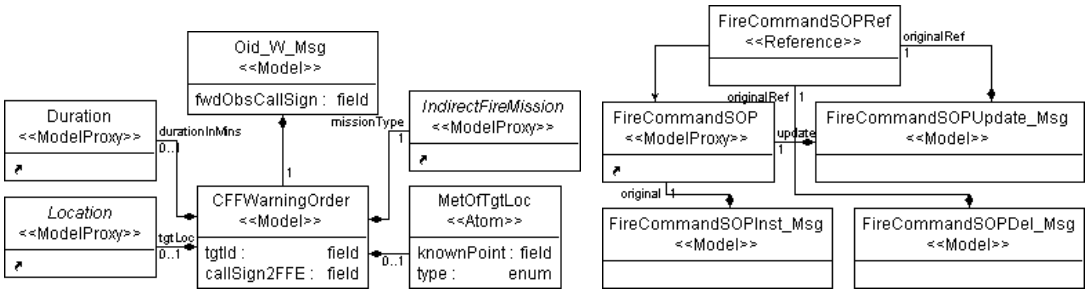


Figure 4.13 a)Msg. for observer identification and warning b)Msgs for FireCommandSOP

Both syntactic and semantic constraints on the selection and formation of message parts are expressed in OCL statements, which are directly bound to the messages themselves or are defined globally at the metamodel level. These constraints in the metamodel are enforced either during model development time or after model construction according to their priority levels. Below is a sample OCL expression for Oid_W_Msg of CFF type message, indicating that "*Target location is only given in immediate suppression or immediate smoke missions*":

```
let missionType = self.parent.connectedFCOs("src",MissionType) in
missionType.name = "Supp" or missionType.name = "ISupp"
```

Message Communications

Message communications are the top level yield of the data model before moving into the behavioral model. The actors, nets and messages introduced up to this point are used in combinations into meaningful message communications.

Structurally, a message communication embodies the message, the sender and the receivers of the message. Apart from one or two exceptions, there is only one sender per message. For this reason the message communication hierarchy is based upon the senders of messages, which are actors. A net cannot be a sender, but only a receiver since its sole function is to relay an incoming message to its member actors. Figure 4.14 illustrates a sample branch of the message communication hierarchy.

Note that every member has a suggestive role name and a cardinality of 1 in its composition relation with the parent communication element. Another point to note is that the sender and the receivers are references, whereas the message is a model element itself. Consequently, in a series of message communications, every message must be a new individual, but the senders and receivers must be existent actors. In cases of multiple receivers, each receiver gets its own copy of the message.

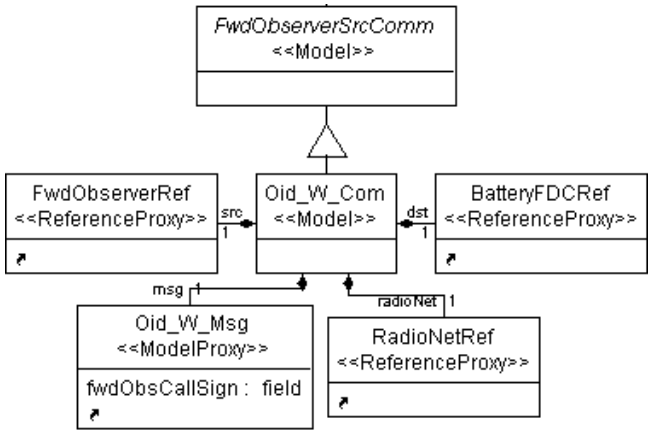


Figure 4.14 A message communication example

Missions

A perusal of the field manuals revealed the types of observed fire missions shown in the mission hierarchy in Figure 4.15. In this classification, observed fire missions are grouped under two fire categories, namely AreaFire and PrecisionFire. This distinction is based on the fact that area fires are conducted with all of the guns in a battery, whereas precision fires

are conducted with usually one, or at most, two guns. Area fires are categorized as adjustment, fire for effect, suppression, immediate suppression, quick smoke, immediate smoke and illumination. Precision fires are destruction and precision registration.

The purpose of area fire is to cover the target area with dense fire so that the greatest possible effect on the target can be achieved [35]. Fire For Effect (FFE) is the most common and important of area fire missions. The observer strives for first-round FFE, provided that if he can locate the target accurately. If the observer cannot locate the target accurately enough to warrant FFE, he conducts an adjustment. Even with an accurate target location, if the current firing data corrections are not available, adjustment may be necessary [36]. In adjustment, fire from the central gun alone is step by step brought onto a designated adjusting point. Fire parameters are refined through observer corrections after each round. FFE is started by the entire battery once a satisfactory adjustment has been obtained.

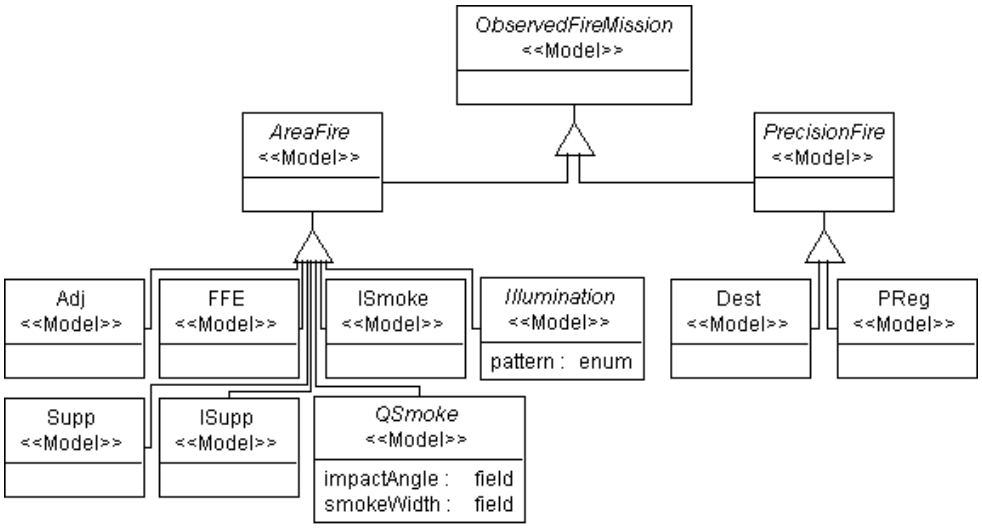


Figure 4.15 The mission hierarchy

Several points need clarification. First of all, adjustment may be conducted in conjunction with smoke and illumination missions as well. In this respect, adjustment can be considered as a preliminary activity rather than a standalone mission. This opens a debate as to discard adjustment as an area fire mission. However, due to adjustment's significance, frequent application, and the field manuals' practice of counting it as a mission on its own, we opted to place adjustment under area fire mission hierarchy.

Second, illumination, suppression and registration missions have several variations, each employing different techniques. There are further variations based on special munitions used in the missions. These are considered out of scope of this thesis.

Finally, since FFE is the most common mission among the others and is usually preceded by an adjustment, adjustment followed by FFE (i.e., AdjFFE) mission is used as the case study of this work. Most of the other kinds of missions are indeed conducted similar to these two with special differences and/or additions.

JC3IEDM as an Upper Level Data Model

JC3IEDM [53] is adopted as an upper level data model for ACM. Particularly, every top-level entity in the data model is specialized from a JC3IEDM element by means of the inheritance mechanism. A simplified hierarchy of the used JC3IEDM elements is readily defined in the data model for the sake of soundness and completeness. This conceptual traceability of the model from JC3IEDM promotes the model's compatibility and recognition. Figure 4.16 shows the top-level FA domain entities (plain boxes) and their extension points with JC3IEDM (shaded boxes).

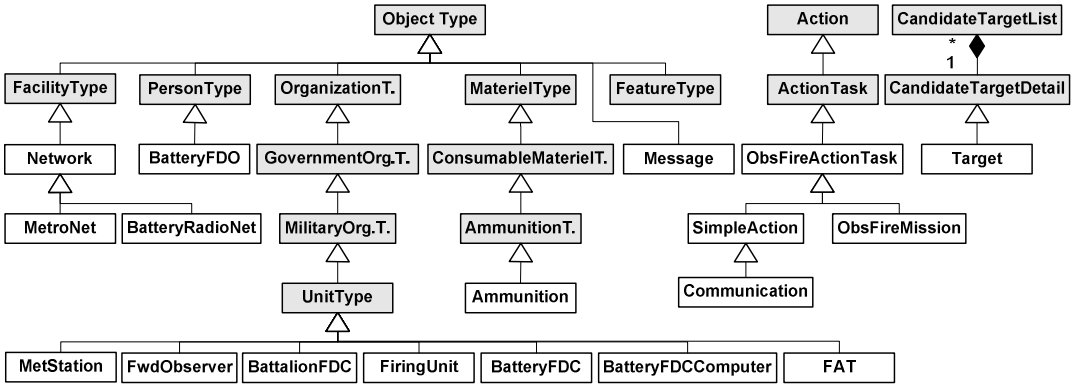


Figure 4.16 FA domain entities as attached to JC3IEDM

4.3.2 Composition of the Behavioral and Data Models

The behavioral part of the ACM is essentially the LSC metamodel [12]. As LSC is extended from MSC, the metamodel also covers the MSC metamodel in its core. MSC is a visual language for specifying the behavior of a concurrent system focusing on the communication among the components of the system. The metamodel covers all the standard MSC features [15][41] and the proposed LSC extensions [14] as a coherent whole.

The data model narration together with the chart samples of the graphical notation should suffice for having a grasp of the LSC notation for the unfamiliar reader. Also an introduction to the MSC and LSC specifications are provided in Section 2.7. Please refer to [12] for more details and examples from the HLA-based simulation domain.

To put it another way, the LSC metamodel provides a generic infrastructure for modeling the discrete communication behavior of a system as a partially ordered set of events (mainly as message passing) between a group of instances. In the context of a specific domain, these generic behavioral elements need to be specialized as the domain's entities. The specializations are naturally derived from instance, message and other elements of the LSC metamodel. The composition of the behavioral and data models is thus achieved by integrating the data model to the LSC model. The integration points of the behavioral and data models are shown in Figure 4.17. It is seen that all of the FA actors and nets are inherited from LSC `Instance` and that the FA domain messages are inherited from LSC `Msg`.

GME's being a configurable toolkit for creating domain-specific modeling environments comes handy in creating the ACM paradigm. The paradigm is the result of importing the existent LSC and FA data metamodels as libraries into GME through its built-in Model Integration paradigm and then defining the integration points in a separate paradigm sheet, as explained above and partly shown in Figure 4.17. Once the ACM is registered as a GME paradigm, a domain specific modeling environment capable of enabling domain experts to build FA mission models is obtained.

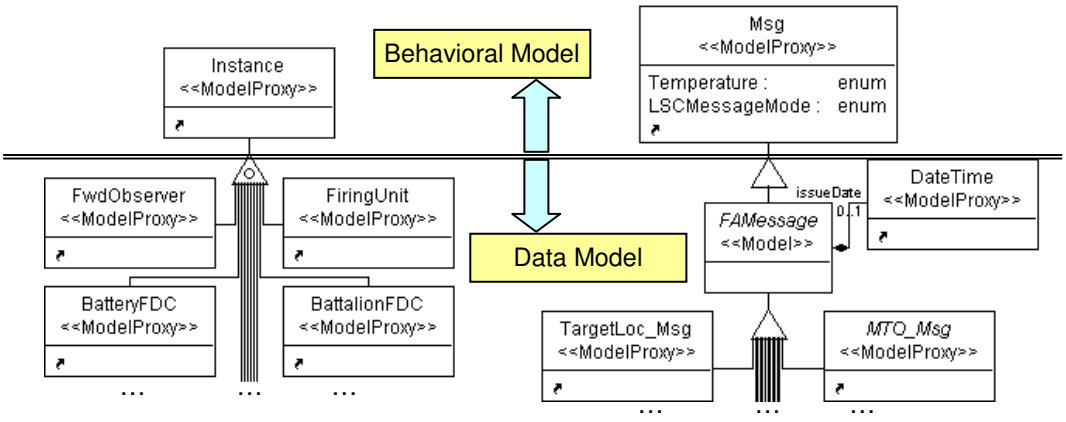


Figure 4.17 Integration of data model to behavioral model (partial view)

4.4 Adjustment Followed by Fire-for-Effect Mission Model

This section describes the AdjFFE mission, the source model for the model transformation case study, in graphical LSC notation. The model includes around 40 LSCs, each being about one page long and an HMSC. In this section, only a set of important charts will be presented. The complete set of LSCs can be found in Appendix A. Since the graphical notation elements are not introduced elsewhere, they will be described wherever they are first encountered.

4.4.1 The Top Level Mission Model

Figure 4.18 fully covers the AdjFFE mission description at the topmost level and sets up the framework for the remaining charts.

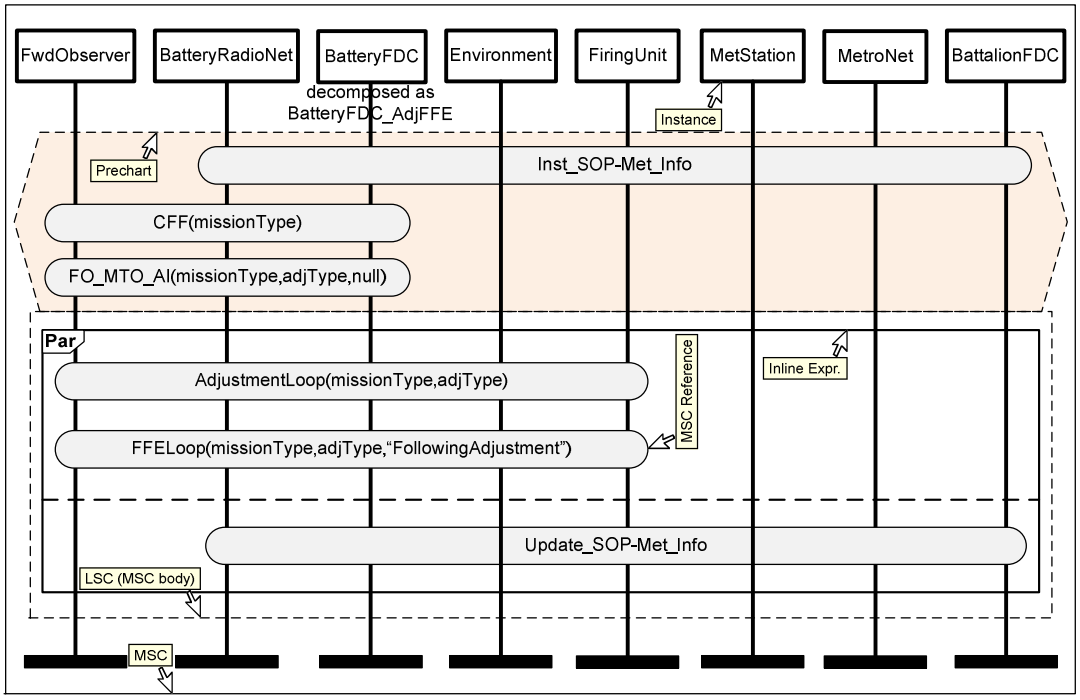


Figure 4.18 Adjustment followed by fire for effect

The horizontal dimension is the structural dimension and the vertical dimension corresponds to the time dimension. On top of the vertical axes are the instance heads, drawn as rectangles with instance names. Vertical lines are the instance axes and thin solid rectangles at the bottom indicate the syntactical end of an axis.

The LSC has a superimposed pre-chart shown as a dashed elongated hexagon indicating that the LSC scenario takes effect, provided the pre-chart has been traversed successfully. The thin rounded rectangles orthogonally crossing the instance axes are references to other MSCs. A reference symbol contains the referenced MSC's name and, if applicable, its actual parameters. If an instance is not involved in a referenced MSC, then its axis is drawn through the reference symbol. The consecutive arrangement of MSC references implies sequential ordering in time. Within the LSC, it is seen that there is a parallel inline expression with two operands, separated by a dashed line.

Finally, note that BatteryFDC is "decomposed as BatteryFDC_FFE". The keyword decomposed below the instance head indicates that the instance is decomposed within at least one MSC to further refine its behavior. Thus an MSC document may be interpreted relative to its own instances only, disregarding any decomposition, or it may be interpreted relative to lower levels of instances by following the decomposition relations. The decomposed BatteryFDC is presented in Section 4.4.2.

The CFF chart, depicted in Figure 4.19, describes the FwdObserver's sending of CFF messages to the BatteryFDC. The messages are also transmitted by the BatteryRadioNet to its members. We devised a simplifying convention that net transmissions are only shown as incoming messages to an MSC reference whose sole instance is the net, in order not to clutter the chart with unnecessary obvious information. The net together with its members are explicitly shown in the defining chart of the called MSC reference.

Messages are shown as directed arrows with labels drawn from the sender to the receiver. A message may originate from and arrive into an instance, an MSC reference or the exterior environment. It may get lost in transmission. A message contacts an MSC reference at a gate, which interfaces the MSC to the outer world. The small black circles are simultaneous regions, meaning that the set of events touching the circle are perceived to happen at the same time. An action is an atomic event represented by a rectangular box attached to an instance axis. The actions are treated in our study as annotations where user defined computations can be entered. The LSC body consists of a guarded optional inline expression where the condition variables being provided as MSC parameters.

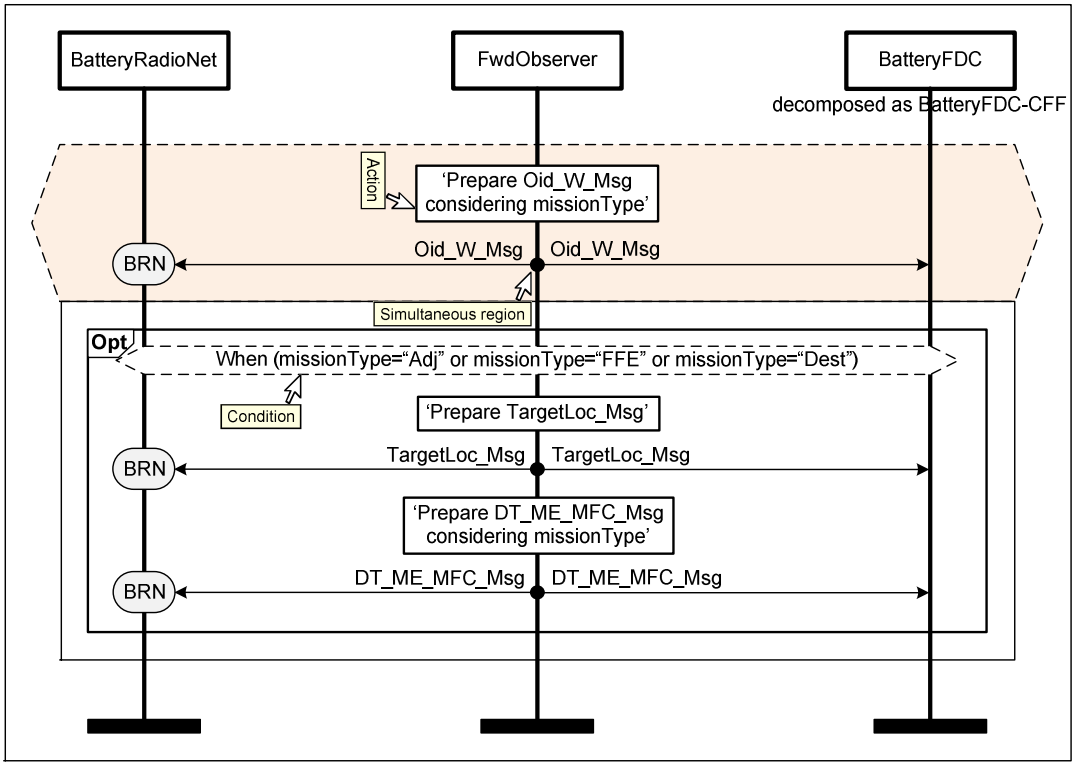


Figure 4.19 Call for fire

Following the CFF and MTO message transmissions, the adjustment loop starts as shown in Figure 4.20. Prior to the loop start, an initial fire command must be prepared and sent, resulting in system state to become AdjNotDone (i.e., adjustment is not done). As long as the adjustment is not done, round shot, spotting observation, shot assessment through correction sending, and subsequent fire command preparation in the light of incoming correction information are executed in sequence. The preparation of correction data and deciding whether the adjustment is accomplished or not require domain specific computations. Thanks to the action mechanism these technical details are abstracted away. The vertical dotted line segment besides the two action events indicates a coregion meaning that the execution order of these two does not matter.

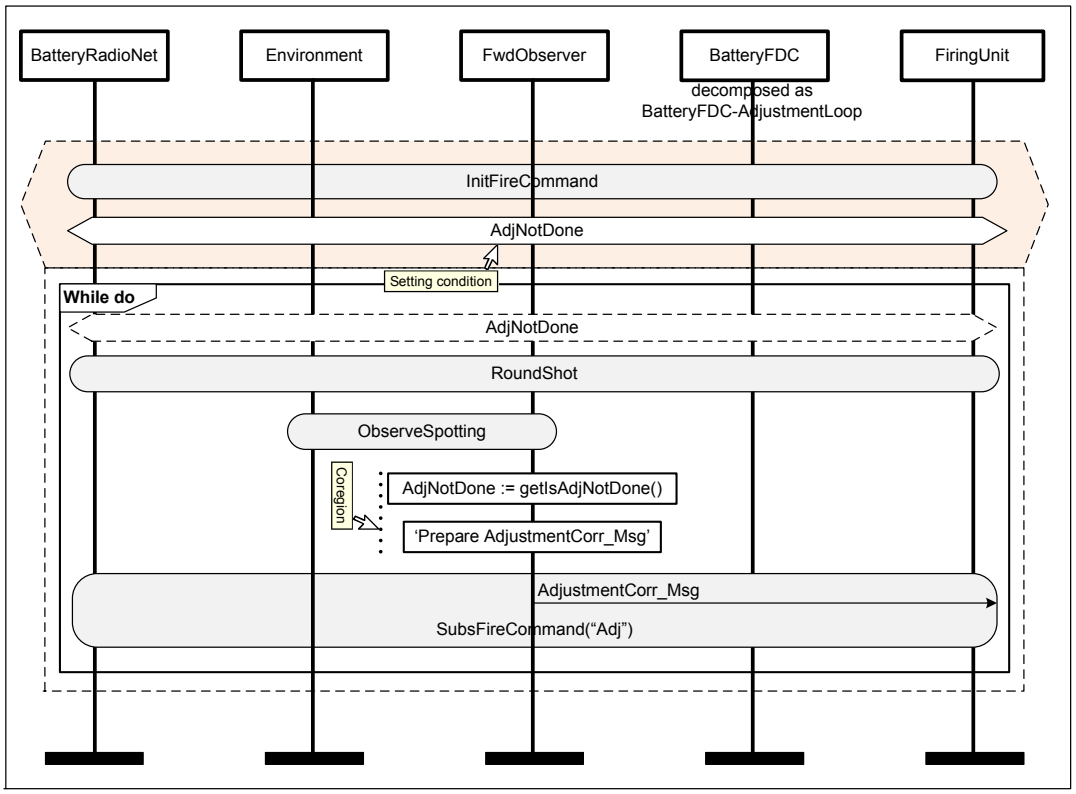


Figure 4.20 Adjustment loop

Note that, concurrently with the MTO and before the initial fire command, there is a fire order message transmission taking place between the sub-instances of BatteryFDC. As these instances are defined in the lower level MSC document called FDC, the fire order message is shown there. For details, refer to Section 4.4.2

The initial fire command chart is depicted in Figure 4.21. Fire commands are prepared within BatteryFDC as the result of a series of detailed computations involving ballistics.

At this stage, only the produced commands are seen to be sent from BatteryFDC to FiringUnit. The decisions on selecting the specific fire commands are extensively guided by the dynamic instance variable metCtrl. Each instance maintains its own copy of metCtrl in order to keep track of its command and control state. We envision the exceptional CheckFiring (CF) and CeaseLoading (CL) events to occur non-deterministically. Whenever the FAT enters into CF or CL states, the instances in question halt in the sense that they do not emit any messages until the BatteryFDC sends cancellation messages for CF or CL.

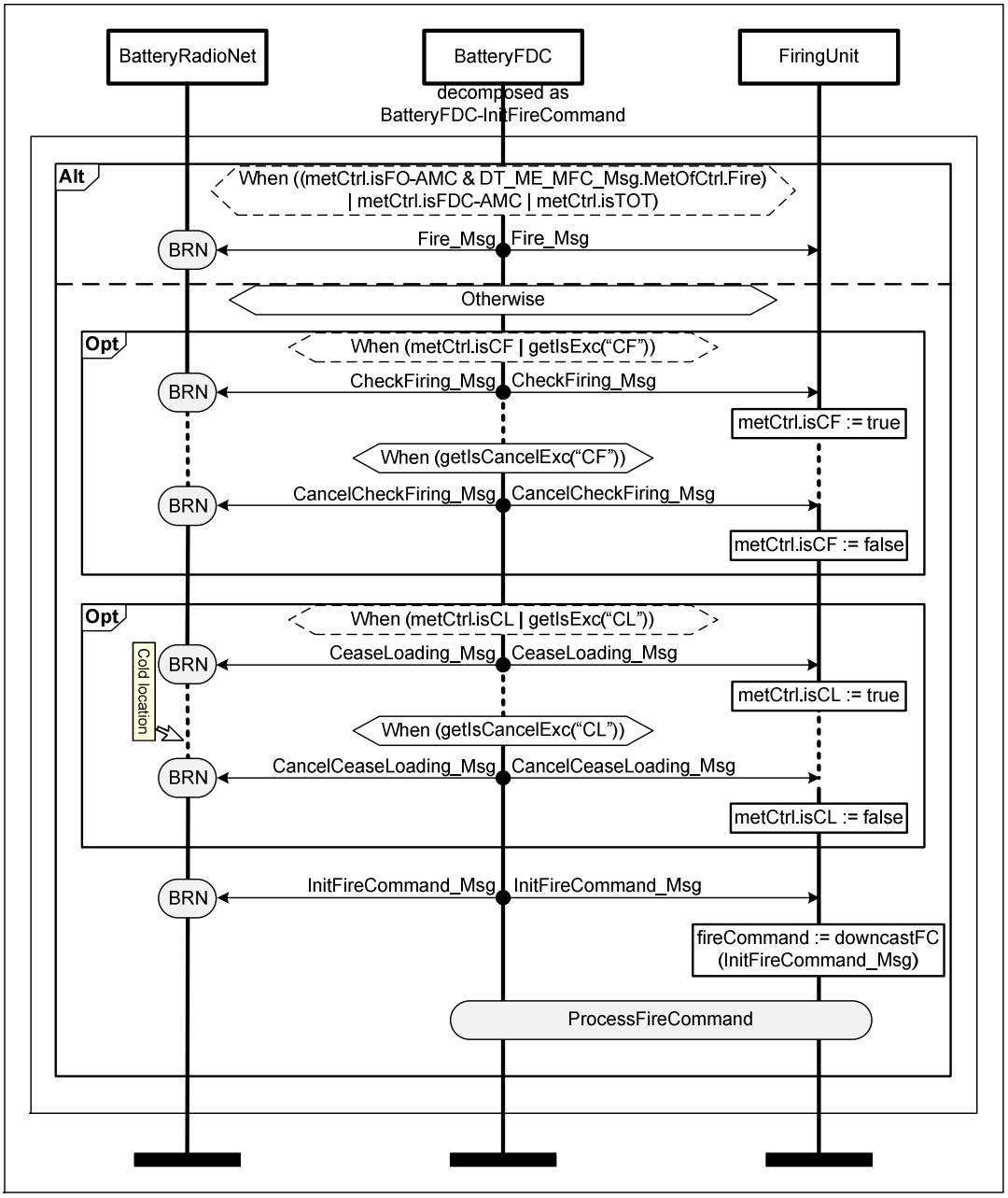


Figure 4.21 Initial Fire Command

There is no restriction on the instances other than emitting messages in due course; that is, they may continue any other activities of their own. This situation is captured by cold locations represented as dashed line segments in the instance axes. A cold location has the semantic that execution may remain at that point indefinitely. The flow ends with an invocation of the ProcessFireCommand chart.

The life cycle of Ammunition, from the moment it is brought into the firing position till its removal after detonation is reflected in the chart of Figure 4.22. Ammunition is created by the Environment instance and made known to the radio net members through an instantiation message. Then the FiringUnit fires the round and the projectile (i.e., ammunition) proceeds traversing its trajectory. The ammunition instantiation message event happens strictly before the firing action event of the FiringUnit, as shown by the dotted arrow between the two. If not explicitly ordered, events that occur on different instances are assumed to take place independently, by definition.

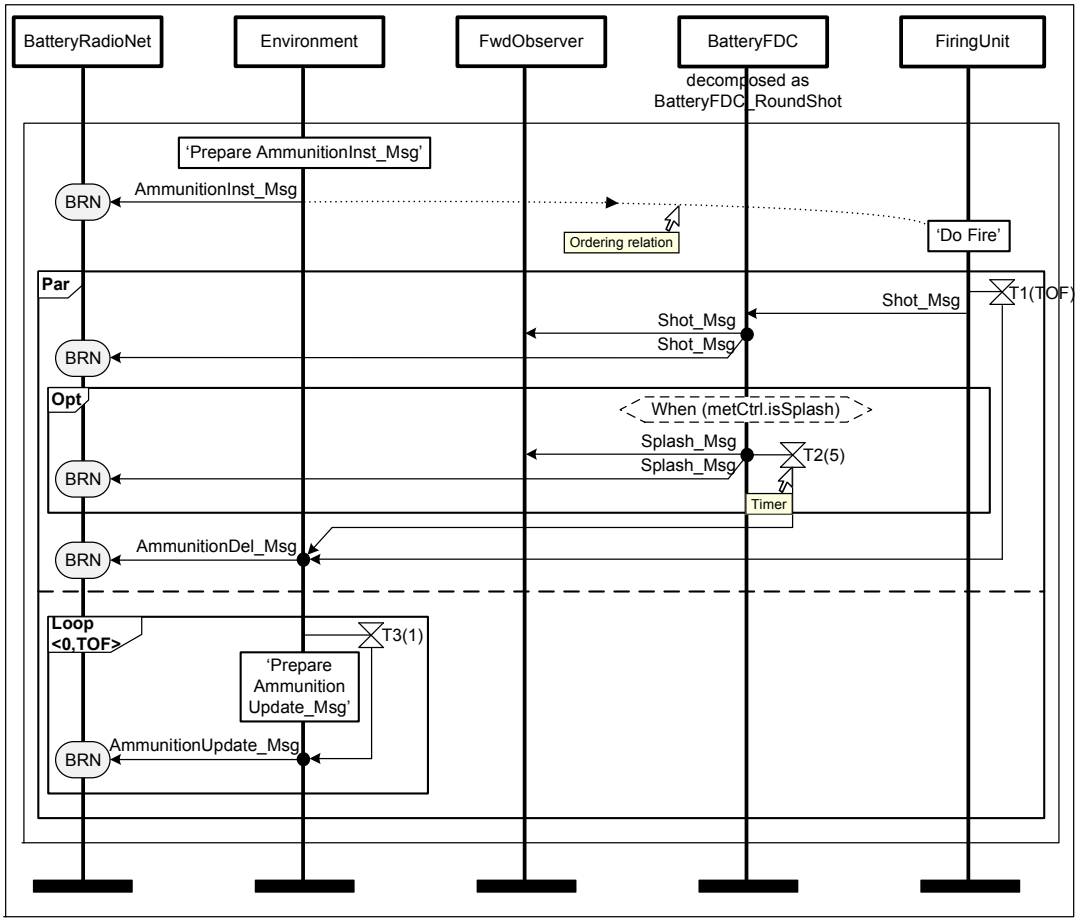


Figure 4.22 Round shot

As the projectile follows its trajectory, Environment sends ammunition position update messages at every second until detonation. Timer events are symbolized with an hour glass having a time value alongside. A timer start event is a horizontal line segment between the

instance axis and the timer symbol. A timer stop event is an L-shaped directed line segment from the timer symbol to the instance axis. After the projectile is shot, a Shot_Msg is sent from FiringUnit to BatteryFDC and then propagated to FwdObserver and the other radio net members. If a warning message was requested before projectile detonation, BatteryFDC sends a Splash_Msg to FwdObserver five seconds prior to detonation.

4.4.2 Instance Decomposition of BatteryFDC

The inner structure and behavior of an instance kind are defined through an MSC document with the same name as the instance kind. To indicate how the behaviors described at different levels of abstraction are related, the behavior of an instance inside an MSC diagram can be specified to be refined in an MSC of the MSC document defining the instance being decomposed.

In the figures of Section 4.4.1 the BatteryFDC instance was tagged as "decomposed" meaning that there is an MSC document called BatteryFDC containing further refinements of the charts that included BatteryFDC as an instance. There are some requirements to be met by the decomposition, which are best explained through exemplifying figures below.

Figure 4.23 illustrates the decomposed chart of AdjFEE as situated in the lower level MSC document refining the BatteryFDC. Note how the MSC references CFF and MTO_Al are refined as viewed from the BatteryFDC's perspective. One important detail that is not present in the upper level chart is the production and sending of the fire order message, since fire orders take place only within the battery FDC scope.

The content of the refined version of CFF is given in Figure 4.24. In terms of inline expressions, the decomposition contains a corresponding inline expression of the same operation and operand structure as the one in the decomposed chart. The inline expression in the decomposition is "extra-global" (i.e., crossing the MSC frame) indicating that the operands are connected to other operands of similar inline expressions when interpreted through decomposition. The harmony between the lower and higher level charts can be clearly followed through the similarities in the structures of the MSC constructs. (Compare Figure 4.19 and Figure 4.24).

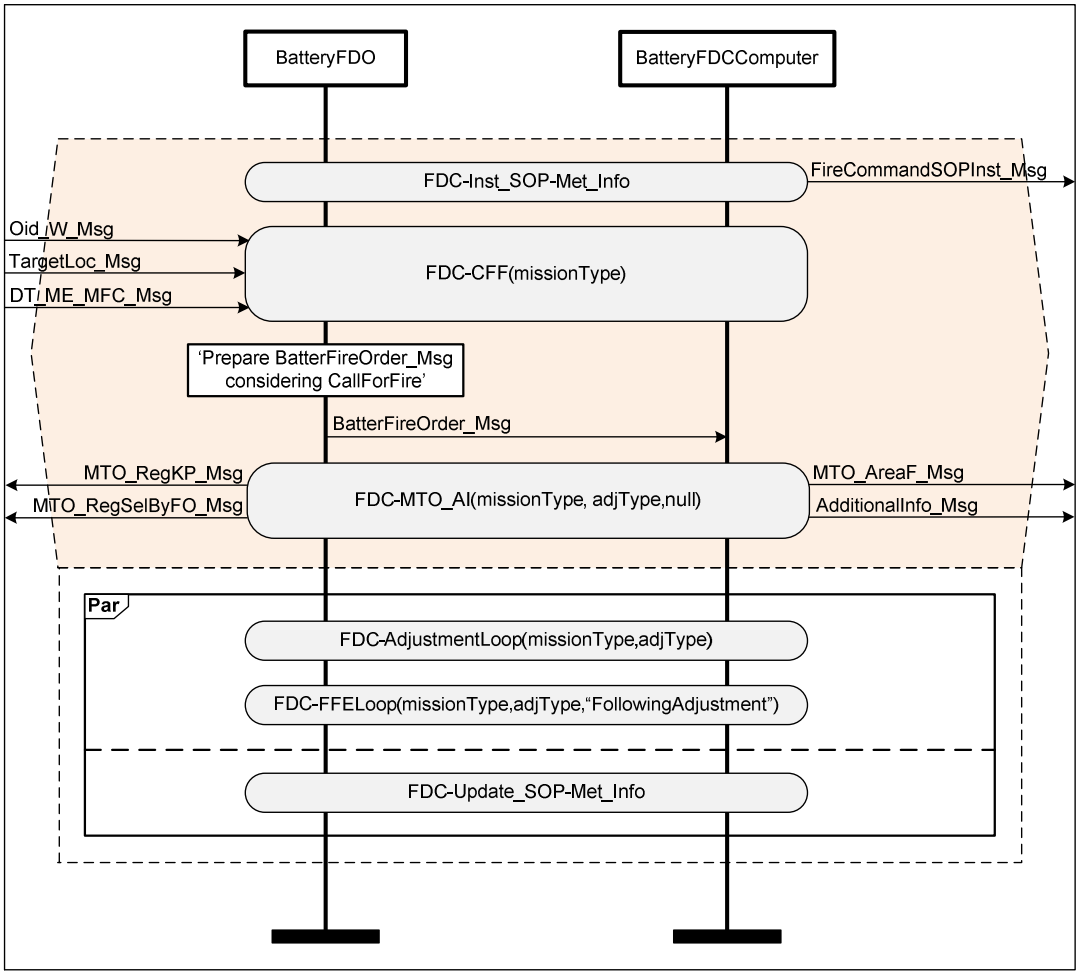


Figure 4.23 Adjustment followed by fire for effect in decomposed BatteryFDC instance

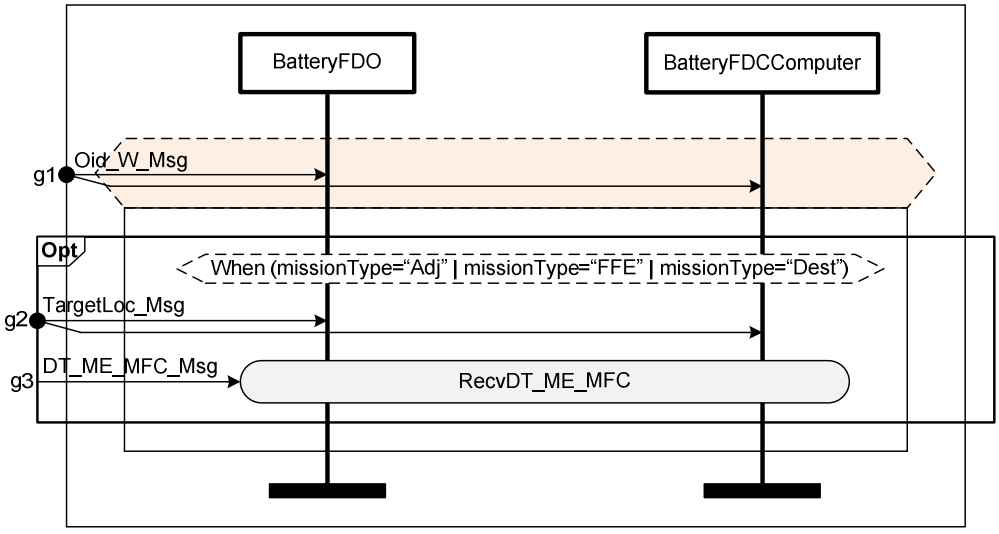


Figure 4.24 Call for fire in decomposed Battery FDC instance

4.4.3 Overview of Missions via a High Level MSC

High-level MSC (HMSC) provides a means to graphically define how a set of MSCs are combined. An HMSC consists of start and end symbols, flow lines, conditions, top level MSC references and inline expressions. A flow line indicates sequential flow, and start and end symbols have the obvious role of scope marking. The latter three have much similar interpretations as the ones found in MSCs.

Figure 4.25 presents the HMSC for the ACM behavioral model. It is a top level view covering which mission executions are started under what conditions. This high level chart first sets the global system states of operation mode and mission type, and later steers the flow towards the desired mission execution. This thesis only covers battery directed operation mode and demonstrates the AdjFFE mission, but the other options are provided for future model extensions. Note that we allow the option for battalion directed operation mode, but do not provide a description for it. Mission type is to be selected among the seven possibilities covered by the present work.

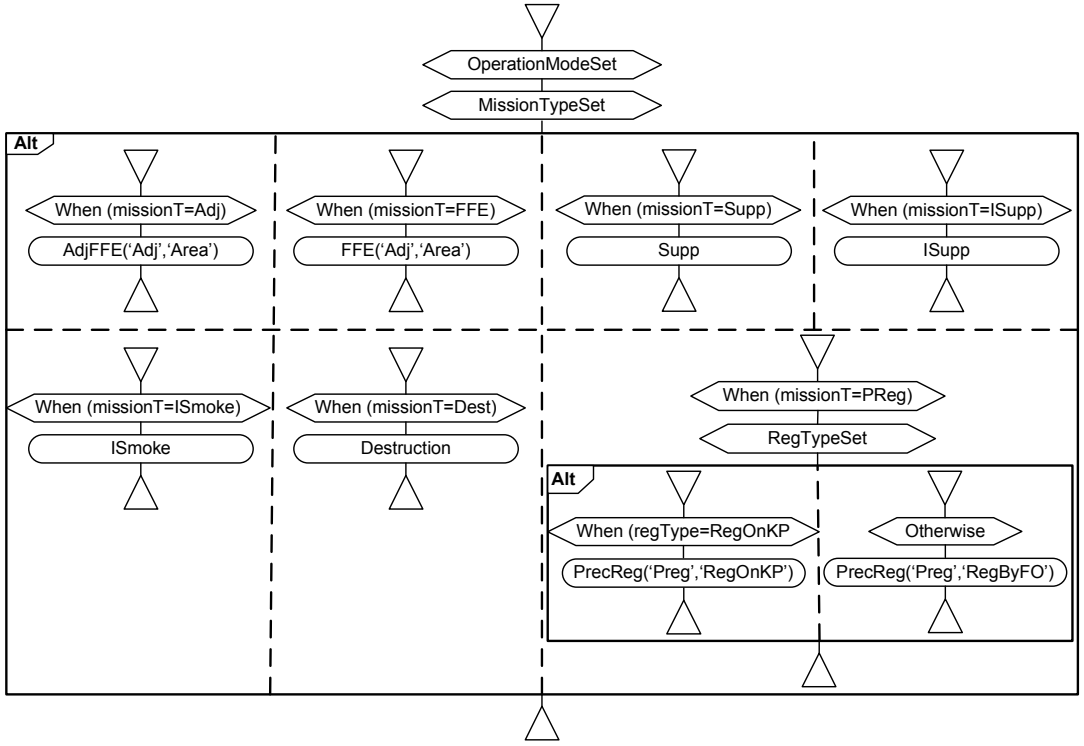


Figure 4.25 High level MSC for FA behavioral model

4.5 Discussions

This section provides a discussion on challenges encountered, lessons learned, assessment of ACM and assessment of using LSC notation in modeling military tasks.

4.5.1 Challenges Encountered

A number of difficulties had to be overcome in both modeling the FA domain and using the LSC metamodel in developing the AdjFFE model. The following paragraphs discuss the notable challenges.

Information on FA observed fire techniques are dispersed in a series of Army field manuals. The narrations of these manuals are fairly informal in that they bear traces of experience and insight obtained in battlefields. Moreover, they presume much background knowledge on the part of the reader, and sometimes even have seemingly incongruent parts. It took a considerable amount of effort to comprehend that content and come up with a coherent domain model. SME consultations also proved to be very fruitful in resolving ambiguities and filling in gaps.

The point of view taken in the manuals dictates the hierarchical chains of command and seniority relationships among the actors. We did not consider military hierarchy in our modeling practice and followed a functional modeling approach instead. Specifically, some of the actors represent humans (e.g., BatteryFDO), some represent units (e.g. BatteryFDC, FiringUnit), and even some represent a mixture of both (FwdObserver, BatteryFDCComputer). All of the actors are selected and organized according to their roles in the observed communication flows in performing FA missions.

Another inconvenience was met in deciding from which JC3IEDM elements to extend the data model elements. Sometimes there were more than one alternative, sometimes there was no obvious candidate and sometimes there was a JC3IEDM entity with the same name as an ACM entity, but having a slightly different meaning. For example, a debated decision was whether to extend the FwdObserver element from JC3IEDM UnitType or MilitaryPostType. For the no-candidate cases, we made up intermediary entities between the JC3IEDM and the entities in question. For the same-name cases, we put an identification tag in front of the entities for preventing name clashes. We used intuition in resolving such situations.

On the behavioral model side, while we enjoyed the convenience in representing the message communications between the actors, we suffered representing global state information. For example, the unary valued setting conditions of MSC was an obvious shortcoming, which we had to relax in modeling for convenience.

An MSC message event has a signature comprised of the message name and the parameter set. The types of the parameters are defined in the data language provided to the

MSC. On the other hand, an FA message is a coherent unit, typically with a nested structure; hence it is inconvenient to flatten it into a list of parameters. In order to adapt the FA message as an MSC message, we set the top level FA message element's name as the name of the corresponding MSC message without parameters. We embed the information content that otherwise would be conveyed by MSC message parameters into FA message definitions.

LSC lacks some well-known utility constructs such as nested exception handling, jumping the flow to another point and global suspension. We strived to avoid these cases; however, in some cases we had to devise workarounds. There were some situations where LSC provided the operators, but they were either insufficient or not applicable under certain conditions. For example it might be the case that a timer starts inside an optional inline operand and timeouts somewhere outside. In this situation we allowed the timer to start inside the operand and go off outside. The interpretation we give is that if the optional operator executes the timer starts, otherwise the timer has no effect.

4.5.2 An Informal Assessment of ACM

This section constitutes an evaluation of the ACM in terms of the approaches taken in domain modeling. Then the selection of tool and technology are critiqued. Consequently the assessment covers a set of recognized CM evaluation criteria such as completeness, traceability, modularity, layering, extensibility, reusability, composability and interoperability of the model.

ACM is a well-focused and framed artifact. Converging to the smaller scale battery level and a limited portion of the overall domain (i.e. observed fire techniques) enabled us to disregard many tactical issues peculiar to the battalion level as well as computational issues, yielding a more compact and comprehensible model. This tight scoping approach makes the model domain specific enough so as to open way for feasible model transformations.

The U.S. Army field manuals [35][36][37] are utilized as the authoritative information sources. In that respect we strived to make use of the original terminology from the field manuals for the model elements. Hence all the data model entities and their attributes can be traced back to the related sections of the field manuals for validation.

We have conducted a series of face validations with an SME. This proved to be very fruitful especially when dealing with the ambiguities inherent in the field manuals.

All of the top-level data model entities are derived from NATO JC3IEDM, establishing the metamodel on a mature and common formalism. This further favors the model in terms of compatibility and recognition. In [45] it is underlined that the Battle Management

Language (BML), which this work can be considered to fall in line with, must use the existing C2 data representations whenever possible. Extending the data model from JC3IEDM clearly echoes this argument.

A salient feature of the model is its capability to illustrate the domain at higher and lower abstraction levels and the seamless integration of the two. In particular, the interactions of the BatteryFDC to other actors and the intra BatteryFDC actors and their interactions are readily demonstrated.

In [49], Davis and Anderson emphasize the relevance of reuse and composability in conceptual models. The FA metamodel promotes these concepts by defining the top level generic model elements as FCOs to provide for easy model extension. An outstanding example exhibiting model reuse and composition is importing the data and behavioral metamodel libraries and then defining specialization relations between the relevant data model elements and the behavioral ones as explained in Section 4.3.2.

To have an overall view of all of the mission descriptions, HMSC mechanism has been utilized. This way, the model user finds a common interface in configuring and selecting a mission description.

On the tool side we believe that deciding on GME was appropriate. GME is a domain-specific, model-integrated program synthesis tool for creating and evolving domain-specific, multi-aspect system models. Due to GME's inherent UML basis, the models are bolstered by being compliant with a common industry standard. (Note that GME's UML support is limited to only class diagrams). This further enabled the model to utilize the tenets of object oriented design such as encapsulation, inheritance and polymorphism. The dynamic semantics of a model is not defined in GME, but this can be introduced through model interpreters that can be plugged into GME. Moreover, the model is capable of representing semantic (business) constraints both on a model element basis and globally in OCL. Finally, it is possible to obtain tool independent XML exports of the models, facilitating interoperability. By this way "silos" are avoided; that is the metamodel is realized in GME, but we are not restricted to GME.

4.5.3 An Assessment of Using LSCs in Modeling of Military Tasks

LSCs are particularly powerful for event-based, rather than state-based, descriptions. The point of view is to capture the observable interactions of an entity, distinguishing between mandatory and optional. The observers can be other system entities or any outside entity, including the environment. The interactions do not have to be only the sending or receiving of messages. More generally, any discrete action by one entity that can be observed by another entity, e.g. shooting, can be modeled as an MSC/LSC message. The

event-based nature of LSCs supports the trace-view of the system behavior. This could be particularly suitable for trace-based applications, such as scenario specification, and course-of-action analysis.

Use of LSC as a practical modeling notation, invariably requires a data model (static model) to be integrated with the action (behavioral) model. The variables, values and expressions communicated by the messages refer to the data model. The action boxes, then, represent internal computations performed on data items by individual instances. The description (though not the ordering) of computations falls outside the scope of LSCs. This issue is best handled by an algorithmic language.

LSCs may not be suitable for representing the execution of tasks that require continuous interactions among entities. Consider, for example, a maneuvering tank platoon. Maintaining its formation and changing the formation when ordered require the tracking of all team members, say, by maintaining line-of-sight, by each member continuously. Representing in LSC how such a task is executed is not convenient. A typical recourse would be to represent the team-wide start and finish conditions of the task and consider it done in the meantime. In the similar vein, the execution of tasks that involve a spatial element, such as illumination patterns, could be handled by action boxes in a way akin to a computational or menial task. It is, then, up to the model transformer or code generator to interpret such conditions and action boxes suitably.

Chart notations, such as MSC, LSC and UML sequence diagrams are often used by software developers to represent certain typical runs of the system being specified. Our approach, in contrast, strives for a complete specification. This explains the extensive use of nested control structures in our charts. With the usual trace-oriented use a much more flat chart structure would suffice.

As the state information is indirectly and implicitly represented in LSCs, they do not readily support system implementation. However, this does not prevent us from animating LSCs, say, for validation purposes, or generating executable code from LSCs, say, for generating prototypes.

4.6 Related Work on Conceptual Modeling

This section points to a selective set of related conceptual modeling techniques, frameworks and approaches in literature that fall in line with the present modeling work. Before starting, a basic understanding of conceptual modeling, formal specification and perspectives from different application domains is worthwhile to provide.

In a broader context, Conceptual Model (CM) is defined in [95] as, “An abstract, idealized and symbolic description of the structure and behavior of the real system, which

is understandable for those from the application domain". Ideally, the CM provides insight into modeler's purpose-related understanding of the structure and behavior of the real system as well as the system knowledge used for modeling. It should also explain the motivation and justification of conducted abstraction, idealization, and selection of the model boundaries. Furthermore, providing the hierarchical organization of the sub-models is desirable. Generally speaking, CMs hardly feature completeness, self-consistency, unambiguousness, direct support for development, maintenance and reuse. Last, CMs can contain comprehensible natural language specifications, i.e. narration, as well as formal specifications in their descriptions.

A formal specification is a solution-oriented, unambiguous symbolic specification of the structure and behavior of the real system, based on a well-defined modeling formalism [95]. Some examples to general purpose formal specifications can be given as UML, Queuing Nets, Petri Nets and DEVS. ACMM and FAMM - the metamodels used in this thesis for representing the field artillery observed fire and HLA domains, are examples to formal specifications tailored for domain-specific modeling. Formal specification supports unambiguousness, efficient implementation of solution, platform independent specification, and automated verification and validation activities such as syntax checking, semantic checking (including self-consistency), control flow and data flow analysis, model checking, and (limited) testing.

There are different conceptions of conceptual modeling from different perspectives such as information systems development, database management systems, knowledge engineering, ontology and simulation. The following subsections provide more insight into some of these.

4.6.1 Conceptual Models of the Mission Space

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 [50] 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) [51] and

the conceptual modeling tool KAMA [52] have further elaborated the vision promoted by the CMMS.

4.6.2 KAMA

KAMA is a conceptual modeling framework that incorporates a notation, a modeling process and a supporting tool for developing mission space conceptual models [52][106]. The process is based on the works of CMMS [50] and Pace [115]. The framework does not mandate the developer to follow the proposed process; any other process is possible, as long as pre-requisite relationships among the diagrams, such as an entity state diagram requiring an entity to be defined or a task flow diagram requiring the existence of a mission or a task, are satisfied. The KAMA notation is a graphical, UML-based specification that provides a domain specific language for conceptual modeling. It is composed of four major packages, namely, Foundation, Mission Space, Structure and Dynamic Behavior – an inspiration from UML Infrastructure specification [17], and defines seven diagrams for representing the structural and behavioral aspects of a model. The tool supports both the notation and the process and provides the developers facilities such as reusing conceptual models from the common repository, filtering diagrams, context-sensitive search, navigation, n-dimensional model viewing, versioning, custom report generation, verification and custom properties management of the model elements.

The case studies conducted with KAMA have revealed that the users had difficulty of comprehension as the diagrams became cluttered. Model development in GME also suffers similar cluttering problems, but the *view aspect* capability of GME [3][33] allows the developer to group related model elements into user defined aspects so that the elements in the same aspect are visible and others are filtered-out when the aspect is active.

Karagöz [52] emphasizes the great value of transforming CMs into simulation design models and admits that using common metamodels for both would ease the process. He further mentions that KAMA's scope does not include that. In our work, we have ACMM and FAMM developed with the same meta-metamodel – metaGME. They also share the same LSC formalism for behavior representation and the ACM2FAM transformation is defined over the metamodels.

The KAMA tool benefits from a common repository similar to the one in DCMF [51] that stores all of the published and accredited conceptual models, model elements and the diagrams, and a simple repository search mechanism, which GME does not (intend to) have support for. On the other hand it lacks two features as stated by the author, namely, metamodel editing and model merging.

4.6.3 Defense Conceptual Modeling Framework

The CMMS initiative was prematurely ended by DoD a few years after its sparkling. It was later independently continued by FOI, the Swedish Defense Research Agency. FOI has refined and enriched the original CMMS concepts and later has evolved its FOI-CMMS work into Defense Conceptual Modeling Framework (DCMF) [51].

A fundamental contribution of DCMF is the introduction of the Knowledge Meta Meta Model (KM3) [96], a meta-metamodel to capture system structures and behavior in an object-oriented and rule based way. The DCMF is an iterative process spanning four major phases governed by different roles of responsibilities. Information is first gathered within the Knowledge Acquisition phase. The Producer role processes unstructured knowledge and transforms it into represented knowledge. To accomplish this, a parsing method must be used. During the Knowledge Representation phase, smaller sections of this data are structured as Knowledge Instances (KI) and validated for storage in the repository by the Controller role. KIs are useful for some purposes, but they are not reusable since they are specific to the scenario data. To get reusable knowledge, KIs are abstracted to the type level, modeled as Knowledge Components (KC) and then validated in the third phase, called Knowledge Modeling. These components are, upon Consumer requests, composed to form Conceptual Models (CM) in the fourth and final phase, Knowledge Use. All the described artifacts are stored in a repository for use and reuse.

In a more recent work [97] the FOI team has investigated enriching DCMF models with semantics in an effort to better conceptualize and reuse knowledge. This is achieved by creating an ontology for Base Object Model (BOM) and producing semantically enriched BOMs as the outcomes of DCMF processes.

4.6.4 Base Object Model

Base Object Model (BOM) is proposed by the SISO to encourage and support reuse, interoperability, composability, and to help enable rapid development of HLA simulations. Conceived in 1997, BOM was standardized by SISO in 2006 [98]. At a high level, BOMs are reusable packages of information representing independent patterns of simulation interplay and are intended to be used as building blocks in the development and extension of simulations. These components can also be composed in larger models (e.g., BOM assemblies). Additionally, interplay within a simulation or federation can be captured and characterized in the form of reusable patterns. These are sequences of events between simulation elements. Implementation of these patterns using HLA object model constructs is also captured in the BOM [99].

Structured in five major parts, a BOM is an XML document that encapsulates the information needed to describe a simulation component. The first part is Model Identification, where metadata about the component is stored. These facts describe the point of contact, what it simulates, how it can and has been used, as well as descriptions aimed towards helping developers find and reuse it. The second part is the Conceptual Model. This part includes what types of actions and events that take place in the component, and is described by a pattern description, a state-machine, and a listing of conceptual entities and events, which, when taken together, describe the flow and dependencies of events and their exceptions. The third part is Model Mapping, and is where conceptual entities and events are mapped to their HLA Object Model representations. This part bridges the Conceptual Model with the HLA Object Model described in the fourth part of the BOM. The fifth section is called Supporting Tables that contain semantic information about events and entities as well as actions that is specified in the Conceptual Model, and are used to provide a human-readable understanding of the patterns described in the BOM.

4.6.5 Ontology as Conceptual Model

Ontologies are structured descriptions that categorize concepts and relationships among concepts within a particular knowledge domain [100]. Ontologies are, like taxonomies, used to classify entities within a domain, but they hold several advantages over traditional taxonomies in that they allow the entities to have properties and relationships. They also allow types of things within a particular domain to be defined as classes, and the meaning of a class is captured via its position within subclass-of (is-a) hierarchy as well as by its properties, relationships, and restrictions. Because ontologies are meant to define a domain and to be shared by many, the most useful ontologies are created by expert groups.

The use of ontologies in M&S has recently emerged as a growing area of research interest as evidenced by the creation of the Discrete-event Modeling Ontology (DeMO) [101], the evaluation of the Command and Control Information Exchange Data Model (C2IEDM) as an interoperability enabling ontology [102], the development of the Process Interaction Modeling Ontology for Discrete-event Simulations (PIMODES) [103], the development of the Component Simulation and Modeling Ontology (COSMO) [104] and the use of domain ontologies in agent-supported interoperability of simulations [105].

4.6.6 JC3IEDM

Joint Command, Control and Consultation Information Exchange Data Model (JC3IEDM) is the core of NATO Reference Model and is also a view model of NATO STANAG 5525 [53]. 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 [54] by additionally including and modeling new joint operational concepts.

There have been numerous efforts in evaluating C2IEDM as an enabling referential data model for semantic simulation interoperability [55][56]. The results turned out to be that C2IEDM could support concepts and entities very well and was sufficient for the formation of relationships. The extensibility of C2IEDM is noted as yet another strength.

Brutzman and Tolk [57] offer recommendations for a framework ensuring interoperability, reusability, and composability for the U.S. Air Force Joint Synthetic Battlespace. They propose benefiting from distributed modeling methods using Model Driven Architecture (MDA). They underline the use of C2IEDM as a common reference model on the semantic level as a promising way to obtain meaningful interoperability between components within joint and combined environments (i.e., system of systems).

4.6.7 Model-Based Approaches

The 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][58]. A well-known MDE initiative is the MDA of Object Management Group (OMG), launched after the broad acceptance of the UML [17], which became the lingua franca for modeling over the past decade. Model transformations are considered the heart of MDA, where the PIM of a system to be constructed, is transformed, or refined, into a PSM [2][20]. Both PIM and PSM conform to their own metamodels, which act as grammars that define these models. Depending on the abstraction layer of the models, the PSM may even be the executable code. If not, it can be further transformed into code through another transformation.

Model Integrated Computing (MIC), an earlier manifestation of MDE, relies on metamodeling to define domain-specific modeling languages and model integrity constraints [3]. 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.

An exemplary MDA approach supporting Program Executive Office Soldier is presented in [59]. The aim is to develop a modeling federation that integrates the capabilities of various existing C2 systems in order to analyze the effects of a soldier on tactical missions. UML chart notations are used to represent the mission descriptions.

CHAPTER V

ACM TO FAM TO CODE TRANSFORMATION

This chapter presents a two-step automatic transformation of a Field Artillery Conceptual Model (ACM) into a High Level Architecture (HLA) Federation Architecture Model (FAM) into executable distributed simulation code. The approach taken adheres to the Model-Driven Engineering (MDE) philosophy. The ACM and the FAM conform to their own metamodels, which are separately built with the Generic Modeling Environment (GME) tool. Both metamodels are composed of data and behavior parts, where the behavior representation in both is based on Live Sequence Charts (LSC). The ACM to FAM transformation is carried out with the Graph Rewriting and Transformation (GReAT) tool and partly hand-coded in C++. Code generation from FAM is accomplished by employing a Java based model interpreter that produces Java/AspectJ code. The code can then be executed on an HLA Run-Time Infrastructure (RTI) engine after weaving the necessary computational aspects. The experience gained in this work provides a step forward for the inspiration of a domain-independent conceptual model transformer for HLA.

The ACM data model defines the field artillery domain entities, and its behavior model defines observed fire missions in LSC form. Likewise, the FAM data model defines the field artillery entities as federates, the federation and HLA messages, and its behavior model defines the fire missions as inter-communicating federates via the RTI, again in LSC form. Adopting a parallel design principle, the ACM to FAM (ACM2FAM) transformations are essentially formulated around the core of data and behavior model transformations, executed in sequence. Before and after these core blocks, come the smaller sets of pre and post rules that set up and tear down the stage for the HLA federation execution. This overview is sketched in Figure 5.1.

Before starting the details of this lengthy transformation process, we would like to provide the user a grasp of how a produced FAM as the result of an ACM2FAM transformation looks like through a FAM LSC in graphical notation. Figure 5.2 partially presents the top level AdjFFE LSC diagram of the produced FAM that corresponds to the

same named ACM LSC of Figure 4.18. Their structural resemblance is apparent with the exceptions of the introduction of the instance named `FieldArtilleryFed` representing the RTI federation execution and the two LSCs at the beginning and the end for federation initialization and tear down, respectively.

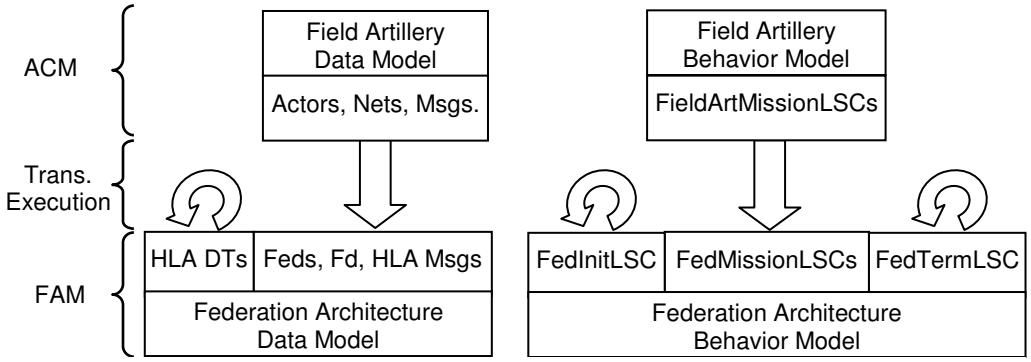


Figure 5.1 An overview of ACM to FAM transformation

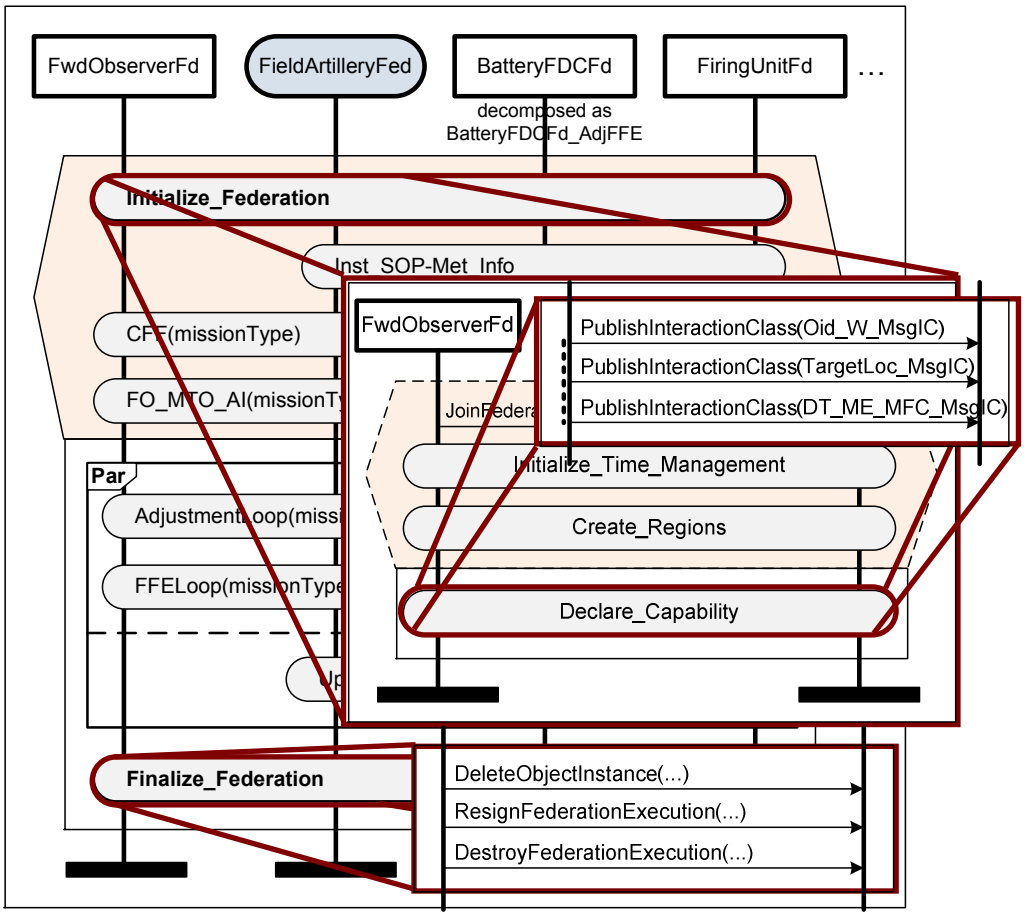


Figure 5.2 Partial view of AdjFFE LSC in a produced FAM

5.1 Setting the Stage for Transformation

The separately and independently built source and target metamodels are exported into an empty transformation model as a preliminary step to start the development of the transformations. The development environment is again GME, as shown in Figure 5.3, but tailored using the GReAT paradigm towards model transformations. Be reminded that transformation development is yet another modeling activity and can therefore be realized in GME. The transformation model consists of source and target models, transformation configuration, transformation blocks and rules and other utility model elements for providing easy global access and cross domain associations to be used during the transformation.

The transformation configuration generally points to the source and target models and metamodels, and the user code library employed by some of the transformation rules. It also designates one of the rules as the start rule.

Cross-links establish cross model associations between the source and target metamodels. Cross-links can be defined not only between different domains but can also be used to extend a domain to provide some extra functionality required by the transformation. By using a separate package for cross-links we are able to specify a larger, heterogeneous domain that encompasses all the domains and cross-references.

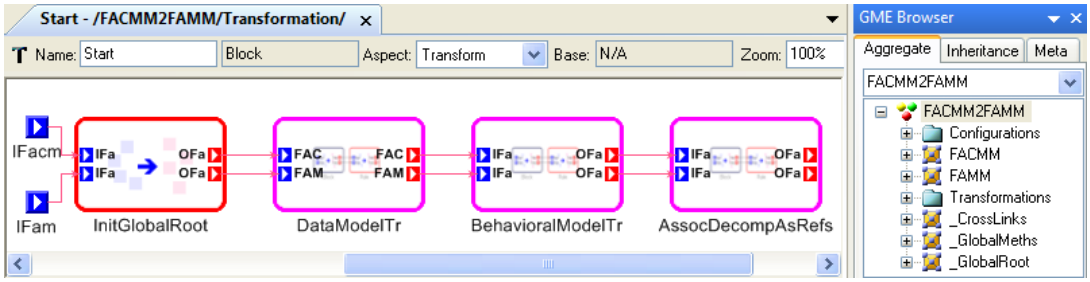


Figure 5.3 The start rule block of ACM2FAM transformation in GME/GReAT

A transformation step always starts pattern matching with an initial context, which is an initial partial binding of the pattern graph. The initial binding reduces the search complexity in two ways, (1) the exponential is reduced to only the unmatched pattern vertices and (2) only host graph elements within a distance *d* from the bound vertex are used for the search, where *d* is the longest pattern path from the bound pattern vertex [6]. This context is passed along from rule to rule via ports during the transformation, similar to parameter passing in

procedural languages. The main weakness of this approach is that the programmer needs to specify context passing through several rules, even if the context is actually used only in one remote, non-adjacent, step. The general idea of the global container is that the objects it contains have global scope; that is, they are accessible throughout the whole transformation, and it is not necessary to pass them along in the context. The capability of eliminating portions of context passing and recurring complex pattern matching is one of the key facilitating factors in terms of the development effort and execution performance in this work.

The transformation definition is comprised of a set of major blocks, which further consist of other blocks, rules, cases or expression references. Table 5.1 summarizes the metrics for the overall ACM2FAM transformation effort, indicating a total of 64 `blocks`, 4 `for-blocks`, 187 `rules`, 13 `tests` (with a total of 55 `cases`) and 21 `references` to other rules. The `DataModelTr` and especially the `BehavioralModelTr` blocks constitute the core of the transformation and are further explained in the subsequent sections.

Table 5.1 Metrics for the ACM2FAM transformation

Transformation Expression	Count	Transformation Expression	Count
Block	64	Test	13
ForBlock	4	Case	55
Rule	187	Expression Reference	21

5.2 Data Model Transformation

Data model transformation corresponds to the structural part of the ACM2FAM transformation. Looking from a FAM perspective, it aims to construct the federation object, the federate objects and the Federation Object Model (FOM) for the federation. The main `DataModelTr` block is shown in Figure 5.4. It is composed of two inner blocks named `ObjectModelTr` and the relatively smaller `FederationStructureTr` that are executed sequentially, in that order.

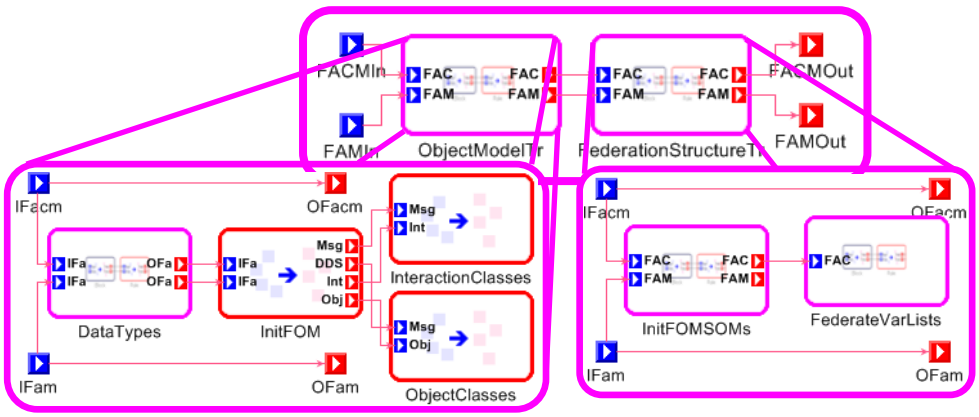


Figure 5.4 The main `DataModelTr` block

5.2.1 Object Model Transformation

Object model transformation basically transforms the set of field artillery message structures that are communicated among domain actors during mission executions into HLA classes. The field artillery messages are represented as free format UML structures with information content provided by the domain. On the other hand, HLA-OMT specification [39] puts forth a data type system. Several OMT tables (attribute, parameter, dimension, time representation, user-supplied tag, and synchronization) provide columns for data type specifications. A data type used in these tables shall be one of simple, enumerated, record, array, and variant record data types. OMT specifies a core set of default data types of basic, simple, enumerated, and array types, that correspond to universally recognized types such as byte, integer, float, boolean and string. The HLA-based distributed simulation model of any domain has to use an arrangement of OMT data types.

As a preliminary step to the field artillery message to OMT class transformation, the `DataTypes` block creates all of the basic, simple, enumerated and array data types that make up the default, predefined HLA data set. Note that there are no default fixed and variant record types. Domain specific ones are later defined in the rules that create interaction and object classes.

The `InitFOM` rule creates containers for interaction classes and object classes and an empty FOM element, which is later filled with interaction and object classes. These two OMT classes are the key elements in FAM data model; they are used frequently throughout the rest of the rules in the transformation. In addition, the stubs that correspond to the other OMT tables are also created in the FOM, of which only dimensions and timestamps are maintained and used, whilst the rest left out of the scope of this thesis.

Then, the transformation flow splits into two parallel branches, where interaction and object classes are created concurrently. The crux of the data model transformation logic is that all non-durable (i.e., stateless, with a life span of only a message transmission period) messages are transformed into interaction classes and durable messages (i.e., stateful, with a life span of the federation execution unless deliberately deleted) are transformed into object classes.

At this point, the user code library interferes to apply operations on the bound objects using the Universal Data Model (UDM) API [34]. 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. In our case, the user code library realizes the actual field artillery message to OMT class transformations programmatically. We have identified and implemented three approaches for transforming an ACM message into an OMT class and its attributes. The library has more than 600 lines of C++ code with public entry methods for the three approaches and eight non-public utility methods. Please refer to [60] for details.

The `InteractionClasses` rule is provided in Figure 5.5 for illustrative purposes, where black colored model elements indicate a pattern to match, and blue colored elements designate the new elements to be created. The code snippet inside an `AttributeMapping` element is executed after GReAT's pattern matcher matches the rule pattern and the effecter makes structural modifications on the matched model elements. The `AttributeMapping` in the figure invokes the user code library's message transformation method.

There are two reasons why an important part of the data transformation is handled by means of a code library. The biggest problem is that a field artillery message is usually deeply structured, possibly having child objects bound to their parents in varying cardinalities. This makes the situation even more complicated, because in order to represent such combinations we would need many patterns, hence rules. For example, if a message can have *n* direct children each having zero or one composition cardinality, then we would need at least *n* parallel rules to cover all possible matched combinations of the source model. On the other hand, by employing the user code library, we only need one rule, no matter how many children with whatever cardinality a message structure may have.

The other reason is obtaining considerable performance gain by directly executing C++ code. This is much faster than first matching a graph and then calling the effecter to execute it. (Be reminded that the sub-graph isomorphism problem, which is involved in every pattern matching step, is NP-complete).

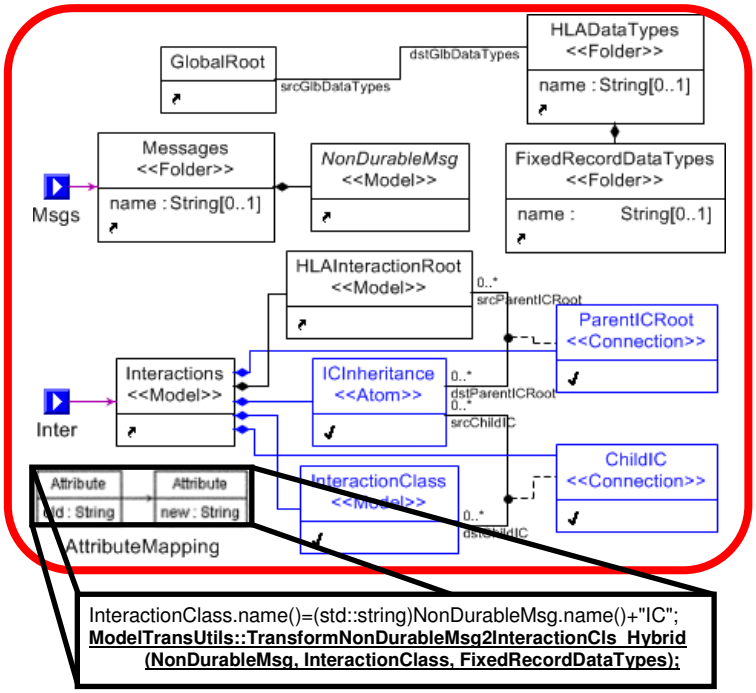


Figure 5.5 The `InteractionClasses` rule

Figure 5.6 illustrates a conceptualized sample field artillery message structure to OMT class transformation. On the left side is a field artillery message structure, named `Msg`, having two components, named `BCmn` and `CProp`, both of which further have a couple of children. Each leaf child has one attribute named `val`, of the type shown. It is assumed that `BCmn` is a common, shared component used by other field artillery messages, and `CProp` is a proprietary component specific to the message in question.

The transformation rule creates the `MsgIC` interaction class on the right hand side through sole pattern matching. The user code library programmatically creates the rest of the structure below `MsgIC`. 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 is transformed into a field of a record type, 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 reflecting the message structure hierarchy. The field name consists of a string of concatenated message structure element names, separated by "_", from the leaf to the root node.

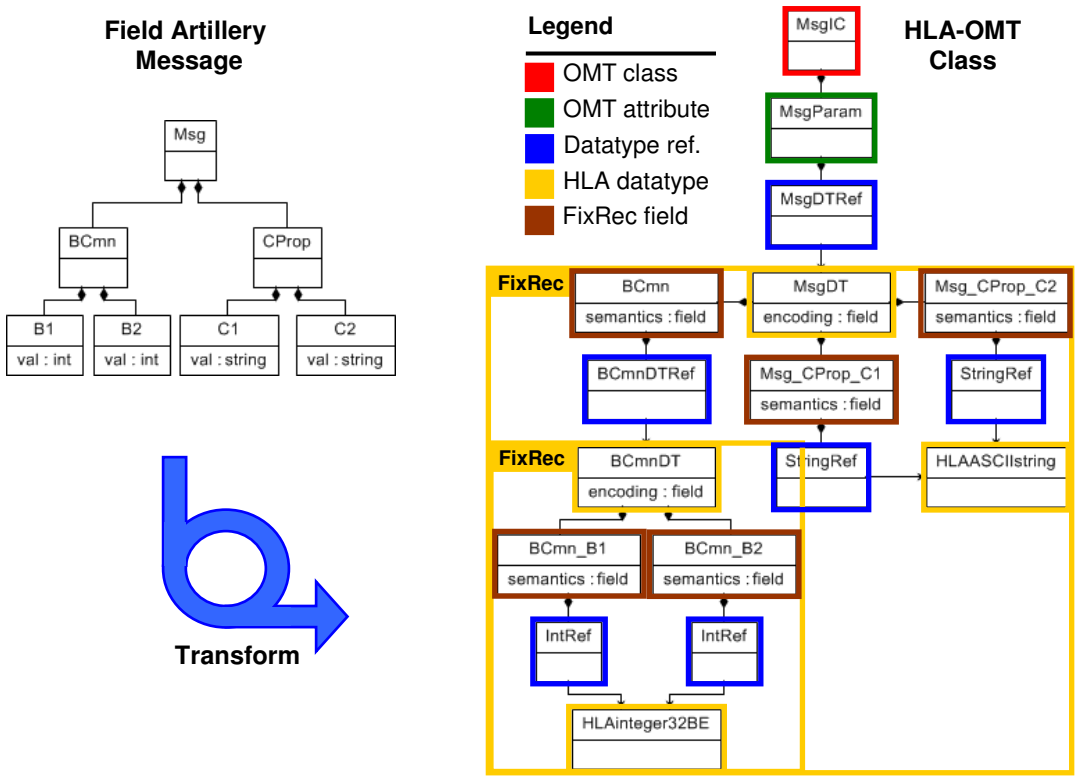


Figure 5.6 A conceptualized field artillery message to OMT transformation

5.2.2 Federation Structure Transformation

The federation structure transformation concludes the data model transformation part. It instantiates the single federation object together with a reference to the FOM that was previously created. It also maps every field artillery Actor and Net to a corresponding HLA federate along with a reference to an associated SOM. In this thesis, SOMs per federate are left as stubs and not developed any further. The FOM is sufficient to capture all the OMT objects participating in the federation execution. Indeed, FOM is what an RTI needs to run a federation [13].

Finally, cross-domain associations establish referencing from each actor/net of the ACM domain to its corresponding federate of the FAM domain. These temporary associations of actor-federate pairs later function as a key enabler in the transformation of message communications among the actors in a field artillery mission to inter-federate communications in the behavioral model transformation step.

The result of the application of the federation structure transformation rules is depicted as a UML sketch in Figure 5.7. It shows a subset of the (hierarchically structured) actor/net collection inside the data model of an ACM, being transformed into their corresponding

federates (flattened) inside the federation structure of a FAM. This one-to-one correspondence is exposed by the blue `has-correspInst` associations. It is further seen that every federate is a `MemberOf` the field artillery federation and that the federation has another association with a reference to the FOM. The naming convention employed in this work assigns the name `FieldArtilleryFed` to the federation element, and an actor/net's name in the field artillery model suffixed by `Fd` to its corresponding federate element.

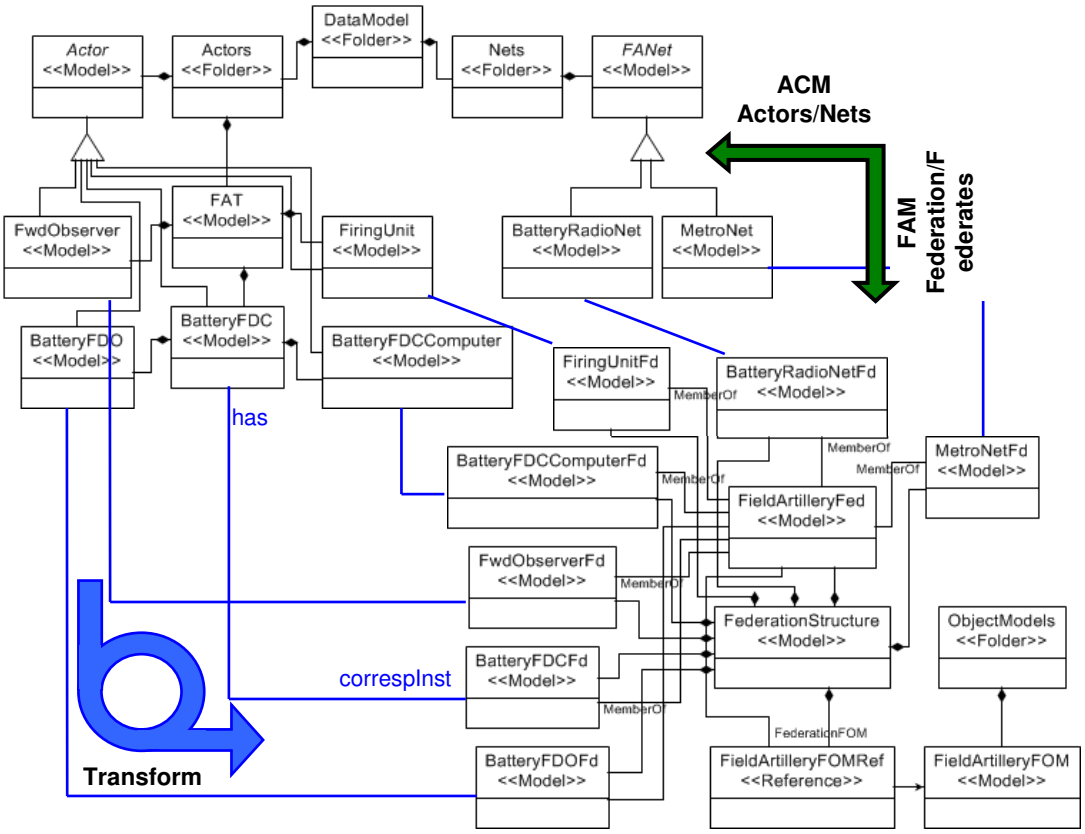


Figure 5.7 Transformation of field artillery actors/nets to HLA federates and federation

5.3 Behavioral Model Transformation

Behavioral model transformation is the bigger and more challenging part of the overall ACM to FAM transformation. It uses the resulting objects of the data model transformation as the instances and message parameters in LSCs that are being produced.

The main block of the behavioral model transformation, `BehavioralModelTr`, is shown in Figure 5.7. `AscGlobalHlaMeths` gets the method library of FAM that contains predefined HLA methods for federation, declaration, object, ownership and time

management. The block contains rules that take copies of all the methods used in the transformation and associate them with the global HLA methods element so that they are readily accessible by the LSC transformation rules. These methods are meant to function as templates; hence their method parameters are left empty. Their copies in the LSCs are assigned parameters with appropriate HLA class instances during the transformation. A simplified and unified sketch of `AscInstanceOfFacm` is also shown in Figure 5.8. The block basically creates `is-InstanceOf` associations between the instances that stand for the same actor element in ACM. An actor instance in the MSC head of an MSC is an instance of the same type of instance in the MSC document head, which in turn is an instance of the canonical actor instance in the data model's `Actors` folder. This chain of associations establishes traceability between the behavior and data sub-models of ACM and provides convenience in subsequent rules. A similar scheme is also applied progressively on the FAM side as the transformation rules construct the model.

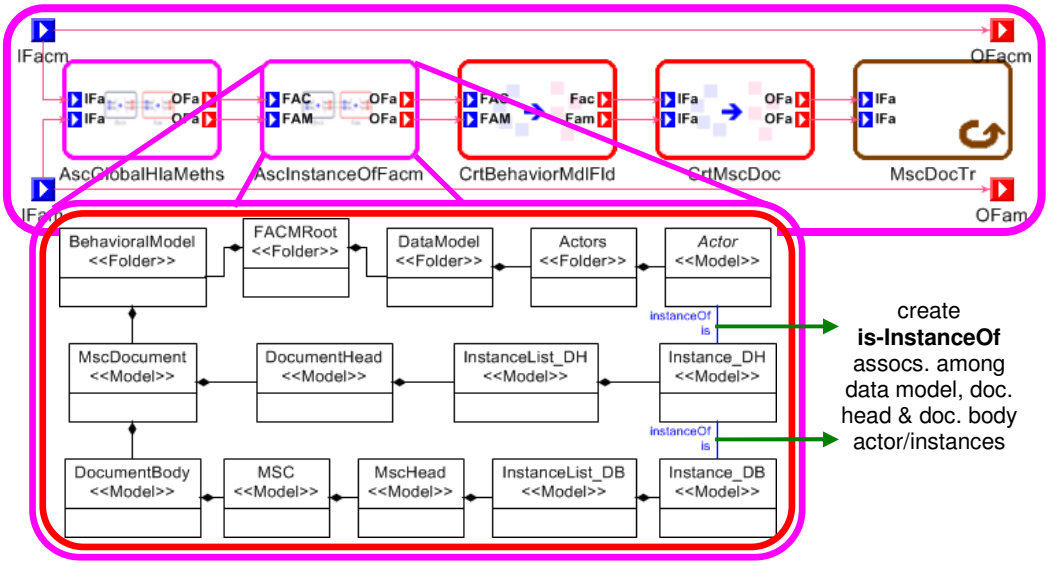


Figure 5.8 The `BehavioralModelTr` and `AscInstanceOfFacm` blocks

The `CrtBehaviorMdlFld` and `CrtMscDoc` rules are triggered one after another for simply creating a FAM behavioral model folder and an MSC document underneath it, provided that their corresponding counterparts are matched in the ACM. A `has-correspMscDoc` association is established between the ACM and FAM MSC documents, since there can be more than one MSC document in a source model and in such a case this

association is necessary for keeping track of MSC references in different documents and during instance decomposition.

5.3.1 MSC Document Transformation

The `MSCDocTr` block is displayed in Figure 5.9. It consists of three sub-blocks, namely, `DocumentHeadTr`, `DocumentBodyTr` and `AscReferences`, executed in that order. All of the blocks and rules within `MSCDocTr` are defined so as to traverse the structure delineated by the MSC metamodel to create a FAM MSC document from an ACM MSC document.

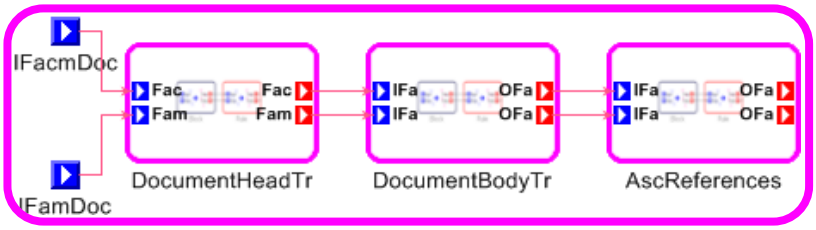


Figure 5.9 The `MSCDocTr` block

To serve as a reference for the reader, part of the metamodel pertaining to the MSC document is reproduced in Figure 5.10. An MSC document consists of a document head and two document bodies of which, the defining part is mandatory and utility part is optional. The MSC body essentially consists of one or more MSCs. The overall LSC/MSC metamodel can be found in [12] in detail. The following two sub-sections describe the document head and body transformations, respectively.

MSC Document Head Transformation

The `DocumentHeadTr` block handles the data definition, message declaration, instance declaration (i.e., the containing clause) and timer declaration parts of the document head of the FAM being constructed. Note also that data definition and message declaration are only addressed as stubs since the content related with these parts are practically provided by the data model.

The instance declaration part of the MSC document head transformation is also one of the key steps in the overall behavioral model transformation. Its role is basically to create federate objects and a federation object derived from the corresponding counterparts found in the federation structure portion of the FAM data model.

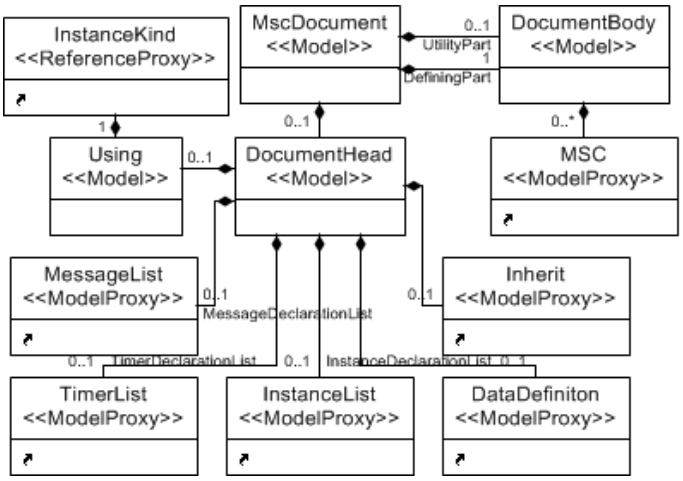


Figure 5.10 Part of the MSC document metamodel [12]

A derived object, which is a deep copy of another structured object, is created inside the attribute mapping code by invoking a UDM API method. This, at the same time creates a sort of inheritance association where, the attribute values of the derived object are kept in sync with the values of the corresponding attributes in the archetype object (i.e. the object that is at the farthest position within the chain of base objects; that is, the one which is not derived from anything) as long as they are modified only through their archetype. Once an attribute's value is modified alone (i.e., directly on the derived object), the attribute becomes de-synched from the archetype, which means that its value is not synchronized to the corresponding attribute's value in the archetype. Please refer to [34] for details.

This architecture is a deliberate design decision in order to have the behavior model content wise backed-up by the data model. With this schema, any attribute update to core federate objects in the data model will be automatically propagated down to the derived objects in the MSC document and from them to the further derived objects in the MSCs of the behavioral model. The benefit of this approach reveals itself when considering this chain of derivations. Another noteworthy choice is calling a generic (in terms of applicability for any type of UDM object) library method instead of employing specific (in terms of being per object type) pattern matching. Otherwise, the pattern matching solution would be bulky, time consuming to define and slower to execute.

MSC Document Body Transformation

The main rule block of the document body transformation, `DocumentBodyTr`, basically transforms the utility and defining parts of an MSC document. Note that it is necessary to

handle the utility part first because the MSCs of the utility part are referenced from within the defining part and therefore they have to exist prior to the defining part transformation.
The MSC document body transformation essentially boils down to MSC transformation. In order to start the process, an empty FAM MSC is created per matched ACM MSC in the given document body. The cross-domain `has-correspMSC` association is established for keeping track of the paired MSCs in subsequent rules. The attribute mapping code copies the chart order index in addition to the name and screen position properties of the ACM MSC to the FAM MSC. The chart order index, although not an artifact of the MSC metamodel, is a crucial annotation that facilitates model interpreters and particularly the code generator, by providing the execution/interpretation order of the MSCs at run-time. Similarly, for multiple documents in a model, the order of the documents may be specified by the document order index [12]. The rule finally delivers both MSCs to the `MSCTrans` block for further building up of the FAM MSC. This fundamental step is elaborated in the following section.

5.3.2 MSC Transformation

MSC transformation is handled by the mainstream `MSCTrans` block, shown in Figure 5.11. Its importance is due to its incorporation of LSC transformation, which virtually is the heart of behavioral model transformation. MSC transformation consists of three consecutive steps that handle MSC head and body transformation, and initialize the federation after the completion of the former two. MSC body transformation essentially boils down to LSC transformation after an empty LSC context is created. LSC transformation is explained in Section 4.3.3.

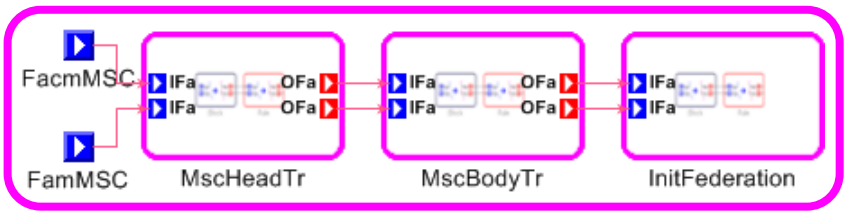


Figure 5.11 The `MSCTrans` block

To serve as a reference for the reader, the top level MSC metamodel is presented in Figure 5.12. An MSC consists of an MSC head and a body. The MSC body can be one of HMSC (High-level MSC), MscBody, LSC or InlineOperand. In this work we practically

use LSC as the MSC body and allow LSCs (having Prechart or Subchart role) and inline operands within LSCs. HMSC and MSC body are not used in this thesis.

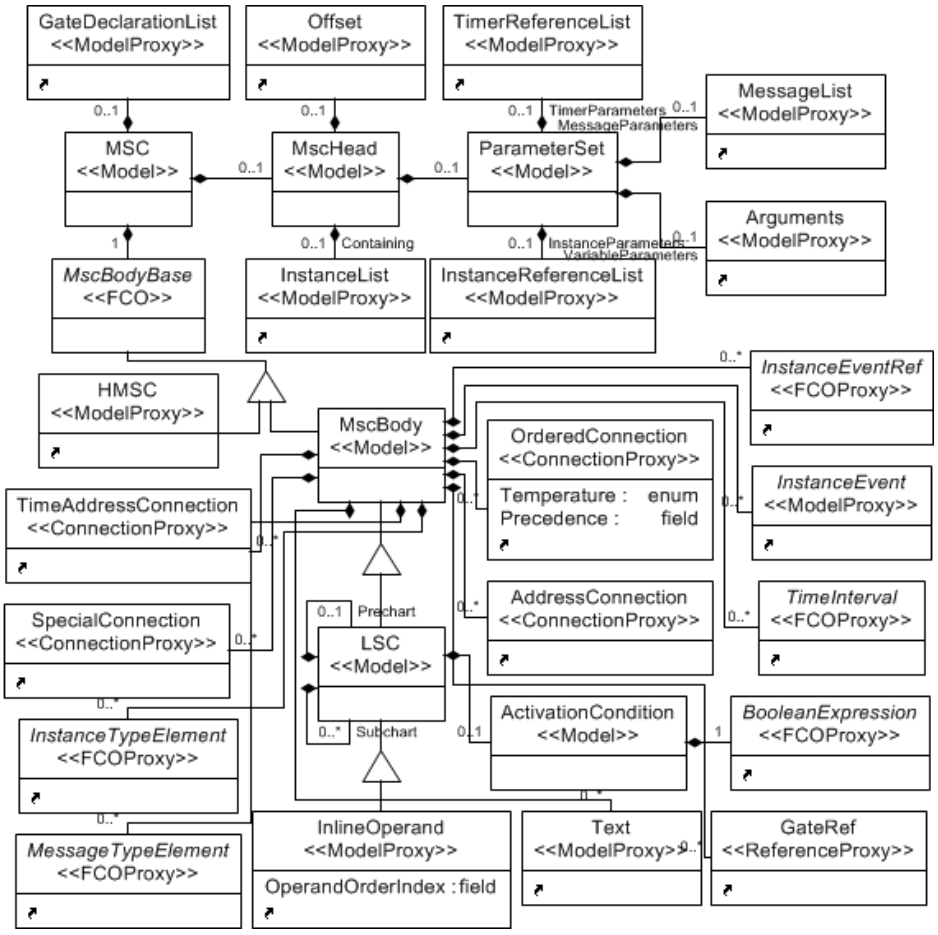


Figure 5.12 Part of the MSC metamodel [12]

MSC Head Transformation

The head part of an MSC is transformed in a four rule block, as shown in Figure 5.13. The head of an MSC houses the instances referenced in the MSC's body, besides other elements. The basic functionality of `MSCHeadTr` is to prepare the instances used in the FAM MSC, by looking at the instances found in the corresponding MSC. Other MSC head components such as offset, parameter set and its subcomponents are either provided explicitly inside the MSC body or considered irrelevant for the purposes of this work and hence, are not covered.

The MSC head transformation also addresses instance decomposition. The MSC specification [15] states that, an instance can be viewed as an abstraction of a whole MSC document (representing a system component) that is participating in a higher level system, hence the mechanism for hierarchical decomposition. The practical outcome of this is the introduction of a separate MSC document per decomposed instance and a new MSC for every MSC in the higher level document that the decomposed instance participates in, which describes the MSC from that instance's lower level perspective. The instance decomposition handling rules are `DecomposeInst` explained in this section and `AssocDecompAsRefs` block residing immediately under the top-level block.

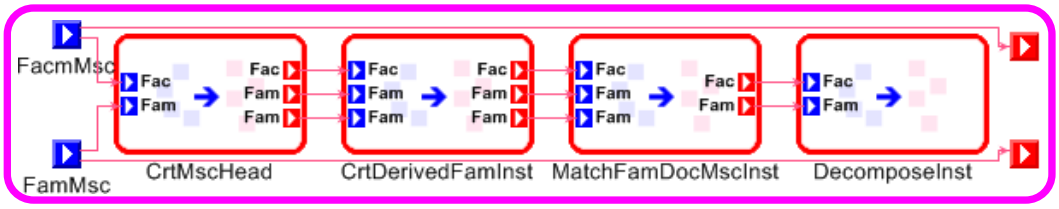


Figure 5.13 The `MSCHeadTr` block

Derived FAM MSC instances are created from the corresponding FAM document head instances and are associated with the ACM MSC instances (using `has-correspInst`) and with the FAM document instance archetypes (using `is-instanceOf`) By this way, structural and one-to-one correspondences are established between and inside ACM and FAM MSCs. This principle is proliferated throughout MSC document, LSC, data element and event transformations. It is a key property of ACM to FAM transformations and provides for traceability and soundness.

If the given ACM MSC instance is defined to be decomposed and has a reference to a specific MSC in the decomposition document, then the FAM instance is also added a decomposed element and a non-assigned MSC reference. These stub references will be assigned later in a post-processing rule after all of the MSC document transformations are completed and hence the entire set of MSCs are created. The attribute mapping code copies property values from the matched source elements to the newly created target elements.

Federation Initialization

Before moving into LSC transformation this section makes a fast forward to explain the federation initialization on the FAM side. The federation initialization is done after an MSC document is transformed head and body-wise (see Figure 5.11). This indicates that it is a post processing step following the full transformation of all the LSCs in the document. (Recall that in a block, a child block or a rule receives input packets after all the packets pass through the previous child.)

The HLA federation initialization activities are done in the `InitFederation` block shown in Figure 5.14. This is a part of the behavioral model transformation indigenous to the FAM domain; that is, there are no associations in the transformation rules to ACM except for the identification of the instances involved. Due to the lack of such an input source, the information content flowing through the federation initialization part is directly embedded inside the transformation rule definitions. This causes the `InitFederation` block to have a substantially hard wired structure. On the other hand, it potentially lends itself for external configuration; that is, the hard wired content can be provided from an outer source, for instance, a GUI front-end, or a configuration file.

The `InitFederation` block handles four preliminary federation execution activities of creating a federation execution, joining federates to the federation execution, initializing time management and declaration management. The federation initialization events are gathered in a sub-chart which itself is placed inside the pre-chart of the top-level FAM LSC. This way, federation initialization is guaranteed to be performed right at the beginning. The subchart is made temperature-wise "*hot*"; hence, mandatory to execute [14]. Since there is no clue from the ACM regarding the execution order of the chart, it is read from a look up table in the user code library; thus, effectively delegated to external configuration.

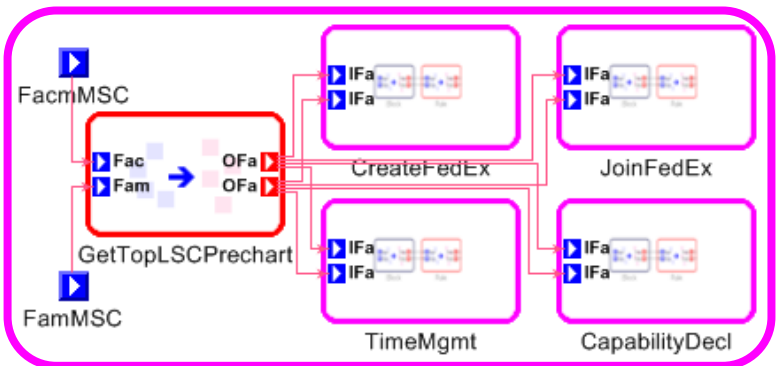


Figure 5.14 The `InitFederation` block

5.3.3 LSC Transformation

The LSC transformation is the heart of ACM2FAM behavioral model transformation. It is the place where the nuts and bolts of the evolution of field artillery inter-entity communications to federate interactions, mediated through the HLA RTI, are defined. The HLA RTI is represented by the `federation` entity specifically introduced in FAM. The LSC transformation process is carried out in the `LSCTrans` block, as overviewed in Figure 5.15. Each pass of the block inputs an ACM LSC and an empty (i.e., stub) FAM LSC, and step by step constructs the FAM LSC as the transformation proceeds through the internal blocks.

The execution order of the sub-blocks does not matter except for the second and the last blocks. The `InstanceRefTr` is a reusable block that has already been utilized in federation initialization. It creates the necessary federate instances (i.e., references) in the FAM LSC by inspecting the ones found in the corresponding ACM LSC. Since these instances are used in the graph patterns of most of the subsequent rules, `InstanceRefTr` must be executed before them. The last block, `SpecialConnsTr`, create associations between two instance events [12] within the LSC and thus need to be executed after ensuring all such events have been created. The activation condition is a boolean condition, which expresses the activation point for a chart [12]. Activation condition transformation is performed in the `ActivationConditionTr` block. There is a simple one-to-one correspondence and equivalence between ACM and FAM activation conditions. The definition of the LSC transformation blocks are generally based on the instance event type categorization of the child elements to be processed in the LSC. These blocks are briefly explained in the rest of the section.

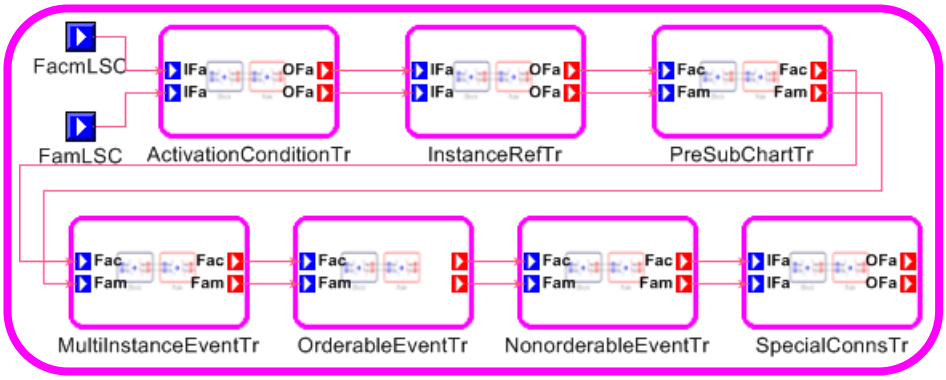


Figure 5.15 The `LSCTrans` block

Prechart and Subchart Transformation

Precharts and subcharts are actually child LSCs that have special role names on the containment associations with their parents. The `PreSubChartTr` block, shown in Figure 5.16, handles the transformation of precharts and subcharts of an LSC. The `CreateSubSchart` rule creates a subchart under the current FAM LSC with the `Subchart` composition role for every subchart of the corresponding ACM LSC. The `CreatePreChart` rule is defined similarly. A notable statement in attribute mapping code (partly shown in the figure) is the call to the `SetInstRefAssocs4LSCChildren` method of the user code library. This method is invoked for all LSC child creations of type LSC (pre/subchart) and multi instance event, including inline expressions (`Loop`, `Opt`, `Exc`, `Par`, `Alt` and `Seq`), references, conditions, otherwise clauses, and LSC idioms [12]. It handles the routine task of creating associations between an LSC's child elements and the relevant instances in the LSC programmatically. This extensively used method could be implemented with transformation rules, but that would require as many rules as the LSC child types listed above and take longer to execute considering the execution speeds of pattern matching vs. direct C++ invocation.

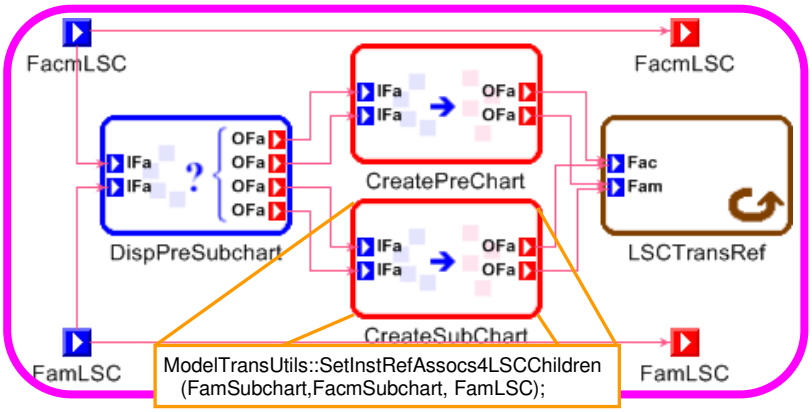


Figure 5.16 The `PreSubChartTr` block

The role of `DispPreSubchart` test is to direct the execution flow to one of the attached rules based on the child element type of the input ACM LSC being a prechart or a subchart. After a child pre/subchart is created under the given FAM LSC, the block ends with a recursive call to the `LSCTrans` block to continue the transformation for the child element, which is yet another LSC.

Multi-Instance Event Transformation

This section explains the transformation of multi-instance events, which constitute a set of frequently used elements, including conditions, otherwises, inline expressions and references (to MSCs). The top-level block, `MultiInstanceEventTr`, is depicted in Figure 5.17. Initially, a child multi instance event of the ACM LSC is matched and dispatched to one of the three alternative transformers together with the FAM LSC.

The `CreateCondition` and `CreateOw` rules perform condition and otherwise transformations, respectively. These rules simply create FAM elements that directly correspond to matched ACM elements. The other types of multi instance events form the family of reference identifications and are handled in the RefIdentTr block, also shown in the figure. Reference identification types are inline expressions and reference. The `CreateReference` and `CreateMSCRef` rules simply create a FAM `Reference` element and a reference to an MSC under that, respectively.

The inline expressions are transformed in the `InlineExpTrans` block. The block initially directs the execution flow to one of the nine inline expression creator rules based on the input ACM inline expression type. Six of these create `alt`, `par`, `opt`, `loop`, `exc` and `seq` elements [15], and three of them create `if-then-else`, `while-do` and `repeat-until` idioms [12]. These rules simply create FAM inline expressions for the given ACM inline expressions and link them together using the `has-correspInlExp` cross-domain association. The attribute mapping codes copy the element properties.

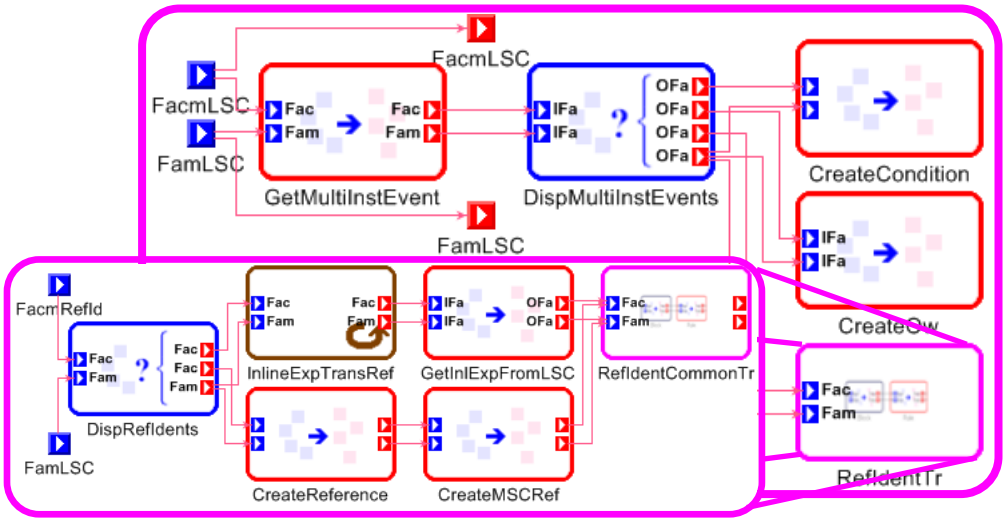


Figure 5.17 The `MultiInstanceEventTr` and `RefIdentTr` blocks

Every inline expression by definition contains one or more inline operands. After the creation of inline expressions, the execution flow joins into a single path to create inline operands. Inline operand is defined to be specialized from LSC [12], in the sense of UML inheritance. The only addition brought in by inline operand is the operand order index property that specifies the order of the operand with respect to the other peer operands within the inline expression. Finally, paired inline operands are recursively handed over to the `LSCTrans` rule for further processing as LSCs.

The `RefIdentCommonTr` is the last, sink block of the `RefIdentTr` block that creates gate, top, bottom and time interval components common for all reference identification type of elements. Time interval transformations further specialize into measurement, singular time and bounded time transformations. All of these rules are quite intuitive and perform ACM to FAM attribute value copying in a straightforward manner.

Orderable Event Transformation

Orderable events are generally the most frequently used set of events in the behavioral model of field artillery scenarios. They form the mechanism for the actual communication among the instances (i.e., actors in ACM terms). The top level `OrderableEventTr` block is shown in Figure 5.18. The block starts by matching and dispatching a LSC contained ACM orderable event to the appropriate rule or block to create its FAM counterpart. The kinds of orderable events handled are action, create, timer event, method event, and message event. Once these events are transformed in their specifics, any general orderings (i.e., before and after) imposed on them are applied in the final block `GeneralOrderTr`.

The `HandleAction` rule is also provided in the figure as an example to explain how a typical orderable event rule works. For any given ACM action, a new FAM action is created in the given parent FAM LSC. Also, the ACM instance that is in association with the matched action is identified. From that, the corresponding FAM instance reference is obtained using the cross-domain association, `has-correspInstRef`. Then a similar association is established between the new FAM action and the FAM instance reference. Finally, the matched ACM action and the created FAM action are passed to the next rule in line.

The timer events, consisting of start timer, stop timer and timeout, form a sub-category of orderable events. The `TimerEventTr` block performs the transformation of timer events. The block initially dispatches a matched ACM timer event and a FAM LSC to one of the three timer event creator rules. After the event creations, instance reference - timer event associations are established in the same manner shown in `HandleAction` rule of

Figure 5.18. Timer events contain references to timer elements. Finally, the references to timers are set for the FAM timer events. In LSC metamodel [12], timer objects are stored in the timer list of the document head of the MSC document. Since the document head is processed before the LSCs, all of the timers should be readily available.

The `MethEventTr` block handles the transformation of call, receive, replyout and replyin events that constitute method call event category. These transformations are quite straightforward and handled similar to the `HandleAction` rule explained above.

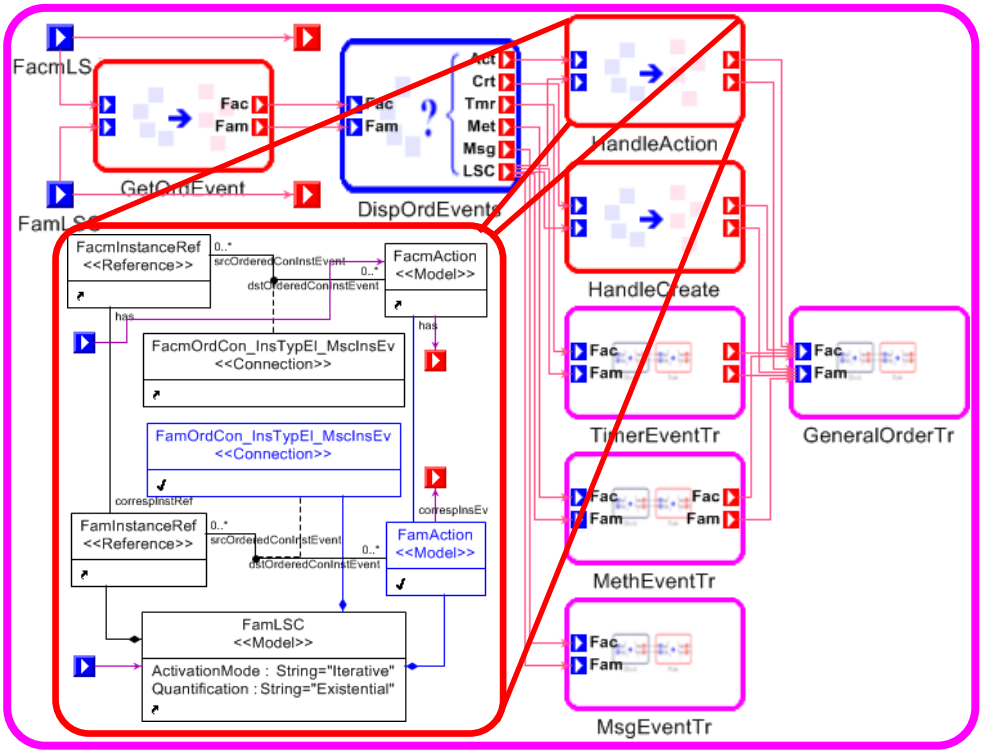


Figure 5.18 The `OrderableEventTr` block

Message Event Transformation

Message events are the most common and important group of orderable events that represent the inter instance (i.e., actor in ACM and federate in FAM terms) communications. Their role is so crucial that message event transformations can be regarded as the crux of this ACM2FAM transformation work. Except for the HLA federation specific initialization and tear-down rules (see Section 4.3.2), most of the other types of transformations are generally dominated by same type of LSC element creating

and value copying in the FAM being constructed per matched LSC element in ACM. On the other hand, message event transformations are more sophisticated in that there are one-to-many event creations that go beyond simple copying of ACM content into FAM. Moreover, these transformations are driven by conditions that take into consideration the type and structure of the ACM data being communicated. This is the primary part where platform (i.e. HLA) specific content is introduced. Because of these, message event rules are more complex and bigger than the other rules.

The main message event transformation block, `MsgEventTr`, is displayed in Figure 5.19. It distributes the incoming packets according to the type of the matched ACM message event. Out and in events are the two kinds message events and are the conjugate of each other in that for two interacting instances A and B, every out-event from A to B implies a corresponding in-event sourced B and targeted A, sequenced in that order [15]. We assume that the transmission of an out-event and its implied in-event are atomic and instantaneous in time, occurring immediately one after another. Because of these, we have conventionally modeled source ACMs having out-events as the sole message event type.

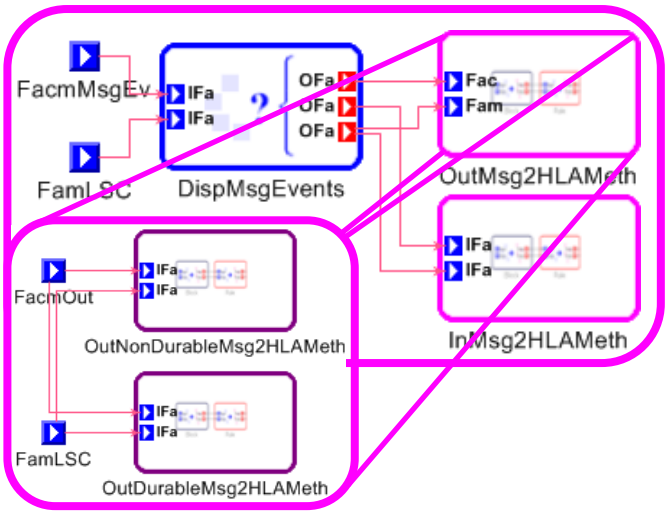


Figure 5.19 The MsgEventTr and OutMsg2HLAMeth blocks

On the other hand, the situation is different in a FAM, in that, any federate to federate communication has to be mediated via the HLA RTI (the federation execution, to be more specific), as dictated by the HLA specification [13]. This loosely coupled communication architecture would normally necessitate an actor A to B out-event transmission in an ACM

to be represented as federate A sending an out-event to the federation first and the federation sending another out-event to federate B. However, instead of having these two explicit outs (and two implicit ins), we have decided to implement one explicit out-event between federate A and the federation and an explicit in-event between federate B and the federation, explicitly employing both in and out-event types. In this setting, if the out-event has order *n*, the in-event is given order *n+1*. (Note that ordering is implemented by incrementing a counter). The approach is diagrammatically illustrated in Figure 5.20. This federate centric event mapping better supports the code generator's code generation strategy which considers each LSC instance (i.e. federate) and its associated events individually while producing the federate base code and computation aspect code [61].

The `OutMsg2HLAMeth` block, also shown in Figure 5.19, handles the transformation of out-events. Within the block, both ACM and FAM input packets are fed to two for-blocks in parallel that are specialized in out-event transformations based on the ACM message payload type. Non-durable message out-events are transformed inside `OutNonDurableMsg2HLAMeth` for-block and durable message out-events are transformed inside `OutDurableMsg2HLAMeth` for-block. Non-durable message transformation is relatively simpler than the durable, because a non-durable message transmission in ACM maps to two HLA message transmissions in FAM, whereas a durable message transmission can map up to six.

Since there are no in-events used in source ACMs, there is no practical need for an in-event transformer counterpart. Therefore the `InMsg2HLAMeth` block is created for the sake of completeness, but left as a stub. The collection of all in-events on the FAM side are created only as a result of out-event transformation as explained above and shown in Figure 5.20.

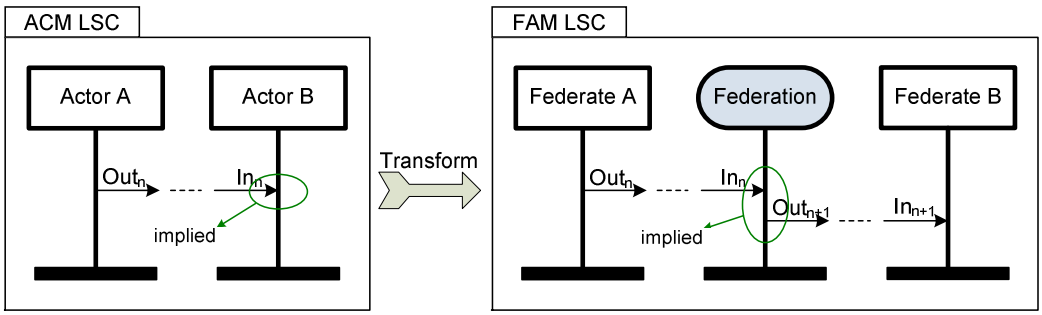


Figure 5.20 ACM out event to FAM out/in event federation execution

Before moving into non-durable and durable message transformations, it is worth to clarify that, the durability and non-durability of ACM message structures is not imposed by a standard or field manual, but is introduced in ACMM at the end of our field artillery domain analysis as a convenience for providing a correspondence to FAM object and interaction classes. Please refer to Section 4.2.1 and [11] for details.

Non-durable Message Transformation

The `OutNonDurableMsg2HLAMeth` block that handles non-durable ACM out-message transformation is sketched in Figure 5.21. The initial rule, `GetNDMsg`, matches and delivers the ACM out-event, non-durable message and FAM LSC to the next rule, and in the meantime, programmatically creates a copy of SendInteraction and ReceiveInteraction HLA methods using the attribute mapping code. The template HLA methods have already been created at a preliminary step, and associated with the global root element for quick access (see Section 4.1). The original methods do not contain any arguments, but their copied instances will have theirs assigned (such as HLA classes and federate references) as the transformation proceeds.

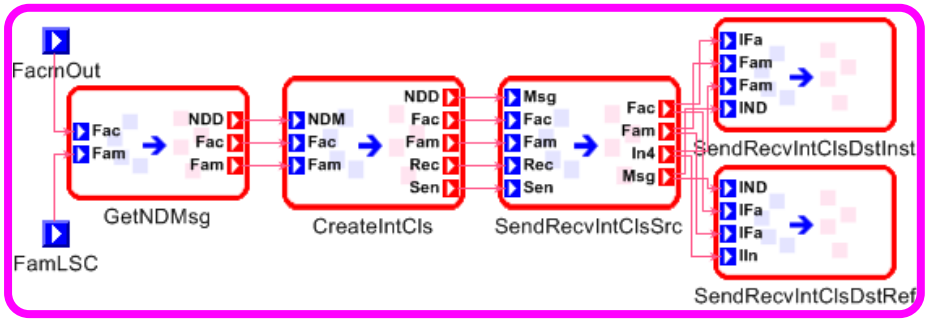


Figure 5.21 The `OutNonDurableMsg2HLAMeth` bock

The `CreateIntCls` rule creates a new interaction class corresponding to the ACM non-durable message in the FAM FOM. It also sets both of the HLA methods to refer to the new interaction class inside their supplied arguments. Finally, it assigns the name of the non-durable message suffixed by "`IC`" as the name of the new interaction class, and invokes the user code library to build the interaction class from the non-durable message.

The `SendRecvIntClsSrc` rule of Figure 5.22 is one of the most crowded rules in the transformation that actually define the federate-to-federate HLA method transmissions via the federation. It first creates a message out-event and associates it with the source instance

(i.e., federate) using an ordered connection. Then it associates the out-event to the send interaction method using a special connection. Finally, it associates the send interaction method to the federation instance using an address connection.

After that, a similar set of activities start for the receive interaction method from the federation to the target federate. First the receive interaction method is associated to the federation instance using an address connection. Then an in-event message is created and associated to the receive interaction method using a special connection.

The last part of the out-event transformation is done by one of the two parallel rules `SendRecvIntClsDstInst` and `SendRecvIntClsDstRef`. They similarly associate the new FAM in-event to a target instance or an MSC `Reference`, respectively.

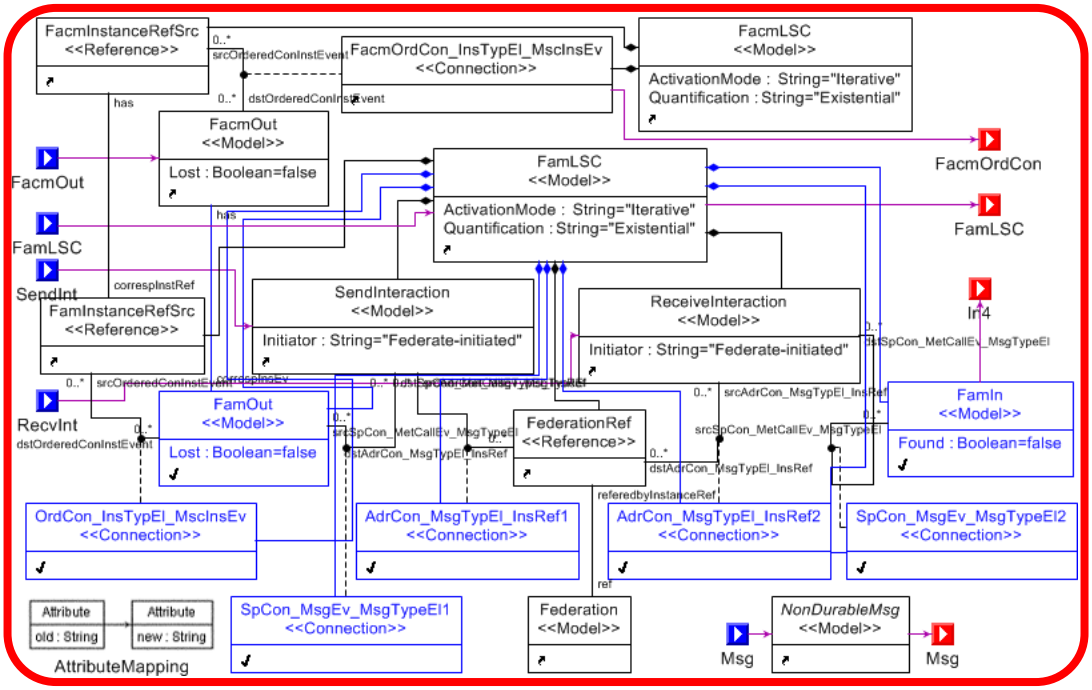


Figure 5.22 The `SendRecvIntClsSrc` rule

The outcome of the non-durable message transformation process is illustrated in Figure 5.23, showing the partial view of an ACM LSC and its corresponding FAM LSC (in abstract syntax) produced as the result of executing `OutNonDurableMsg2HLAMeth` transformation block. The model element stereo-types are tagged in the figure. In the source LSC is seen an `Oid_W message` out-event sent from `FwdObserver` to `BatteryFDC`. On the produced LSC, this corresponds to two HLA message event transmissions. The figure

shows the FwdObserver federate sending a message out-event of SendInteraction to the field artillery federation, and BatteryFDC federate receiving the corresponding message in-event of ReceiveInteraction from the federation. Sequencing (i.e., precedence attribute) information of the message transmissions are annotated in the callout boxes of the figure. The precedence value of the ACM message event is copied to the initial FAM message event, and its auto-incremented value is assigned to the second event. The algorithm used ensures a conflict-free generation of ordering values throughout the FAM LSC. Both of the HLA methods have references to the same interaction class of Oid_W_MsgIC type, which corresponds to the transformed ACM message, as their supplied arguments (not shown in the figure).

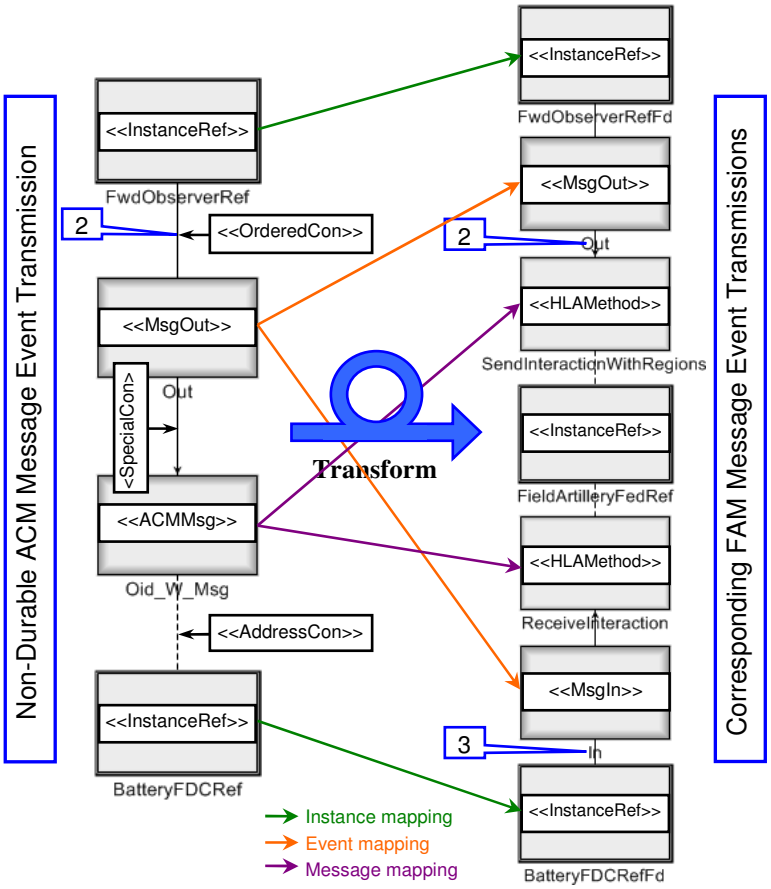


Figure 5.23 Partial view of non-durable message transformation and its result in FAM

Durable Message Transformation

Durable message transformation is the most complicated of the LSC instance event transformations. Figure 5.24 displays the big OutDurableMsg2HLAMeth bock. It is

defined methodologically similar to `OutNonDurableMsg2HLAMeth` block, most notably being about three times in size. Therefore it is regarded redundant to explain the transformation in detail, but appropriate to provide an overview of the differences.

The durable messages in ACM are defined to be of three types; namely, instantiation, update and deletion (please refer to Section 4.2.1 and [11]). There are three parallel courses of transformations that address message out-events of each durable message type. An ACM instantiation type message out-event maps to six FAM HLA message out-events. The mapping cardinalities of an out-event for update and delete types are both one to two.

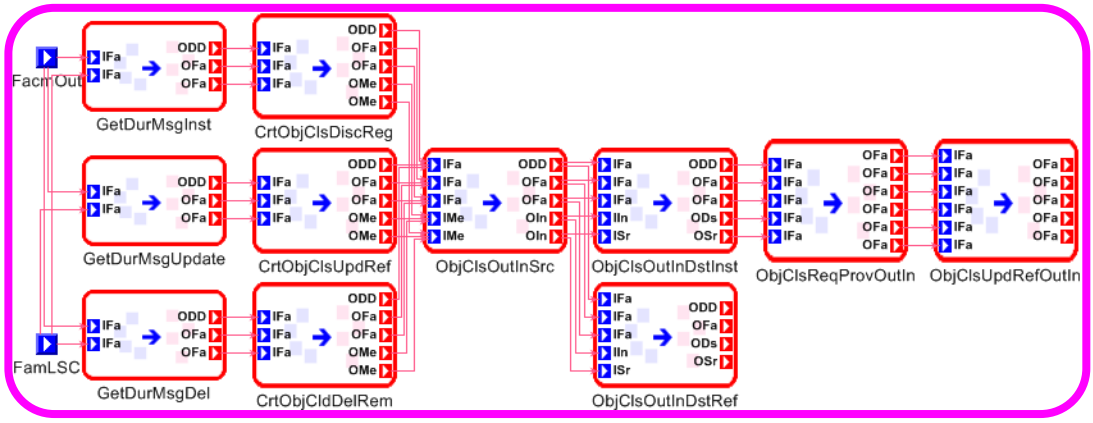


Figure 5.24 The `OutDurableMsg2HLAMeth` bock

The outcome of the instantiation type of durable message transformation process is illustrated in Figure 5.25, showing the partial view of an ACM LSC and its corresponding FAM LSC (in abstract syntax) produced as the result of executing `OutDurableMsg2HLAMeth` transformation block. The model element stereo-types are tagged in the figure. The source LSC has a single `FireCommandSOPInst` message sent from the `BatteryFDC` to the `FiringUnit`. On the produced LSC, this corresponds to six HLA message out-event transmissions. The figure shows the `BatteryFDC` federate sending a message out-event of `RegisterObjectInstance` to the field artillery federation, and the `FiringUnit` federate receiving a following message in-event of `DiscoverObjectInstance` from the federation. Then the `FiringUnit` federate sends a message out-event of `RequestAttributeValueUpdate` to the field artillery federation, and the `BatteryFDC` federate receives a following message in-event of `ProvideAttributeValueUpdate` from the federation. Finally, the `BatteryFDC`

federate sends a message out-event of `UpdateAttributeValues` to the field artillery federation, and then the `FiringUnit` federate receives a message in-event of `ReflectAttributeValues` from the federation. Ordering information of the message transmissions are annotated in the callout boxes of the figure and are produced similarly to non-durable message transformation case explained above. Both of the HLA methods in each of the three message event transmissions have references to the same object class of `FireCommondSOPInst_MsgOC` type, which corresponds to the transformed ACM message, as their supplied arguments (not shown in the figure).

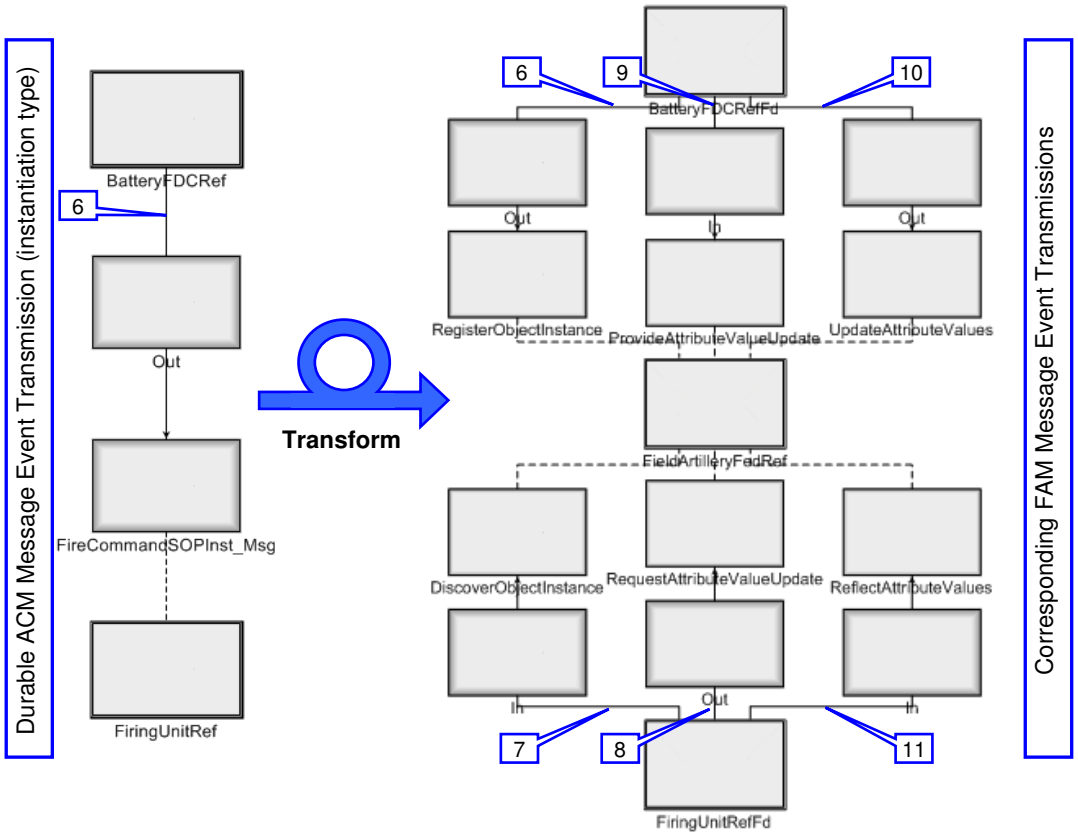


Figure 5.25 Partial view of (instantiation type) durable message transformation and its result in FAM

Non-orderable Event Transformation

The non-orderable events constitute the set of instance events that do not require an explicit ordering of execution. A relative execution order among the events of an instance is implicit along the axis line of an instance; that is, the events that are attached higher up

along the axis execute before the ones attached lower. However, no claim can be made about the execution order of two disjoint events on separate instance axes without using explicit ordering [15]. The `NonorderableEventTr` block performs the transformation of non-orderable events. The block matches and dispatches the input packets to one of the handler rules according to the type of the ACM non-orderable event. The block is defined similar to the `OrderableEventTr` block in Figure 5.18 and its handler rules are defined similar to the simple rules shown there. The handler rules perform the transformation of method, end-method, concurrent, end-concurrent, suspension, end-suspension, stop, end-instance, invariant, end-invariant and simultaneous region.

Special Associations Formation

Most of the LSC transformation blocks and rules are related with instance event transformations, which generally involve associations between instance events and instances. This top down instance event driven approach successfully addresses the majority of the LSC transformation spectrum. However, there is a small set of LSC structures not covered up to now that does not involve instances, such as special associations pertaining only to events. The `SpecialConnsTr` block placed at the end of the LSC transformation path, is responsible for the transformation of those parts. It is deliberately positioned as the last LSC transformation block because it requires all of the FAM LSC entities to be already created and available by the time it starts execution; otherwise, it is likely to miss the transformation of some special associations.

Figure 5.26 shows the `SpecialConnsTr` block, which is the transformer for special associations. There are three kinds of special connections used in this work, namely, the ones that associate simultaneous regions to instance events, timer starts to timer events and general order elements to ordered events. An in-depth examination of the LSC specification and the metamodel might reveal some more special association types, but they are out of the scope of this work and indeed are rarely used in practice. The figure additionally shows the `AscSimRegToInstEv` rule as an illustrative example. For any ACM simultaneous region that is specially associated with an instance event, the rule matches their corresponding FAM simultaneous region and the instance event by utilizing their cross-links to FAM. Then a similar kind of special association is established between the two FAM elements. The other two special connection transformations are defined with the same approach.

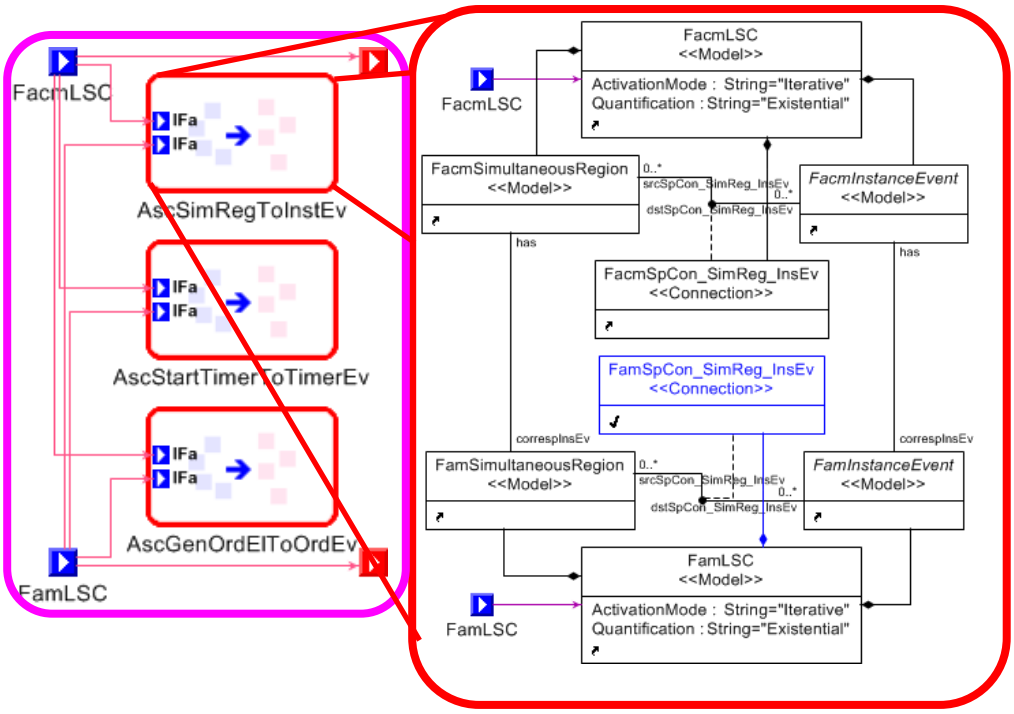


Figure 5.26 The `SpecialConnsTr` block

5.4 Multiple Instance LSC to Binary Instance LSC Transformation of FAM

The behavioral transformation of ACM2FAM is a one to one MSC transformation from ACM to FAM; that is, a corresponding element of the same type is created on the FAM side for each MSC document, MSC and LSC of ACM. Furthermore, the content of an LSC is transformed as described in Section 5.3 in detail. At the end of the transformation, an equal number of federates to the number of actors in an ACM LSC plus one federation instance are created in the corresponding FAM LSC.

However, a FAM with this structure does not fully comply with the input requirements of the code generator. As explained in [61], the code generator by design expects and generates code only for one instance (i.e., federate application) in an LSC. If there are more than one instances, exercises have shown that code is generated only for the first one and the others are simply ignored. The LSC instance is the focal element in the code generation process, and ultimately all LSC instances are generated in separate class files and they are declared and used in the diagram code generated from the LSC diagram. Instance codes drive the simulation and each instance runs in its own thread.

Under these circumstances, a generated FAM has to be refactored into an organization completely processible by the code generator. In simplest terms, every LSC that contains

multiple federate applications has to be stripped down into as many *binary* LSCs as the number of federate applications, each containing one federate application and the federation. This process is depicted in Figure 5.27. In this way, every binary LSC only contains its federate's mutual communication with the federation – a closer organization towards a local, federate-oriented view. Note that the stripping process may end-up in loss of event orderings in binary LSCs that were implicitly known in multi LSCs due to transitive chaining of events among the instances.

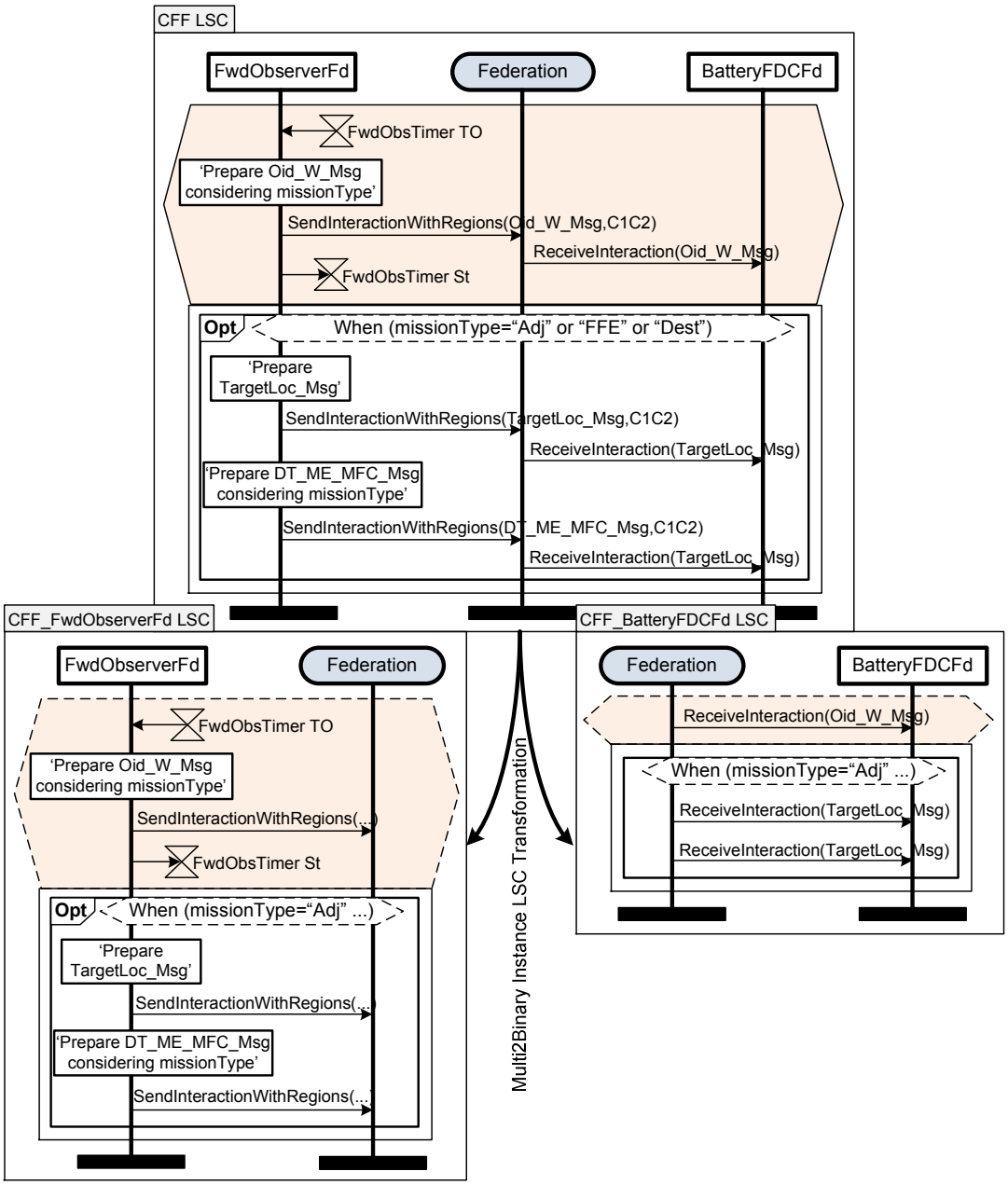


Figure 5.27 Stripping a multi-instance FAM LSC into binary-instance LSCs

In an effort to adapt a FAM produced as the result of an ACM2FAM transformation for code generation, we have developed yet another GReAT transformation that refines a FAM having multiple instance LSCs into another FAM having two instance (i.e., one for the federate and one for the federation) LSCs. The following sub-sections elaborate on this transformation, named *Multi2BinaryLSC*.

5.4.1 Initializing Multi2BinaryLSC Transformation

The Multi2BinaryLSC is configured first to create a copy of the input model and then perform the transformation on that copied model. The start block is shown in Figure 5.28. It consists of the bigger `Multi2BinaryTr` that handles the transformation without handling the MSC references. The last block, `AscMSCRefs_M2B`, is a kind of post-processing step that binds the unbound MSC references among each other.

The `Multi2BinaryTr` block consists of two rules, one block and one rule defined in sequence. The first rule creates temporary associations between the MSC document, federation and environment elements for easy referencing in subsequent rules. The second rule simply matches each MSC in an MSC document and passes it to the main block. The last rule deletes all of the multi instance MSCs after they are stripped down.

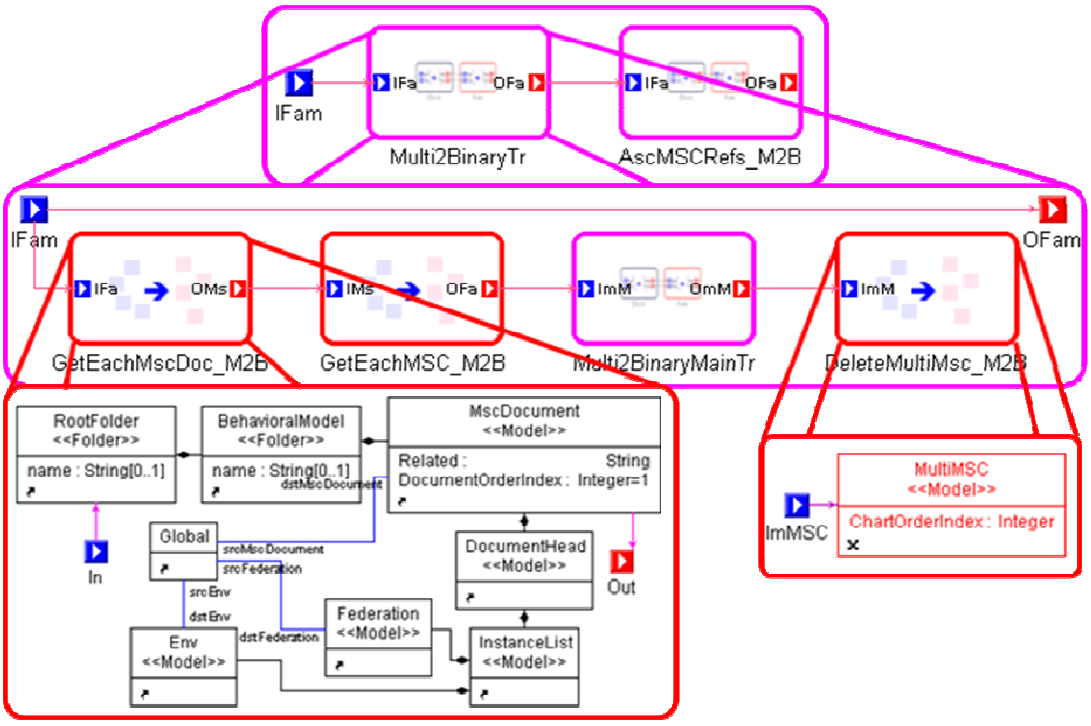


Figure 5.28 The `Start` and `Multi2BinaryTr` blocks

5.4.2 Creating Binary MSCs and LSCs per Federate

The main transformation block is expounded in Figure 5.29. It consists of an initializer rule, `InitBinaryMSCLSC`, and a reference to the LSC transformer block. `InitBinaryMSCLSC` is a crucial rule where the crux of the multi-to-binary stripping is done. It matches every federeate application in the multi-MSC and creates a binary MSC and LSC for each. It also creates a new MSC head and instance list for the binary MSC and creates a new federate application inside the instance list corresponding to the matched federate of the multi-MSC.

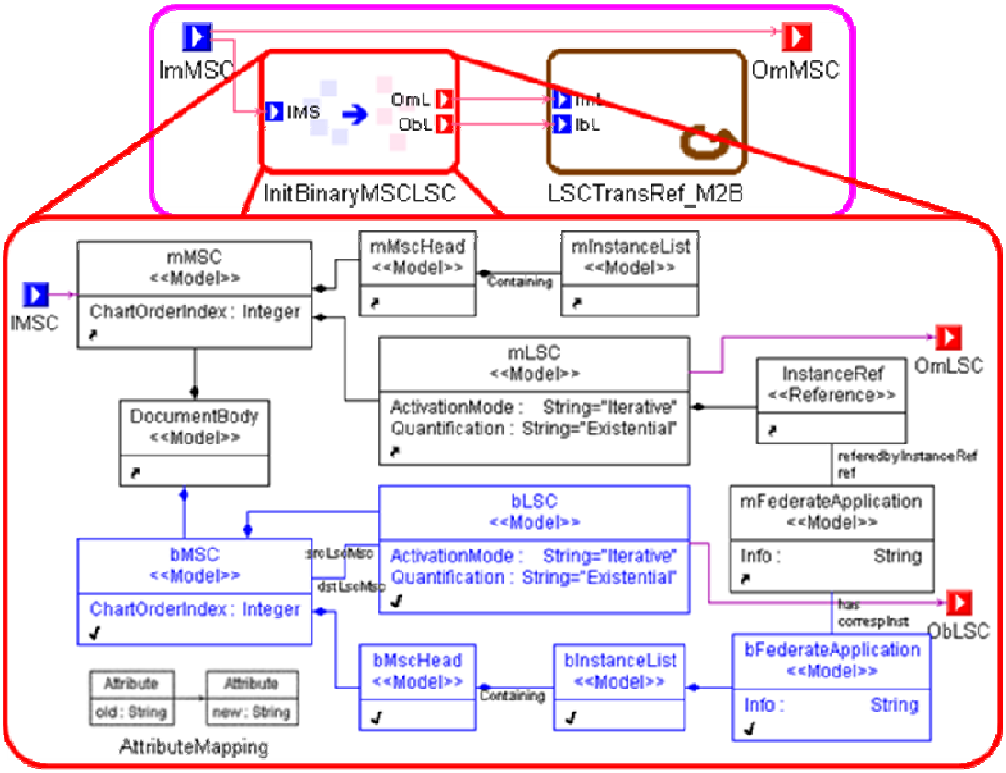


Figure 5.29 `Multi2BinaryMainTr` block and `InitBinaryMSCLSC` rule

The attribute mapping code of `InitBinaryMSCLSC` rule is presented in Table 5.2. The naming convention for binary MSC (and LSC) name is the concatenation of the multi MSC (and LSC) name with the name of the federate application in question. Another issue to resolve during the stripping process is to calculate the positions in terms of coordinates and the chart order index properties of the binary MSCs inside the document body. Chart order index specifies the execution order of an MSC with respect to others in the containing MSC

document. Every invocation of the `GetNextChartOrderIndex()` method of the user code library returns an ever increasing relative offset value that initially starts from 0, and is reset per multi MSC. Finally, the activation modes and quantifications of the binary LSC are copied from the multi LSC.

Table 5.2 `AttributeMapping` code of `InitBinaryMSCLSC` rule

```
bMSC.name()=(std::string)mMSC.name()+"_"+(std::string)mFederateApplication.name();
bLSC.name()=(std::string)mLSC.name()+"_"+(std::string)mFederateApplication.name();
bFederateApplication.name()=mFederateApplication.name();
bMscHead.name()=mMscHead.name();
bInstanceList.name()=mInstanceList.name();
int chartOrderIndCnt=ModelTransUtils::GetNextChartOrderIndex();
int yPos = 100*(1+chartOrderIndCnt);
char yPosStr[10];
_itoa_s(yPos, yPosStr, 10);
bMSC.position()="(100,"+string(yPosStr)+")";
bMSC.ChartOrderIndex()=(__int64)mMSC.ChartOrderIndex()+chartOrderIndCnt;
bLSC.position()=mLSC.position();
bMscHead.position()="(100,100)";
bLSC.ActivationMode()=mLSC.ActivationMode();
bLSC.Quantification()=mLSC.Quantification();
```

5.4.3 Multi to Binary LSC Transformation

The bulk of the transformations consist of multi to binary LSC transformations, collected under the `LSCTrans_M2B` block which is presented in Figure 5.30. The block is generally organized in a similar structure with ACM2FAM's LSC transformation as explained in Section 5.3.3. Differences of special interest are emphasized in this section. Note that within this and all of its subordinate blocks, the LHS context is the multiLSC and the RHS context is the binaryLSC, as input by the blue ports and output by the red ports.

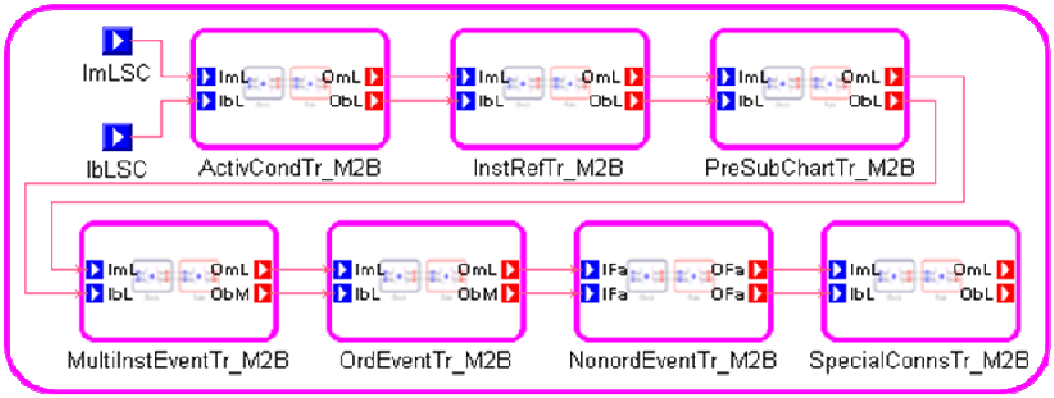


Figure 5.30 `LSCTrans_M2B` block

A unique approach is in setting the instance references in a binary LSC to their corresponding federate applications in the instance list of the binary MSC, as shown in Figure 5.31. The key facilitator is the temporary `has-correspInst` association that has already been established in a previous `InitBinaryMSCLSC` rule execution. The attribute mapping code (not shown in the figure) copies the name and position values of the multi instance reference to the binary instance reference.

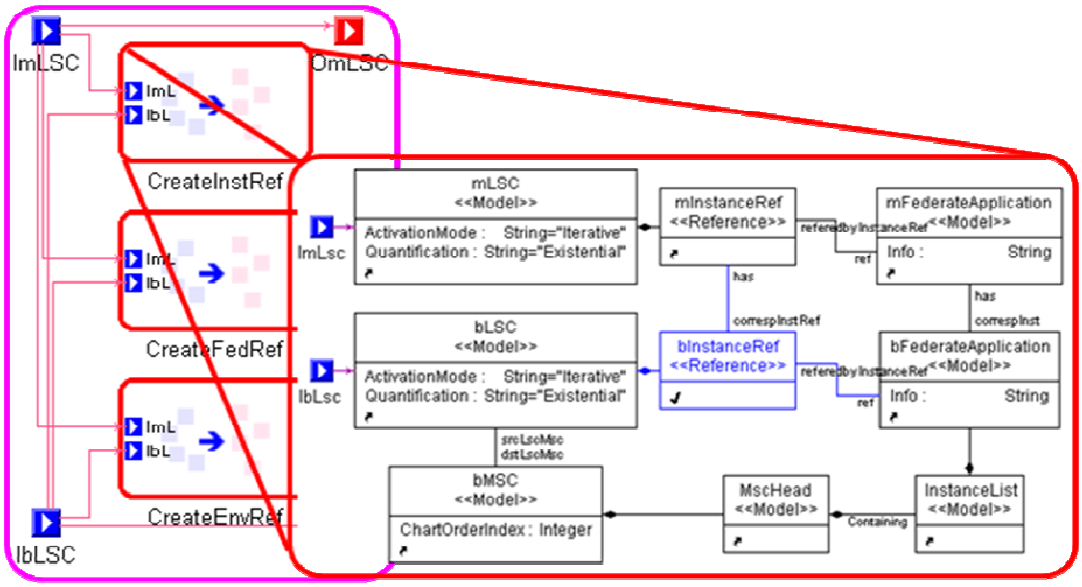


Figure 5.31 `InstRefTr_M2B` block and `CreateInstRef` rule

Another interesting rule to demonstrate is the binary subchart creator rule depicted in Figure 5.32. The originality here is not in the rule pattern, which is more or less the same as its ACM2FAM counterpart, but in the employment of a new temporary association and a guard expression. The `srcLscMsc-dstLscMsc` association is used to maintain a direct link from every subordinate LSC type; that is, prechart, subchart or inline operand, to its MSC ancestor, eliminating the need for the commonly used and expensive `GetMSC4LSC` block, which is explained in Section 6.5.2 and shown in Figure B.59 with details. The `ifHasCorrespInstRef()` method checks whether the given multiSubchart contains an instance reference that has a corresponding instance reference inside the given binaryLSC. It acts as a filter to uniquely identify the binaryLSC inside which to create a binarySubchart that corresponds to the multiSubchart. The guard is extensively used in the rest of the `LSCTrans_M2B` rules.

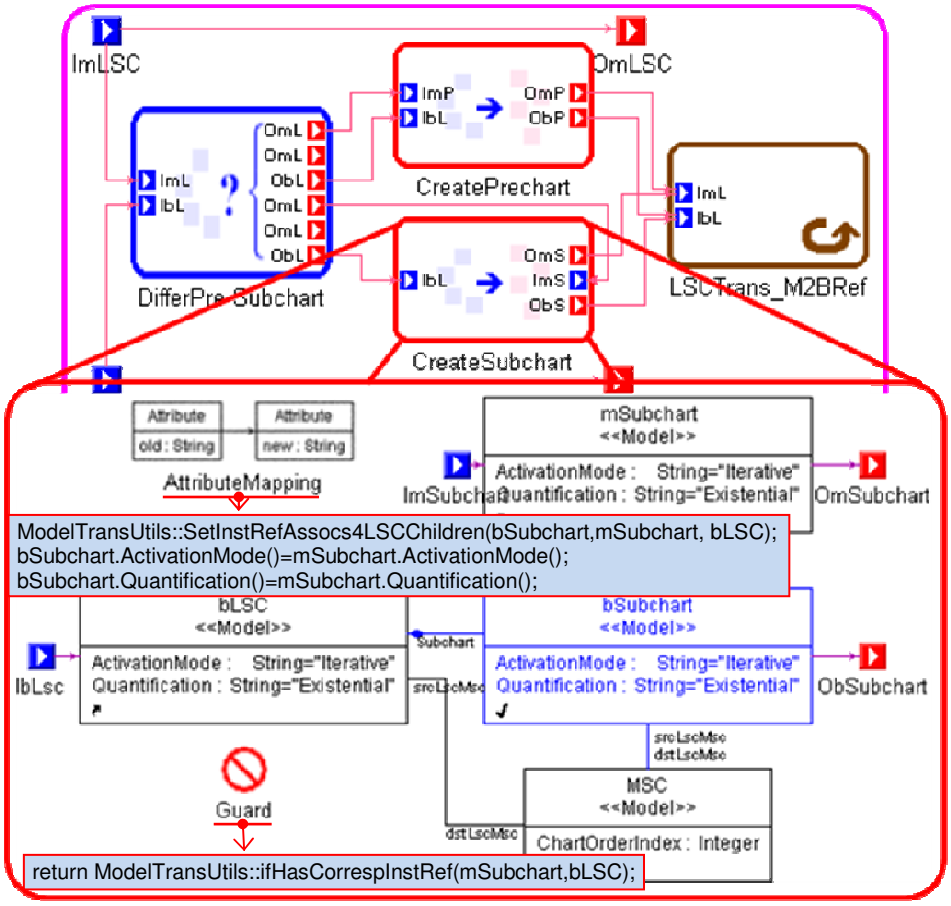


Figure 5.32 `PreSubChartTr_M2B` block and `CreateSubchart` rule

The attribute mapping code is the same as its ACM2FAM counterpart, associating instance references of the parent binary LSC to the child binary subchart and copying activation mode and quantification values from multiSubchart to binarySubchart.

The final representative rule set is the blocks that handle `out` and `in` message events to and from federates and the federation. `HandleOut` block and `FederationOutFederate` rule are shown in Figure 5.33. In the figure it is seen that for every `out` message event from the federation reference that sends an HLA method to a federate in the multiLSC, a similar `out` message event and HLA method are created from the corresponding federation reference to the corresponding federate in the binaryLSC. The other message event handling rules are defined similarly. A newly introduced user code library method is `DeepCopyMgaObject()` that traverses the Mga object provided as the second parameter and creates a copy of its structure in the stub object given as the first parameter. This is an alternative to the `CreateInstance()` UDM API method that is extensively used in the ACM2FAM transformation with a difference in usage context and parameter preparation.

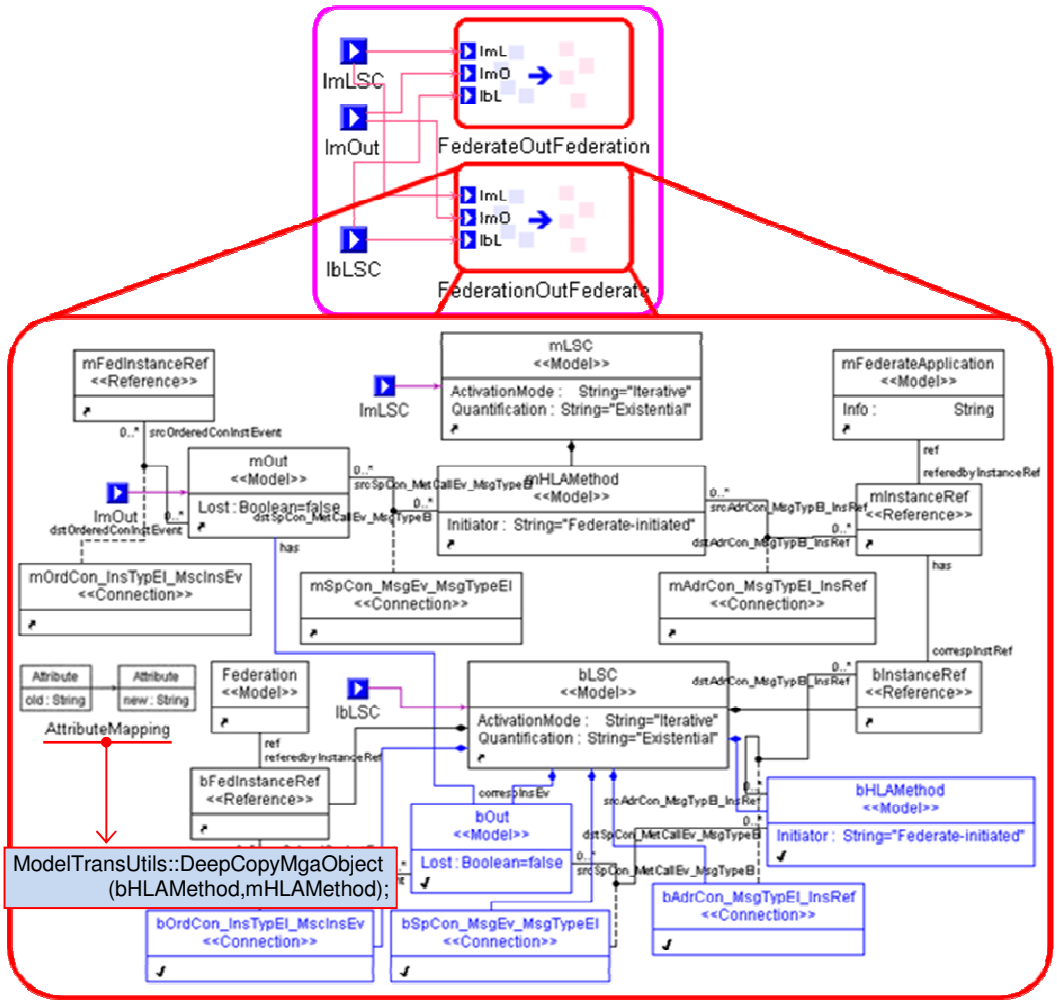


Figure 5.33 `HandleOut` block and `FederationOutFederate` rule

5.5 FAM-to-Simulation Code Generation and Execution

Referring back to Figure 1.1, the content presented up to this point constitutes the first phase of the overall transformation process, where ACM-to-FAM transformation is explained in detail. In the second phase, the produced FAM is fed to the code generator to produce federate source codes, federation source code and useful artifacts such as FOM Document Data (FDD). The details of the code generation are presented in [61].

Aspect Oriented Programming (AOP) [62] paradigm is adopted in generating HLA-based distributed simulation code. The AOP approach provides the separation of cross-cutting concerns. In our case, this allows us to generate code so as to exercise LSCs in a computation-free manner. Then application-specific computational (and other non-communication) aspect advices are hand woven onto the generated base code. On the other hand, HLA-specific portions of the code are automatically woven into the base code generated from the LSC.

The LSC instance is the focal element in code generation. The federation and the federates are all specialized from the LSC instance element. All LSC instances are generated in separate class files and they are declared and used in the diagram code generated from the LSC diagram. A snapshot of the generated source code folders with the source files per binary-instance LSC is presented in Figure 5.34.

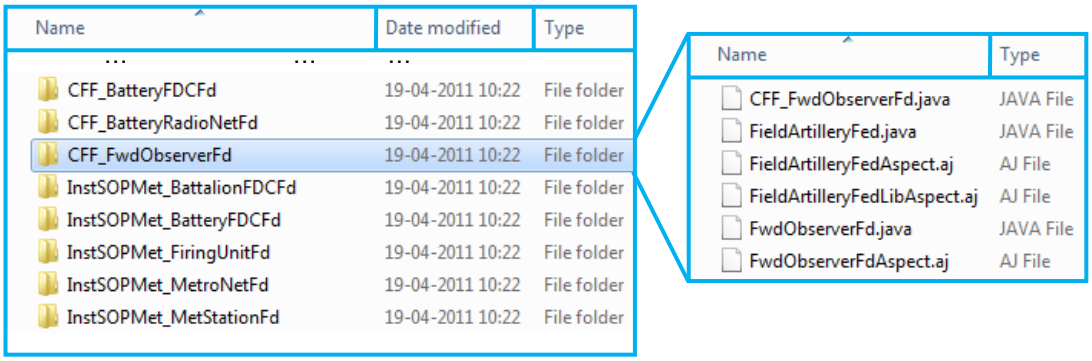


Figure 5.34 Sample generated source code folders and files view

Figure 5.35 depicts the static structure of the generated federate application. Each instance runs in its own thread. For every LSC message out-event, an RTI ambassador method call is made, and for every LSC message input event, a federate ambassador method callback is generated. The LSC instance aspect code intercepts the RTI ambassador method calls. It executes developer written computation code (e.g., modifying method arguments and value of arguments) and then redirects the call to the RTI with the computation code in effect. On the RTI side, in addition to LSC, an aspect code (RTI Instance Aspect) is generated for every federation execution. This aspect code catches the RTI callback methods and forwards them to the LSC Instance (federate application) code. Then in the LSC instance aspect code, the result of a callback (with all arguments) is made available to the developer.

The code generator creates an Eclipse project and stores the generated Java and AspectJ codes in the project root folder. We use an AOP-enabled Eclipse installation to weave the aspects and run the simulation code. (Eclipse gains AOP capability by installing AspectJ Development Tools software on it. AspectJ [63] is an aspect-oriented extension for the Java programming language.) An Eclipse screenshot of the generated code from the FAM model of an *AdjFFE* mission scenario [11] of the field artillery domain is displayed in Figure 5.36. The details of code generation for the AdjFFE case study are presented in [84].

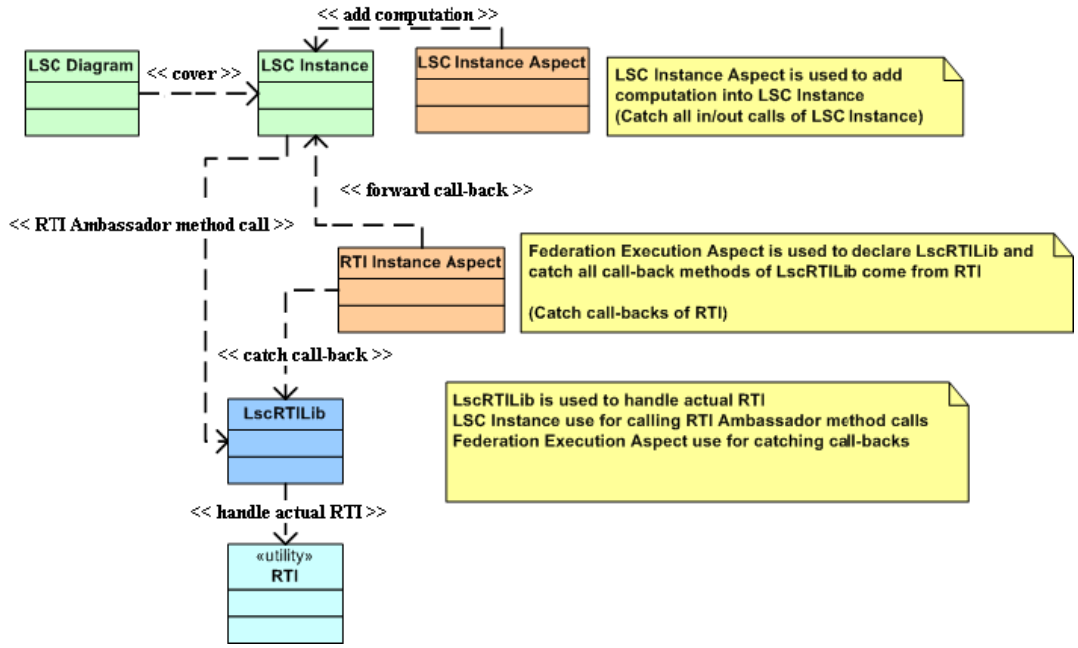


Figure 5.35 Static structure of a generated federate application [61]

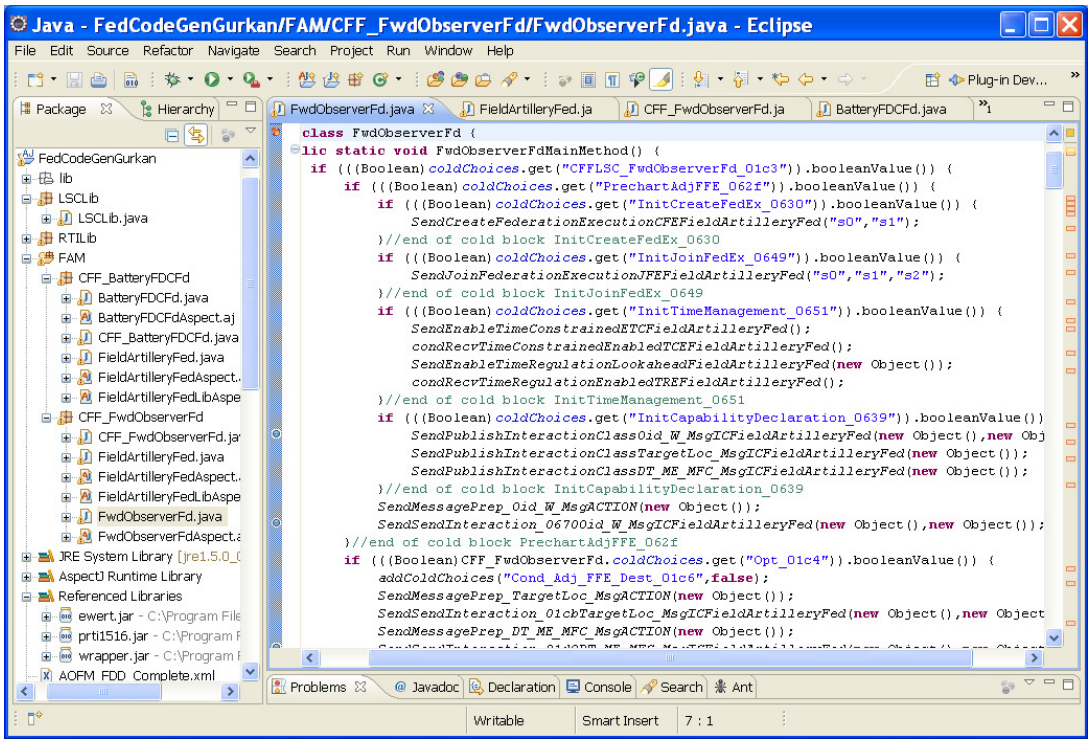


Figure 5.36 A screenshot of the generated *AdjFFE* mission code in Eclipse

After the aspect codes are written and the source is compiled the simulation is run for execution. Currently the code generator supports the RTI implementation developed by Pitch Technologies (certified for IEEE-1516), named pRTI. A pRTI screenshot of an *AdjFFE* simulation execution is shown in Figure 5.37.

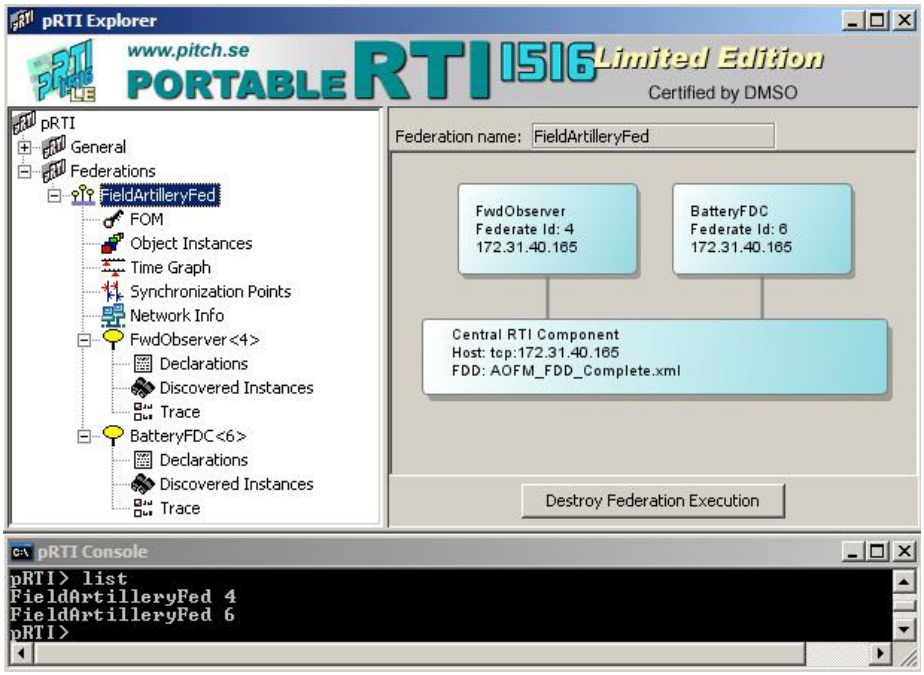


Figure 5.37 A screenshot of an *AdjFFE* simulation execution in pRTI

5.6 Analysis of the Transformations

Wijngaarden and Visser [64] identify three fundamental aspects of transformation mechanics, namely, scope, staging and direction. Although these aspects are intended for evaluating transformation approaches in a broader sense, it can still be referred in assessing this specific transformation work. Besides these, modularity is recognized as a key aspect to achieve reusable and adaptable transformation definitions [65]. Internal transformation composition [66] is an issue which is closely related to modularity. Composition of transformation definitions requires a proper modular construct, providing a composition operator, such that separate transformation definitions can be created as composable units. In this section we analyze our two phased transformation work from the points of view of modularity, internal transformation composition, staging, scope and direction.

5.6.1 Modularity Analysis

Modularity is a key factor in developing reusable and maintainable transformation definitions. Through a modular construct, decomposition and composition are possible. Transformation reusability is facilitated if a transformation unit has a specification, so that the developer only needs to know what is transformed into what, but not how the transformation is done. ACM2FAM transformations clearly comply with this principle because Section 5.2.1 explains which parts of the ACM data model are transformed into which parts of the FAM data model, and Section 5.3.3 does the same for the behavioral models. These statements are made prior to how the transformations are actually explained in detail. Figure 5.1 provides an overview of the modular breakdown of the ACM2FAM transformation. Following this breakdown, the whole transformation is defined as a set of hierarchical transformation blocks, down to individual transformation rule level. Moreover, the use of expression references for recurring transformations provide us transformation rule and block reuse.

5.6.2 Internal Transformation Composition Analysis

The behavior representation formalism LSC/MSC provides a comprehensive instance decomposition construct [15]. The inner structure and behavior of an instance kind is defined through an MSC document with the same name as the instance kind. To indicate how the behaviors described at different levels of abstraction are related, the behavior of an instance inside an MSC diagram can be specified to be refined in an MSC of the MSC document defining the instance being decomposed.

In this thesis, `BatteryFDC` is a decomposed instance that is further refined into another MSCDocument of its own, which models the internal organization and workflow of the fire direction center of a battery. Work is under way to complete the `BatteryFDC` transformation. From a vertical perspective, we perform ACM2FAM and BatteryFDC2FAM transformations in composition. From a lateral perspective, the two phased end to end transformation presented in this paper is logically a composition of ACM2FAM and FAM2Code in sequence. Moreover, at a finer level, data and behavior model transformations within ACM2FAM are also performed as a composition of the two. The horizontal and vertical internal transformation compositions performed in this work are summarized in Figure 5.38. As a final remark, GReAT does not provide a composition operator in the sense of [66]. The closest construct can be the expression reference [15] that is used to invoke another transformation rule or block, but this not the same as invoking a separate, independent transformation.

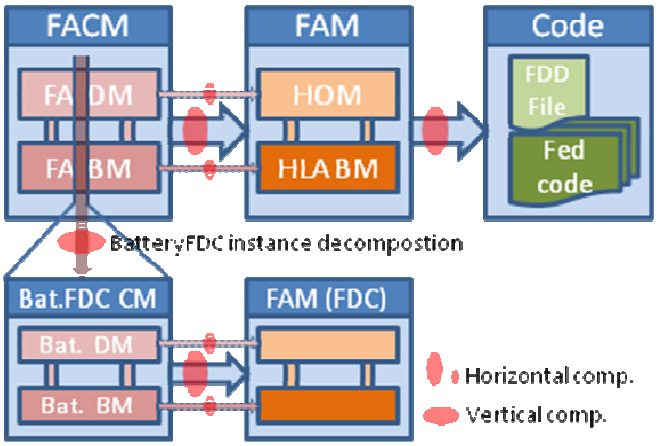


Figure 5.38 Horizontal and vertical internal transformation compositions

5.6.3 Staging Analysis

Staging is the ability of a transformation tool to split a transformation definition into several stages. Usually, model transformation languages are single-stage tools, that is, the transformation execution consists of applying the transformation rules as a whole. In contrast, a multi-stage generate approach allows a transformation definition to be split into several independent stages, each one generating a part of the target model, and then one or more final merging stages connect the results of the previous stages.

This aspect is inherent in our two-phased transformation mechanism, as seen in Figure 1.1. The first phase transforms an ACM into FAM and the second phase transforms the generated FAM to simulation code. Note that in this thesis, the stages are literally loose in that, the first phase is a GReAT transformation, while the second is actually a model interpretation over the FAM model. The only constraint is that ACM2FAM transformation must precede FAM2Code and that the output of the former is the input of the latter. It is necessary to note that GReAT itself is a single-stage tool, but our end-to-end mechanism is a staged approach. Finally, we do not have the so called final merging stage.

5.6.4 Scope Analysis

Scope is the area of a model (either source or target) covered by a single transformation step, where a transformation step is usually a single rule application. The pivot of a transformation step is defined as the main source element from which a rule resolves. Four main types of transformation steps can be identified between a piece of source model and a piece of target model, namely local (source) to local (target scope), local to global, global to local and global to global transformations.

In a local to local transformation step, a source element can be directly translated to a target element. All the information needed to create the target element is accessible (i.e. can be easily reached) from the source element. Many of the transformation rules in this work are of local to local type. Especially, most of the LSC transformation rules are local to local, except for the ones associated with the payloads of the communicated messages. This is not surprising since the LSC metamodel is used for behavior modeling of both domains and we would generally expect behavior preservation across domains. After all, a simulation model is a representation of the real world in a different formalism.

In a local to global transformation step, a source element is transformed into several target elements. Usually, one of these target elements is part of the piece of target model being generated by the rule, while the other target elements need to be allocated in a different part of the target model. These target elements are referred to as non-local results. A considerable amount of rules in this work are of this type. This is mostly due to the fact of transforming a PIM into a PSM which require the introduction of model elements regarding the platform and other target domain specific aspects. Examples to this category on data model transformation side include rules that initialize HLA data types, FOM and its sub-container elements, rules that create and populate interaction and object classes and rules that create the federation structure elements. On the behavioral model transformation side are the rules that create federation initialization and tear-down, and especially, the orderable event transformation rules for durable and non-durable messages, which are the most complex and platform specific content adding rules, are typical examples.

In a global to local transformation step, additional information is needed to create a target element from a source element. This additional information is not easily accessible from the source element being transformed (i.e. the pivot), but a complex query is needed. This kind of situations stemmed frequently in this work. GReAT has two handy mechanisms in easing this burden, called “global container” and “cross-links”. Global container contains elements that have global scope; that is, they are accessible throughout the whole transformation, and it is not necessary to pass them along in the context. Cross-links establish cross model associations between the source and target metamodels (see Section 5.1 for both constructs). Many rules spread throughout the data and behavior transformations employ cross-links or global containers, and thus are examples of global to local transformation step.

Global to global transformation is a combination of the previous two situations. They usually involve complicated rules. We have tried to avoid such cases as much as possible since pattern matching is an expensive operation. Our approach was to divide the

transformation into a set of serial and/or parallel smaller rules, each exhibiting a local to local or local to global pattern. In spite of our efforts, the orderable event transformation rules for durable and non-durable messages are still quite complex and are global to global type of transformations.

An interesting aspect of ACM2FAM transformations related with the scope topic is the employment of the user code library. This library that we've written in C++ is invoked from within various transformation rules to complement the graph based transformations with a high level programming language support. It utilizes the flexible and powerful UDM API [34] to manipulate the GReAT transformation model to completely handle or partially assist a number of the transformation rules. This resulted in a reduction in the number and complexity of the transformation rules (i.e., simplify global-to-global rules into local-to-global, or global-to-local and the like).

5.6.5 Direction Analysis

Finally, direction refers to whether a transformation is controlled by the structure of the source model (source-driven) or by the structure of the target model (target-driven). Since this is a characteristic of the underlying transformation language, there is not much to discuss on this topic here. It should be enough to state that GReAT is a source-driven transformation language and thus, so is the ACM2FAM transformation.

5.7 Related Work on Model Transformations

Although there is a wide range of works in the literature that are focused on behavioral or data model transformations, to our knowledge, the transformation of a fully-fledged conceptual model to an executable model has not been reported before. We treat both data and behavior on equal grounds in our transformation perspective and put forth a two-phased transformation framework for PIM-to-PSM and PSM-to-executable code transformations. The section starts with transformations targeting simulation models, which are the ones that adopt similar approaches to this work, and continues, in decreasing relevance, with others.

5.7.1 Transformations Targeting Simulation

The past few years have seen the publication of ontologies for a large number of domains. The modeling and simulation community is beginning to see potential for using these ontologies in the modeling process. There is a group researchers who tend to formulate CMs as ontologies and then follow a transformation path towards (executable) simulation models. Silver et al. [107] suggests a tool called *Ontology Driven Simulation* (*ODS*) that establishes relationships between domain ontologies and a modeling ontology

and then uses the relationships to instantiate a simulation model as ontology instances. Then, translating these instances into XML based markup languages and then into executable models for various software packages are also possible. As a case study, they map Problem-oriented Medical Records Ontology (PMRO) [108], which represents a hospital emergency department, to the Discrete Event Modeling Ontology (DeMO) [101], which describes a discrete event simulation from state, event, activity, and process oriented world views. The DeMO instances were then translated into to Extensible Process Interaction Markup Language (XPIML) [109], which then could be translated by the tool into executable models for either JSIM or ARENA tools. This work is principally similar to ours in that it employs a multi-phased transformation technique from a domain model down to code. Yet, the source domain ontology to DeMO and DeMO to XPIML transformations are not done automatically, but manually with the help of GUI supported mapping tools.

Another ontology-based work is presented by Durak et al. [110], where they semi-automatically transform the Trajectory Simulation ONTology (TSONT) [111] models to two different target models. Two different tools are used for two different programming paradigms. For object oriented programming paradigm, the OWL2UML tool transforms OWL ontologies to UML class diagrams with user guidance [112]. Then, Platform Independent Framework Architecture or trajectory simulation reuse infrastructure is constructed by means of user guided transformations. For function oriented programming paradigm, TSONT2SIM tool [110] generates MATLAB Simulink block definitions by transforming the trajectory simulation function definitions captured in TSONT. This double targeted transformation work requires man in the loop in its processes and currently does not produce executable code.

Küçükyavuz et al. [93] propose a method for transforming KAMA [52] mission space conceptual models into simulation space BOMs [98]. They have established mappings from KAMA elements and attributes to BOM elements and attributes in a tabular format. They have demonstrated the applicability of their approach on a radar warning receiver mission space model. They have discovered that there were some fields in KAMA that had no correspondence in BOM and vice versa and argue that this should be anticipated since KAMA and BOM have different concerns and abstraction levels. Unlike ours, this work presents sketchy mappings without any transformation automation and code generation. In our work, we also have partial correspondence between ACM and FAM, but we obtain a complete transformation from ACM to FAM thanks to the information embedded inside the transformation rules.

Etienne et al. [90] report on French military's work to improve interoperability between their simulations. They have investigated the feasibility of two aspects of the MDE approach for their needs: high quality code design through a prototype of Domain Specific Language (DSL), and model transformation, through an HLA code generator, associated to the former DSL. They demonstrated a case study based on a simple tank platoon mission. An interesting aspect of their work is that they use the same modelling and model transformation tools used in this thesis, namely, GME and GReAT. Consequently, they concluded that the MDE appears to be powerful in easing dialog between officers and engineers, and enabling short development or modification cycles through drag/drop model reuse and automated HMI generation, for instance.

The CAPSULE study [91] is another French military contracted work that aims to apply the MDA approach to the M&S domain to investigate the degree of portability, interoperability and reuse MDA can offer for their simulations. The study was conducted in three stages, where, in the first stage, a suitable state of the art model transformation tool is selected (which turned out to be MIA-Transformer). In the second stage, the feasibility of applying such an approach on three existing simulation "frameworks" (HLA, Escadre, Ligase) by using a technique they called "MOF transformation" is investigated, and in the final stage, the design and development of a demonstrator that addresses HLA and Ligase target simulation platforms is done. During the study, a meta-PIM and three meta-PSMs for the three target simulation platforms were created, all being in XMI format. The generated PSMs were opened in Rational Rose, and C++ skeleton code was generated automatically. Note that the reported transformations covered only data models; behaviour was not included.

Experiences of Raytheon Missile Systems on MDE are highlighted and summarized in [92]. For the last several years, Raytheon has been employing auto-code processes and tools to facilitate rapid deployment of models and algorithms into Integrated Flight Simulations (IFS). The paper demonstrates the concrete benefits of employing MDE approach at Raytheon through several benchmark charts and tables. It concludes that the MDE processes significantly reduces overall cost and readily allows fidelity enhancements, yielding better system performance assessment and characterization. They utilize MDE approach only for direct code generation from relatively small models; there is no usage of any intermediary PIM to PSM transformations.

The purpose of PEO Soldier Simulation Road Map study [88] is to continue to build a capability for Program Executive Office (PEO) Soldier to assess the platoon level effectiveness of different soldier equipment architectures using distributed simulation. The

capability is being built by means of Army's Modeling Architecture for Technology, Research, and Experimentation (MATREX), which is an implementation of a unified Army federation to support distributed engineering-level analysis within a greater force-on-force environment. At its core, MATREX provides an RTI, a FOM, and a middleware independent capability that allows simulation developers to move with agility from different implementations of HLA or Test and Training Enabling Architecture (TENA). These capabilities are enabled by a set of components and tools. Key components include battle command management services which implement federation services for communications, situation awareness, and command and control. The Protocore tool is a simulation architecture development environment that allows federation developers to design a FOM and automatically generate source code for participating simulations that interact with that FOM in a middleware independent fashion. This capability is based on a transformation from a PIM specification, the FOM to a PSM specification, such as HLA 1.3. In this sense, MATREX is a realization of MDA in support of federated simulation. Within the scope of the study, the PIM for a PEO Soldier scenario is demonstrated to be transformed into its corresponding PSM. They use UML sequence diagrams to describe their selected scenarios, whereas we use LSCs for the complete behavioral specification of the missions of interest.

Ambrogio et al. [113] introduces a model-driven approach that allows automating most of the activities that are traditionally carried out manually to implement a DEVS-based simulation from a high-level model of the system under simulation. The paper illustrates the set of UML profiles and model transformations that endow simulation developers with an automated approach that produces a significant portion of the final simulation code. As a case study, the production of a DEVS/SOA simulation for a basic queuing system is presented. The model-to-model transformations specified in ATL [7] are executed by use of the ATL engine provided by the Eclipse tool, while model-to-text transformations specified in Xpand are executed by use of the openArchitectureware tool [114]. Specific and mostly automated processes have been introduced to yield not only the code but also the configuration data for the DEVS/SOA platform, so as to produce a DEVS/SOA simulation ready to be executed. This work is a two-phased model transformation effort similar to ours, however, in its present form, the approach only produces the core skeleton of Java classes that implement DEVS models. Work is in progress to deal with the inclusion of UML-based abstract models that specify the simulation logic (i.e., behavior) as well. We incorporate the computational part of the simulation logic in the form of advises to be woven into the generated AspectJ code.

5.7.2 Automata, State Chart, State Diagram Transformations

Szemethy [67] introduces a tool that performs transformations from high-level domain-specific models of the time-triggered language, Giotto, which is used to describe embedded systems, into analysis models represented in timed automata. It uses the same modeling and transformation tools employed in this work, namely GME and GReAT. It is an early case study demonstrating PIM to PSM transformations of the kind shown in this thesis. According to the time-triggered paradigm, all activities of the system must be strictly periodic, with possibly different frequencies in different modes of operation. The activities in our domains, on the other hand, are majorly event driven. Time triggering exists in only a few specific places to initiate various (sub) scenarios in a field artillery mission. Our work has a wider and more diverse scope in terms of the source and target models. The number and complexity of our transformation rules also outrange theirs. Since the concepts and functionality of their source and target domains are closely related, the mapping of Giotto entities to timed automata entities is simple and straightforward. The only sophistication in the course of their transformation is the generation of timed automata instruction sequences from Giotto timing constraints.

There is a body of work dealing with the translation of sequence diagrams to state charts, see for example, [68][69][70]. The approaches in these examples differ from traditional graph transformation approaches, where the transformations are specified over the abstract syntax. Gronmo and Pedersen [68] base their transformation on the concrete syntax of both domains. Ziadi et al. [69] and Sun [70] define their transformations by pseudo-code operating on algebraic definitions. Our transformation is defined over the abstract syntax of the source and target domains and hence any input model and its produced output model are guaranteed to be correct by construction. Owing to the simplicity and small size of the sequence diagrams and state charts, the set of transformation rules in these works are relatively smaller. In our opinion this characteristic facilitates defining the transformations over concrete syntax. The field artillery data model, HLA OMT and LSC domains, on the other hand, are much bigger and complex. In order to cope with this complexity, our transformation additionally incorporates a fast user code library that leverages the execution performance.

Van Amstel et al. [71] have developed a transformation from Algebra of Communicating Processes (ACP) into UML state machines. Using the Rhapsody tool they generate code to execute the produced UML state machine and the action dispatcher. By this way, the execution of an ACP model is simulated. On the other hand, its behavior preservation is limited to execution trace equivalence. In our transformation, both data and

behavior are preserved and this is traceable through the rule definitions. Note that we are performing transformation between two totally unrelated domains and our source model has smaller information content than the target model. Information is not lost, but increased and the transformation rules are where this is done.

5.7.3 LSC to Code Transformations

Code generation from behavioral specifications in LSC is an ongoing challenge for researchers [72]. There is also a body of literature dealing with transforming LSCs to some executable form; in particular, state charts [73][74]. We favor executable code generation directly from LSC as this approach tends to yield more readable code. Harel and Marelly [43] propose a play-in/play-out engine to capture behavioral requirements. The Play-Engine automatically constructs the behavioral model in LSCs, and then provides a simulation of the execution of the LSC diagrams by playing out different scenarios. In contrast, our metamodeling approach, due to its data model integration capability, provides the opportunity to extend or tailor the code generator or interpreter in accordance with the data model.

5.7.4 Schema Transformations

In software engineering, the functional requirements of the system are formally specified in a conceptual schema, or a conceptual (data) model. Conceptual schemas are described in a conceptual modeling tool/language such as GME or UML. Schema translation has been considered an important practical problem in the fields of databases and information systems [75]. The topic has nowadays gained more momentum due to the need for translation between ontology languages and for translation between models in the sense of MDA.

The MDA of OMG specifies three system viewpoints and three corresponding default system models: a CIM, PIM and a PSM. Semantics of Business Vocabulary and Business Rules (SBVR) defines the metamodel for documenting the semantics of business vocabulary, business facts and business rules. Business rules in SBVR are structured by logical semantic formulations, which facilitate their automation in software systems. In fact, SBVR specifies a metamodel to describe CIMs and UML is the standard language proposed by OMG to build PIMs (consequently, the conceptual schemas).

Raventós and Olivé [76] propose an automatic approach to translation between schemas modeled in UML and SBVR vocabularies and rules, and vice versa. The authors have formulated this translation as a particular application of the more generic problem of schema translation. Both the source and target schemas used in the translations are instances of metaschemas which are MOF-compliant [16]. The main contribution of their

approach is the extensive use of object-oriented concepts in the definition of translation mappings, particularly the use of operations (and their refinements) and invariants, both of which are formalized in OCL.

To facilitate the application of their approach, they have developed a transformation tool framework on top of Eclipse tool, that allows designers to model the UML context schema, generate the corresponding SBVR instance and finally obtain a natural language description of the schema (in Structured English) [77]. The UML/OCL-to-SVBR transformation is formalized in the ATLAS Transformation Language and the SVBR-to-text transformation is implemented in MOFScript. A surprising aspect of their study, is that they follow a PIM-to-CIM and CIM-to-structured natural language transformation direction, which is the opposite of most MDE practices.

5.7.5 Web Services Transformations

Heckel and Lohman [78] propose a model-driven approach to the development of reactive information systems, such as dynamic web pages or web services, modeling their typical request-query-update-response pattern by means of graph transformation rules. The transformation is carried out using story diagrams which is a graph transformation language based on UML and Java. With the transformations, source models in UML are transformed into contracts expressed in the Java Modeling Language (JML). Unlike ours, this work is a single step MDD effort that does not further attempt to generate executable code from the produced JML models.

Another work in the Web domain is the UML-based Web Engineering (UWE) approach [79], where rule-based transformations written in ATL are defined for all model-to-model transitions, and model-to-code transformations pertaining to web content, navigation and presentation. First, business process models are transformed to UML activity diagrams by the ATLAS transformation engine. Then a run-time environment built on top of the Spring framework performs direct execution of the generated activity models. In a more recent work [80], a graph transformation approach is taken to refine business-oriented architecture models to service oriented architecture models, focusing on the ability of dynamic reconfiguration typical for Service Oriented Architecture (SOA). The authors have formally defined the refinement relations from the component-based business level architectural style to the SOA style in UML, but the work is still under way to implement the transformations in Graph eXchange Language (GXL), the language supported by both the AGG transformation tool and CheckVML, a model checker for graph transformation systems.

5.7.6 Transformation by Example

A noteworthy approach to model transformation, so called transformation by example, is proposed in [81]. The authors view model transformation essentially as a combinatorial optimization problem where the transformation of a source model is obtained by finding, for each of its constructs, a similar transformation in an example base. Two strategies based on two search-based algorithms, namely particle swarm optimization and simulated annealing, are employed. The approach is illustrated and evaluated on the well-known case of transforming UML class diagrams to relational schemas. This work is unique in not requiring metamodels and transformation definitions for the source and target models.

5.7.7 Miscellaneous UML-based Transformations

Braga [82] proposes an automatic and validated code generation process from Role-Based Access Control (RBAC) policies into aspect code. They have developed a transformation from SecureUML, a RBAC policy specification language, to AAC, a simple abstract aspect-oriented language. Both languages are specified by metamodels defined in UML. The transformation essentially maps each entity and its associated RBAC policy in the source model to an abstract entity class and an aspect in the target model. The abstract class represents an interface that a concrete implementation of the controlled component must implement. The aspect implements the access control constraints that must hold when a component's method is called. As the last step, AspectJ code is generated from the produced AAC model. The transformation is implemented as a Java application on top of an OCL evaluator named ITP/OCL. This work purely takes an RBAC perspective on a generic entity-relationship data model formation. The metamodels and the transformation do not incorporate any sort of behavior representation that would capture the processes or workflows in a domain.

UML2Alloy [83] is a tool which transforms a subset of UML class diagrams and OCL constraints into the Alloy language, so that the generated specifications in Alloy can be automatically analyzed by the Alloy Analyzer, a tool used for identifying design faults in a software specification. This work differs from ours in that it employs transformations for UML model analysis with the motivation to catch design faults at earlier stages of software development lifecycle, whereas we are transform a conceptual domain model down to its executable simulation model. Another major difference is that UML2Alloy uses the SiTra model transformation framework, which is a minimal, Java based library that simply facilitates a style of programming that incorporates the concept of transformation rules. SiTra is a primitive tool compared to GReAT in terms of the offered transformation capabilities.

CHAPTER VI

DISCUSSIONS AND FUTURE RESEARCH DIRECTIONS

This thesis has presented an end to end comprehensive model transformation endeavor from the field artillery conceptual model, ACM, to the HLA federation architecture model, FAM. The resulting FAM is further processed through the code generator to generate executable simulation code. The ACM and FAM both consist of data and behavioral parts and the transformations revolve around transforming the two parts in sequence. In the data model transformation, ACM domain actors are transformed into federates and the communicated message structures are transformed into HLA classes. The behavior model transformation is based on transforming ACM LSCs that represent domain actor communications to FAM LSCs that represent the corresponding HLA federate communications via the federation execution inside the RTI. The extra platform specific content and logic required for FAM is provided through the transformation rules, and the user code library employed by the transformation.

6.1 Discussions on ACM Model and ACM2FAM Transformation

This section briefly discusses on ACM and ACM2FAM transformation. Appendix D provides hints and recommendations derived from our experience in realizing ACM2FAM transformation for future model transformation developers of GReAT.

Within the scope of this thesis, ACM has initially been developed in order to lay the groundwork for transformations. ACM's information content is obtained from the US Army field manuals in the public domain. ACM's data model is based on message formats and JC3IEDM, and its behavior model is based on LSC, whose metamodel was developed in another work together with FAM. The challenges encountered and an evaluation of employing LSC notation in observed fire mission modeling is shared with the community [11]. Another intention for developing ACM is to bring the attention of the CM community to the employment of chart notations in describing military tasks, which has not been done before in the literature.

The transformation from an ACM to FAM provides the ability to exercise the resulting federation architecture. In a fully automated exercise, intra-federation communication will follow the specified patterns; the communicated values, being randomly generated, will not be correct. This can be regarded as a first-cut simulation of the exercise. Taking a step towards complete federate application generation, the developer has to weave the computation logic onto the generated code.

A notable downside of the transformation is its poor performance especially when source models get bigger. This is accountable for every rule execution boiling down to solving the sub-graph isomorphism problem on the input model and the match pattern. This burden is ameliorated by breaking rules into reasonably small chunks and providing as much initial binding on the match pattern as possible. Another facilitator is the employment of a C++ user code library that programmatically aid in transformations. This provides a two-fold gain in that, first, the execution of the code library is faster, and second, it saves from tediously defining many similar transformation rules.

6.2 Discussions on FAMM and the Code Generator

FAMM has been developed in a previous work and tested together with the code generator in various exercices [85][86][87], wherein the FAMs were manually developed in close coordination with the code generator team. Eventually, base codes were successfully generated, aspects were woven and the resulting codes were successfully run on RTI. On the other hand, problems and unforeseen issues emerged when we started testing FAMs that were automatically produced as results of ACM2FAM transfomations. Even before being able to test code generation for auto-produced FAMs, the FAMM itself needed various modifications in order for it to be used as a target (meta)model in GReAT transformations. In short, changes were required on both FAMM and the code generator in order to have the two-phased end-to-end ACM to executable code transformation vision to flourish in practice. This section informally asseses FAMM and the code generator for their usability in graph-based model transformations based on our experience, and categorically summarizes the required changes. The summary of changes done in FAMM and the code generator are provided in Appendix C. All of the new FAMM versions along with a change log per version as well as the modified code generator source are available through the thesis distribution CD. The source code also reports the changes done in comments.

6.2.1 Discussion and Assessment of FAMM

FAMM has been developed in GME as a metamodel for building HLA-based distributed simulation models [85]. Its development was closely coordinated with the development of

FAM to Java/AspectJ code generator [86]. Although its prospective role as the target model for transformations from ACM was also taken into account in its design, it was never tried in GReAT tool until it was fully completed. We started encountering various issues from the moment we have imported FAMM into GReAT for use in ACM2FAM transformation definition. The paragraphs below briefly explain the adaptation process of FAMM to make it compatible with GReAT transformations. Note that issues mentioned in this sections are specific to FAMM's usage in GReAT. Otherwise, it is flawless as a GME domain model for HLA.

Before starting with the issues, it is worthwhile to recall that GReAT transformation models (of `UMLModelTransformer` paradigm) are first processed by the *`GReAT Master Interpreter`* to generate C++ code of the metamodels and transformation definition and then the *`Graph Rewrite Engine`* executes this code to actually perform transformations.

All of the `connection` elements that are used in associating more than one pair of modeling elements had to be avoided. (Because if a `connection` element is used more than one time as source-to-connector or connector-to-destination, then the code generator generates duplicate method definitions for those `connection` parts, which result in compile errors.) This is achieved by building a `connection` hierarchy so that ambiguities in code generation are eliminated.

All of the modeling element, role or attribute names that are at the same time C++ reserved words (such as `if`, `else`, `for`, `string`, etc) had to be renamed for obvious reasons. Not as obvious as these were, the GReAT interpreter generating utility methods (such as "`Create`") which had the same name with some other FAM elements. Such name clashes, which could only be detected by trial and error, also had to be resolved.

Some FAM elements in different paradigm sheets were named the same. This does not cause any problem as far as GME modeling is concerned; however, the GReAT interpreter produces duplicate class names for those elements, which result in syntax errors at compile time. Thus, name uniqueness had to be enforced throughout the entire FAM.

This last action is not taken due to an obligation, but just for convention. A reference that points to all of a super class's child classes is made to refer to the super class only, in order to reduce redundancy.

6.2.2 Discussion and Assessment of the Code Generator

Before starting the discussion on code generator, it is worthwhile to mention about a post-processing work that has to be done on a produced FAM, in order to comply with the requirements of the code generator on LSC structure. The code generator expects an LSC to only cover a single federate and its communication with the federation. The developers of

the code generator had the motivation that such a local federate view would better facilitate code generation. Consequently, the Multi2BinaryLSC transformer was developed in order to refine FAMs having multi instance LSCs into FAMs having binary instance LSCs, as explained in Section 5.4.

After FAMM was sanitized of the aforementioned issues that prevented it from being used in ACM2FAM transformations, we could start generating FAMs that correspond to various AdjFFE ACM mission models. Then these FAMs were opened in GME and the Code Generator (CodeGen) plug-in was run on them to generate simulation base codes. However, things did not go as expected again and run-time errors were thrown. The problems generally had to do with the imperfection of the CodeGen, because our AdjFFE FAMs were automatically generated and they were correct-by construction due to their compliance to FAMM. Note that the fixes done on the original FAMM introduced nothing that would have negatively affected the CodeGen's execution. The reason for these issues, we think, is that the development of CodeGen was majorly steered by the samples that were manually created during FAM testing. The scope and representative power of those samples were not as far-reaching as AdjFFE FAMs. As a result, some of the permissible FAM structure combinations were simply missed by the CodeGen. These patterns were revealed during modeling with an ACM perspective and mindset. In addition to these, the CodeGen simply had some syntactic and semantic flaws in its code generation logic and shortcomings in FAM coverage that we have discovered during our exercises. The rest of the section summarizes the issues fixed in the CodeGen.

Since CodeGen traverses a given FAM to generate code, most of the modifications that we had to make in FAM had corresponding change requirements in CodeGen. These include; writing getter methods for the newly introduced connection types in FAMM, and calling them in appropriate places, reflecting any FAMM modeling element name change (either due to reserved name clash, duplicate name definition, or convention) in the code. In the original CodeGen, only the "DefiningPart" of an MSCDocument was processed, but the similar "UtilityPart" was commented out. We have opened up the comment since AdjFFE FAMs contained both defining and utility parts.

Other more serious issues include; all `OrderedConnections` have `priority` attributes as dictated by the MSC metamodel, which indicate relative execution order among the events. CodeGen processes the `OrderedConnections` inside a list data structure that is indexed by the connections' priorities. CodeGen overlookingly assumes that the list index starts with one and sequentially increases in ones. This assumption might be valid for a manually constructed FAM, but AdjFFE FAMs are auto generated from their

corresponding ACMs. Even if we try to arrange the list index as required, we end up in non-sequentially ordered priorities in the produced FAMs, because the number of `OrderedConncetions` at least doubles due to extra HLA-RTI communications. We have devised more sophisticated data structures and algorithms to correctly work with a sparsely filled `OrderedConnection` list. If the code was left intact, then either no or semantically incorrect and missing code was generated.

The *AStyle* plug-in, which is a source code indenter, formatter, and beautifier for Java did not work for some reason and caused run-time error. We had to abandon its usage.

Occasionally, null value checks for variables were not written, causing null pointer exceptions at runtime. Such issues are the results of making assumptions on the input FAMs and quick-and-dirty coding practice. We have corrected almost all of these cases.

The generated code for inline operand "`Opt`" contained a syntax error. `Opt` was most probably never tested by the CodeGen team before. We have correctly added the "`.`" field accessor before the "`coldChoices`" field of the active LSC.

Since CodeGen was designed as a GME plug-in, there was no "main" Java method to launch it standalone in Eclipse environment. We have defined a main method inside the `LSCCodeGen` class that invoked the main interpreter method with the relevant parameters.

6.3 A Comparison to MDA

This section explores the questions of where the artifacts used in the overall MDE activities of this thesis lie with respect to concepts and standards advocated by MDA and how our models align with MDA's triple modeling viewpoints.

6.3.1 Our Artifacts Associated with MDA Standards

Object Management Group (OMG) introduces a four-layer metamodel hierarchy for defining modeling, metamodeling, and meta-metamodeling languages and activities in [17]. Table 4.2 relates the different levels of models used in this thesis to OMG's modeling hierarchy. Besides that, Figure 1.2 shows the abstraction levels of these domain and transformation models with respect to OMG's hierarchy. Taking this one step further, Figure 6.1 associates the concepts and standards that OMG has put into its MDA vision with the MDE artifacts employed in this thesis. According to the figure, metaGME, the meta-metamodel of GME, is functionally equivalent to MOF of OMG at M3 level. ACMM and FAMM, the metamodels of the source and target domains of this MDE work, are functionally equivalent to UML of OMG at M2 level. Finally, UMT, the metamodel of the FACM2FAM transformation presented in this thesis, is functionally equivalent to QVT of OMG at M2 level.

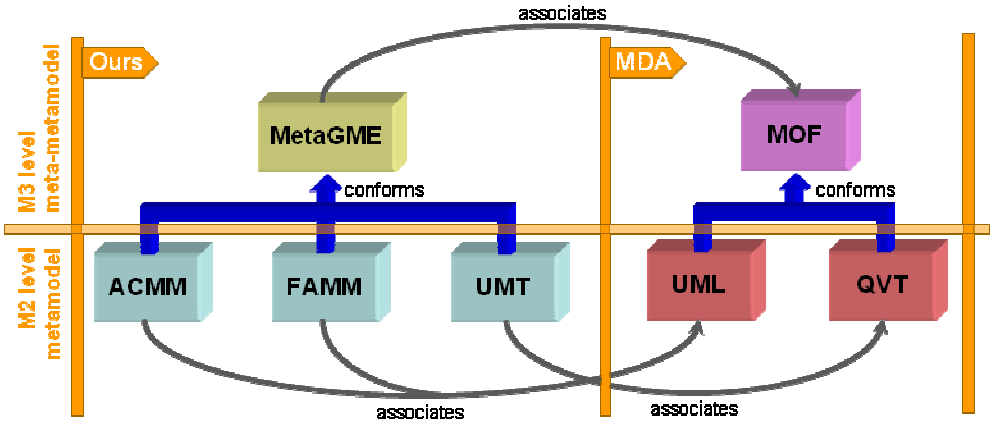


Figure 6.1 Associating our metamodeling artifacts to OMG standards

6.3.2 Our Models from MDA's Modeling Viewpoints

MDA establishes three different modeling viewpoints [21], called CIM, PIM and PSM. The highest level of abstraction is the Computation Independent Model (CIM). This is a conceptual model that identifies the concepts and processes important on the business level. This is easily mappable to the missions and means identified on the operational level. The main artifacts are use cases. The Platform Independent Model (PIM) capture concepts and processes in software engineering artifacts of class and object hierarchies, activities, sequences, and other means showing the roles of each component. PIMs are very close to conceptual models that already use vignette and scenario elements motivating the various possible actions and their sequencing. If this conceptual model is mapped to a concrete platform, e.g. the middleware to be used, the result is a Platform Specific Model (PSM). In the optimal case, the PSM can be used to produce code, as all information needed is available.

Considering the above definitions, ACMM can best be classified as a PIM since it captures the field artillery observed fire domain entities and missions in a UML-based notation, yet avoiding any simulation-specific details. The classification of field artillery messages in ACMM as durable or non-durable is not evident from the authoritative publications we have consulted. Yet, this distinction is free from any simulation notion and can even be regarded as a good modeling practice that facilitates building families of message structures based on usage characteristics. Eventually, this distinction promotes concise ACM2FAM transformations.

FAMM, being the HLA domain model, incorporates all the necessary details to represent any HLA-based distributed simulation. Similar to ACMM, it uses a UML-based

notation, hence built out of software engineering artifacts. Different than ACMM, it defines constructs that provide interfaces to the underlying implementation platform, which is HLA/ RTI. It is relatively straightforward to generate executable simulation code from a FAM than directly from an ACM.

Having identified ACMM and FAMM as the PIM and the PSM, there is nothing much left to identify as a CIM in our work. The narrative model of the field artillery observed fire domain provided in Section 2.5 possesses CIM characteristics. It explains the concepts, processes and missions of FA concisely, at a natural language level. It incorporates use case information, but not in a formal way. It can serve as a (part of) user requirements specification of the system to be developed, built as a result of the analysis of the authoritative references about the FA domain. Consequently, the FA narrative model seems comparable to a CIM.

6.4 Towards a Domain-Independent CM Transformer for HLA

The experience gained in this thesis has shown that domain to domain transformations are doable to the degree of success in mapping the source domain's actors and communicated data structures to the target domain, and in mapping each set of appropriate behavior elements of the source domain to the behavior elements in the target domain. The design of these groupings and mappings is the most challenging task of the transformations. The behavioral model transformation of ACM2FAM essentially being an LSC-to-LSC transformation brings a degree of ease to the process and opens possibility for automation.

6.4.1 The Transformation Definition Experience

This section presents a summary of the applied model transformation definition process in the course of this thesis, which has evolved based on our experience. It is intended as a useful reference for researchers studying graph-based model transformations (in GReAT).

Admittedly, defining transformations for ACM2FAM in GReAT has been a manual and cumbersome undertaking. Working on two large source and target metamodels such as ACMM and FAMM surely has a major role in that. In this first experience most of the transformation rules have each been defined individually and separately since there were no artifacts at our disposal to reuse or utilize. We could start reusing some of the previous rule patterns in subsequent rules as we progressed through the process. Along with that, GReAT's being a declarative and visual tool facilitated tackling with the burden of this tedious model transformation work.

We have defined a work breakdown of the overall transformation into fundamental modules, such as data model transformation, behavioral model transformation, and a

number of others under these. We first started developing the smaller and lower level modules, which have already been refined into legitimate conception levels. Then we merged the smaller modules into larger ones, and finally connected the behavioral transformation module after data transformation module. Modules were represented as GReAT transformation `blocks` with varying depths, eventually ending up in `rules`. Similar to `function` definitions in programming languages, `rules` and `blocks` have explicit input and output interfaces consisting of `ports`. As the `rules` were defined, they were connected to one another and subsequently `blocks` were connected similarly. This development style of GReAT provided an implicit and convenient means for transformation sequencing. As the development continued, some `rules` and several `blocks` turned out to be reusable with some tweaking. The 21 `references` used in the transformation is an indication of the degree of block/rule reuse.

We have generally adopted a spiral development approach, where `blocks` and `rules`, and even modules on a larger scale, were occasionally refactored after discussions among the research team. Since GReAT transformation rules are defined over source and target metamodel elements, even small changes on these could have significant effects on the transformation definitions. GReAT documentation [6] explains how updates to metamodels are reflected in the transformations. Model migration is based on internal identifiers of the model elements, where an old metamodel element reference in a transformation rule is directed to the new metamodel element that has the same id. Once the migration is done, the old metamodel is manually deleted from the transformation definition. For the unmatched model elements in the transformation definition, the associated transformation rule elements are left unassigned. Our experience has proved that model migration in GReAT is easier said than done.

We have experienced several model updates in the course of this thesis, of which a few caused catastrophic effects, causing the migration process to fail unexpectedly or ending up part of the transformation definition being lost. Apparently some of changes in structure, inheritance or other type of associations of the metamodel were such that the migration engine of GReAT could not cope with them. In such cases, there was no way but to roll-back changes on the metamodel controllably until the migration worked, and then modify the transformation definition so that it would prevent failing when the changes on the metamodel were reapplied. Alternatively, we could directly prune the transformation definition to a safe point that would tolerate model migration. Then, in any case, we had to redefine the pruned parts of the transformation definition in accordance with the new metamodel. This is really a painful process, so we advise transformation writers in GReAT

to invest their time and effort for obtaining well designed and stable metamodels in the beginning before they actually start defining the transformations. Subsequent changes at metamodel level end up in expensive reworks on transformation definitions and unnecessary skidding to bring the transformation back on tracks.

6.4.2 The Highlights of ACM to FAM Transformation

In this section we highlight the corresponding key elements of ACM and FAM matched during the transformation, in an effort to identify the points of abstraction in a CM that facilitates designing a domain-independent CM transformer for HLA.

Model transformations are usually defined from more conceptual (e.g. less platform specific) to more implementation-oriented (e.g. more platform specific) models. This generally implies that many source-to-target model mappings are possible. Besides that, the target model is likely to have extra data elements, such as actors and message structures. Also, it usually has extra behavior patterns, such as system initialization, complementary communications via the extra actors and system shut-down, for which the source model provides no clues. Extra behavior patterns and all sorts of book keeping, which do not have direct correspondence in the source model, contribute to the level of difficulty in defining the transformations.

In an effort to couple the key source and target model elements participating in ACM2FAM transformation in the light of the above points in a nutshell, every field artillery actor is mapped to a federate; every non-durable message is mapped to an interaction class; every durable data element is mapped to an object class; the federation element is brought in as a collection of communicating federates, every actor to actor non-durable message communication is mapped to a federate to federate communication via the federation (executing on the HLA RTI), using a pair of send/receive interaction class messages; every actor to actor instantiation type of durable message communication is mapped to a federate to federate communication via the federation, using three pairs of register/discover object instance, request/provide attribute value update and update/reflect attribute values messages; every actor to actor update type of durable message communication is mapped to a federate to federate communication via the federation, using a pair of update/reflect attribute values messages; every actor to actor delete type of durable message communication is mapped to a federate to federate communication via the federation, using a pair of delete/remove object instance messages; the default HLA types (that serve HLA classes) are brought in; federation initialization is introduced in a preliminary LSC by creating the federation execution, joining the federates to the federation, initializing time management, and declaring capabilities; federation destruction

is brought in to the final LSC by resigning the federates from the federation and destroying the federation execution; finally, the rest of the FACM LSC parts are directly (i.e., one-to-one) mapped to equivalent FAM LSC parts.

6.4.3 Designing the Domain-Independent HLA Transformer

Generalizing over the specific model transformation work presented, an interesting research question would be whether it is possible to develop a domain-independent transformation from any conceptual domain model to the HLA simulation model, FAM. As summarized in Section 6.4.2, the experience of ACM2FAM transformation has been useful to identify the "hot" points of FAM that would play a pivotal role in generalizing the transformation perspective from ACM2FAM to *AnyCM*2FAM. These points would be used in bridging the source model to FAM in defining the transformations. Once these mapping points are bound, we have the incentive that it is potentially viable to carry-out the model transformation as a domain-independent LSC-to-LSC transformation.

A PSM is naturally expected to cover the content conveyed by its corresponding PIM and introduce extra, lower level, platform related information. Returning to our work, FAM enriches the information content with the HLA-based distributed simulation concepts, such as federation, declaration, object, ownership, time and data distribution management and HLA default data types. These extras have their places in both the data and behavioral models and need to be addressed during a Conceptual Model (CM) to FAM transformation. In this thesis, this addressing is directly done (i.e., hard coded) inside the transformation rules, hence preventing the use of the transformation with other source domains.

Therefore, the first step forward in obtaining a domain-independent HLA transformer should be to devise a mechanism which guides the model transformer in matching the relevant elements of the source CM with the aforementioned points in FAM. Also, the user code library needs to be adapted for the parts pertaining to the new source CM. Figure 6.2 shows the architecture of the envisioned Domain-Independent HLA Transformer (DIHT).

DIHT would be a FAM transformation framework that provides a GUI-based front-end adapter to tailor a given CM towards FAM transformation. The adapter's role would be to let the user graphically configure the transformation's source domain dependent content. Tailoring is accomplished by fitting the conceptual model to a so-called abstract *FAM-oriented CM template*, which is partly sketched in Figure 6.3 in metamodel form. This CM template is derived and generalized from the experience of this thesis. Fitting is used in the sense of hooking appropriate user-designated CM elements to the extension points in the template model, using an inheritance mechanism in the sense of object oriented programming. The framework assumes that the CM consists of actors communicating

stateful (i.e., durable) and/or stateless (i.e., instantaneous or volatile) data elements with each other. Also, the CM is supposed to use the same LSC metamodel as FAM's for its behavior representation.

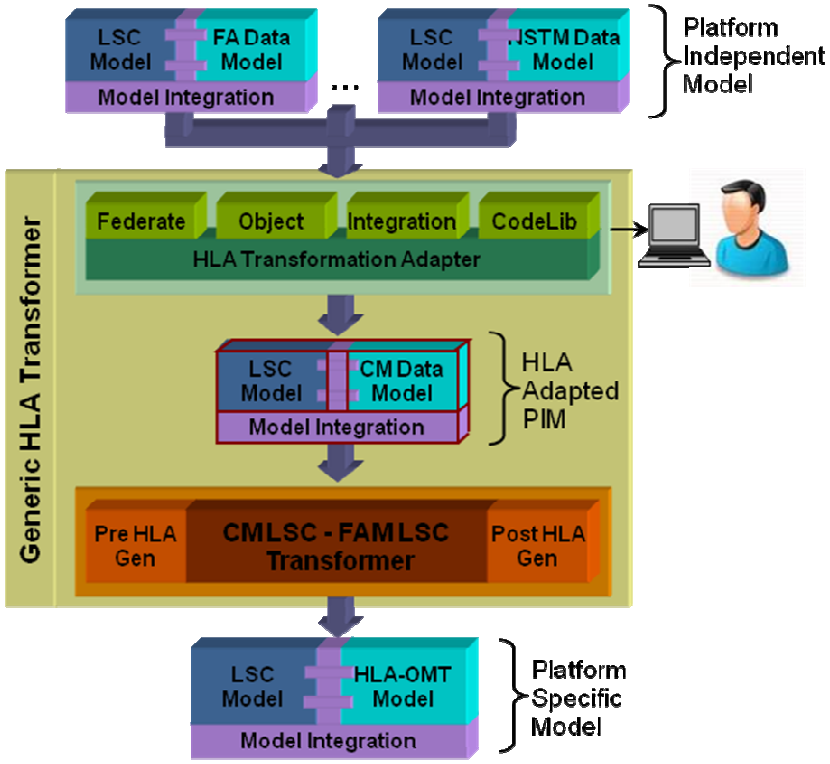


Figure 6.2 The envisioned domain-independent HLA transformer

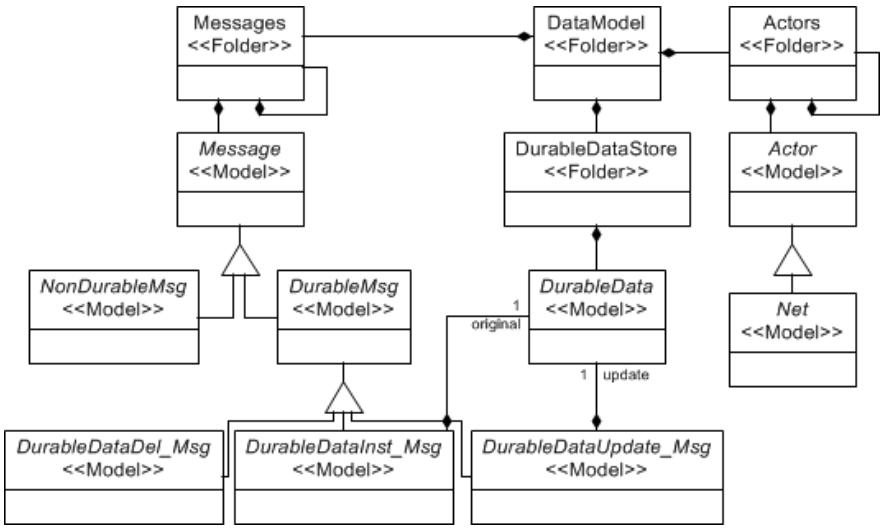


Figure 6.3 The overview of the FAM-oriented CM template

The result of the adaptation process is an intermediary model which is a unification of the template and the given CM. This composite model is then fed to the *CM-to-FAM LSC transformer* to produce the FAM. Specifically, the CM actors and nets are mapped to HLA federates and the CM data elements are transformed into HLA classes by invoking the configured user code library methods. The LSC-to-LSC transformations are carried out using the template model elements, independent of the CM elements in question. This fact can be seen in the LSC transformation rules of Section 5.3.3. The specific CM elements are only accessed inside the code library, which is effectively detached from the transformations. The Pre and Post HLA generation parts of the transformer are independent of the source model and only generate the HLA prerequisities, federation initialization and shut down parts of the FAM mentioned in Section 6.4.2.

With this architecture, obtaining an executable simulation model for another domain would be a matter of developing its data model and integrating the data model with the LSC model to obtain a complete CM of the domain. Then the DIHT would be used to adapt the CM for HLA through the front end tool and then the rest of the transformation would be performed automatically over the LSCs.

6.5 Future Research Directions

This section points to three main future research directions, namely, the development of a domain-independent HLA transformer, investigating the possibility for higher order transformations, and leveraging this work with BOMs that represent intra-federate state and behavior.

6.5.1 Domain-Independent HLA Transformer

This thesis has provided the ground laying work for a future Domain-Independent HLA Transformer (DIHT) that can transform any CM to FAM, provided that the CM is formulated as entities communicating stateful (i.e., durable) and/or stateless (i.e., non-durable) data elements with each other, and is based on the LSC metamodel for behavior representation. The user needs to pre-process the CM by a front end tool to integrate it with the so-called FAM-oriented CM template. Then the resulting intermediate form would automatically be transformed to FAM. In this scheme, the specific CM elements would only be accessed inside the code library, which is effectively detached from the transformation definition (Please refer to Section 6.4 for details).

6.5.2 Possibilities for Higher Order Transformations

A promising future research direction is to investigate the possibility for higher order transformations. The natural starting point is to identify and formulate transformation

patterns that emerge from this work. Then by defining higher-order transformation rules these patterns can be generated. These higher-order rules would potentially be reusable in other CM transformation tasks. The primary motivation for higher order consideration of a transformation component is it having potential for reuse or it being a representative of a large set of rules that are similar in structure, suitable for auto generation once parameterized or externally configured. As a quick start, we would like to point some of the more obvious rules and patterns in ACM2FAM transformation that have the potential for being subjects of higher order transformations.

In data model transformation part, we make use of the default HLA methods defined by the IEEE1516 standard. These methods, which were created as a FAM model in a previous work, are imported as a library into the stub FAM whose remaining parts will be built by the transformation rules. Actually, this library usage is the reason why we start transformations with a stub FAM; otherwise, we could completely create and build the FAM on the fly. Later in transformations, we create deep copies of these methods into FAM LSCs, modify and use them in LSC message transmissions. The same case holds for the default HLA data types: They are also imported as a library into the stub FAM, but for some reason, perhaps unnecessarily, we also manually create the needed default HLA data types in the FederationModel folder of the root folder of the FAM, and refer them from the HLA attributes and parameters then on. We consider that higher order transformation rules can be written to generate all of the default HLA methods and data types, so that the need for the library import mechanism, which might not always be available, can be eliminated.

In behavior transformation part, there are more opportunities for higher order transformations. The promising areas are the transformation rules that construct FAM parts having no direct correspondence with ACM. Principally, almost all of the federation initialization and tearing down rules seems to be suitable for generation via higher order transformations. These include rules dealing with federation creation, joining to federation, initializing time management, capability declaration for federates, deleting object classes, resigning federation, etc.

We would like to draw the reader's attention to a specific case: Currently the knowledge for publish/subscribe declarations of interaction classes are embedded inside transformation rules, requiring two blocks and two rules for each and every interaction class. The situation is even more complicated with object classes. Considering the amount of overhead involved, it can be concluded that handling capability declarations with ordinary transformation rules is definitely infeasible and requires an efficient delegation mechanism. (Currently the declarations for all of the classes are not done due to the burden). Among the

alternatives are, defining higher order transformation rules, employing a user code library to solve the problem through the UDM API and writing a preprocessing model interpreter that runs over the source ACM to extract these relationships, annotate the model(s), perhaps in crosslinks packages, so as to ease the job of subsequent rules that actually handle capability declarations. The overall process seems to be suitable for parameterization and whichever alternative is selected, it can benefit from this fact.

There are some transformation rules and blocks that we have reused throughout the transformations. These can also be considered as candidates for higher order generation. However, we advise a case by case analysis of generic reuse potential for each to decide whether it is actually worth going for higher order transformation. One suitable generic candidate is the "get MSC parent of a LSC" idiom (i.e., GetMSC4LSC block) that we have commonly used. To summarize, an LSC might happen to have more than one ancestor LSC, prechart, subchart, or inline operand (note that each of these "is-a" LSC). Above this ancestor chain comes always a parent MSC. Sometimes it necessitates accessing this parent MSC when only the lowest child LSC is available within the rule context; hence we invoke the GetMSC4LSC block. Such blocks and rules that generically work on the model structure in a context-free manner are good candidates for higher order transformations.

Finally, all of the utility UDM API methods that are invoked inside the user code library and some of the higher-level, user-defined ones can be delegated to higher order transformations. Indeed it would be an interesting exercise to work out these functionalities into higher order transformations. One outstanding example is the `CreateInstance` method that we commonly use to create deep copies of model elements into a given container. Actually, this capability is crucial for our transformations and currently this UDM method call is the only way to achieve it.

6.5.3 Using BOMs for Intra-Federate Modeling

FAM adopts an inter-federate modeling perspective within a federation. The state, behavior or processes inside a federate are not emphasized. This, however, does not necessarily mean that it is all together impossible to model what is inside a federate with FAM. LSC, the behavior representation formalism that FAM uses, provides the "instance decomposition" mechanism just for this purpose. It allows an instance (note that federates are modeled as instances in FAM), to be represented as a standalone MSC document of its own, on a lower scale, thus "decomposed". We have demonstrated the decomposition of `BatteryFDC` in Section 4.4.2. An MSC document is comparable to a federation within the context of this thesis. The analogy to the HLA world is that a federate, depending on its internal organization, can behave like or is a federation on a lower scale; or reversely, a

federation can be wrapped as a federate in another higher scale federation. Having said these, the instance decomposition mechanism cannot provide the means for the co-existence of inter-federate and intra-federate modeling of a federation. This is a crucial deficiency of instance decomposition.

BOM is an open standard that aims to encourage and support reuse, interoperability, composability, and to help enable rapid development of HLA simulations [98][99]. At a higher level, BOMs are reusable packages of information representing independent patterns of simulation interplay and are intended to be used as building blocks in the development and extension of simulations. These components can also be composed in larger models e.g., BOM Assemblies. The *Conceptual Model* part, which is one of the five parts of a BOM, contains information that describes the patterns of interplay of the component. This part includes the types of actions and events that take place in the component, and is described by a pattern description, a state-machine, and a listing of conceptual entities and events, which, when taken together, describe the flow and dependencies of events and their exceptions. This organization of BOM makes it a very convenient formalism to model intra-federate state and behavior, an issue not addressed in FAMM.

We support instance decomposition in this work, and have demonstrated its usage in graphical LSC notation during BatteryFDC modeling. The decomposition of an instance yields another (lower level) MSC document for the decomposed instance besides the main MSC document inside the `BehavioralModels` folder of ACM. Although there are no formal associations established between the main document's and decomposed document's LSCs currently, they can easily be identified by the employed naming convention, as advised by the MSC standard [15]. At the end of an ACM2FAM transformation, all of the corresponding MSC documents are generated on the FAM side. This way we have all the necessary information to create BOMs for the decomposed instances, hence federates. The code generator creates federation code per MSC document in a FAM. Therefore, the information in subordinate MSC documents has to be consolidated into the main document as BOMs. In order to achieve this, the FAMM definition must be enriched with a BOM metamodel in the first place. Then, a model interpreter for FAM can be written to carry the intra-federate knowledge embedded in decomposed MSC documents as BOM components inside the main MSC document. Eventually this will give the opportunity for both inter-federate and intra-federate state and behavior being present in a federation definition. Of course, the code generator has also to be extended in order to process BOMs in a given FAM.

CHAPTER VII

CONCLUSION

This thesis has presented a comprehensive graph-based model transformation work from the field artillery conceptual model (ACM) to HLA federation architecture model (FAM). The work was undertaken to understand the difficulties involved from a mission space model to an executable simulation model adhering to the Model-Driven Engineering (MDE) philosophy. Both ACMs and FAMs are formally defined conforming to their metamodels, ACMM and FAMM, respectively. ACMM has been developed within the scope of this thesis to serve as a realistic source model for the transformations. ACMM is comprised of a behavioral component, based on Live Sequence Charts (LSCs), and a data component based on UML class diagrams. Using ACMM, the *Adjustment Followed by Fire For Effect* (*AdjFFE*) mission, which serves as the source model for the model transformation case study, is constructed.

The ACM2FAM transformation, which is defined over metamodel-level graph patterns, is carried out with the Graph Rewriting and Transformation (GReAT) tool. Data and behavior are preserved while transforming an ACM into its corresponding FAM. In fact the result of the execution of the transformation rules is an increase in the "information content" of the models from source to target. The extra platform specific information required for FAM is provided through the transformation rules, and a user code library. The user code library is written to facilitate the model transformations in terms of improved execution performance and saving from the tedium of graphically defining many uninteresting transformation rules.

Another transformation named Multi2BinaryLSC is also developed, to be applied as a pre-processing step on a produced FAM before feeding it to the code generator. In essence, Multi2BinaryLSC accomplishes transformation from a global view of the federation to the collection of local views of the federates. Multi2BinaryLSC strips down a FAM's LSCs having more one than one federate and the federation into a set of LSCs having only one

federate and the federation. This way code generation is also facilitated in that it can generate code one federate at each run.

A second phase transformation is applied by a code generator to produce executable simulation code in Java/AspectJ from a FAM. Computation logic has to be woven onto the generated (aspect) code in order to provide legitimate values for the data structures at runtime. The resulting code can then be executed on an HLA Run-Time Infrastructure.

The metamodels used for the domain and transformation modeling in this thesis have one to one correspondences with the standards advocated by the Model Driven Architecture (MDA) of OMG.

The model transformer presented in this thesis is analyzed against published model transformation analysis studies in literature.

The experience gained in this thesis is a step forward in designing a domain-independent model transformer for HLA from any conceptual model that is based on LSC for behavioral representation. As a future study, a conceptual model of another domain can be developed in parallel to building the domain-independent HLA model transformer in the light of the recommendations and guidance drawn out of this thesis. Another future research direction is to investigate the utility of higher order transformations; that is, developing higher level, declarative rules to define the recurring patterns of ordinary transformation rules. We have identified with justifications the parts of the transformations amenable for generation through higher order transformations. Finally, another interesting further study would be to enrich this work with BOM formalism so as to incorporate intra-federate modeling capability to complement the existing inter-federate modeling within a federation.

REFERENCES

[1] D.C. Schmidt, Model-driven engineering, IEEE Computer, vol. 39, no. 2, pp. 25–32, 2006.

[2] S. Sendall, W. Kozaczynski, Model transformation: The heart and soul of model-driven software development, IEEE Software, vol. 20, no. 5, pp. 42–45, 2003.

[3] A. Ledeczi, A. Bakay, M. Maroti, P. Volgvesi, G. Nordstorm, J. Sprinkle, G. Karsai, Composing domain-specific design environments, IEEE Computer, vol. 34, no.11, pp. 44–51, 2001.

[4] G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific Publishing Co. Pte. Ltd., 1997.

[5] J. Bézivin, Advances in Model Driven Engineering, in: Jornadas de Ingeniería del Software y Bases de Datos (JISBD), San Sebastian, Spain, September, 2009.

[6] A. Agrawal, G. Karsai, S. Neema, F. Shi, A. Vizhanyo, The design of a language for model transformations, Software and System Modeling, vol. 5, no. 3, pp. 261–288, 2006.

[7] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, P. Valduriez, ATL: a QVT-like transformation language, in: Companion of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 719–720, Portland, OR, October, 2006.

[8] L. Geiger, A. Zündorf, Tool modeling with FUJABA, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 148, no. 1, pp. 173–186, 2006.

[9] A. Tolk, Avoiding another green elephant –A Proposal for the next generation HLA based on the Model Driven Architecture, in: Fall Simulation Interoperability Workshop (SIW), Orlando, FL, September, 2002.

[10] S. Parr, R. Keith-Magee, Making the case for MDA, in: Fall Simulation Interoperability Workshop (SIW), Orlando, FL, September, 2003.

[11] G. Özhan, H. Oguztüzün, P. Evrensel, Modeling of field artillery tasks with Live Sequence Charts, The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology (JDMS), vol. 5, no. 4, pp. 219–252, October, 2008.

[12] O. Topçu, M. Adak, H. Oğuztüzün, A metamodel for federation architectures, ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 18, no. 3.10, 2008.

[13] IEEE 1516 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules, September, 2000.

[14] W. Damm, D. Harel, LSCs: Breathing life into Message Sequence Charts, Formal Methods in System Design, vol. 19, no. 1, pp. 45–80, 2001.

[15] Z.120, Formal Description Techniques (FDT) - Message Sequence Charts. Pre-published Recommendation Telecommunication Standardization Sector of International Telecommunication Union, 2004.

[16] OMG 2006, Meta Object Facility (MOF) core specification, Version 2.0, Available Specification formal / 2006-01-01.

[17] OMG 2007, Unified Modeling Language: Infrastructure, version 2.3, Technical Report formal/2010-05-03.

[18] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood, Transformation: The missing link of MDA, in: International Conference on Graph Transformation (ICGT), pp. 90–105, Barcelona, Spain, October, 2002.

[19] T. Mens, P. Van Gorp, A Taxonomy of model transformation, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 152, pp. 125–142, 2006.

[20] A. Kleppe, J. Warmer, W. Bast, MDA Explained: Practice and Promise, Addison Weslesy, 2003.

[21] OMG 2003, MDA Guide Version 1.0.1. Object Management Group, http://www.omg.org/mda, last accessed on August 18, 2011.

[22] S. Kent, Model Driven Engineering, Lecture Notes In Computer Science, Vol. 2335 Proceedings of the Third International Conference on Integrated Formal Methods, pp. 286–298, 2002.

[23] M. Nagl, Formal Languages of Labeled Graphs, Computing, vol. 16, pp. 113–137, 1976.

[24] A. Habel, Hyperedge Replacement: Grammars and Languages, Lecture Notes in Computer Science (LNCS), vol. 643, Springer-Verlag, Berlin, 1992.

[25] A Habel, Hypergraph Grammars: Transformational and algorithmic aspects, Journal of Information Processing and Cybernetics EIK, vol. 28, pp. 241–277, 1992.

[26] H. Ehrig, M. Pfender, H. J. Schneider, Graph Grammars: an algebraic approach, In Proceegings IEEE Conf. on Automata and Switching Theory, pp. 167–180, 1973.

[27] M. Löwe, Algabraic approach to single-pushout graph transformation, Theoritical Computer Science, vol. 109, pp. 181–224, 1993.

[28] D. Blostein, A. Schürr , Computing with Graphs and Graph Rewriting, Technical Report AIB, pp. 97–8, Fachgruppe Informatik, RWTH Aachen, Germany, 1997.

[29] J.D. Lara, H. Vangheluwe, AToM3: A Tool for Multi-formalism and Meta-modelling, in: 5th international Conference on Fundamental Approaches To Software Engineering, April, 2002. Lecture Notes In Computer Science (LNCS), vol. 2306, pp. 174-188, Springer-Verlag, London, 2002.

[30] G. Taentzer, Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems, Ph.D.Thesis, TU Berlin, Shaker Verlag, Germany, 1996.

[31] F. Marschall, P. Braun, Model Transformations for the MDA with BOTL, in: Workshop on Model Driven Architecture: Foundations and Applications, pp. 83-90, University of Twente, Enschede, The Netherlands, 2003.

[32] D. Varro , G. Varro , A. Pataricza, Designing the automatic transformation of visual languages, Journal of Science of Computer Programming, vol. 44, no. 2, pp. 205-227, 2002.

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

[34] A. Bakay, E. Magyari, The UDM framework, Institute for Software-Integrated Systems, Vanderbilt University, Nashville, TN, October, 2004.

[35] FM 6-30, Tactics, Techniques, and Procedures for Observed Fire, http://www.globalsecurity.org/military/library/policy/army/fm/6-30/index.html, 1991, last accessed on August 18, 2011.

[36] FM 6-40, Tactics, Techniques, and Procedures for Field Artillery Manual Cannon Gunnery, http://www.globalsecurity.org/military/library/policy/army/fm/6-40/index.html, 1999, last accessed on August 18, 2011.

[37] FM 6-50, Tactics, Techniques, and Procedures for The Field Artillery Cannon Battery, http://www.globalsecurity.org/military/library/policy/army/fm/6-50/index.html, 1990, last accessed on August 18, 2011.

[38] Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification (IEEE 1516.1), 2000.

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

[40] Standard for IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP- IEEE 1516.3), 2003.

[41] Z.120 – Annex B, Formal Semantics of Message Sequence Charts, Recommendation of Telecommunication Standardization Sector of International Telecommunication Union (ITU-T), 1998.

[42] M. Brill, W. Damm, J. Klose, B. Westphal, H. Witteke, Live Sequence Charts: An Introduction to Lines, Arrows, and Strange Boxes in the Context of Formal Verification, Springer-Verlag LNCS, vol. 3147, pp. 374-399, 2004.

[43] D. Harel, R. Marelly, Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine, Springer-Verlag, 2003.

[44] S. Robinson, Conceptual Modelling for Simulation Part I: Definition and Requirements, Journal of the Operational Research Society, vol. 59, pp. 278–290, 2008.

[45] W.P. Sudnikovich, J.M. Pullen, M.S. Kleiner, S.A. Carey, Extensible Battle Management Language as a Transformation Enabler, SIMULATION, vol. 80, pp. 669–680, 2004.

[46] U. Schade, M.R. Hieb, Improving Planning and Replanning: Using a Formal Grammar to Automate Processing of Command and Control Information for Decision Support, International C2 Journal, vol. 1, no. 2, pp. 9-90, 2007.

[47] D. Barak, D. Harel, R. Marelly, InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming, IEEE Transactions on Software Engineering, vol. 32, no. 7, pp. 467-485, July 2006.

[48] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming, vol. 8, no.3, pp. 231-274, 1987.

[49] P.K. Davis, R.H. Anderson, Improving the Composability of DoD Models and Simulations, Journal of Defense Modeling and Simulation, vol. 1, no. 1 pp. 5 -17, 2004.

[50] J. Sheehan, T. Prosser, H. Conley, G. Stone, K. Yentz, J. Morrow, Conceptual Models of the Mission Space (CMMS): Basic Concepts, Advanced Techniques, and Pragmatic Examples, in: Spring Simulation Interoperability Workshop, Orlando, FL, March 1998.

[51] V. Mojtahed, M.G. Lozano, P. Svan, B. Andersson, V. Kabilan, DCMF – Defence Conceptual Modelling Framework, A FOI methodology report, FOI-R–1754—SE, November 2005.

[52] N.A. Karagöz, A Framework for Developing Conceptual Models of the Mission Space for Simulation Systems, Ph.D. thesis, Department of Information Systems, METU, Ankara, Turkey, 2008.

[53] The Joint Command, Control and Consultation Information Exchange Data Model, JC3IEDM – UK – DMWG Edition 3.1a, Greding, Germany, 2007.

[54] The C2 Information Exchange Data Model, C2IEDM Main-US-DMWG Edition 6.1, Greding, Germany, 2003.

[55] C. Turnitsa, A. Tolk, Evaluation of the C2IEDM as an Interoperability-Enabling Ontology, European Simulation Interoperability Workshop, 05E-SIW-045, Toulouse, France, June 2005.

[56] C. Turnitsa, A. Tolk, Ontology of the C2IEDM –Further studies to enable Semantic Interoperability, Paper 05F-SIW-084, Fall Simulation Interoperability Workshop, Orlando, FL, September 2005.

[57] D. Brutzman, A. Tolk, JSB composability and Web services interoperability via Extensible Modeling & Simulation framework (XMSF) and Model Driven Architecture (MDA), In Proceedings of Enabling Technologies for Simulation Science: VIII, vol. 5423, pp. 310-319, Orlando, FL, April 2004.

[58] J. Bezivin, On the Unification Power of Models, Springer Verlag, Journal of Software and Systems Modeling, vol.4, no.2, pp. 171-188, 2005.

[59] A. Tolk, R. Kewley, R. Landaeta, T. Litwin, A Systems Engineering Process for driving System of Systems Development with Operational Requirements, in: 29th American Society for Engineering Management Annual Conference, West Point, NY, November, 2008.

[60] G. Özhan, A.C. Dinç, H. Oguztüzün, Model-integrated development of field artillery Federation Object Model, in: Second International Conference on Advances in System Simulation (SIMUL), pp.109–114, Nice, France, August, 2010.

[61] M. Adak, O. Topçu, H. Oguztüzün, Model-based code generation for HLA federates, Software — Practice & Experience, vol. 40, no. 2, pp. 149–175, 2010.

[62] T. Elrad, M. Akşit, G. Kiczales, K. Lieberherr, H. Ossher, Discussing aspects of AOP, Communications of the ACM, vol.44, no. 10, pp. 33–38, 2001.

[63] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting started with AspectJ, Communications of the ACM, vol. 44, no. 10, pp. 59–65, 2001.

[64] J. v. van Wijngaardeen, E. Visser, Program transformation mechanics: A classification of mechanisms for program transformation with a survey of existing transformation systems, Technical Report, Utrecht University, May, 2003.

[65] I. Kurtev, K. van den Berg, F. Jouault, Rule-based modularization in model transformation languages illustrated with ATL, in: ACM Symposium on Applied Computing (SAC), pp. 1202–1209, Dijon, France, 2006.

[66] A. Kleppe, MCC: A model transformation environment, in: 2nd European Conference on Model Driven Architecture, LNSC, pp. 173–187, 2006.

[67] T. Szemethy, Case study: Model transformations for time-triggered systems, in: International Workshop on Graph and Model Transformations (GRaMoT), Tallinn, Estonia, September, 2005.

[68] R. Grønmo, B. Møller-Pedersen, From sequence diagrams to state machines by graph transformation, in: International Conference on Model Transformation (ICMT), pp. 93–107, Malaga, Spain, June, 2010.

[69] T. Ziadi, L. Helouet, J. M. Jezequel, Revisiting statechart synthesis with an algebraic approach, in: International Conference on Software Engineering (ICSE), pp. 242-251, Edinburg, UK, May, 2004.

[70] X. Sun, A model-driven approach to scenario-based requirements engineering, MSc. thesis, School of Computer Science, McGill University, Montreal, Canada, 2007.

[71] F. van Amstel, M.G.J. van den Brand, Z. Protić, T. Verhoeff, Transforming process algebra models into UML state machines: Bridging a semantic gap?, Theory and Practice of Model Transformations - Lecture Notes in Computer Science, vol. 5063, pp. 61–75, 2008.

[72] S. Maoz, D. Harel, From multi-model scenarios to code: compiling LSCs into AspectJ, in: ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, pp. 219–230, Portland, OR, November, 2006.

[73] Y. Bontemps, P. Heymans, P.Y. Schobbens, From live sequence charts to state machines and back: a guided tour, IEEE Transactions on Software Engineering, vol. 31, no. 12, pp. 999–1014, 2005.

[74] I.Krüger, R. Grosu, P. Scholz, M. Broy, From MSCs to statecharts, in: International Workshop on Distributed and Parallel Embedded Systems, pp.61–71, Scholoß Eringerfeld, Germany, October, 1998.

[75] J.J. van Griethuysen, Concepts and terminology for the conceptual schema and the information base, Secretariat ISO/TC97/SC5, American Standards Institute in New York, 1982.

[76] R. Raventós, A. Olivé, An object-oriented operation-based approach to translation between MOF metaschemas, Data & Knowledge Engineering, vol. 67, no. 3, pp. 444–462, 2008.

[77] J. Cabot, R. Pau, R. Raventós, From UML/OCL to SBVR specifications: A challenging transformation, Information Systems, vol. 35, no. 4, pp. 417–440, 2010.

[78] R. Heckel, M. Lohmann, Model-driven development of reactive information systems: from graph transformation rules to JML contracts, International Journal on Software Tools for Technology Transfer (STTT), vol. 9, no. 2, pp. 193-207, 2007.

[79] A. Kraus, A. Knapp, N. Koch, Model-Driven Generation of Web Applications in UWE, In 3rd International Workshop on Model-Driven Web Engineering (MDWE), CEUR-WS, vol 261, 2007.

[80] L. Baresi, R. Heckel, S. Thöne, D. Varro, Style-based modeling and refinement of service-oriented architectures, Software and Systems Modeling (SoSyM), vol. 5, no. 2, pp. 187–207, 2006.

[81] M. Kessentini, H. Sahraoui, M. Boukadoum, O. Benomar, Search-Based Model Transformation by Example, Software and Systems Modeling (SoSyM), Special Issue of MoDELS 20), pp. 1–18, 2010

[82] C. Braga, From Access Control Policies to an Aspect-Based Infrastructure: A Metamodel-Based Approach, in MoDELS, pp. 243-256, 2008.

[83] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, On Challenges of Model Transformation from UML to Alloy, Software and Systems Modeling (SoSym), Special Issue on MoDELS 2007, vol. 9, no. 1, pp. 69-86, 2008.

[84] G. Özhan, H. Oguztüzün, Generating simulation code from federation models: A field artillery case study, in: European Simulation Interoperability Workshop (EuroSIW), 11E-SIW-007, The Hague, Netherlands, June, 2011.

[85] O. Topçu, Metamodeling for the HLA Federation Architectures, Ph.D.Thesis, Department of Computer Engineering, METU, Ankara, Turkey, 2007.

[86] M. Adak, Model-Based Code Generation for the High Level Architecture Federates, Ph.D.Thesis, Department of Computer Engineering, METU, Ankara, Turkey, 2007.

[87] A. Molla, K. Sarıoğlu, O. Topçu, M. Adak, H. Oğuztüzün, Federation Architecture Modeling: A Case Study with NSTMSS, in: Fall Simulation Interoperability Workshop (SIW), 07F-SIW-052, Orlando, Florida, USA, September pp. 16-21, 2007.

[88] R. Kewley, A. Tolk, T. Litwin, PEO Soldier Simulation Road Map V – The MATREX Federation, Technical Report DSE-TR-0802, Operations Research Center of Excellence, Westpoint, NY, September, 2008.

[89] M. G. Lozano, V. Mojtahed, A Process for Developing Conceptual Models or the Mission Space (CMMs) – From Knowledge Acquistion to Knowledge Use, in: Fall Simulation Interoperability Workshop, 05F-SIW-038, Orlando, FL, September, 2005.

[90] S. Etienne, L. Xavier, V. Olivier, Applying MDE for HLA federation rapid generation, in: European Simulation Interoperability Workshop, 06E-SIW-066, Stockholm, Sweden, June, 2006.

[91] E. Guiffard, D. Kadi, J. P. Mochet, R. Mauget, CAPSULE: Application of the MDA methodology to the simulation domain, in: European Simulation Interoperability Workshop, 06E-SIW-026, Stockholm, Sweden, June, 2006.

[92] J. Carlaftes, B. Collins, D. Fiehler, Auto-Code Based Model-Driven Engineering Techniques for Simulation Development, in: Spring Simulation Interoperability Workshop, 10S-SIW-015, Orlando, FL, April, 2010.

[93] F. Küçükyavuz, N. A. Karagöz, O. Demirörs, Constructing Bridges between Mission Space Conceptual Models and BOM, in: Spring Simulation Interoperability Workshop, 11S-SIW-036, Boston, MA, April, 2011.

[94] Ö. Özdikiş , U Durak , H. Oğuztüzün, Tool support for transformation from an OWL ontology to an HLA Object Model, in: 3rd International ICST Conference on Simulation Tools and Techniques, Malaga, Spain, March, 2010.

[95] D. A. Brade, Conceptual Modeling Meets Formal Specifications, in: Spring Simulation Interoperability Workshop, 04S-SIW-138, Orlando, FL, April, 2003.

[96] B. Andersson, M. G. Lozano, V. Mojtahed, The Use of a Knowledge Meta Meta Model (KM3) when Building Conceptual Models of the Mission Space, in: Fall Simulation Interoperability Workshop, 05F-SIW-040, Orlando, FL, September, 2005.

[97] V. Mojtahed, E. O. Svee, J. Zdravkovic, Semantic enhancements when designing a BOM-based conceptual repository, in: European Simulation Interoperability Workshop, 10E-SIW-021, Ottawa, Canada, July, 2010.

[98] SISO, Base Object Model (BOM) Template Specification, SISO-STD-003-2006, 2006.

[99] SISO, Guide for Base Object Model (BOM) Use and Implementation, SISO-STD-003.1-2006, 2006.

[100] T. Gruber, A translation approach to portable ontology specifications, Knowledge Acquisition, vol. 5, pp. 199–220, 1993.

[101] J. Miller, G. Baramidze, P.Fishwick, Investigating ontologies for simulation and modeling, in: 37th Annual Simulation Symposium, pp.55–71, 2004.

[102] A. Tolk, Evaluation of the C2IEDM as an interoperability-enabling ontology, in: European Simulation Interoperability Workshop, Toulouse, France, June, 2005.

[103] L. W. Lacy, Interchanging discrete-event simulation process interaction models using the web ontology language – OWL. PhD Dissertation, Department of Industrial Engineering and Management Systems, University of Central Florida, 2006.

[104] Y. Teo, C. Szabo, CODES: An integrated approach to composable modeling and simulation, in: 41st Annual Simulation Symposium, pp.103–110, 2008.

[105] L. Yilmaz, S. Paspuleti, Toward a meta-level framework for agent-supported interoperation of defense simulations, The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology (JDMS), vol. 2, pp. 161–175, 2005.

[106] U. Eryılmaz, N.A. Karagöz, KAMA: A Tool for Developing Conceptual Models For C4ISR Simulations, in: European Simulation Interoperability Workshop, 09E-SIW-020, İstanbul, Turkey, July, 2009.

[107] G. A. Silver, O. A. Hassan, J. A. Miller, From domain ontologies to modeling ontologies to executable simulation models, in: 39th Winter Simulation Conference, Washington D.C., December, 2007.

[108] WC3 Health Care and Life Science Group, A Problem-Oriented Medical Records Ontology, http://esw.w3.org/topic/HCLS/POMROntology, 2006, last accessed on August 18, 2011.

[109] G. A. Silver, L. W. Lacy, J. A. Miller, Ontology Based Representations of Simulation Models Following the Process Interaction World View, in: Winter Simulation Conference, pp. 1168–1176, Monterey, CA, 2006.

[110] U. Durak, H. Oğuztüzün, S. K. İder, Ontology-based domain engineering for trajectory simulation reuse, International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 8, pp.1109–1129, 2009.

[111] U. Durak, H. Oğuztüzün, S. K. İder, An ontology for trajectory simulation, in: Winter Simulation Conference, Monterey, CA, 2006.

[112] O. Özdikiş, User guided transformations of OWL ontologies to UML class diagrams, M.Sc. Thesis, Department of Computer Engineering, METU, Turkey, 2007.

[113] A. D'Ambrogio, D. Gianni, J. L. Risco-Martín, A. Pieroni, A MDA-based Approach for the Development of DEVS/SOA Simulations, in: Spring Simulation Interoperability Workshop, Orlando, FL, April, 2010.

[114] openArchitectureware, an MDA/MDD generator framework, http://www.openarchitectureware.org/, last accessed on August 18, 2011.

[115] D. K. Pace, Conceptual Model Descriptions, in: Spring Simulation Interoperability Workshop, 99S-SIW-025, Orlando, FL, March 1999.

APPENDIX A

ADJFFE MODEL LSCS IN GRAPHICAL NOTATION

This appendix presents all of the LSCs for the *Adjustment Followed by Fire For Effect* (*AdjFFE*) mission model in graphical notation. Each LSC is provided with a brief description of its purpose, execution conditions and logic.

Figure A.1 shows the top-level chart, `AdjFFE`, which provides an overall coverage of the mission. Its LSC activation mode is iterative and quantification is existential [14] (from this point on only the values of activation mode and quantification will be mentioned for the sake of brevity). The chart includes all of the eight instances, a prechart and a body with a parallel inline expression with two inline operands. The prechart consists of references to three MSCs and the parallel expression has references to two MSCs in its first operand and has a reference to one MSC in its second operand. The referred MSCs (and their contained LSCs) are presented in subsequent figures.

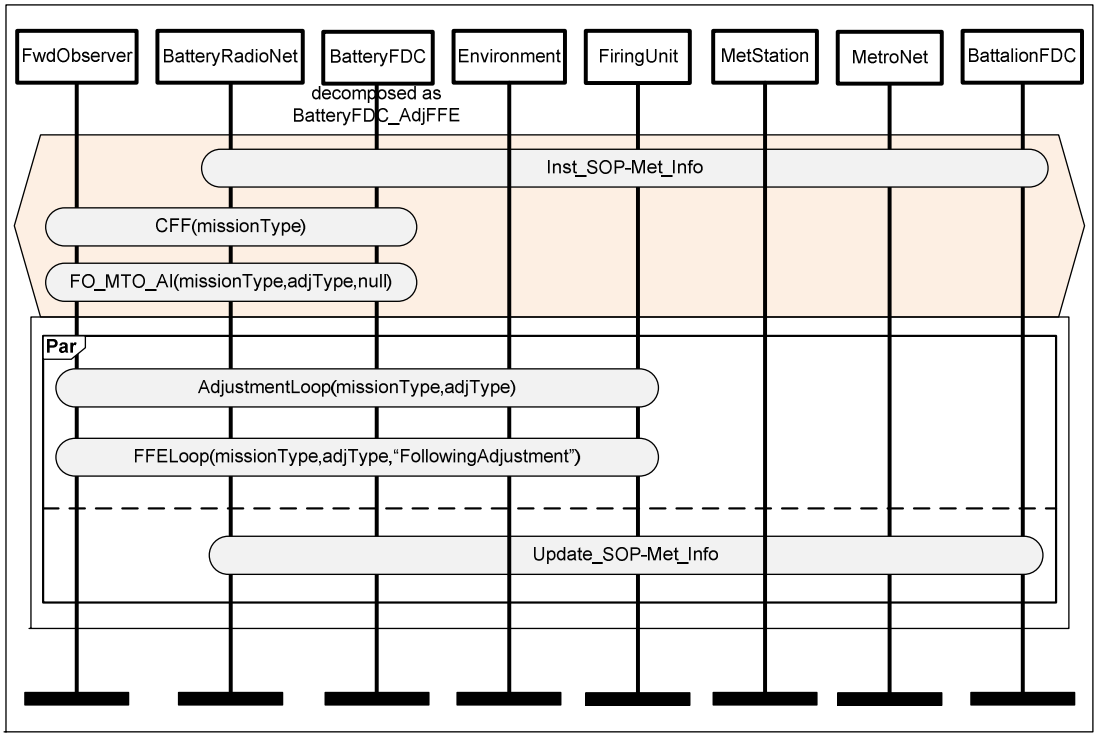


Figure A.1 The Main LSC for AdjFFE mission LSC

The iterative and universal call for fire chart, `CFF`, is depicted in Figure A.2. It describes the call for fire request made by the forward observer to the battery FDC. It consists of the preparation and sending of one mandatory and two optional messages. The optional messages are sent if the mission type is given as `adjustment`, `FFE` or `destruction`. All the messages sent to the battery FDC are also simultaneously sent to the battery radio net.

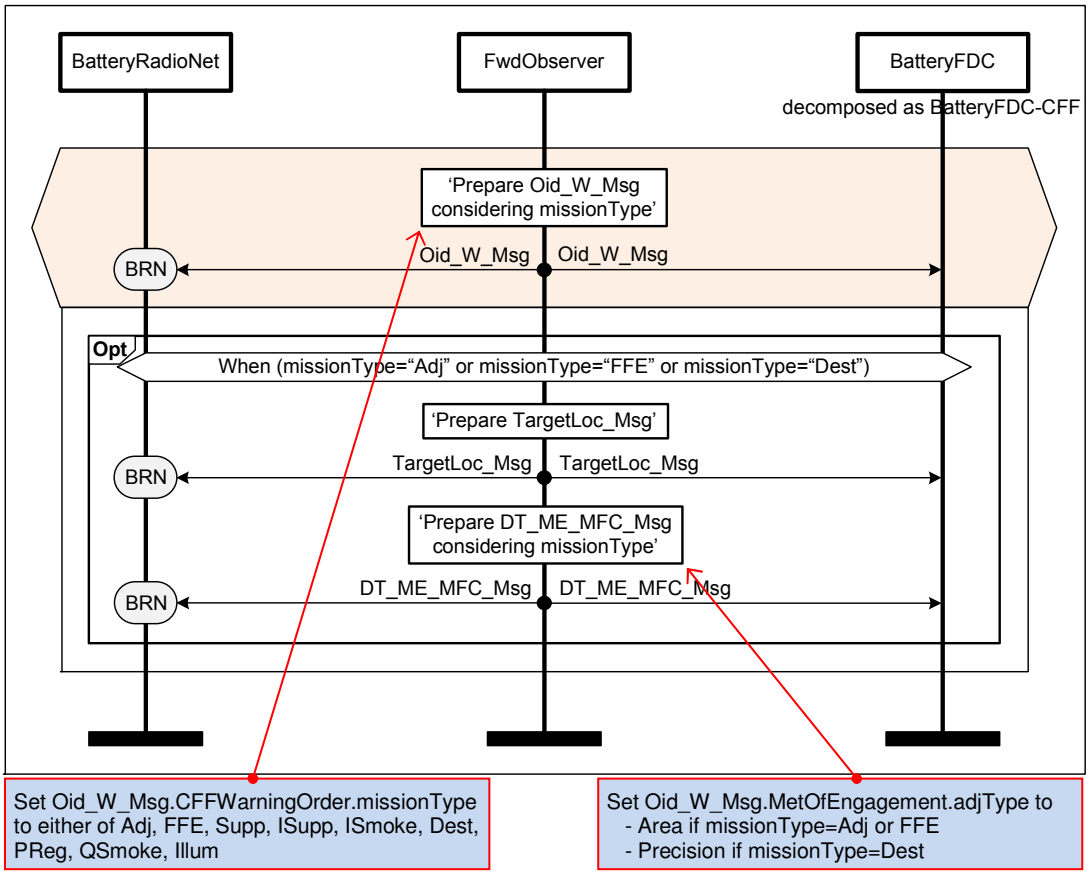


Figure A.2 Call for fire LSC

The iterative and universal `FO_MTO_AI` chart is illustrated in

Figure A.3. It covers the messages sent by the battery FDC to the forward observer in response to a previous CFF request. It consists of three alternatively sent `MTO`, one optional `additionalInfo` and one mandatory `fireOrder` (not shown in the figure – sent within decomposed `BatteryFDC`) messages. All of the messages sent to the battery FDC are also simultaneously sent to the battery radio net.

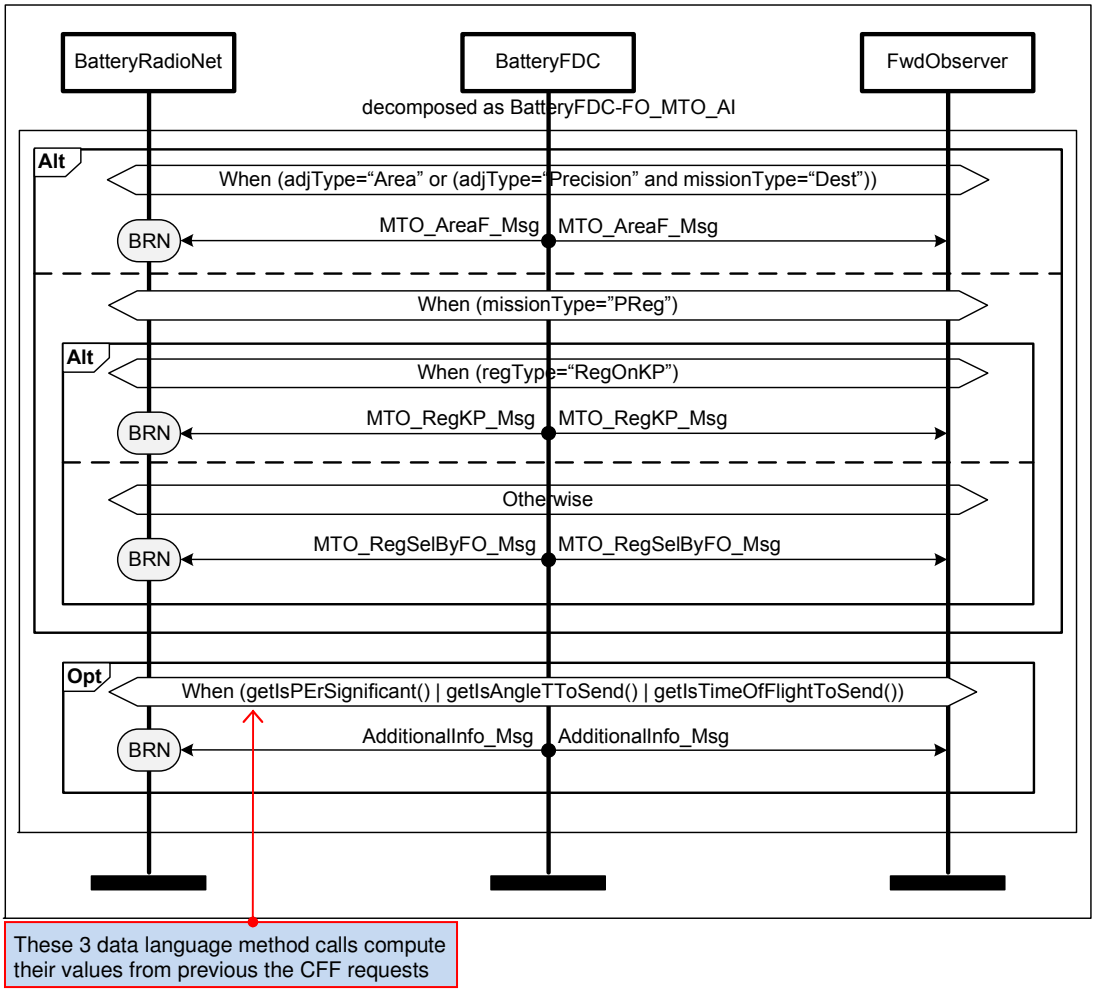


Figure A.3 FO_MTO_AI (Fire Order, Message to Observer, Additional Information) LSC

The iterative and existential `AdjustmentLoop` chart is sketched in Figure A.4. Its activation condition [14] is that the `cannotObserve` flag of the `methodOfControl` part of the `CFF` message be `false`. The chart starts with a prechart those references to an MSC handling the initial fire command preparation and sending. Then it loops until the adjustment is complete, cycling through rounds firing, spotting observation, adjustment correction and subsequent fire command generation steps. When the adjustment is decided to be done in the last loop cycle, the `missionType` is set to `FFE`. The system method `getIsAdjNotDone()` computes its value from an `ObservedSp_Msg` sent within the `ObserveSpotting` MSC.

The invariant and universal `ObserveSpotting` chart is shown in Figure A.5. Its activation depends on the detonation of the ammunition. `Environment` tests the observed spotting in a four operand alternative expression and sends a spotting message accordingly.

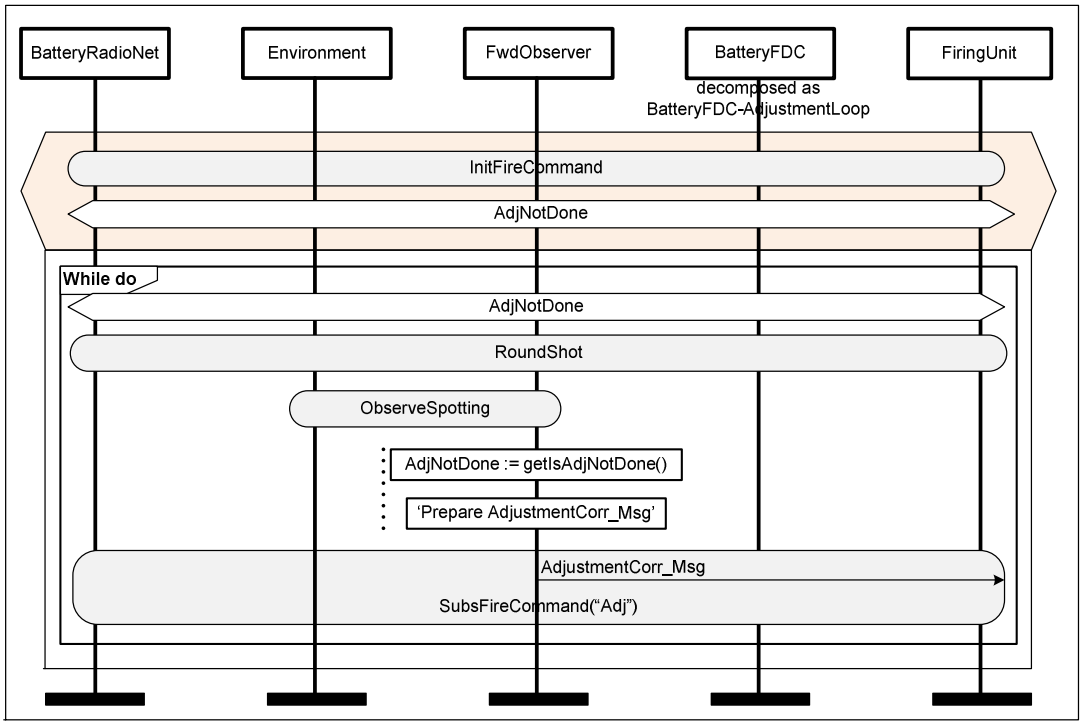


Figure A.4 Adjustment loop LSC

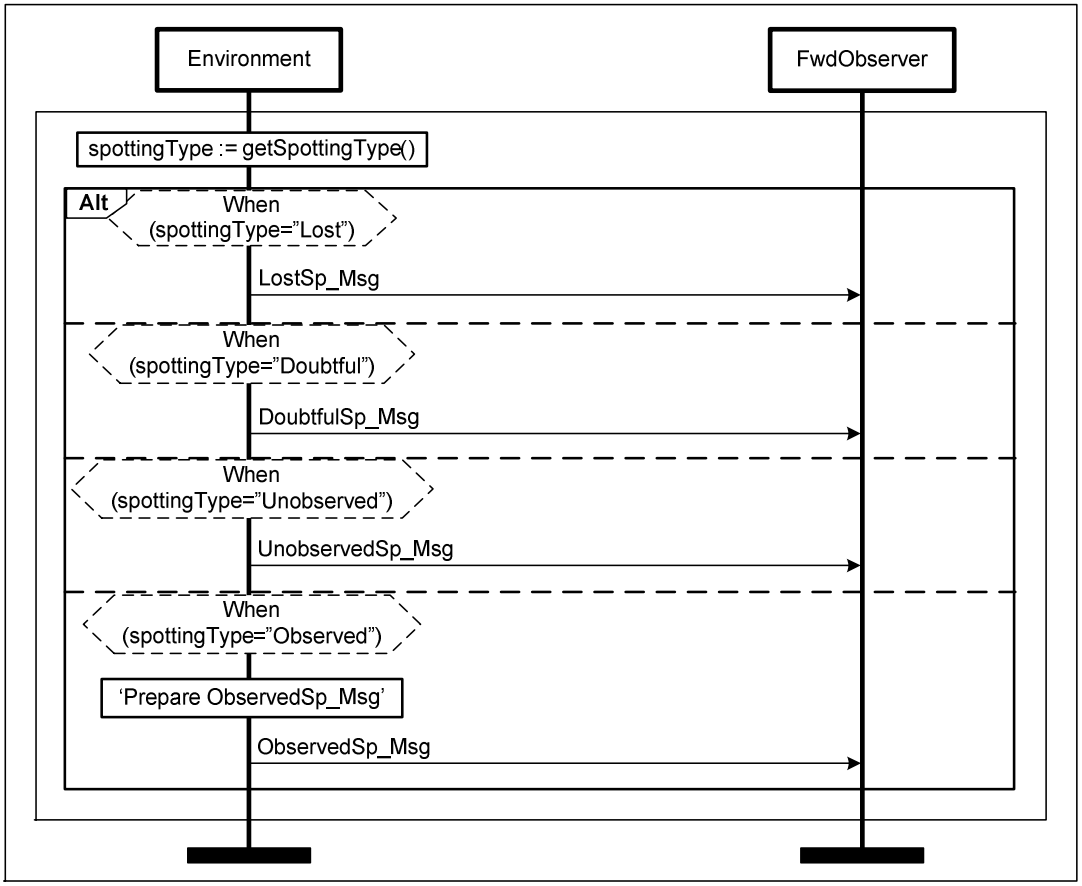


Figure A.5 Observe spotting LSC

The iterative and universal `InitFireCommand` chart is depicted in Figure A.6. It covers the generation of an initial fire command message based on value combinations of the method of control variable. It consists of an alternative expression with two operands. The second operand has two optional blocks followed by an initial fire command message transmission and ends with a call to an MSC that processes the fire command. All of the messages sent to the firing unit are also simultaneously sent to the battery radio net.

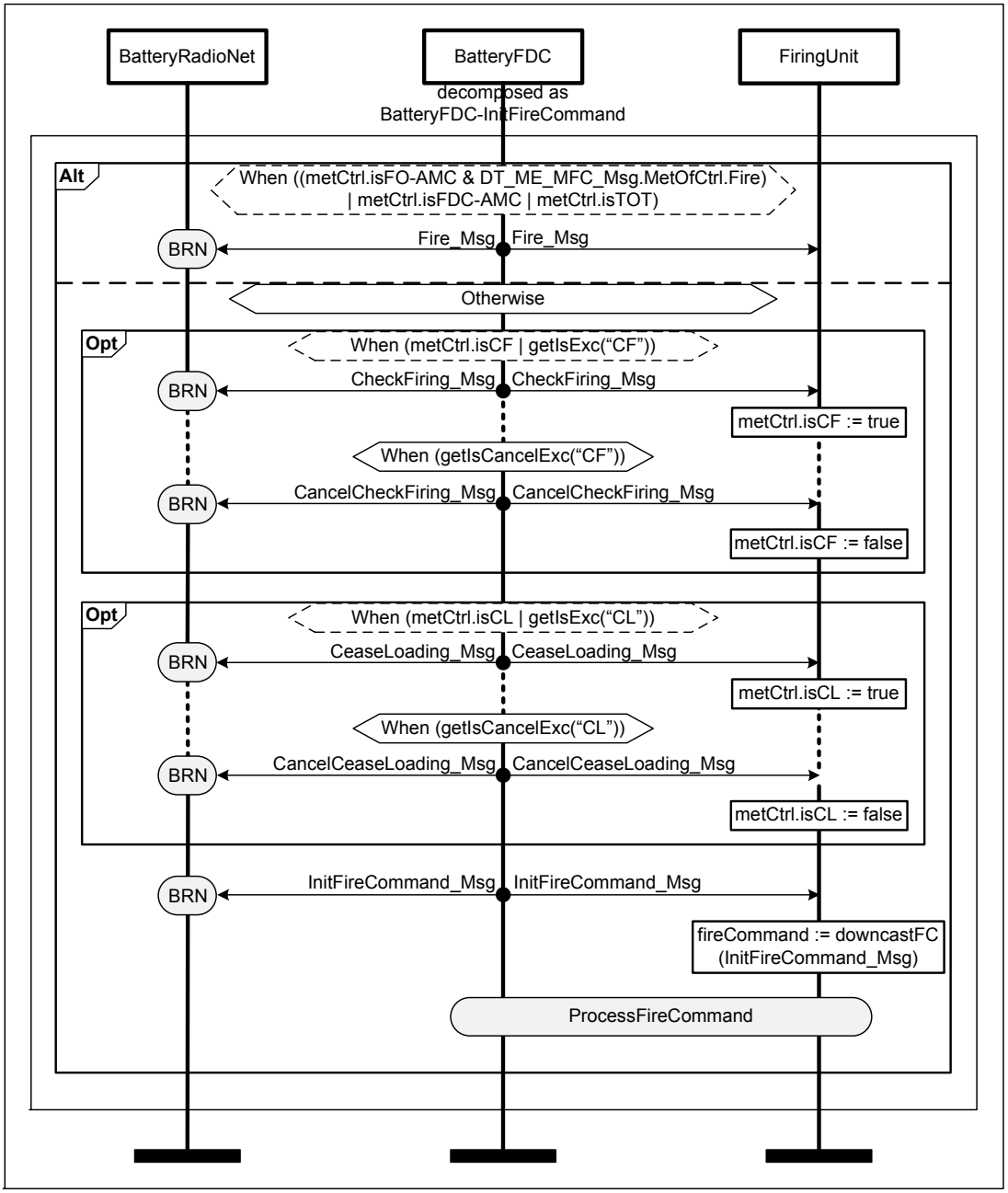


Figure A.6 Initial fire command LSC

The iterative and universal `InitFireCommand` LSC is illustrated in Figure A.7. Once the initial fire command is received, this chart is used for managing further communication between the firing unit and the battery FDC. It consists of an LSC body with four optional blocks that are entered based on various properties of the received fire command and set various properties of the method of control variable based on those values. The last two optional blocks house `Ready`, `Fire` and `Laid` message transmissions.

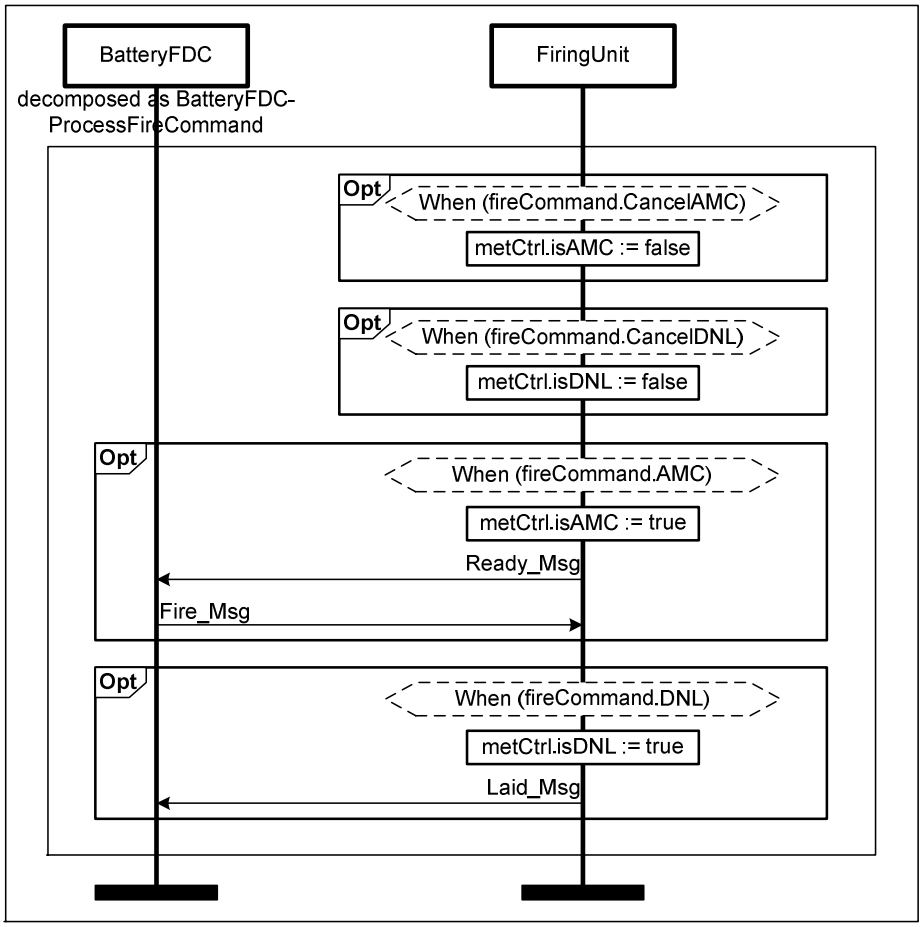


Figure A.7 Process fire command LSC

The iterative and universal `SubsFureCommand` LSC is sketched in Figure A.8. It is used for managing fire commands sent after the initial one. It consists of a prechart and an LSC body with three-operand alternative expression. The prechart starts execution with two gates that input messages from the external world and relay them to two MSCs. The alternative expression covers the generation of a subsequent fire command message based on value combinations of the method of control variable. The third operand has two

optional blocks followed by a subsequent fire command message transmission and ends with a call to an MSC that processes the fire command. All of the messages sent to the firing unit are also simultaneously sent to the battery radio net.

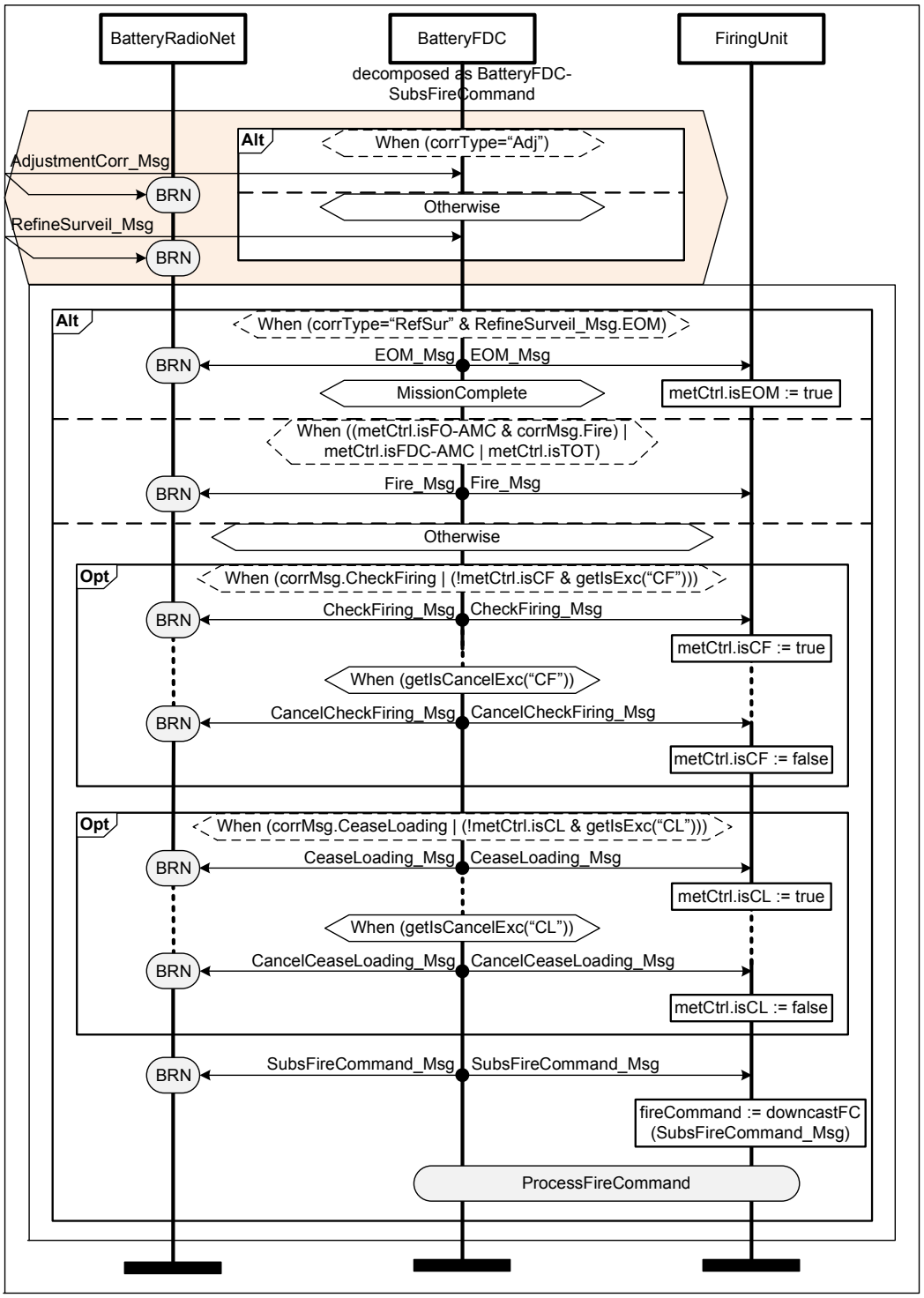


Figure A.8 Subsequent fire command LSC

The initial and universal durable message instantiator chart, `InstSOPMet`, is shown in Figure A.9. It simply houses the preparation and transmission of battalion fire order SOP, fire command SOP and computer meteorology report instantiation messages.

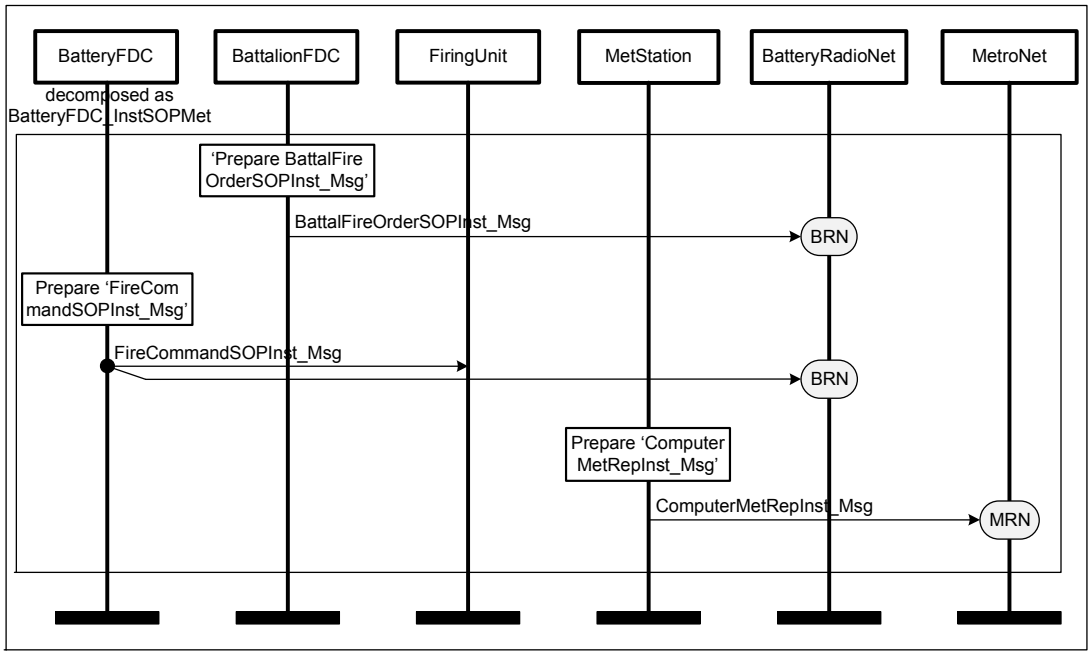


Figure A.9 Instantiation type of durable messages LSC

The initial and universal durable message updater chart, `UpdateSOPMet`, is depicted in Figure A.10. It consists of a parallel expression with three operands that contain references to MSCs handling battalion fire order SOP, fire command SOP and computer meteorology report updating.

The initial and universal battalion fire order SOP updater chart, `BattalionFOUpdate`, is illustrated in Figure A.11. The battalion FDC periodically prepares and sends update messages for the battalion fire order SOP to the battery radio net until the mission completes or fails. The message update period is timer based. After the while-do loop exits, the final message sent is deletion for the SOP.

The initial and universal fire command SOP updater chart, `FireCommandUpdate`, is sketched in Figure A.12. The battery FDC periodically prepares and simultaneously sends update messages for the fire command SOP to the firing unit and the battery radio net until the mission completes or fails. The message update period is timer based. After the while-do loop exits, the final message sent is deletion for the SOP.

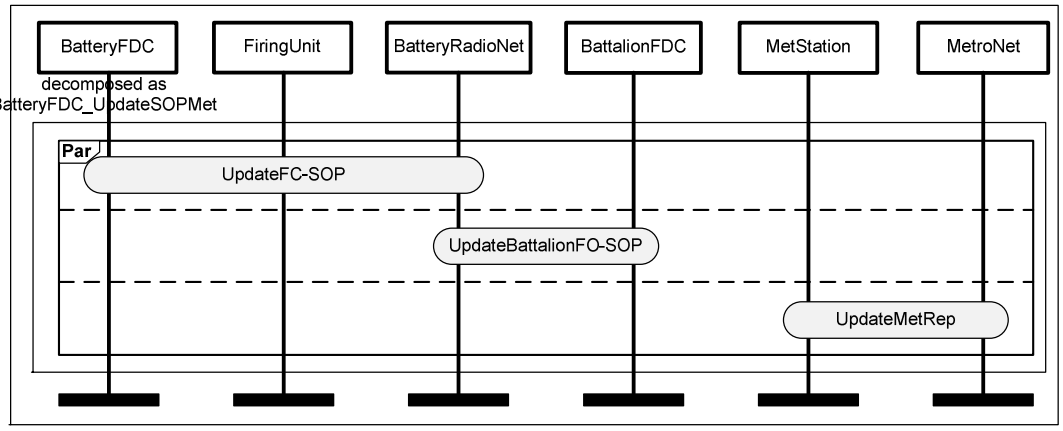


Figure A.10 Overall update type of durable messages LSC

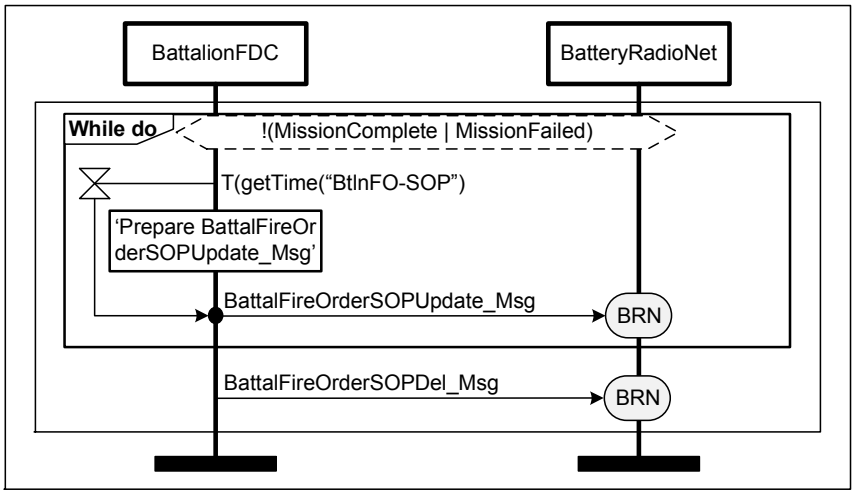


Figure A.11 Battalion fire order update and delete LSC

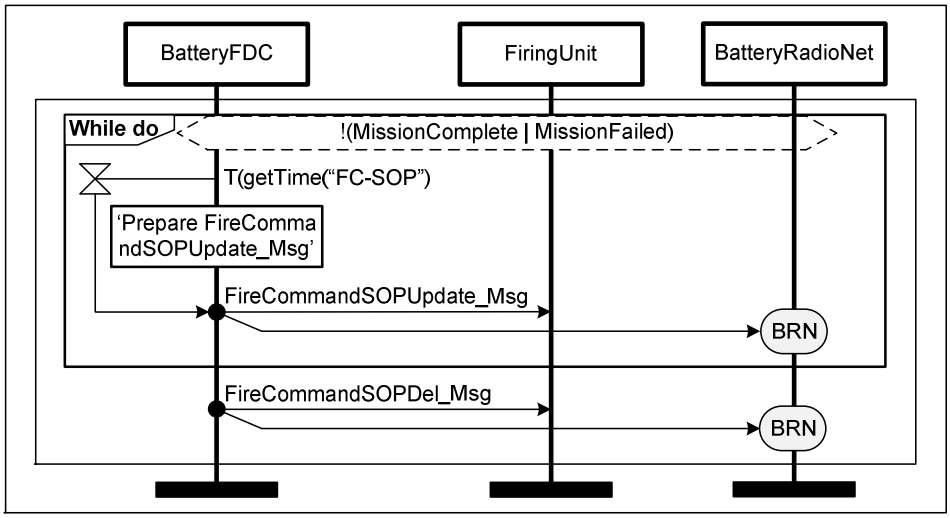


Figure A.12 Fire command update and delete LSC

The initial and universal meteorology report updater chart, `MetRepUpdate`, is shown in Figure A.13. The meteorology station periodically prepares and sends update messages for the computer meteorology report to the metro net until the mission completes or fails. The message update period is timer based. After the while-do loop exits, the final message sent is deletion for the meteorology report.

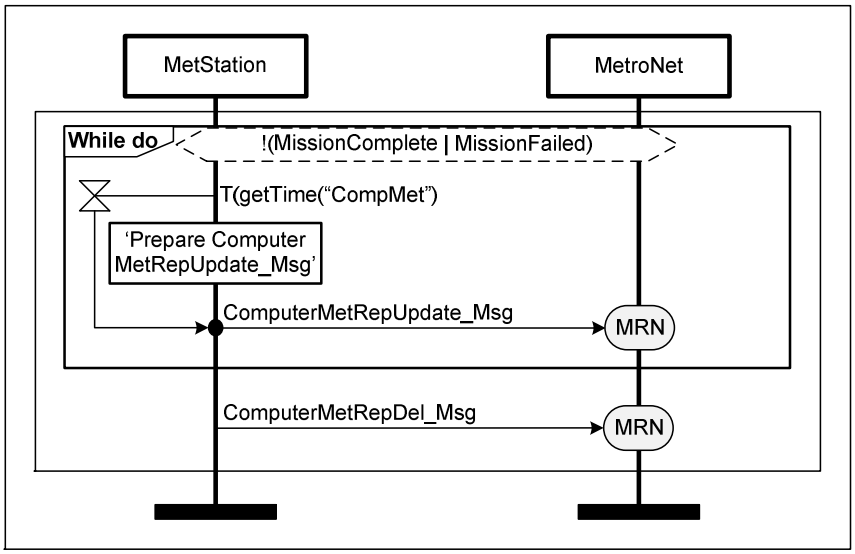


Figure A.13 Metro report update and delete LSC

The invariant and existential `RoundShot` LSC is depicted in Figure A.14. It models a durable ammunition object's life cycle from the moment of its creation, to being fired, to being updated throughout its flight for trajectory changes, to its detonation and finally deletion. The `fire` action is explicitly ordered after the ammunition's instantiation. The ammunition's flight and its trajectory updates take place in a parallel expression. The time of flight and ammunition update period are both timer based, where the time of flight is acquired through a system variable and update period is constant. After the ammunition is fired, a series of two `shot` messages and an optional `splash` message are also transmitted.

The invariant and existential `VolleyShot` LSC is illustrated in Figure A.15. It is very similar to the `RoundShot` LSC with the difference that `VolleyShot` has a three-operand parallel expression where the first operand's optional expression and the second operand are controlled by the `volleyShotType` variable, which effectively makes their executions mutually exclusive. When `volleyShotType` is `initial`, then the LSC behaves exactly as the `RoundShot` LSC. When it is `final`, a rounds complete message is sent after firing.

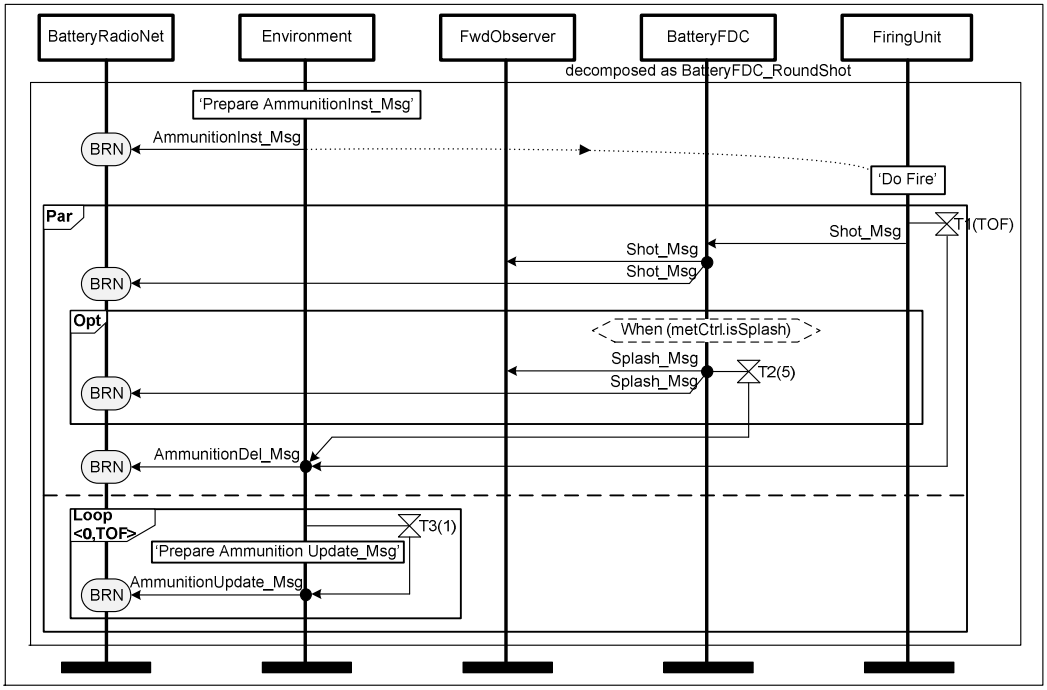


Figure A.14 Round shot LSC

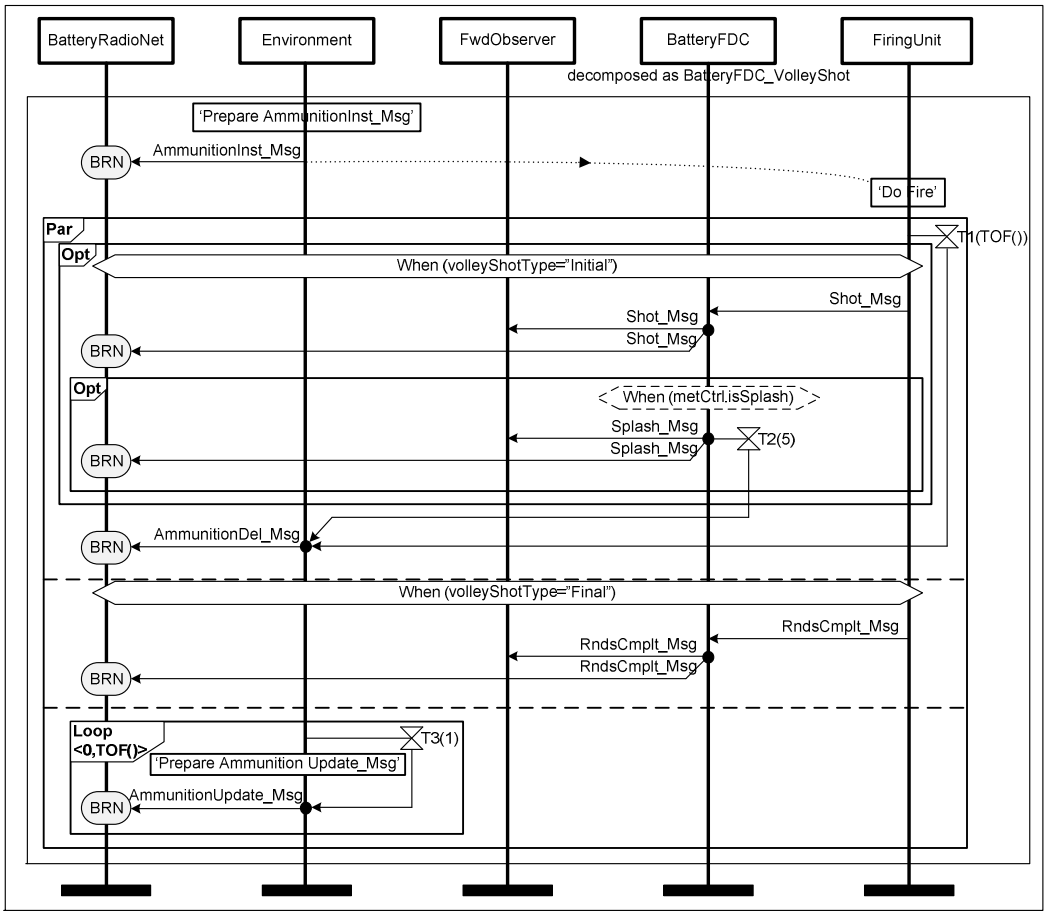


Figure A.15 Volley shot LSC

The invariant and existential `VolleyFire` LSC is sketched in Figure A.16. Its activation condition is a fire command with quadrant elevation being sent. The chart body starts with an `initial` volley shot followed by a spotting observation. If the number of rounds to fire is greater than 2, then rounds-2 `intermediate` volley shots are fired with observations. If the number of rounds to fire is greater than 1, then a `final` volley shot is made. Note that all of the events in this LSC are references to either `VolleyShot` (with a parameter) or `ObserveSpotting` MSCs.

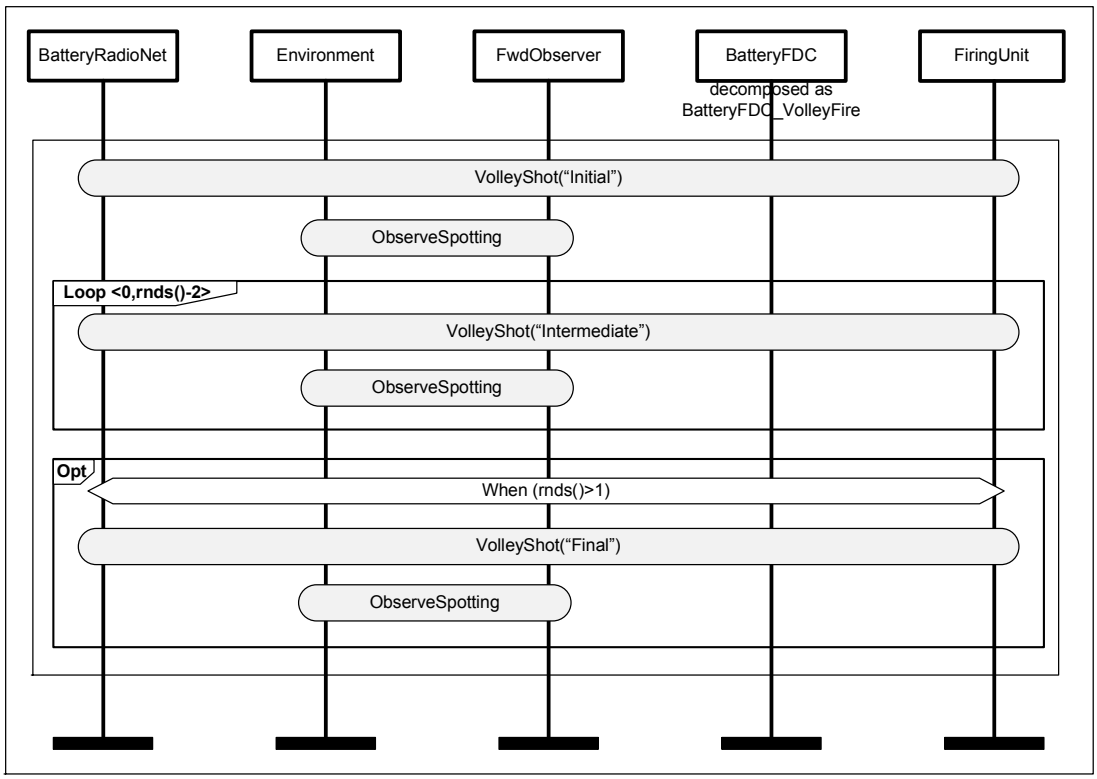


Figure A.16 Volley fire LSC

The invariant and universal `MetroNet` LSC is shown in Figure A.17. It models all of the incoming meteorology report messages to the meteorology net and their distribution within the net members. There are instantiation, update and delete types of computer and ballistic meteorology reports. The LSC body loops receiving and distributing these six messages in an alternative expression of six operands until the mission completes or fails.

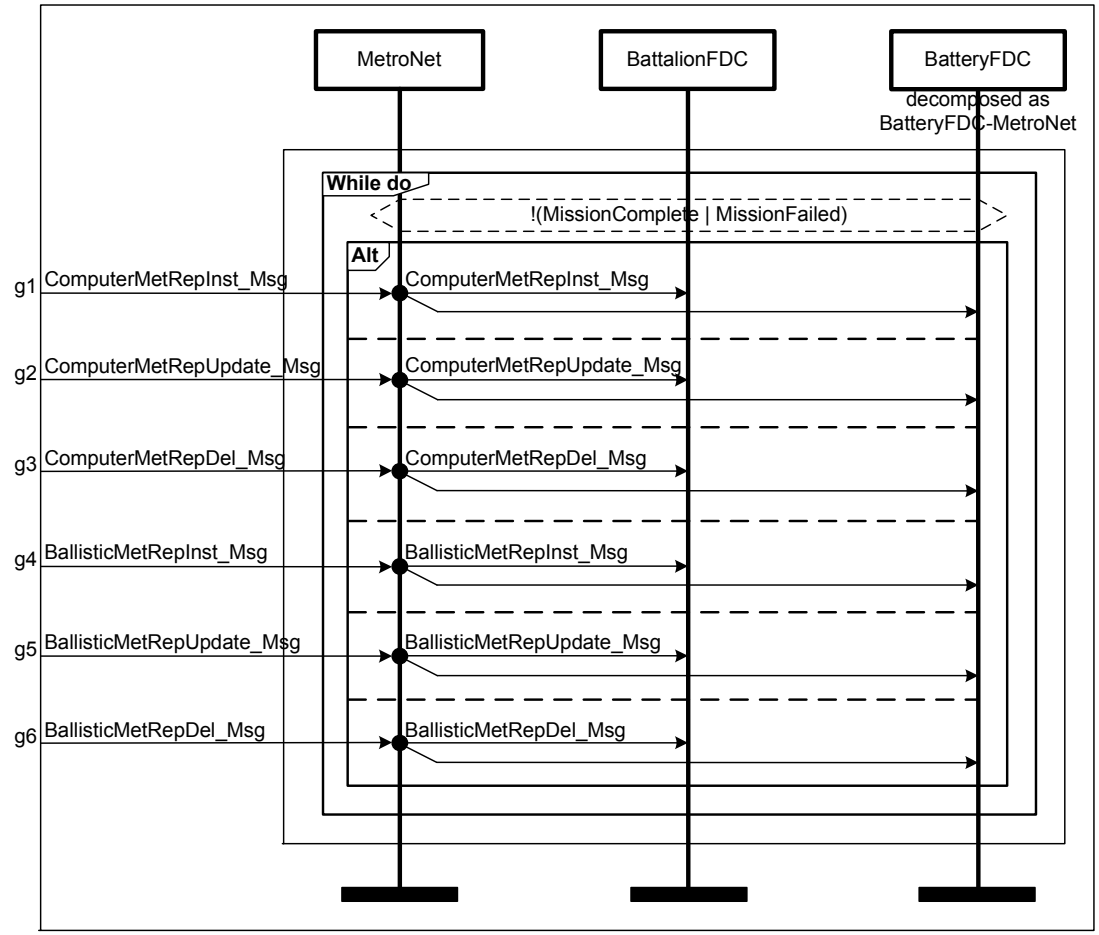


Figure A.17 Metro net LSC

The invariant and universal `BatteryRadioNet` LSC is depicted in Figure A.18. It models all of the incoming messages to the battery radio net and their distribution within the net members. There are 23 types of incoming messages through the gates. The LSC body loops receiving and distributing these messages in an alternative expression of 23 operands until the mission completes or fails. The loop also includes a reference to the `BattalionIntervention` MSC after the alternative expression, to check whether there is an intervention on the mission by the battalion. One of the incoming messages is the `EOM` message. When this message is received, the global `MissionComplete` flag is set and the mission ends successfully.

The iterative and universal `BattalionIntervention` LSC is shown in Figure A.19. This chart is not a main stream chart and is used to provide an upper command intervention on the mission. It consists of an optional block controlled by an external system method. If the block is entered the mission is aborted, variables are reset and a new fire order is issued.

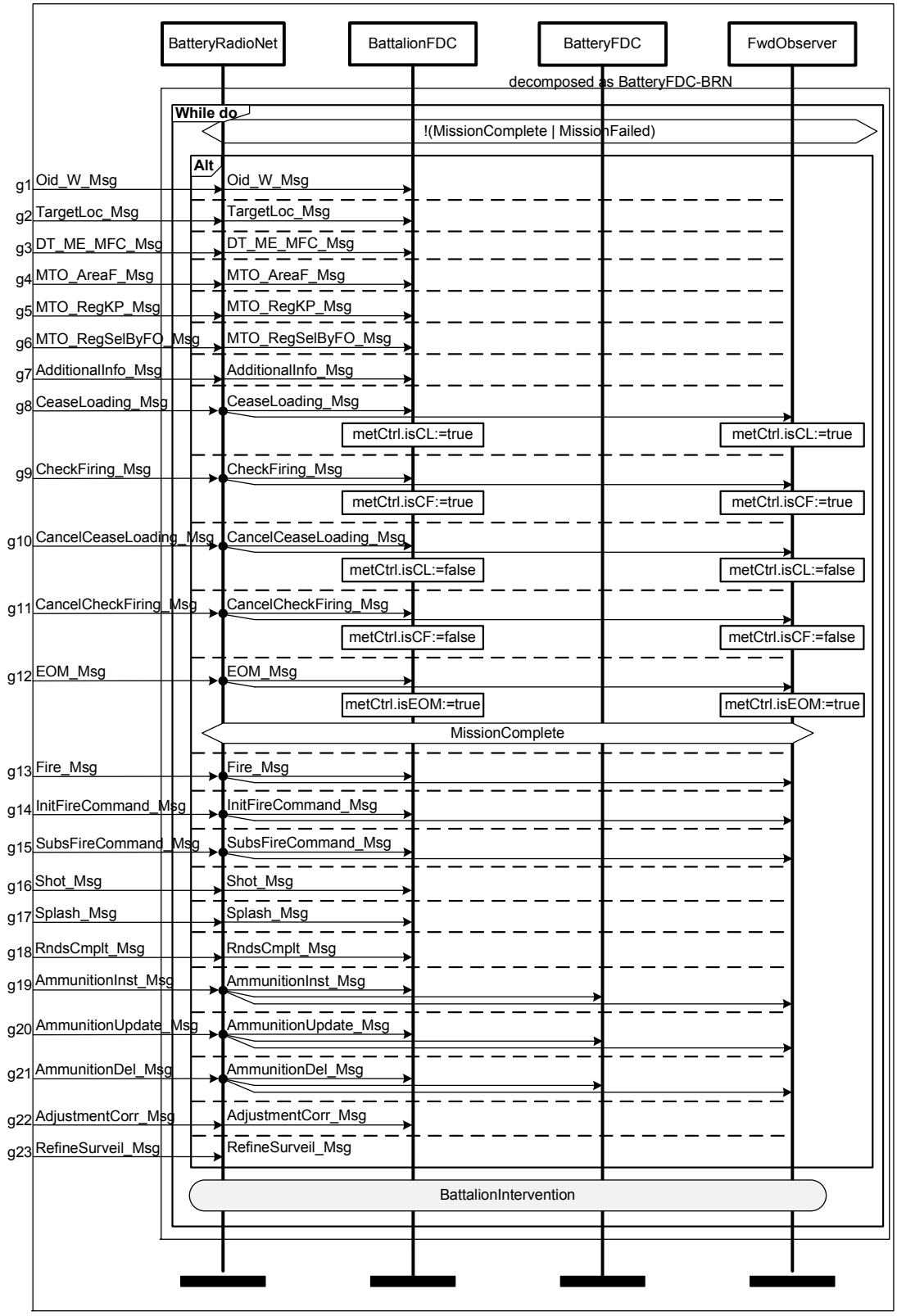


Figure A.18 Battery radio net LSC

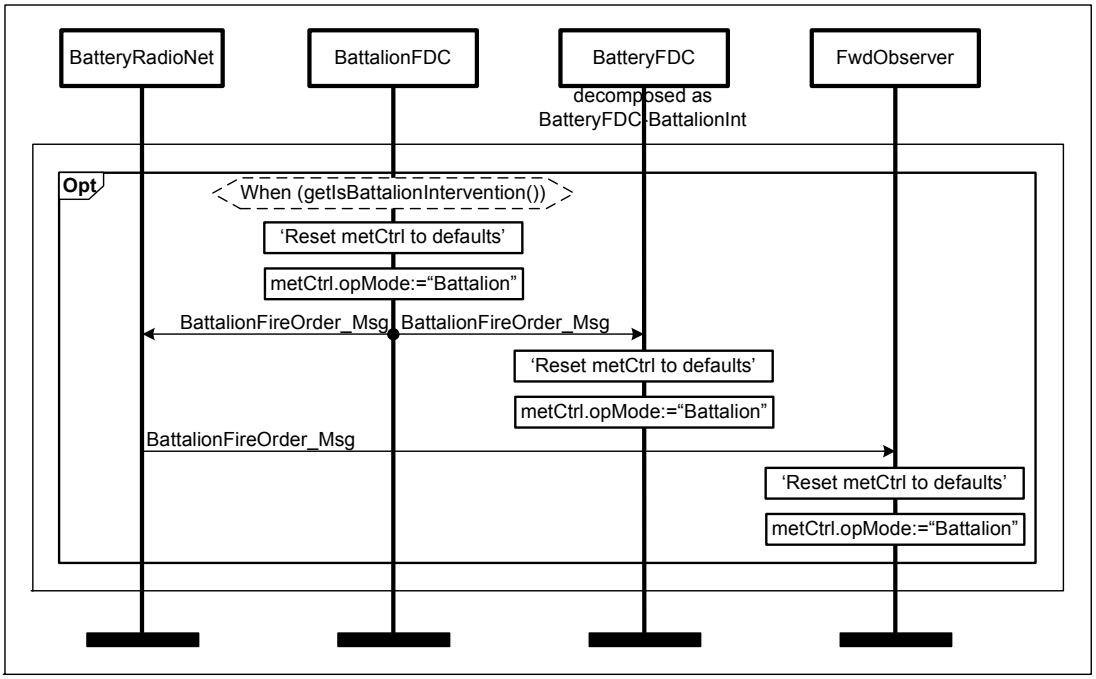


Figure A.19 Battalion intervention LSC

The iterative and existential `FFELoop` LSC is illustrated in Figure A.20. The chart can be entered without a prior adjustment step, or after an adjustment that ends with a correction message transmission indicating that the adjustment is done and fire for effect can be started. The fire for effect chart models the scenario where all the guns in a battery fire their rounds with the same fire parameters. `FFELoop` starts with a prechart that issues an initial fire command if the mission is being performed without an adjustment. The chart body usually ends after a call to `VolleyFire` that yields a satisfactory result. If the outcome is accurate, but insufficient, then volley shots are repeated in a loop until a different result is obtained. If the result is worse; that is, inaccurate and insufficient, then the mission is restarted from adjustment stage. In any case, a last refinement and surveillance message with `EOM` flag set is sent into the `SubsFireCommand`, which in turn, ends the mission.

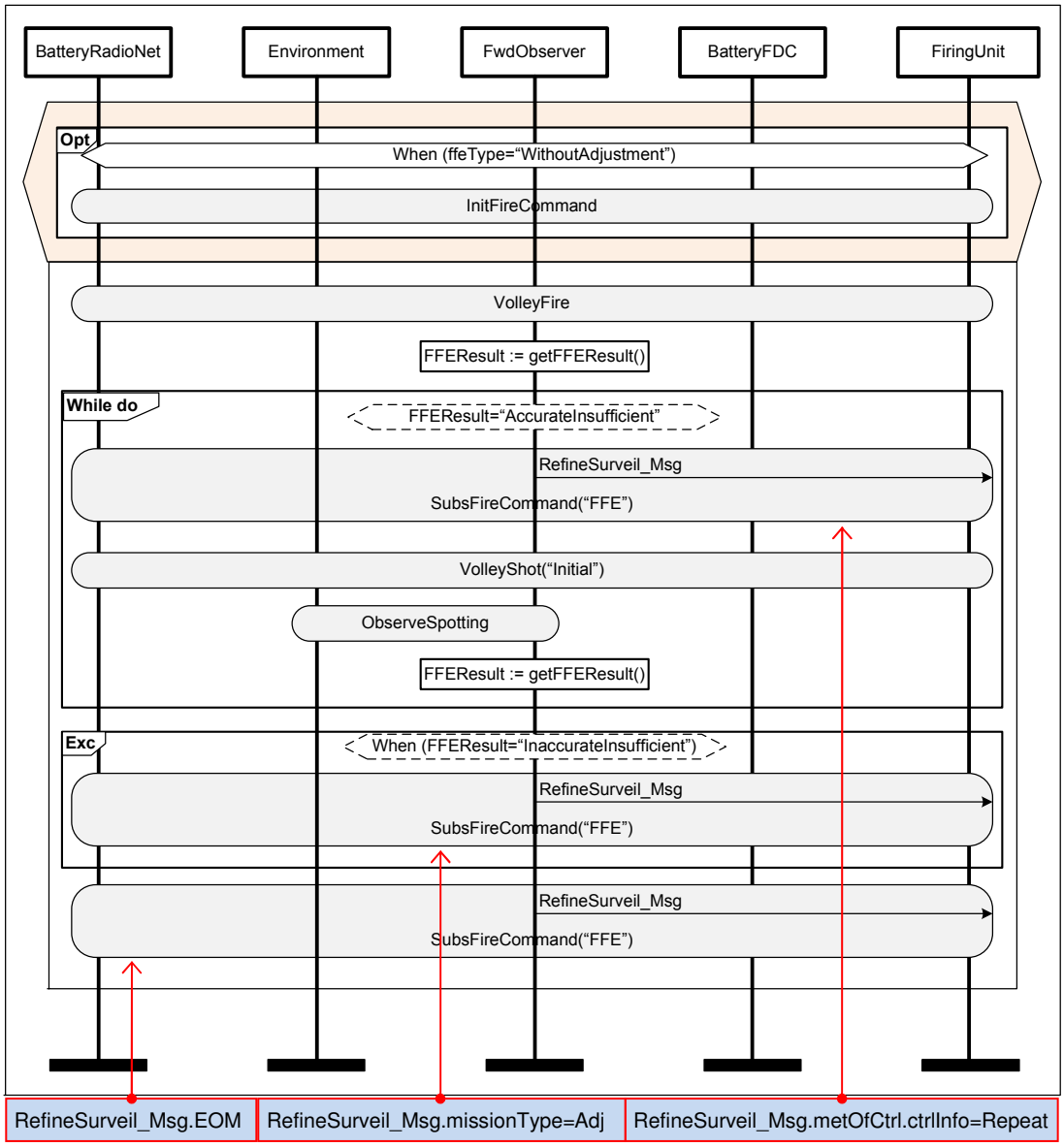


Figure A.20 Fire for effect loop LSC

APPENDIX B

ACM TO FAM MODEL TRANSFORMATION RULES

This Appendix presents the set of most prominent ACM to FAM model transformation blocks and rules as implemented in GReAT-configured GME. Although there are an abundance of blocks and rules depicted (about 130), it is still half of the total number. Only relevant and representative blocks and rules are included. The full set can be found in the transformation definition file accompanied with the thesis CD. The presented blocks and rules are usually compact enough and self explanatory. Overall explanations are provided at section heads and specifics are provided above the figures where deemed necessary.

B.1 Start Block

The `start` block is shown in Figure B.1. It presents a top-level view of the overall transformations. It is seen that ACM2FAM transformation consists of the global container's initialization, data model transformation, behavioral model transformation and binding the calls to decomposed instance document's MSCs from the main document's MSCs.

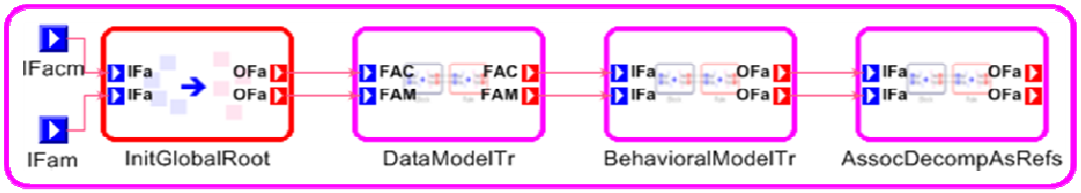


Figure B.1 Start block

The initialization of the global container is sketched in Figure B.2. The general idea of the global container is that the objects it contains have global scope; that is, they are accessible throughout the whole transformation, and it is not necessary to pass them along in the context. The capability of eliminating portions of context passing and recurring complex pattern matching is one of the key facilitating factors in terms of the development effort and execution performance in this work.

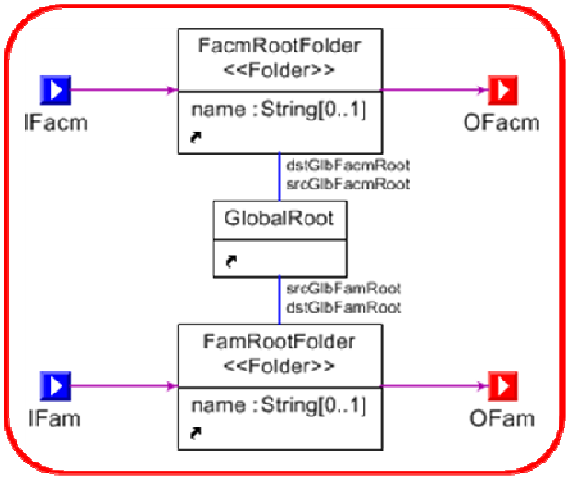


Figure B.2 InitGlobalRoot rule

B.2 Data Model Transformation

Data model transformation corresponds to the structural part of the ACM2FAM transformation. Looking from a FAM perspective, it aims to construct the federation object, the federate objects and the Federation Object Model (FOM) for the federation. The main `DataModelTr` block is shown in Figure B.3. It is composed of two inner blocks named `ObjectModelTr` and the relatively smaller `FederationStructureTr` that are executed sequentially, in that order.

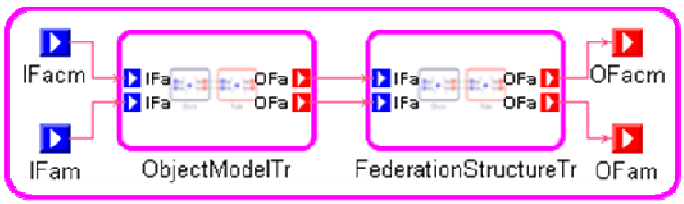


Figure B.3 DataModelTr block

B.2.1 Object Model Transformation

Object model transformation, whose top-level block is seen in Figure B.4, basically transforms the set of field artillery message structures that are communicated among domain actors during mission executions into HLA-OMT classes. The field artillery messages are represented as free format UML structures with information content provided

by the domain. On the other hand, HLA-OMT specification [39] puts forth a data type system. OMT specifies a core set of default data types of basic, simple, enumerated, and array types, that correspond to universally recognized types such as byte, integer, float, boolean and string.

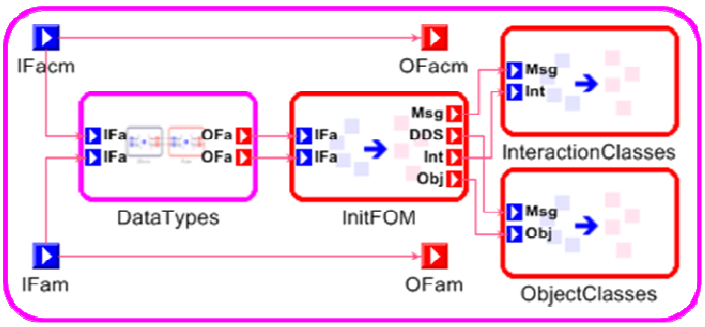


Figure B.4 ObjectModelTr block

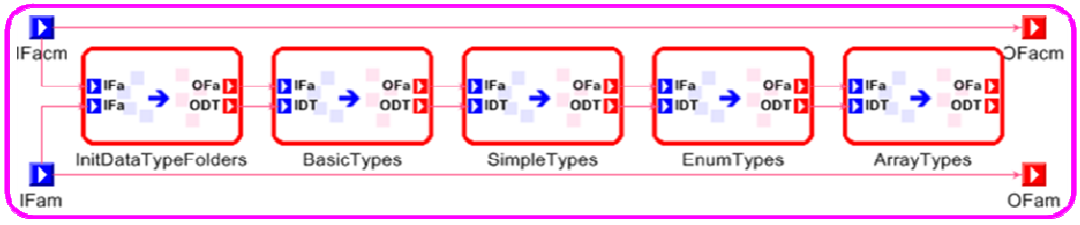


Figure B.5 DataTypes block

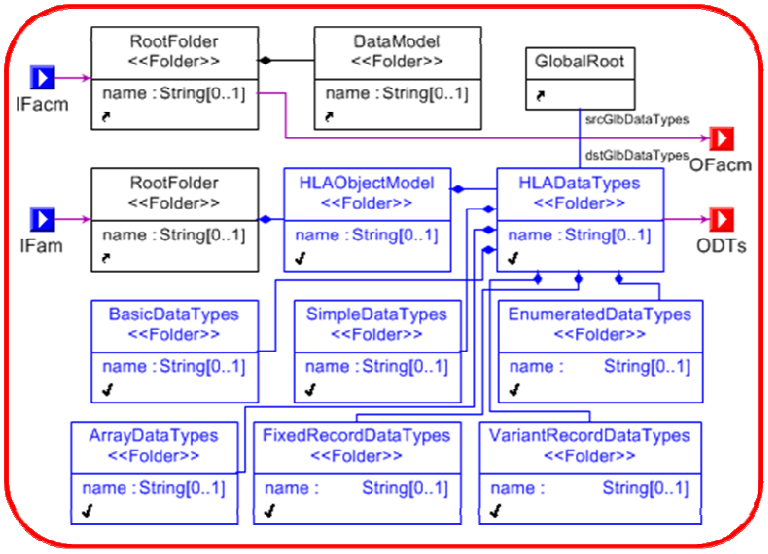


Figure B.6 InitDataTypeFolders rule

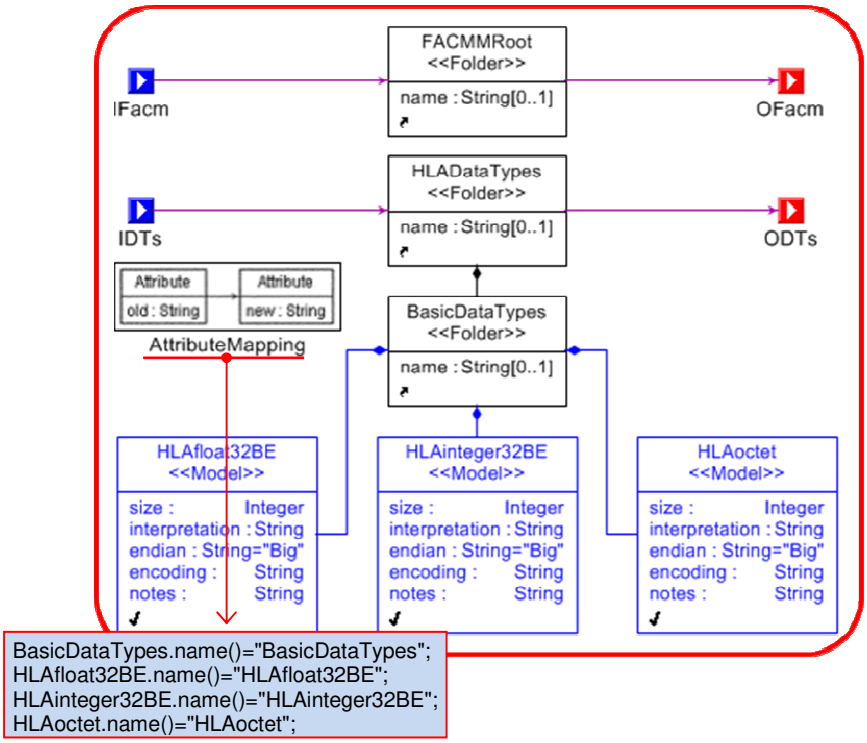


Figure B.7 BasicTypes rule

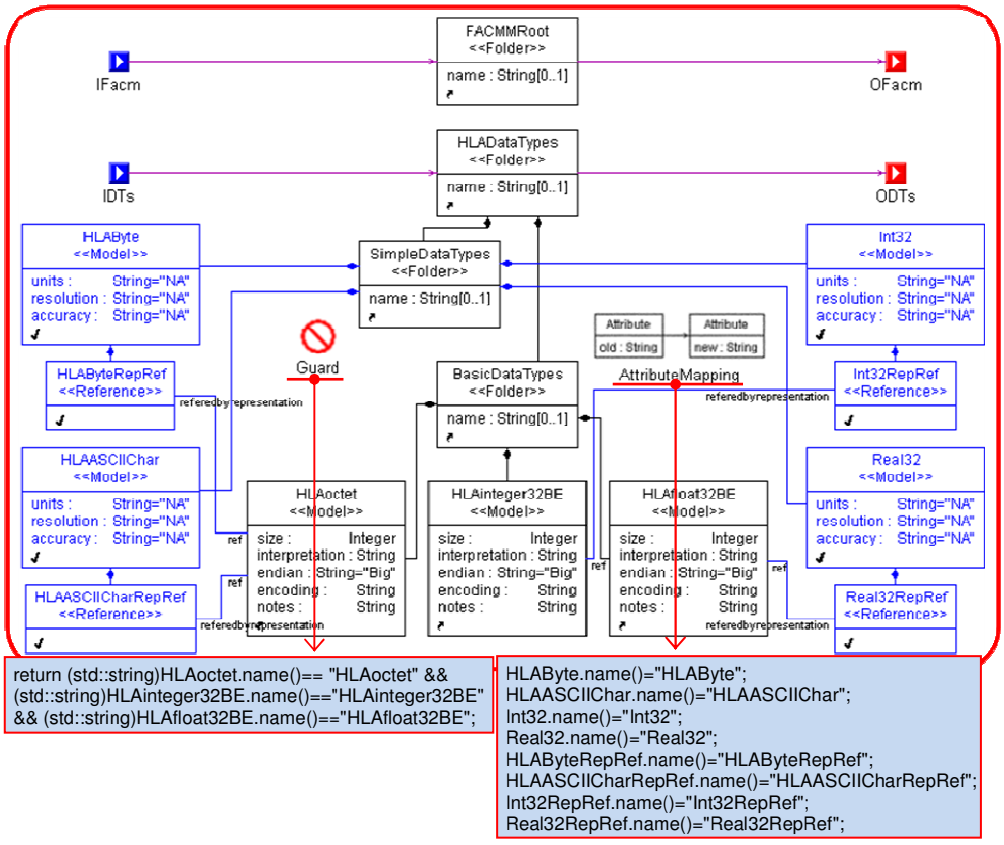


Figure B.8 SimpleTypes rule

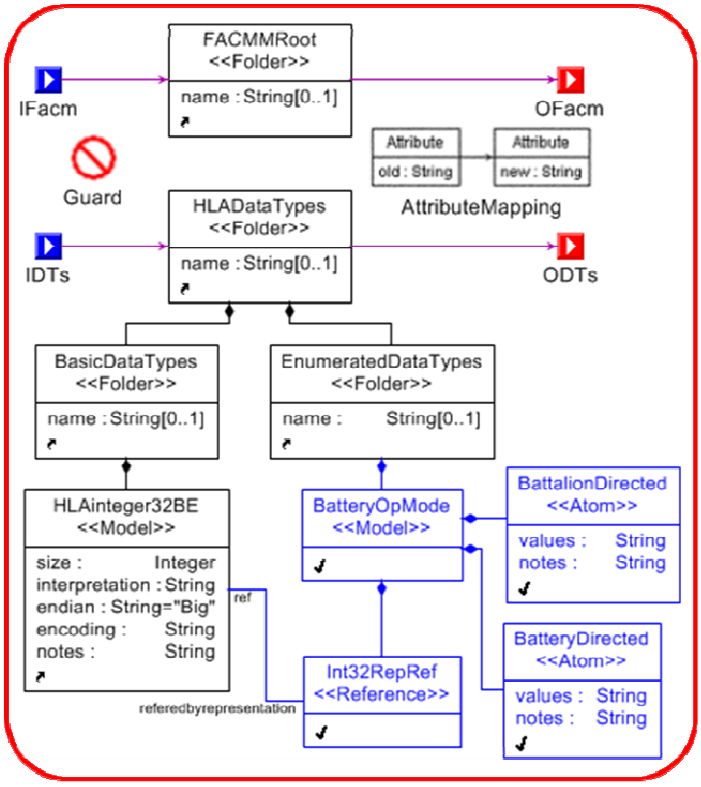


Figure B.9 EnumTypes rule

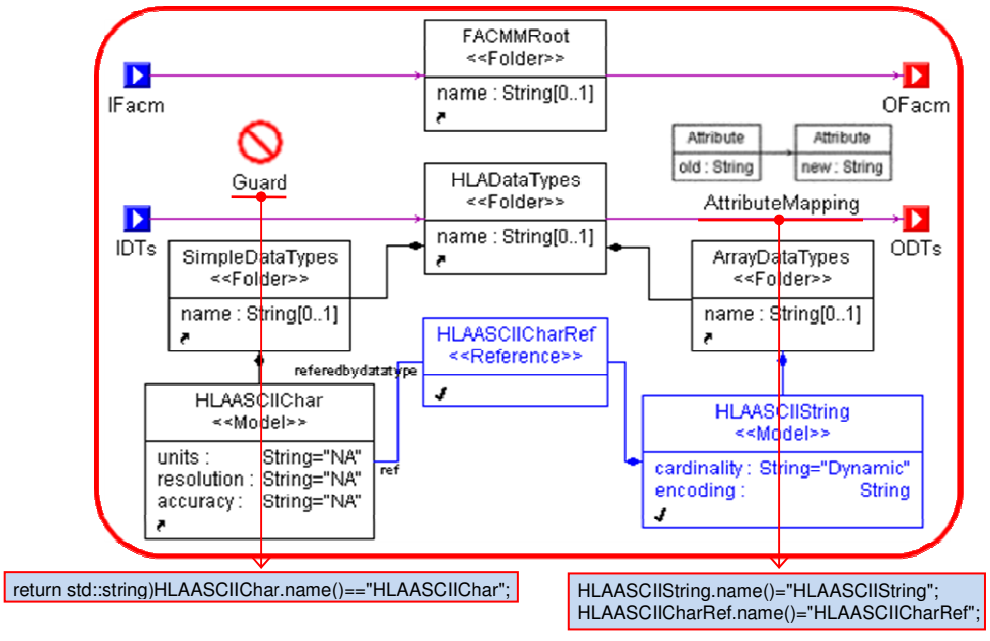


Figure B.10 ArrayTypes rule

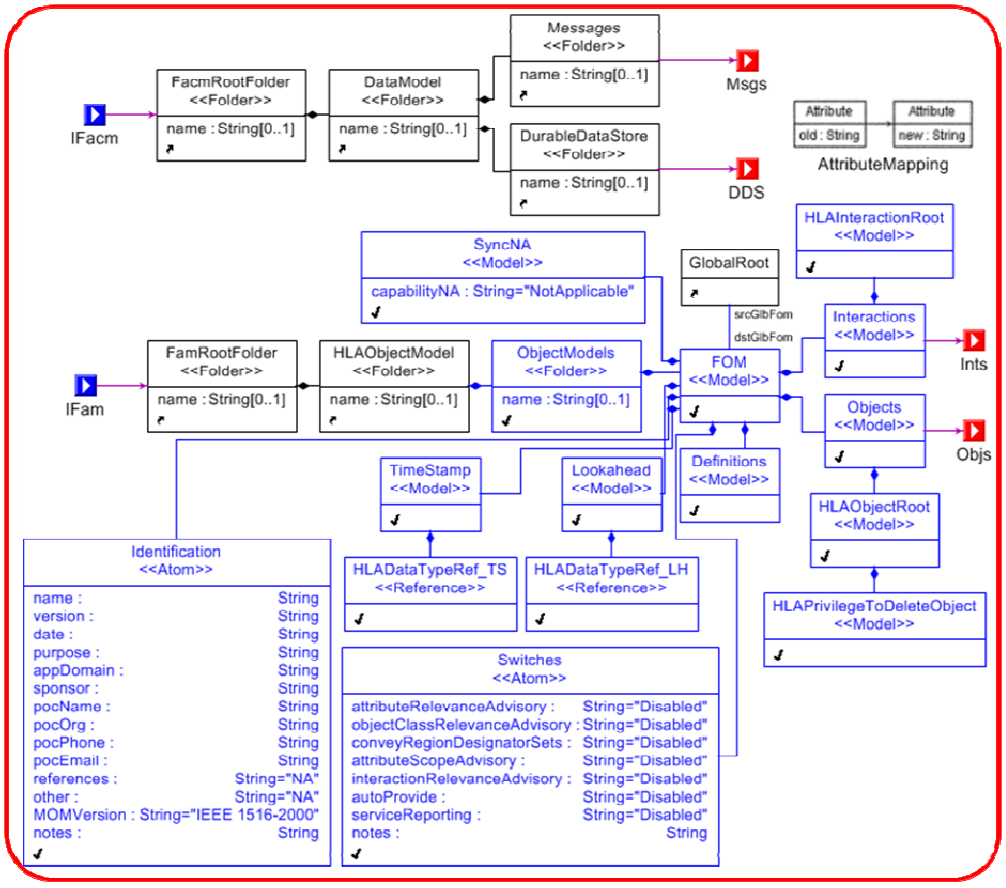
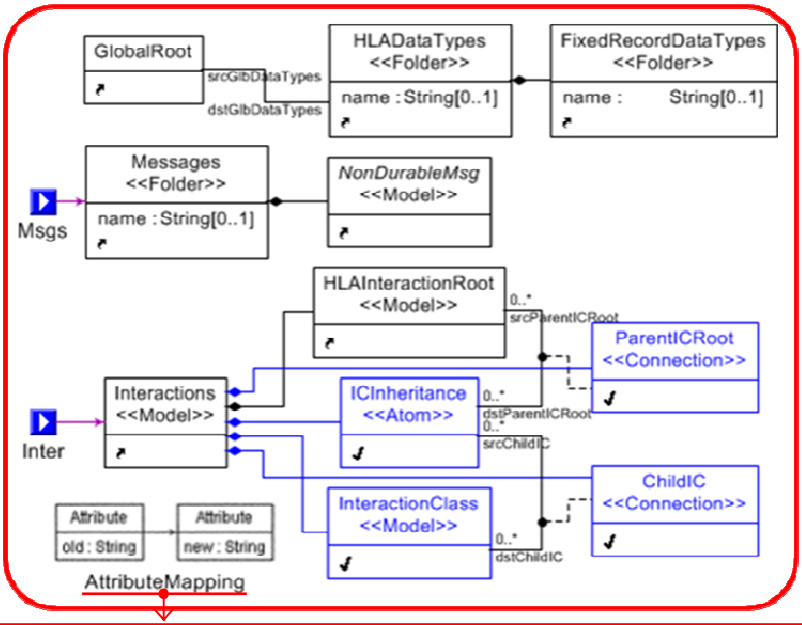


Figure B.11 InitFOM rule




```
InteractionClass.name()=(std::string)NonDurableMsg.name()+"IC";
//ModelTransUtils class is in FADM2HOM_Utils.cpp, which is in UserCodeLib, as declared in Configurations/CodeLibrary
ModelTransUtils::TransformNonDurableMsg2InteractionCls_Hybrid(NonDurableMsg, InteractionClass, FixedRecordDataTypes);
```

Figure B.12 InteractionClasses rule

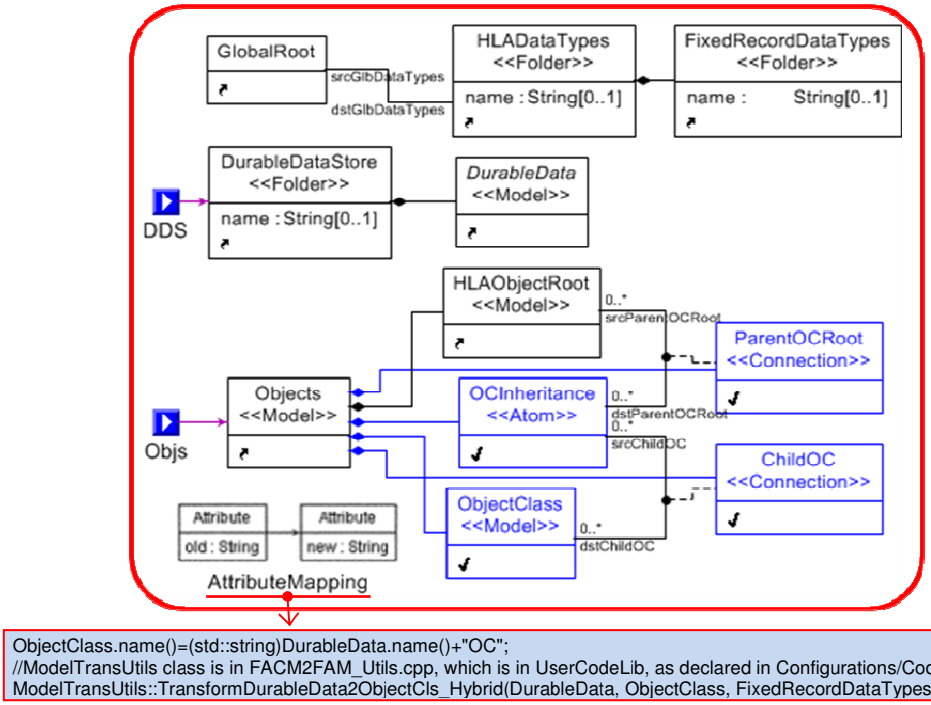


Figure B.13 ObjectClasses rule

B.2.2 Federation Structure Transformation

The federation structure transformation concludes the data model transformation part. It instantiates the single federation object together with a reference to the FOM that was previously created. It also maps every field artillery `Actor` and `Net` to a corresponding HLA federate along with a reference to an associated SOM. In this thesis, SOMs per federate are left as stubs and not developed any further.

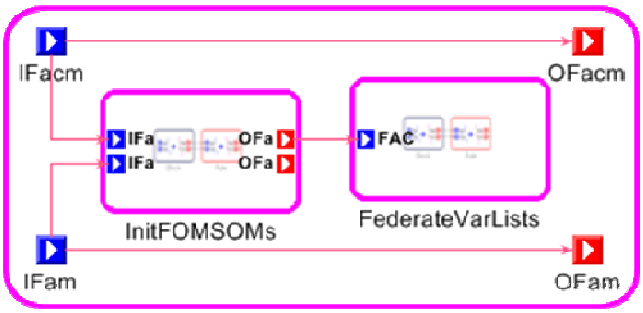


Figure B.14 FederationStructureTr block

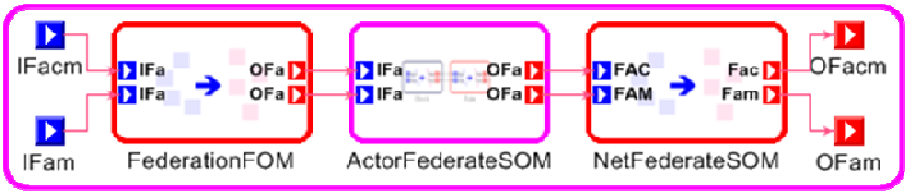


Figure B.15 InitFOMSOMs block

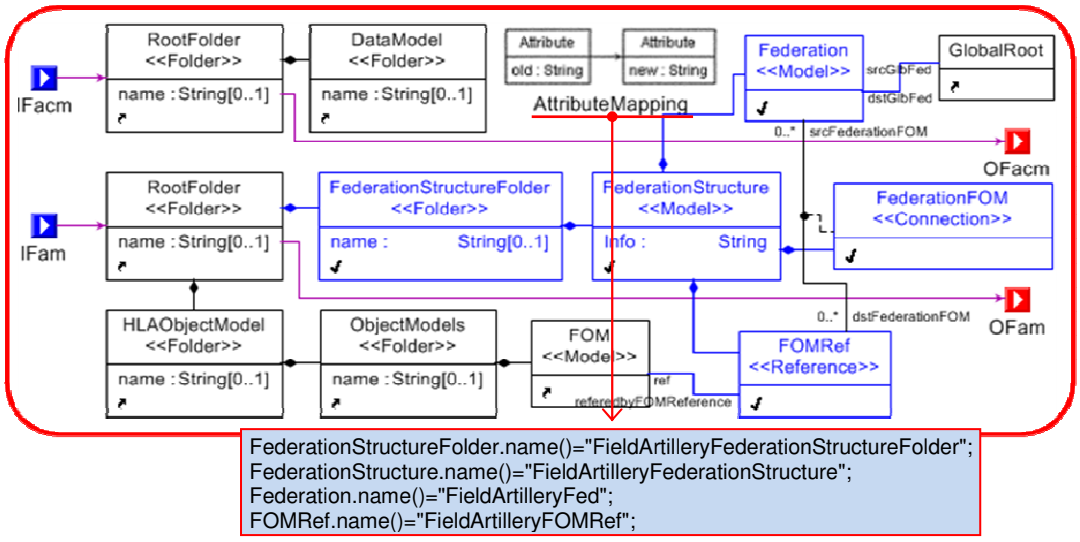


Figure B.16 FederationFOM rule

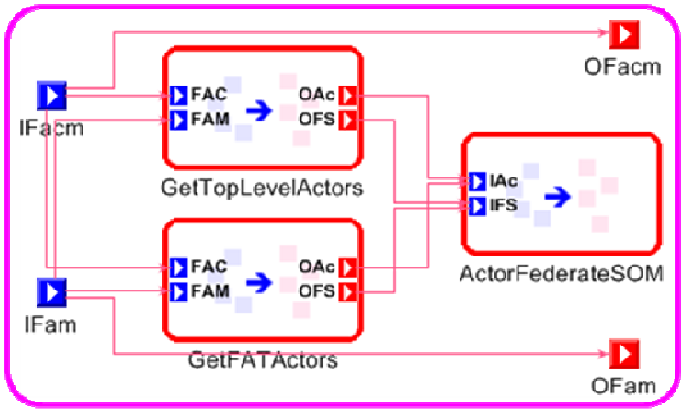


Figure B.17 ActorFederateSOM block

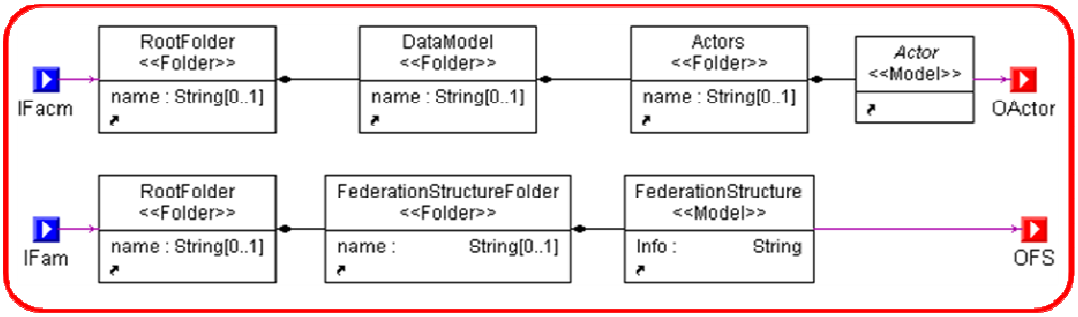


Figure B.18 GetTopLevelActors rule

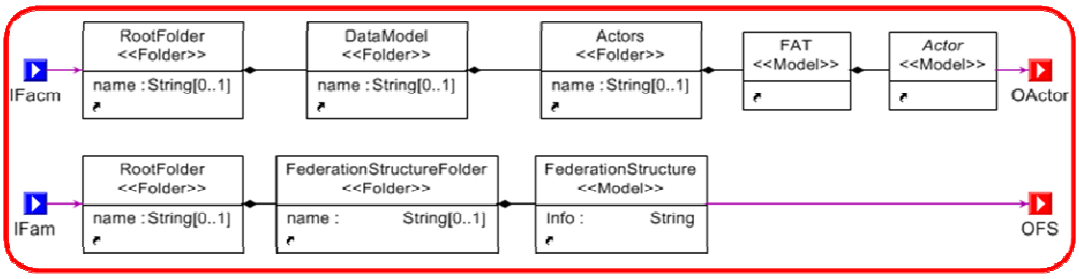
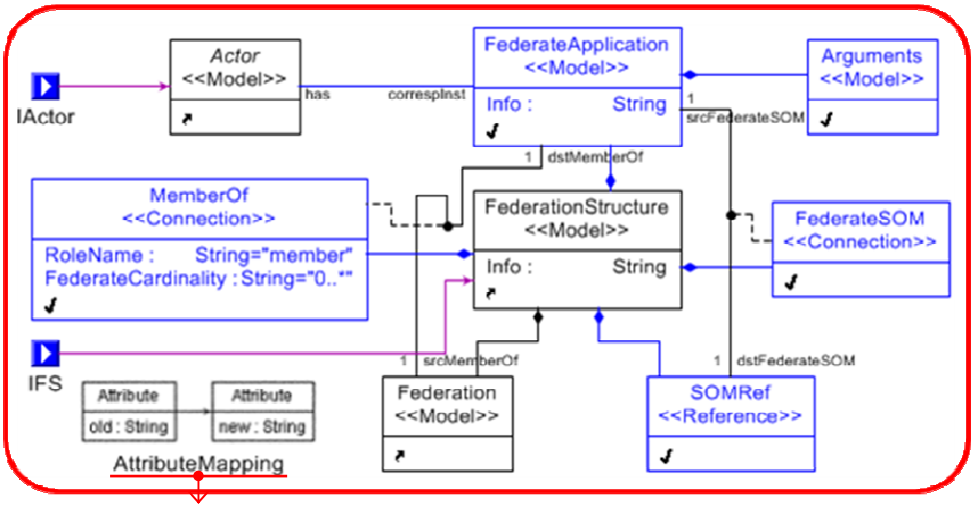


Figure B.19 GetFATActors rule




```
string temp; Actor.GetStrValue("name", temp); temp+="Fd";
FederateApplication.SetStrValue("name", temp); Actor.GetStrValue("name", temp);
temp+="SOMRef"; SOMRef.SetStrValue("name", temp);
Arguments.name()="Variables";
```

Figure B.20 ActorFederateSOM rule

B.2.3 Initializing Variable Lists Per Federate

Every HLA federate has a variable list by definition [39]. The rules in this section create the variable lists of the set of federates that correspond to the 8 actors and nets defined by ACMM. The top-level variable list creation block is illustrated in Figure B.21. Every federate has self specific variables of different data types. This makes the variable definition by transformation rules a tedious and frustrating process. The generation of these rules might be handy by employing higher order transformations with a text-based variable configuration per federate. Luckily, the variable lists of the federates are not directly used by the subsequent transformation rules and the code generator. Thus we have only created and filled-in the variable list for the `BatteryFDC` federate for the sake of not skipping an HLA defined component, and left the others as stubs.

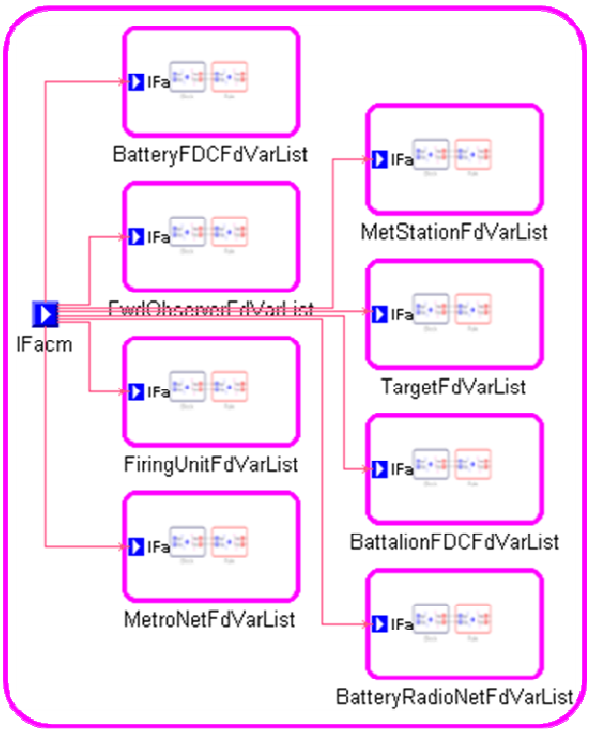


Figure B.21 FederateVarLists block

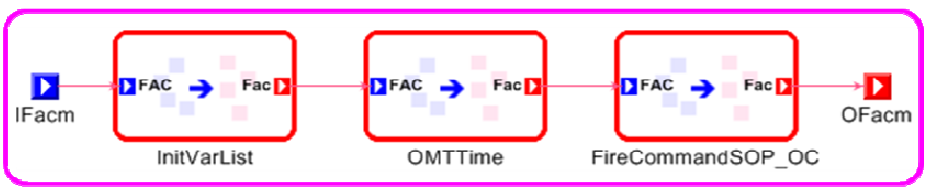


Figure B.22 BatteryFDCFdVarList block

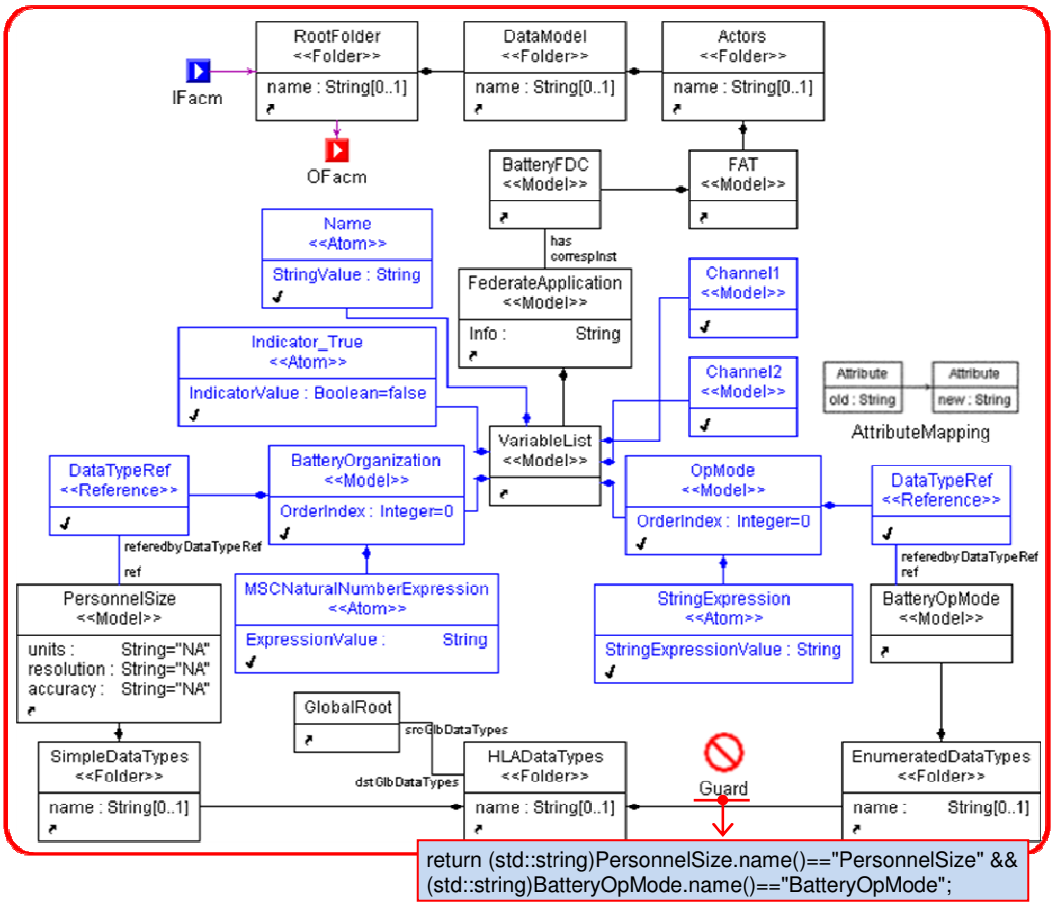


Figure B.23 InitVarList rule

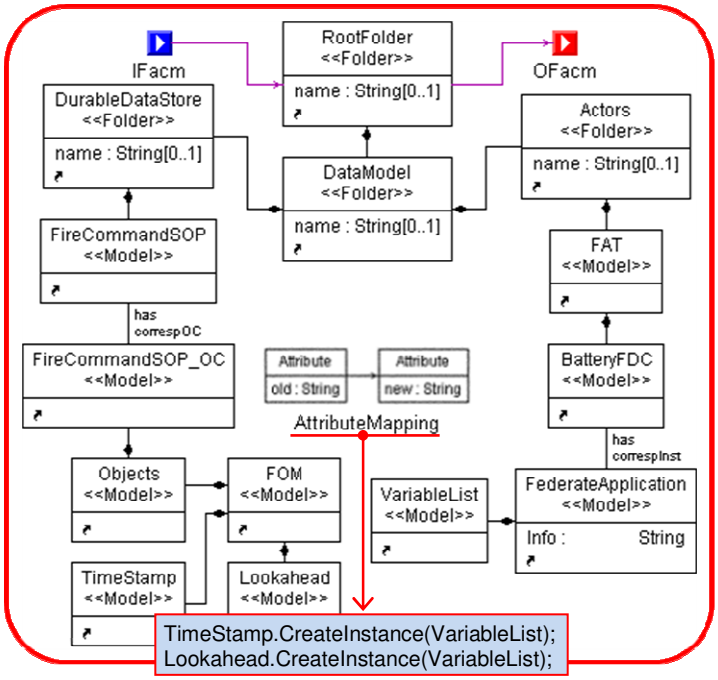


Figure B.24 OMTTime rule

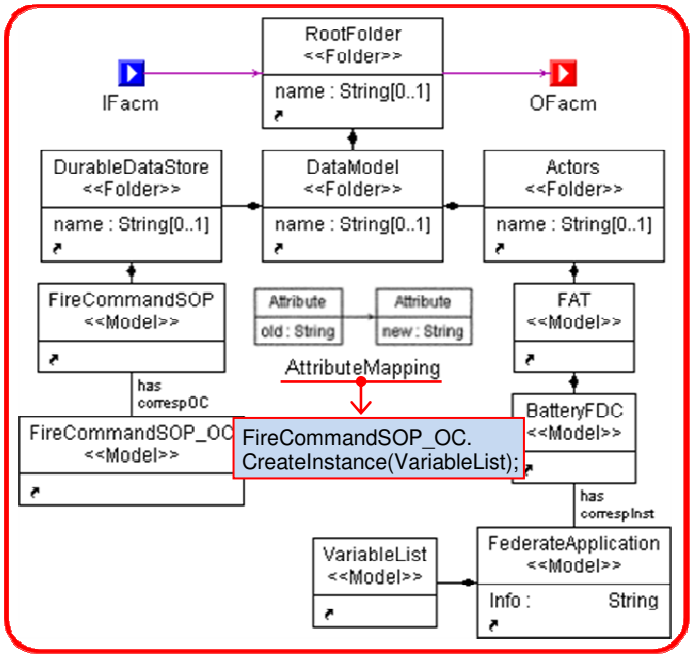


Figure B.25 FireCommandSOP_OC rule

B.3 Behavioral Model Transformation

Behavioral model transformation is the bigger and more challenging part of the overall ACM2FAM transformation. It uses the resulting objects of the data model transformation as the instances and message parameters in LSCs that are being produced. The main block of the behavioral model transformation, `BehavioralModelTr`, is shown in Figure B.26.

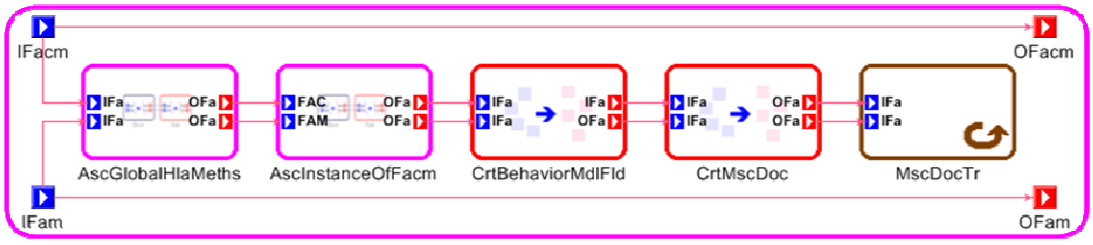


Figure B.26 BehavioralModelTr block

The `AscGlobalHlaMeths` block, as expounded in Figure B.27, gets the method library of FAM that contains predefined HLA methods for federation, declaration, object, ownership and time management. The block contains rules that take copies of all the

methods used in the transformation and associate them with the global HLA methods element so that they are readily accessible by the LSC transformation rules. These methods are meant to function as templates; hence their method parameters are left empty. Their copies in the LSCs are assigned parameters with appropriate HLA class instances during the transformation.

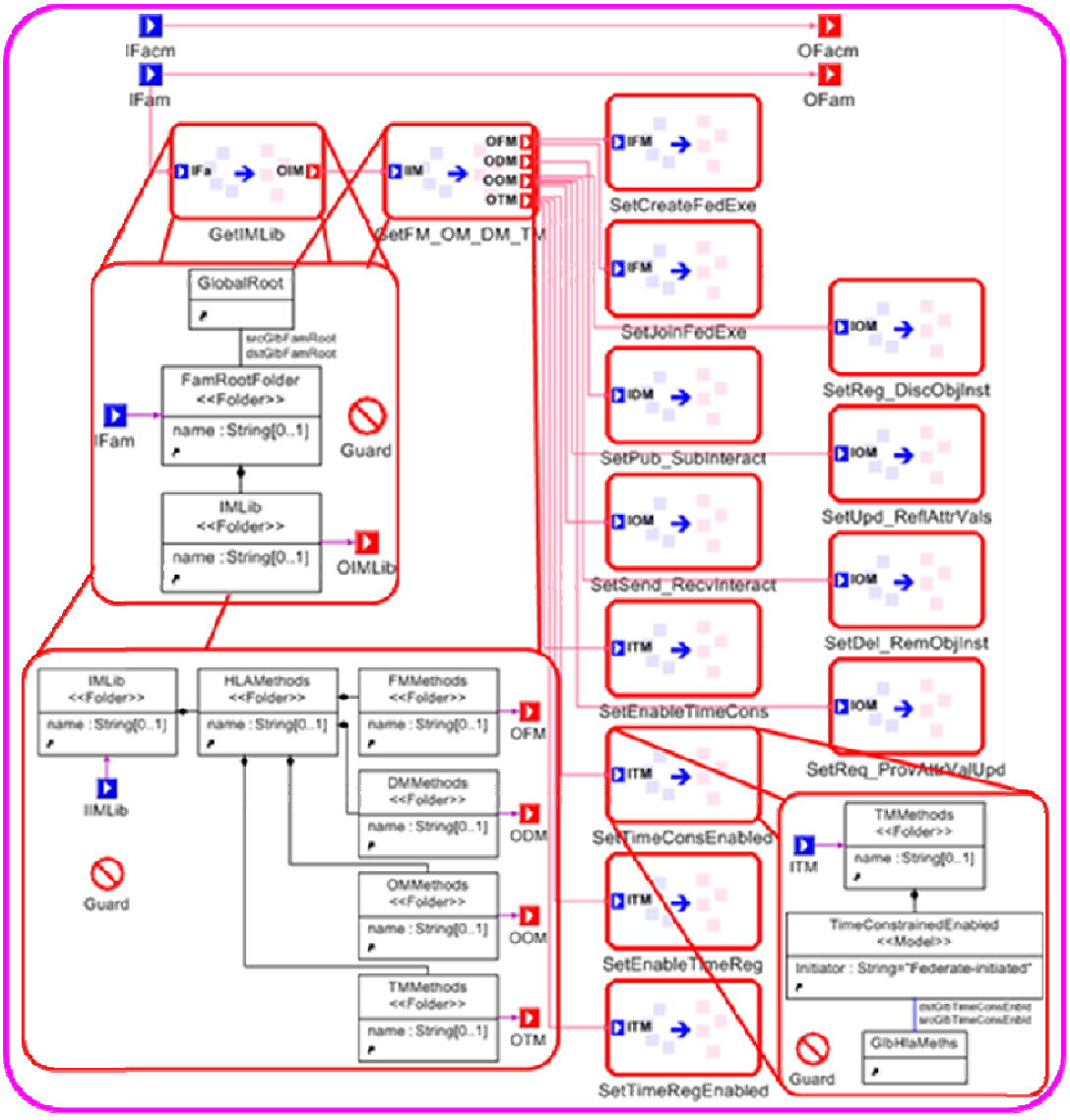


Figure B.27 AscGlobalHlaMeths block

The `AscInstanceOfFacm` block and its subordinate blocks and rules are displayed in Figure B.28, Figure B.29, Figure B.30, Figure B.31 and Figure B.32. The block basically creates `is-InstanceOf` associations between the instances that stand for the same actor

element in ACM. An actor instance in the MSC head of an MSC is an instance of the same type of instance in the MSC document head, which in turn is an instance of the canonical actor instance in the data model's `Actors` folder. This chain of associations establishes traceability between the behavior and data sub-models of ACM and provides convenience in subsequent rules. A similar scheme is also applied progressively on the FAM side as the transformation rules construct the model.

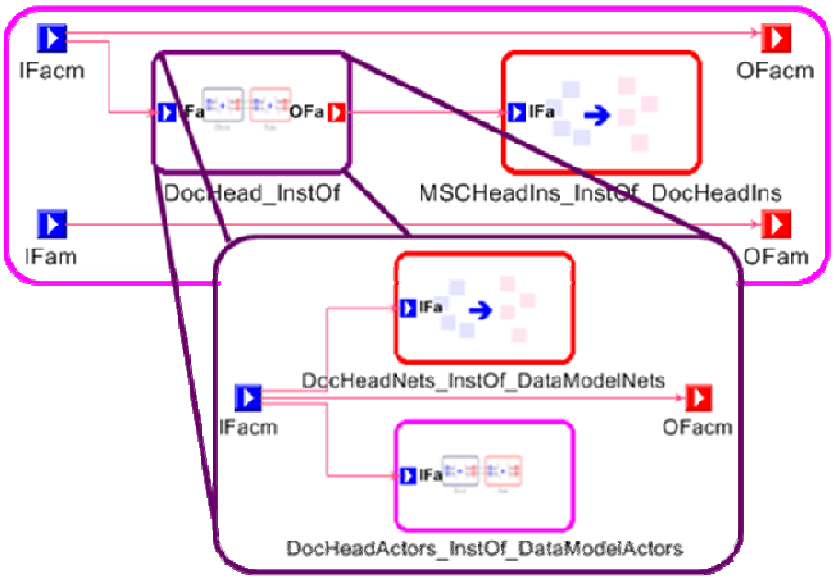


Figure B.28 AscInstanceOfFacm block and DocHead_InstOf for-block

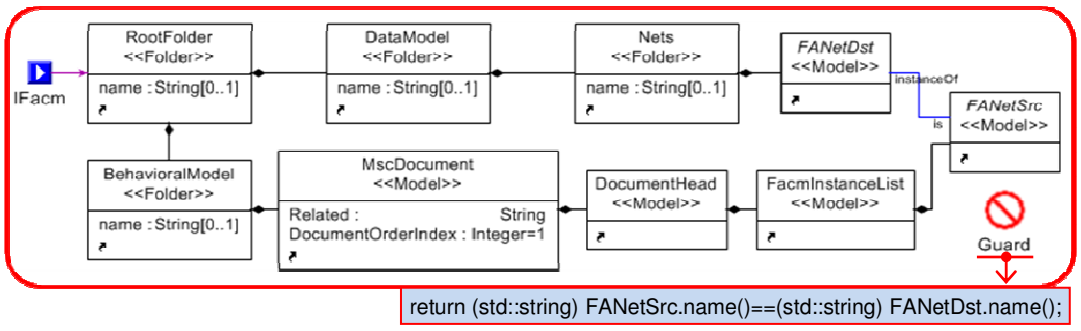


Figure B.29 DocHeadNets_InstOf_DataModelNets rule

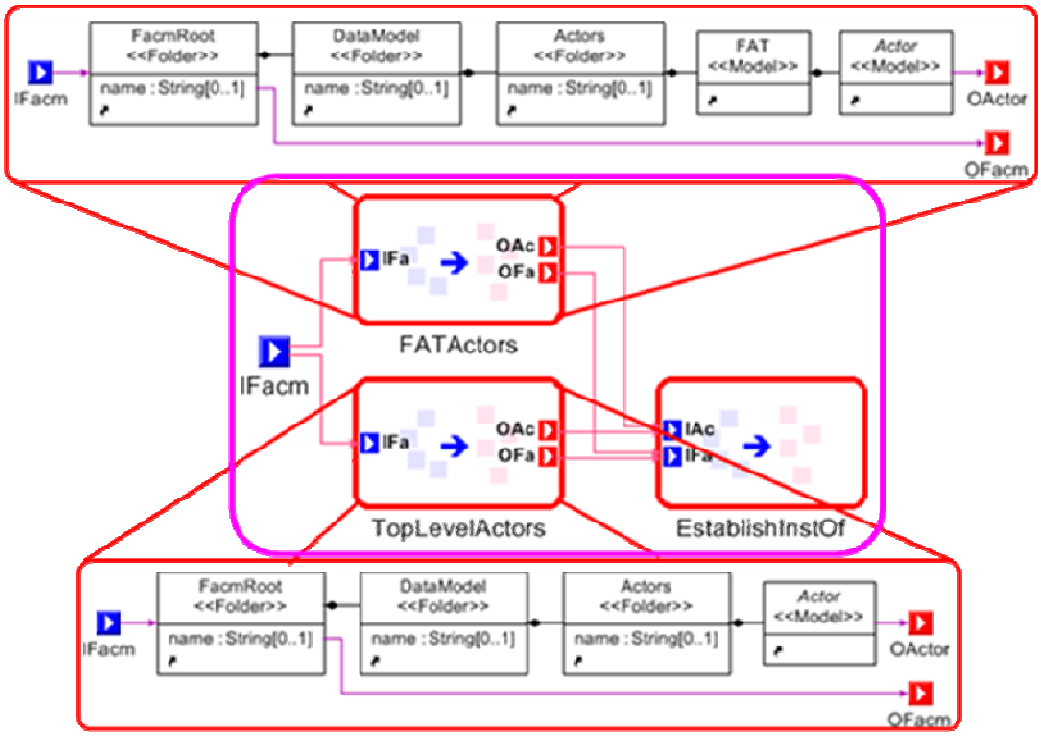


Figure B.30 DocHeadActors_InstOf_DataModelActors block

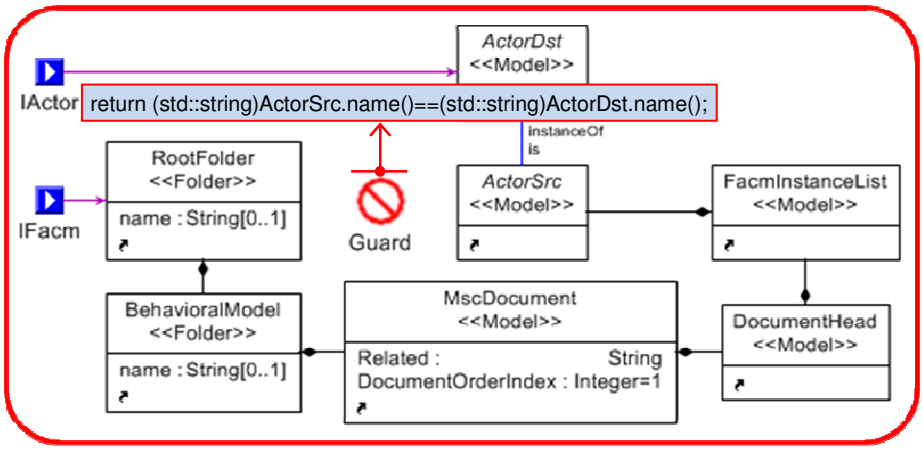


Figure B.31 EstablishInstOf rule

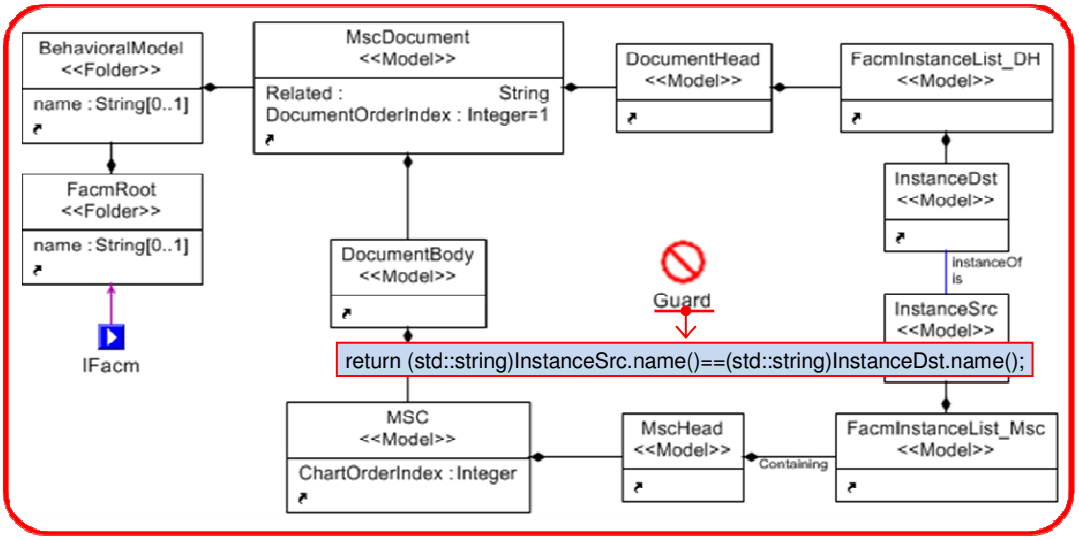


Figure B.32 MSCHeadIns_InstOf_DocHeadIns rule

The `CrtBehaviorMdlFld` and `CrtMscDoc` rules are triggered one after another for simply creating a FAM behavioral model folder and an MSC document underneath it, provided that their corresponding counterparts are matched in the ACM. A `has-correspMscDoc` association is established between the ACM and FAM MSC documents, since there can be more than one MSC document in a source model and in such a case this association is necessary for keeping track of MSC references in different documents and during instance decomposition.

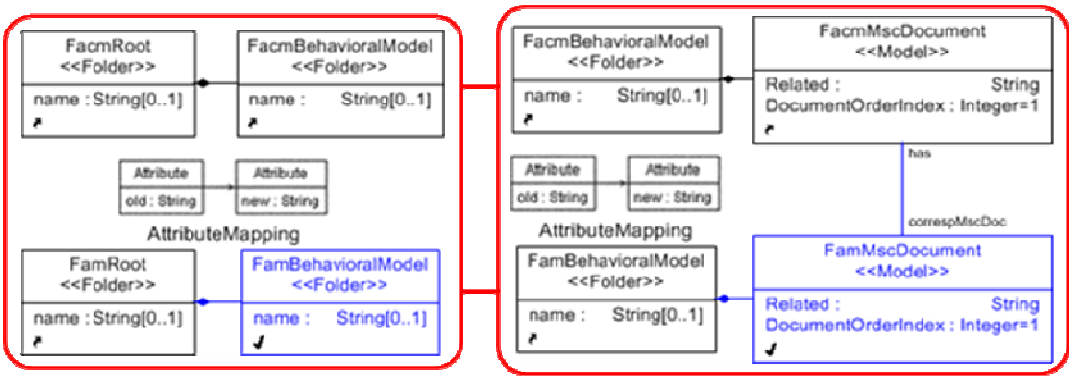


Figure B.33 CrtBehaviorMdlFld and CrtMscDoc rules

B.3.1 MSC Document Transformation

The `MSCDocTr` block is shown in Figure B.34. It consists of three sub-blocks, namely, `DocumentHeadTr`, `DocumentBodyTr` and `AscReferences`, executed in that order. All of the blocks and rules within `MSCDocTr` are defined so as to traverse the structure delineated by the MSC metamodel to create a FAM MSC document from an ACM MSC document.

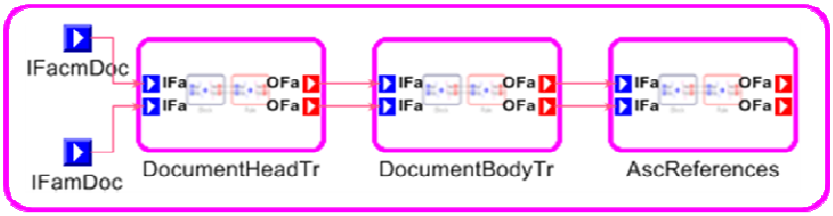


Figure B.34 MscDocTr block

The `DocumentHeadTr` block handles the data definition, message declaration, instance declaration and timer declaration parts of the document head of the FAM being constructed. Note also that data definition and message declaration are only addressed as stubs since the content related with these parts are practically provided by the data model.

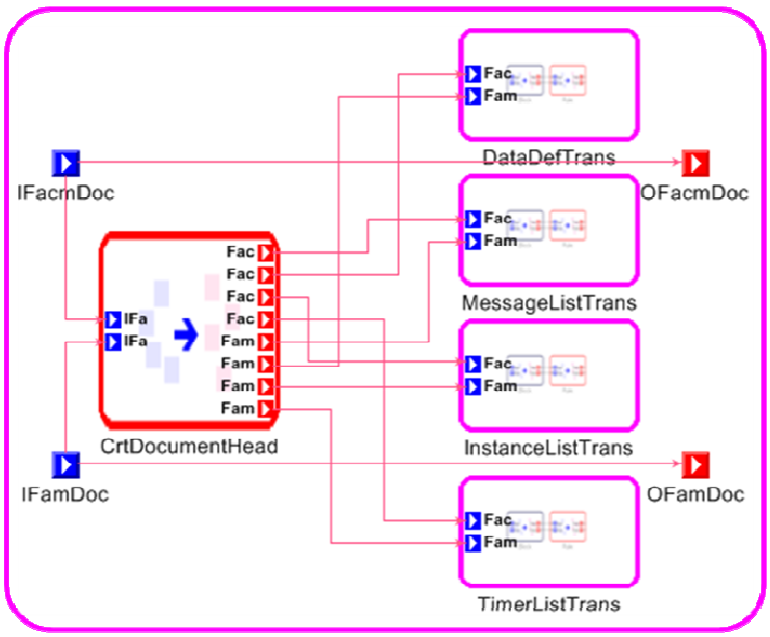


Figure B.35 DocumentHeadTr block

The instance declaration part of the MSC document head transformation is also one of the key steps in the overall behavioral model transformation. Its role is basically to create federate objects and a federation object derived from the corresponding counterparts found in the federation structure portion of the FAM data model.

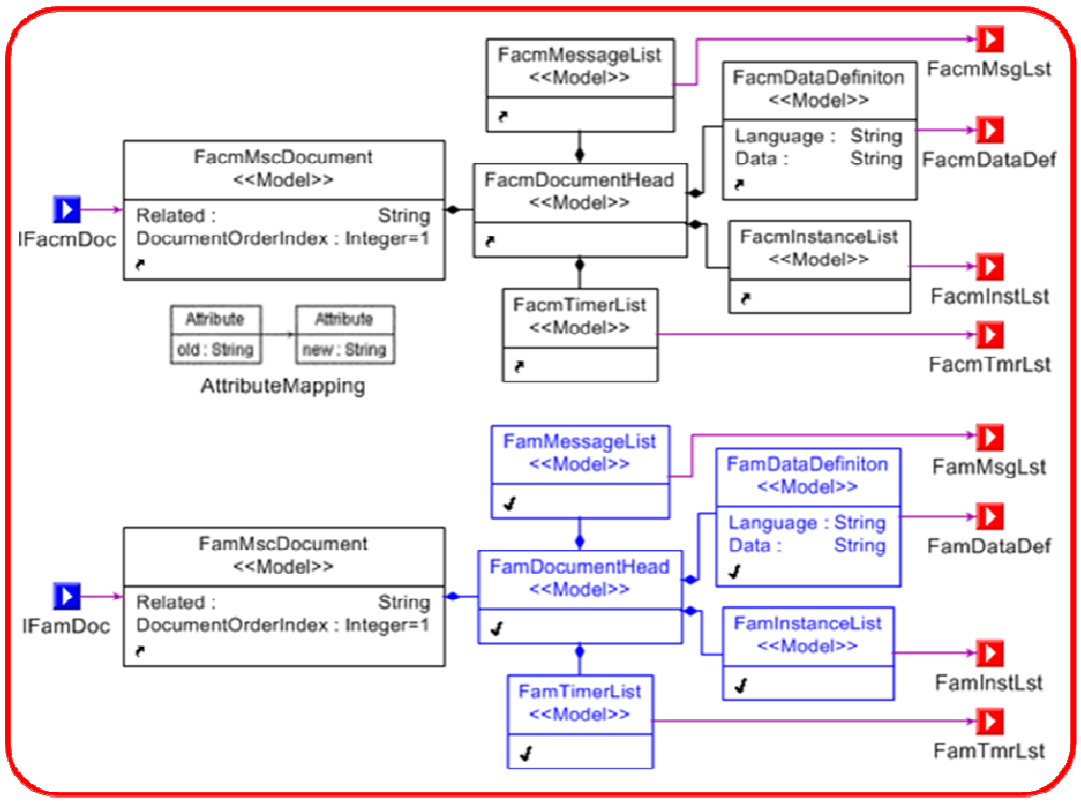


Figure B.36 CrtDocumentHead rule

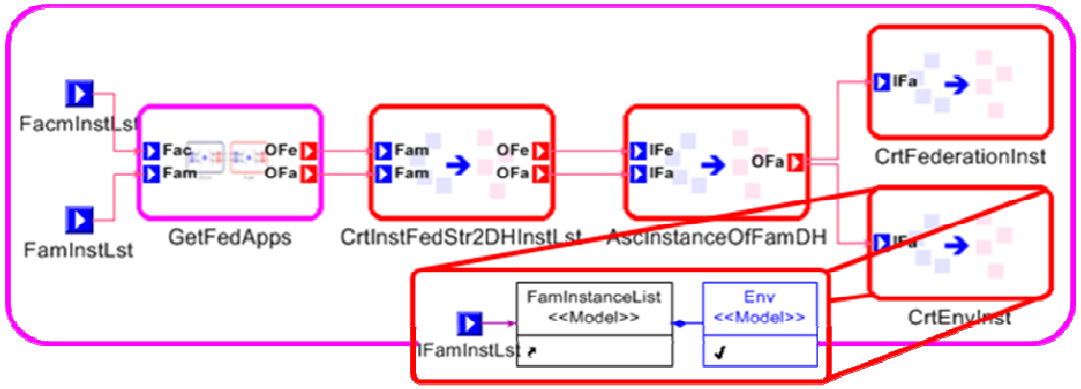


Figure B.37 InstanceListTrans block

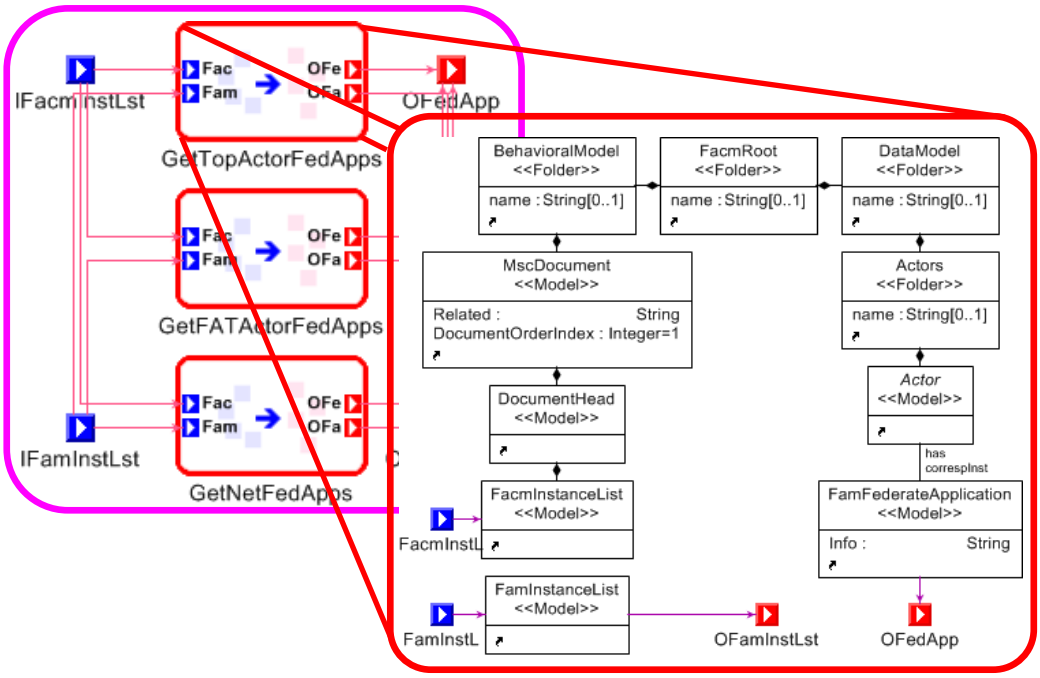


Figure B.38 GetFedApps block and GetTopActorFedApps rule

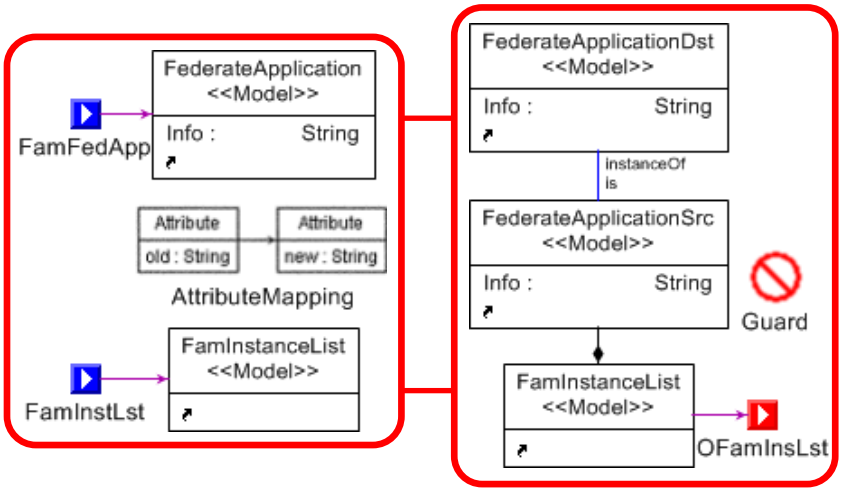


Figure B.39 CrtInstFedStr2DHInstLst and AscInstanceOfFamDH rules

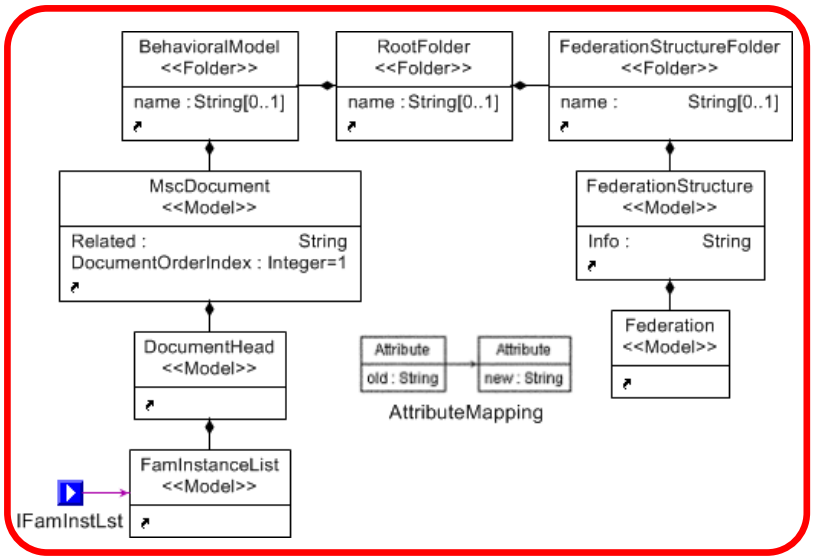


Figure B.40 CrtFederationInst rule

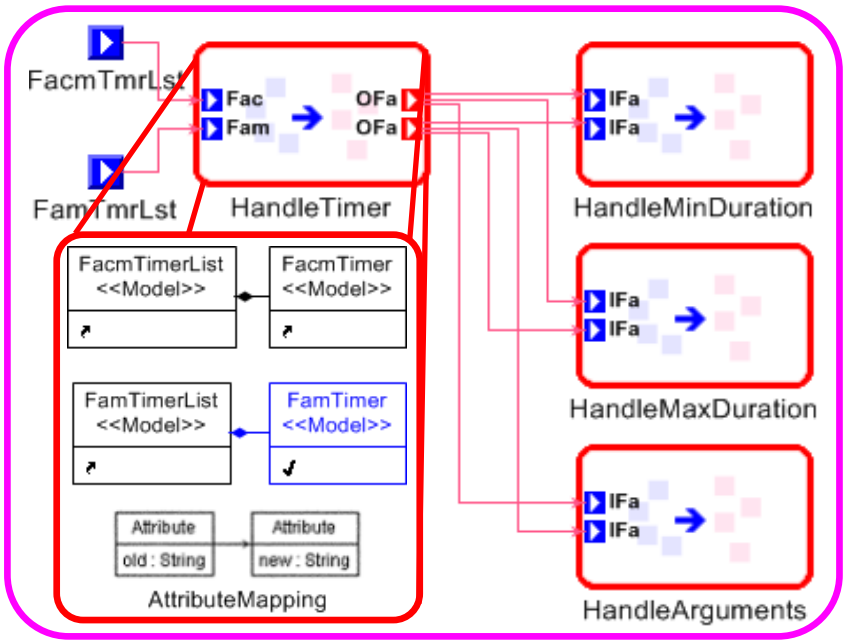


Figure B.41 TimerListTrans and HandleTimer rule

The MSC document body transformation, whose top-level block is illustrated in Figure B.42, essentially boils down to MSC transformation. In order to start the process, an empty FAM MSC is created per matched ACM MSC in the given document body. The cross-domain `has-correspMSC` association is established for keeping track of the paired MSCs

in subsequent rules. The attribute mapping code copies the chart order index in addition to the name and screen position properties of the ACM MSC to the FAM MSC. The chart order index, although not an artifact of the MSC metamodel, is a crucial annotation that facilitates model interpreters and particularly the code generator, by providing the execution/interpretation order of the MSCs at run-time. Similarly, for multiple documents in a model, the order of the documents may be specified by the document order index [12].

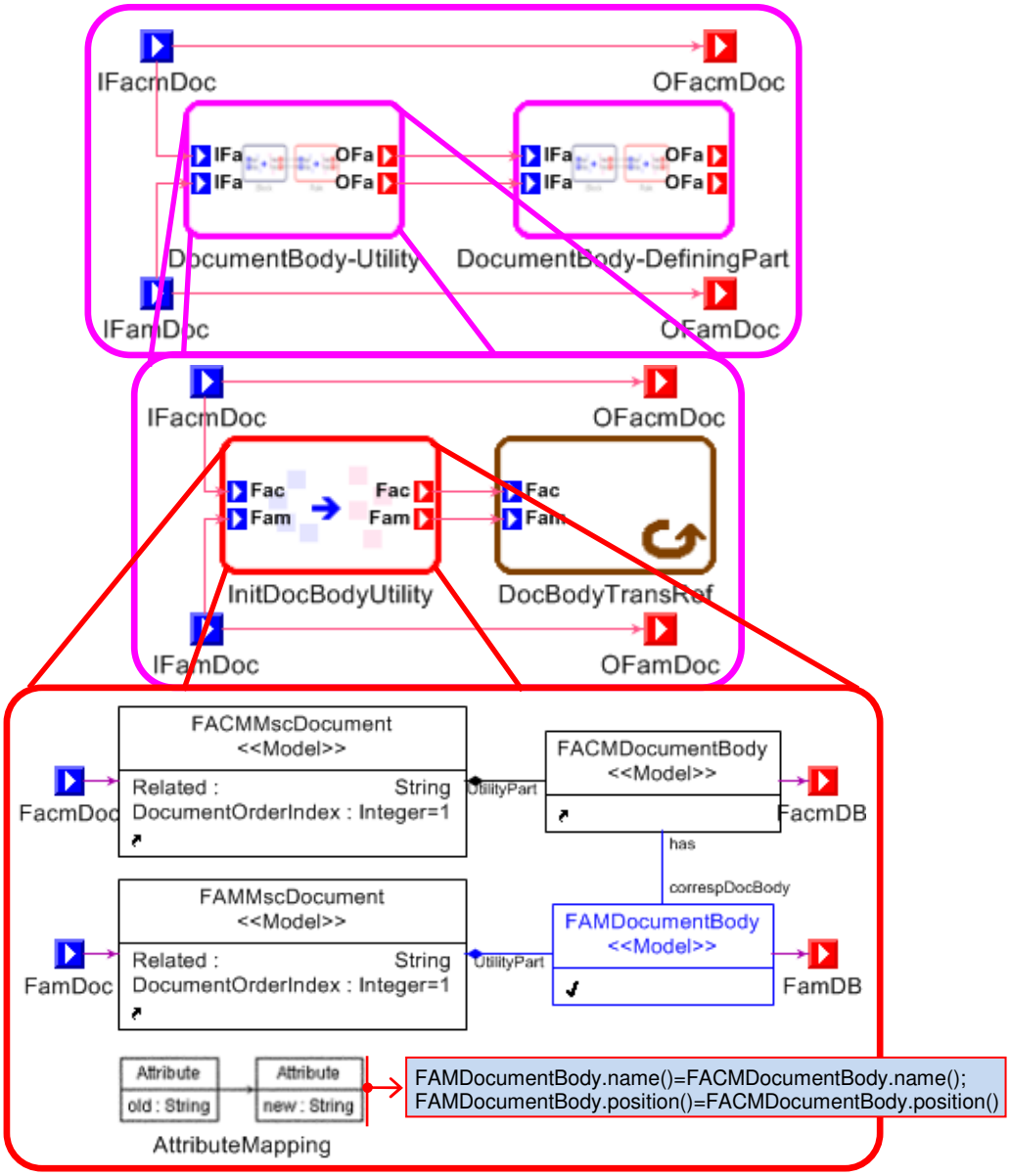


Figure B.42 DocumentBodyTr and DocumentBody-Utility blocks and InitDocBodyUtility rule

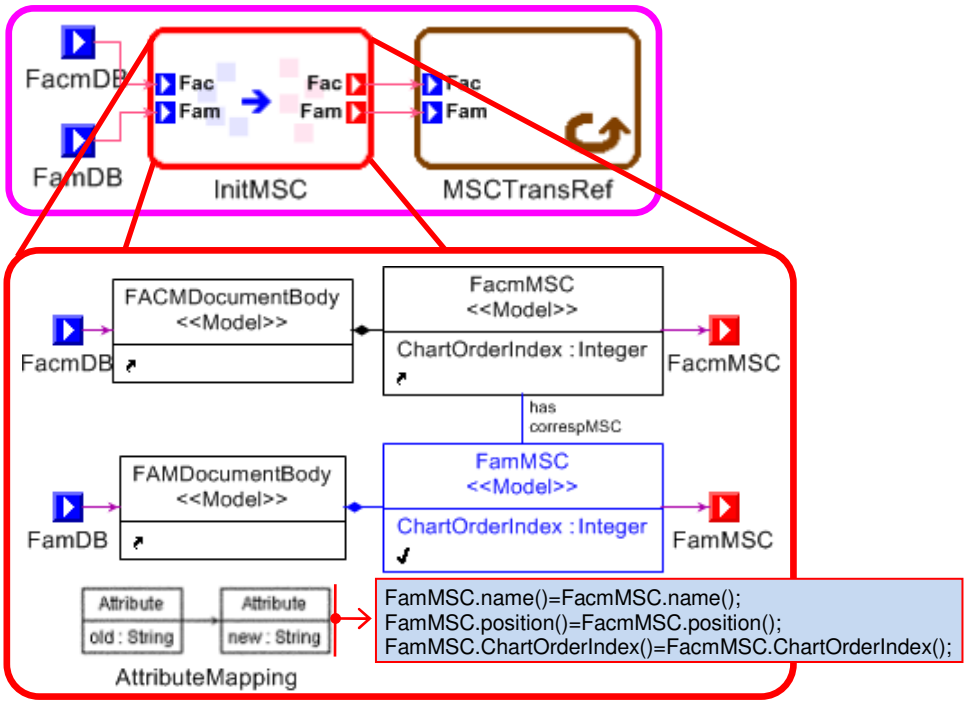


Figure B.43 DocBodyTrans block and InitMSC rule

B.3.2 MSC Transformation

MSC transformation is handled by the mainstream `MSCTrans` block, shown in Figure B.44. Its importance is due to its incorporation of LSC transformation, which virtually is the heart of behavioral model transformation. MSC transformation consists of three consecutive steps that handle MSC head and body transformation, and initialize the federation after the completion of the former two. MSC body transformation essentially boils down to LSC transformation after an empty LSC context is created. LSC transformation rules are further found in Section B.3.4.

The head part of an MSC is transformed in a four rule block. The head of an MSC houses the instances referenced in the MSC's body, besides other elements. The basic functionality of `MSCHeadTr` is to prepare the instances used in the FAM MSC, by looking at the instances found in the corresponding MSC. Other MSC head components such as offset, parameter set and its subcomponents are either provided explicitly inside the MSC body or considered irrelevant for the purposes of this work and hence, are not covered. The MSC head transformation also addresses instance decomposition.

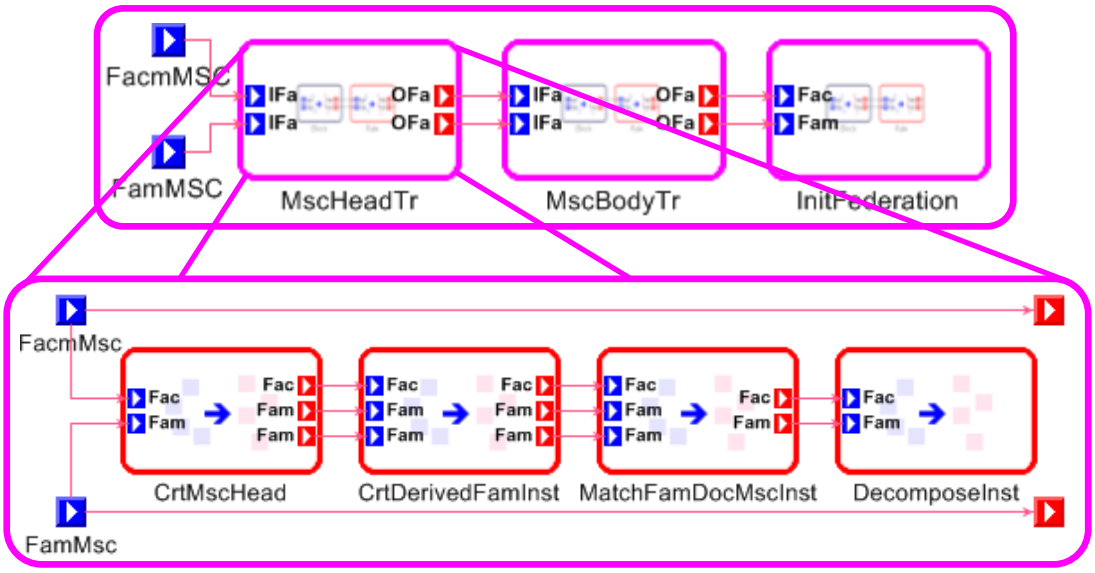


Figure B.44 MSCTrans and MscHeadTr blocks

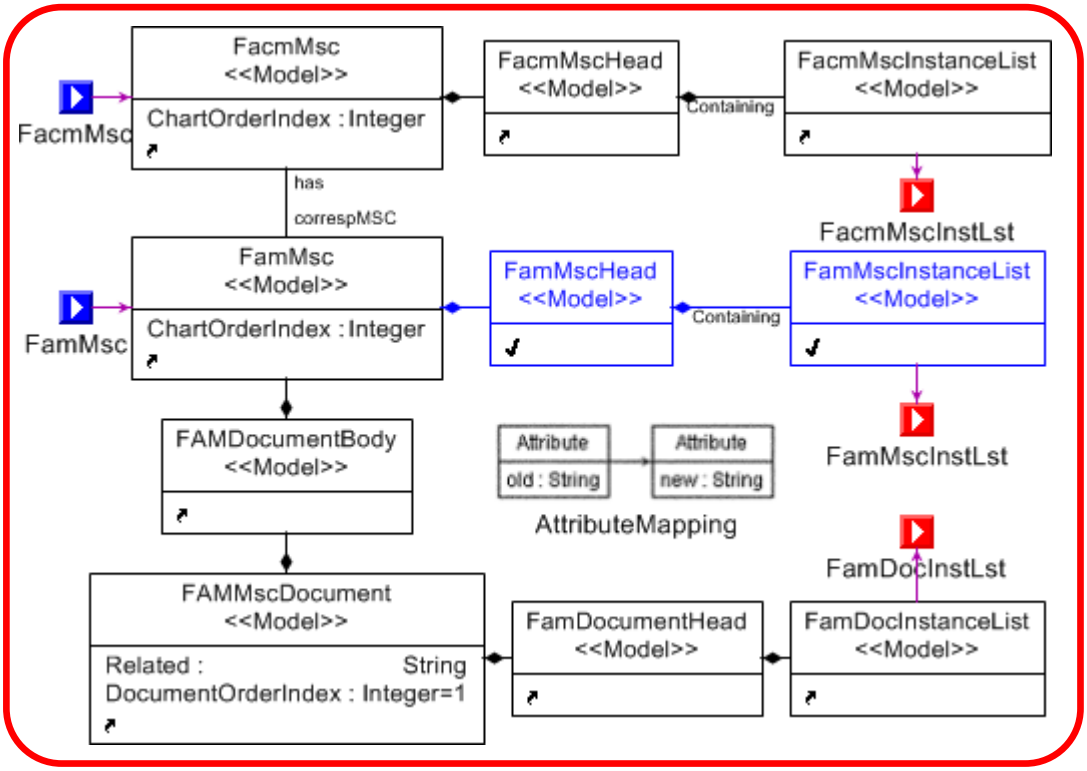


Figure B.45 CrtMscHead rule

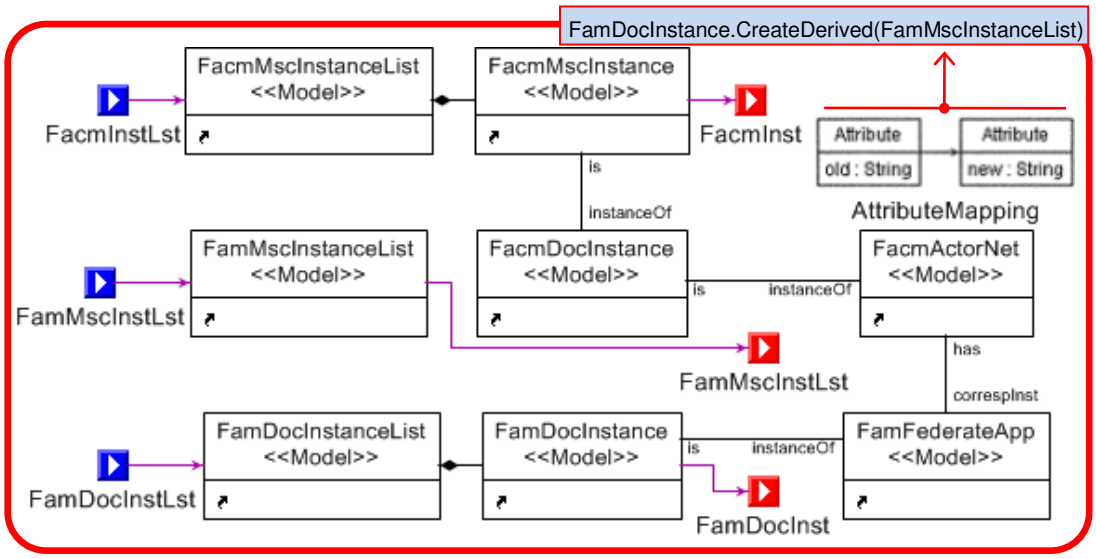


Figure B.46 CrtDerivedFamInst rule

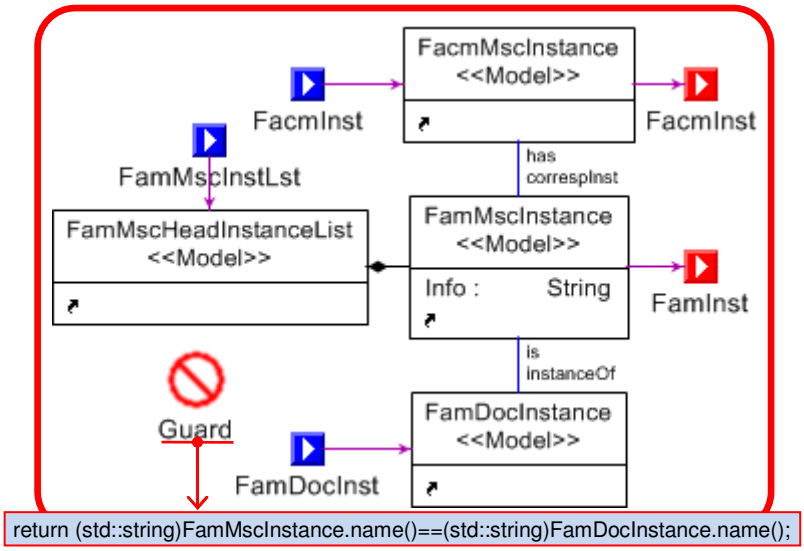


Figure B.47 MatchFamDocMscInst rule

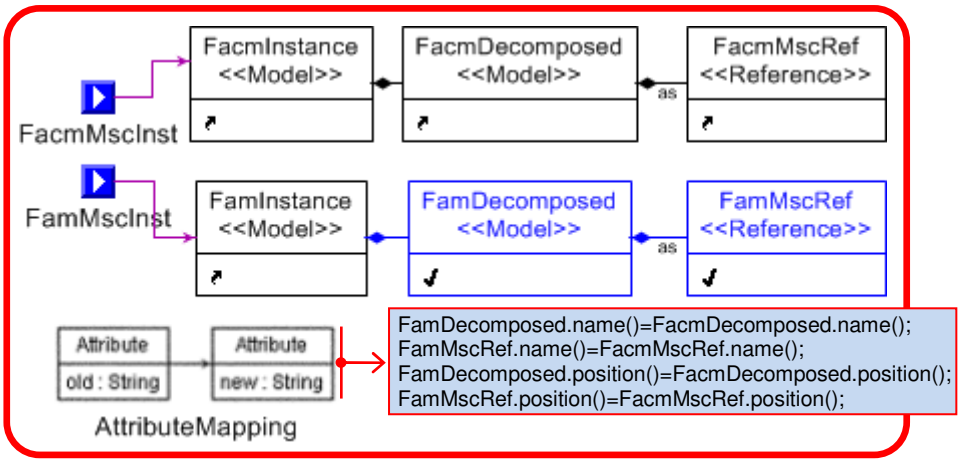


Figure B.48 DecomposeInst rule

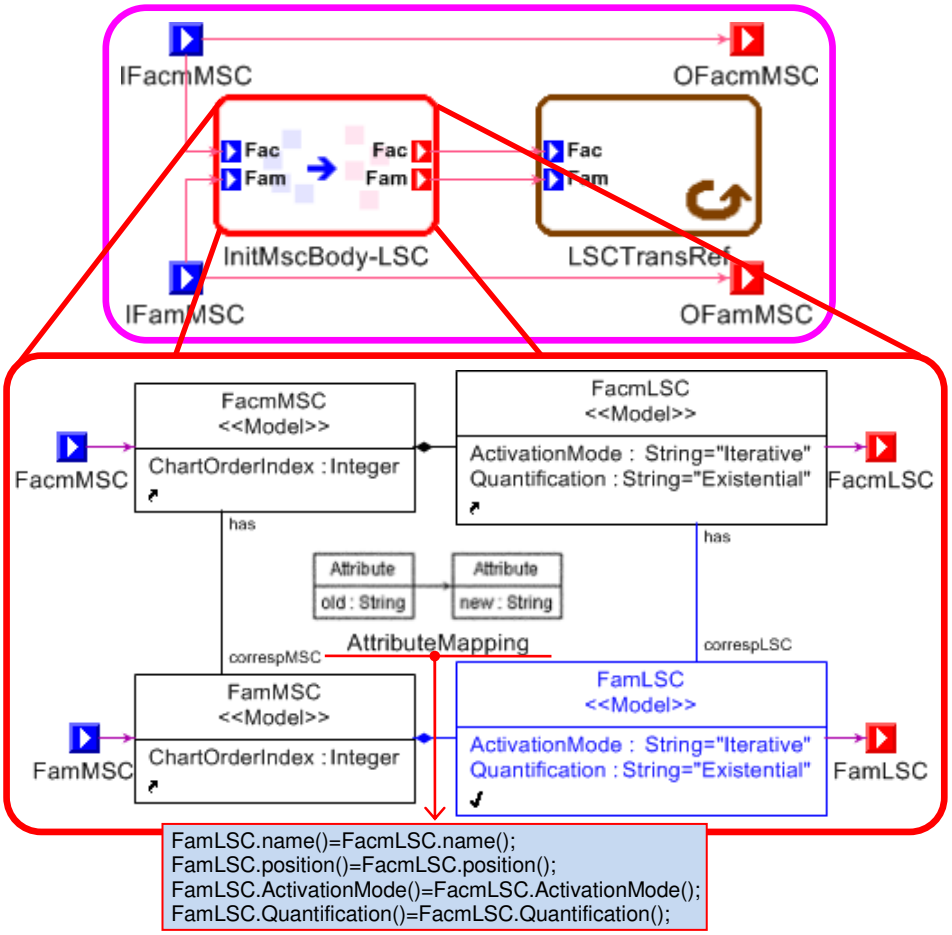


Figure B.49 MscBodyTr block and InitMscBody-LSC rule

B.3.3 Federation Initialization

Before moving into LSC transformation this section makes a fast forward to explain the federation initialization on the FAM side. The federation initialization is done after an MSC document is transformed head and body-wise. This indicates that it is a post processing step following the full transformation of all the LSCs in the document

The HLA federation initialization activities are done in the `InitFederation` block sketched in Figure B.50. This is a part of the behavioral model transformation indigenous to the FAM domain; that is, there are no associations in the transformation rules to ACM except for the identification of the instances involved. Due to the lack of such an input source, the information content flowing through the federation initialization part is directly embedded inside the transformation rule definitions.

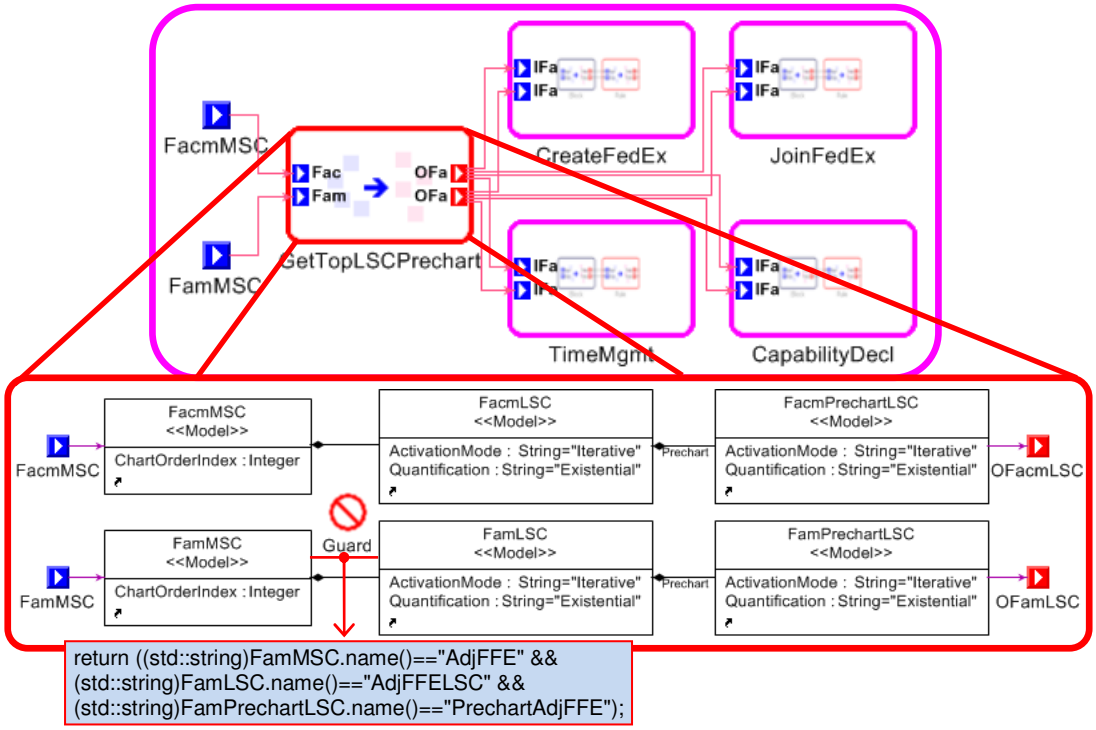


Figure B.50 InitFederation block and GetTopLSCPrechart rule

The `InitFederation` block handles four preliminary federation execution activities of creating a federation execution, joining federates to the federation execution, initializing time management and declaration management. The federation initialization events are gathered in a sub-chart which itself is placed inside the pre-chart of the top-level FAM LSC. This way, federation initialization is guaranteed to be performed right at the

beginning. The subchart is made temperature-wise "*hot*"; hence, mandatory to execute [14]. Since there is no clue from the ACM regarding the execution order of the chart, it is read from a look up table in the user code library; thus, effectively delegated to external configuration.

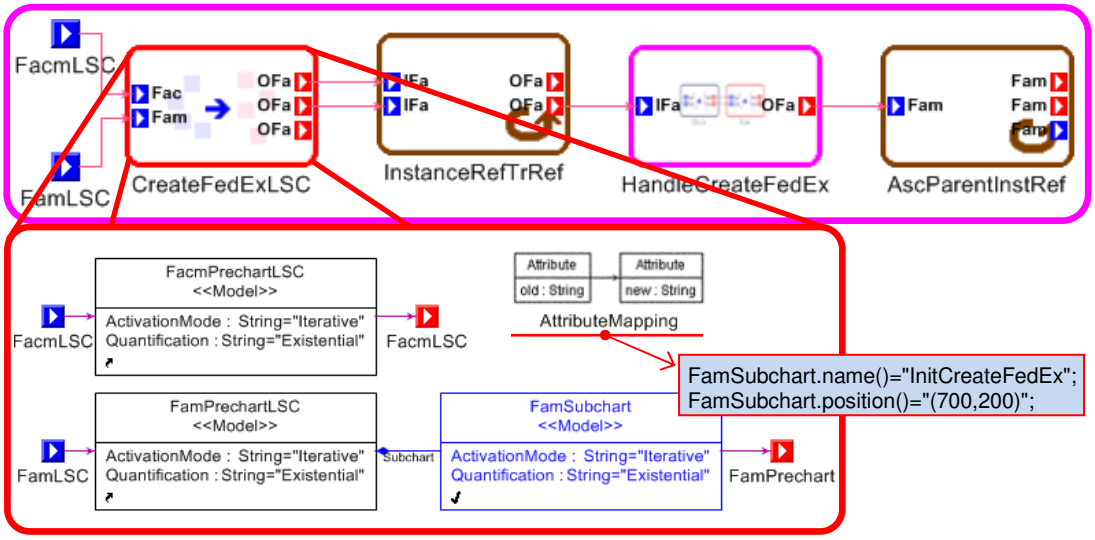


Figure B.51 CreateFedEx block and CreateFedExLSC rule

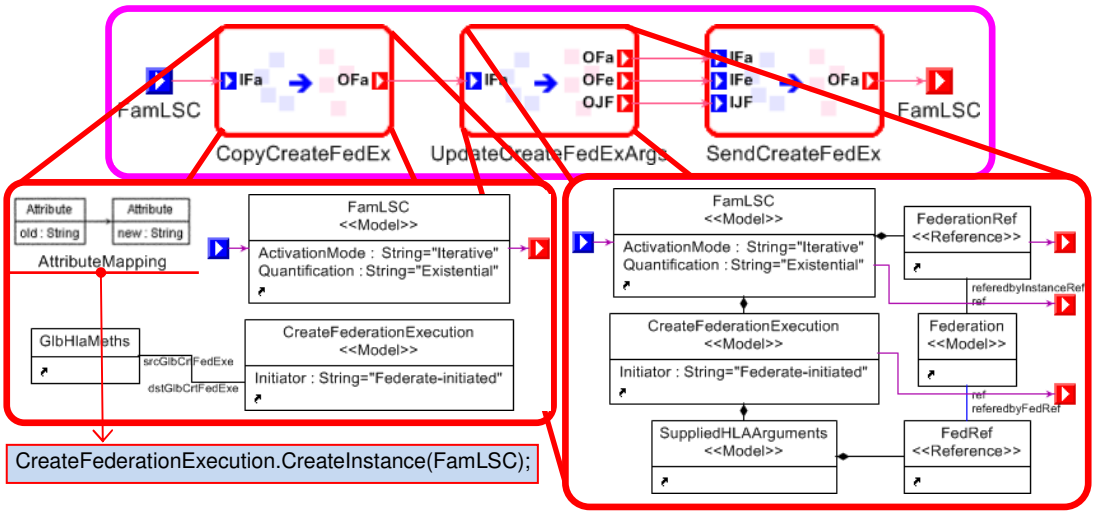


Figure B.52 HandleCreateFedEx block and CopyCreateFedEx and UpdateCreateFedExArgs rules

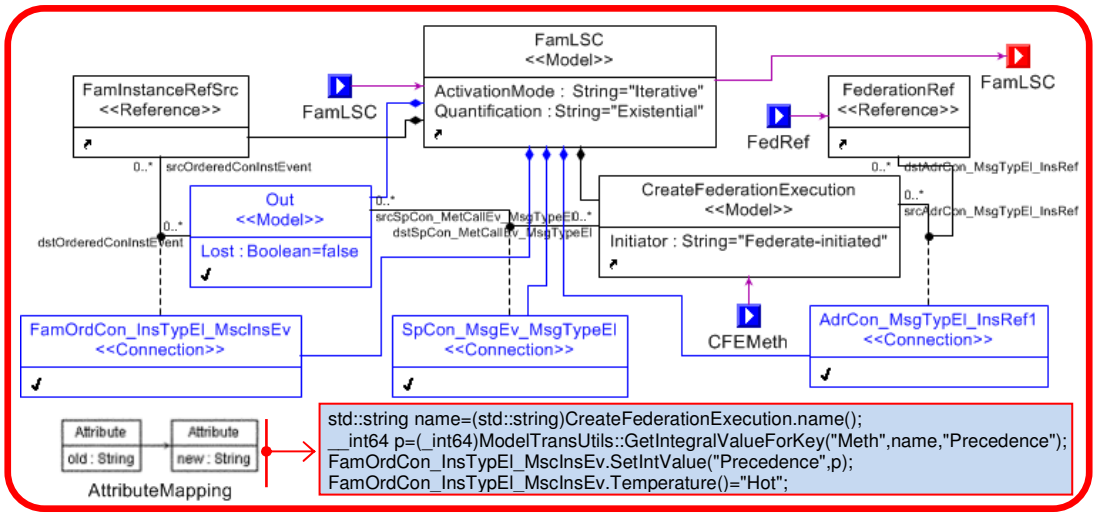


Figure B.53 SendCreateFedEx rule

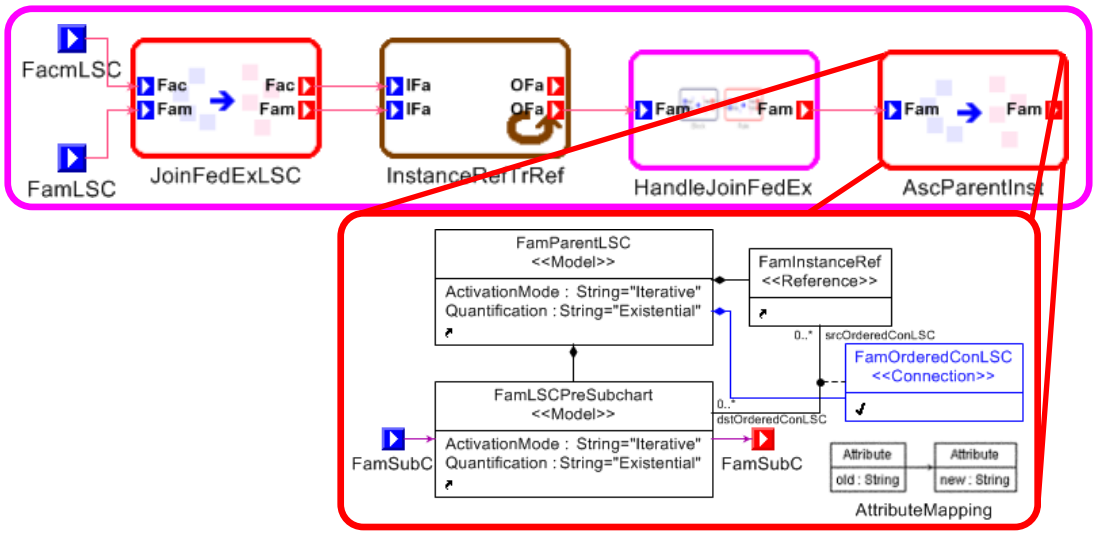


Figure B.54 JoinFedEx block and AscParentInst rule

B.3.4 LSC Transformation

The LSC transformation is the where the nuts and bolts of the evolution of field artillery inter-entity communications to federate interactions, mediated through the HLA RTI, are defined. The LSC transformation process is carried out in the `LSCTrans` block, as overviewed in Figure B.55. Each pass of the block inputs an ACM LSC and a stub FAM LSC, and step by step constructs the FAM LSC as the transformation proceeds through the internal blocks.

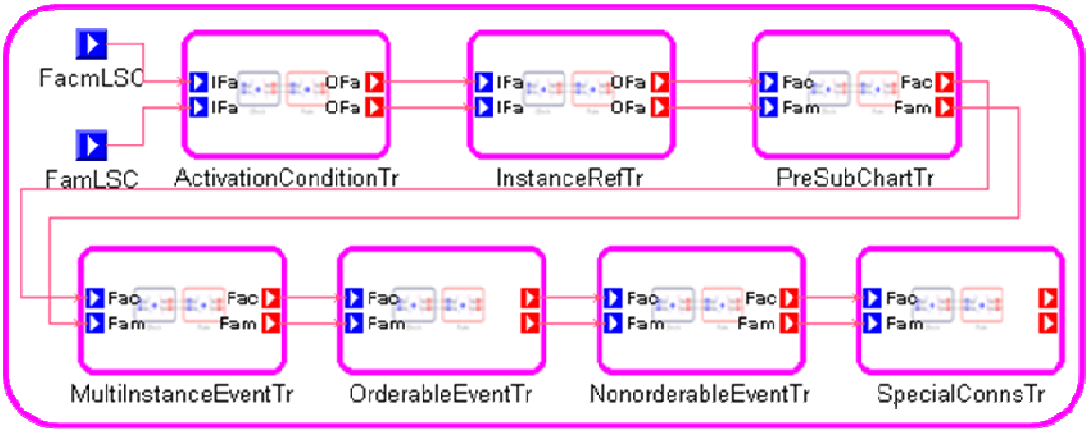


Figure B.55 LSCTrans block

Activation condition transformation is performed in the `ActivationConditionTr` block, as illustrated in Figure B.56. There is a simple one-to-one correspondence and equivalence between ACM and FAM activation conditions. The definition of the LSC transformation blocks are generally based on the instance event type categorization of the child elements to be processed in the LSC.

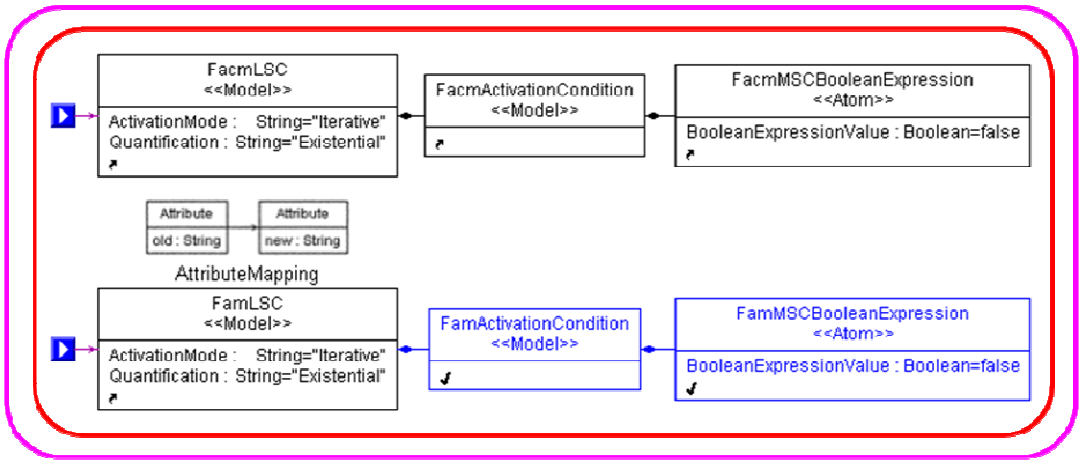


Figure B.56 ActivationConditionTr block and ActivationCondition rule

B.3.4.1 Instance Reference Transformation

The execution order of the sub-blocks of the `LSCTrans` block does not matter except for the second and the last blocks. The `InstanceRefTr` depicted in Figure B.57 creates the necessary federate instances (i.e., references) in the FAM LSC by inspecting the ones found

in the corresponding ACM LSC. Since these instances are used in the graph patterns of most of the subsequent rules, `InstanceRefTr` must be executed before them. The last block, `SpecialConnsTr`, create associations between two instance events [12] within the LSC and thus need to be executed after ensuring all such events have been created.

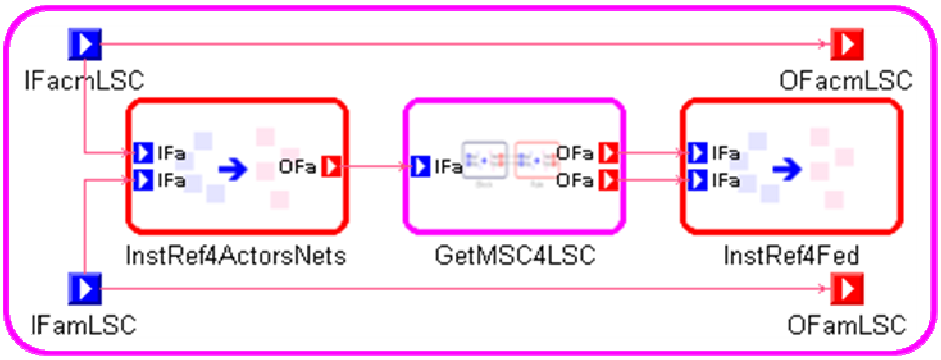


Figure B.57 InstanceRefTr block

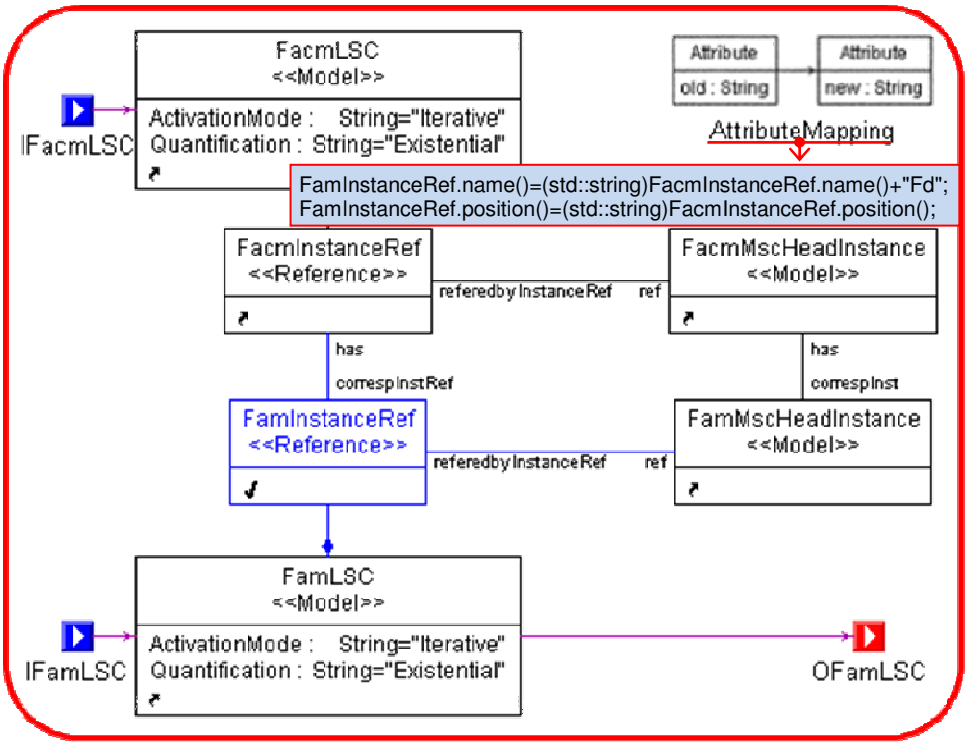


Figure B.58 InstRef4ActorsNets rule

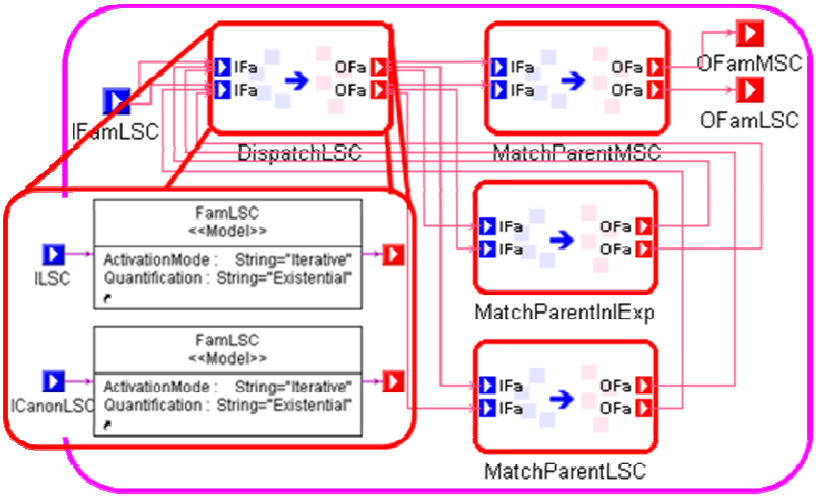


Figure B.59 GetMSC4LSC block and DispatchLSC rule

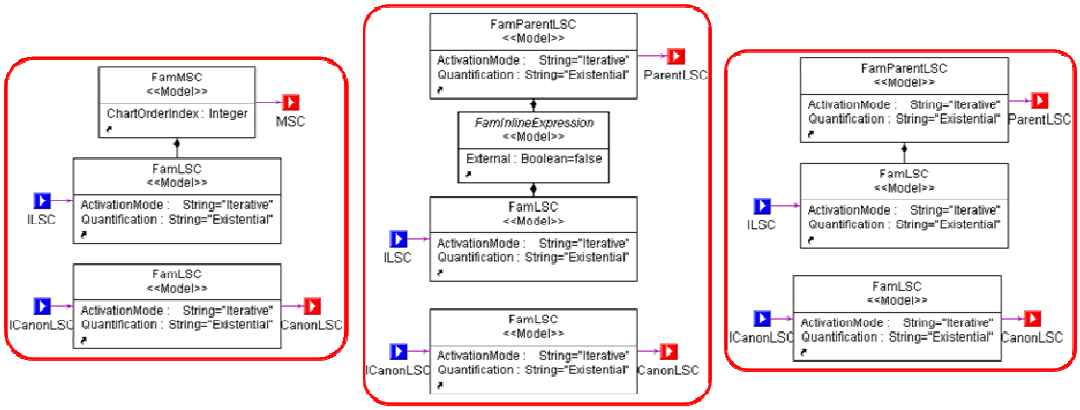


Figure B.60 MatchParentMSC, MatchParentInlExp and MatchParentLSC rules

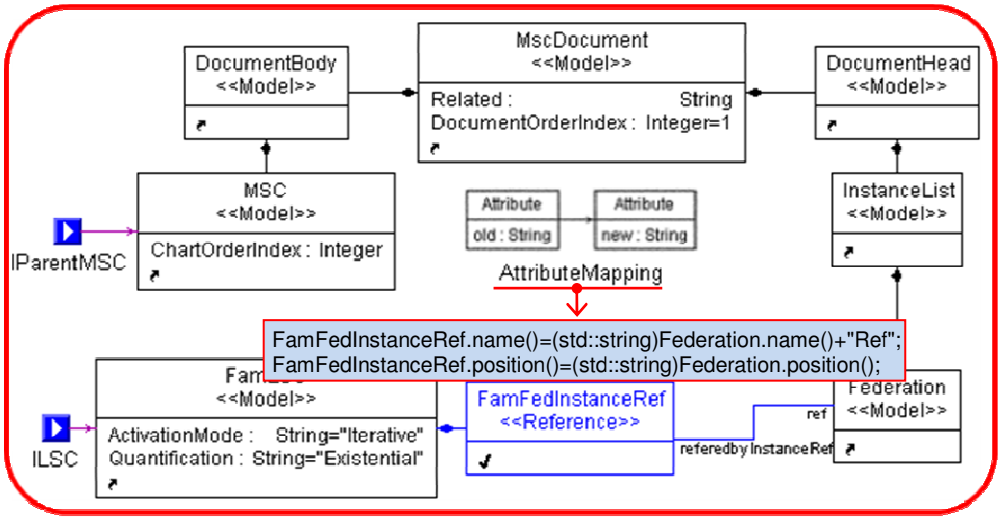


Figure B.61 InstRef4Fed rule

B.3.4.2 Prechart and Subchart Transformation

Precharts and subcharts are actually child LSCs that have special role names on the containment associations with their parents. The `PreSubChartTr` block, shown in Figure B.62, handles the transformation of precharts and subcharts of an LSC. The `CreateSubSchart` rule creates a subchart under the current FAM LSC with the `Subchart` compsition role for every subchart of the corresponding ACM LSC.

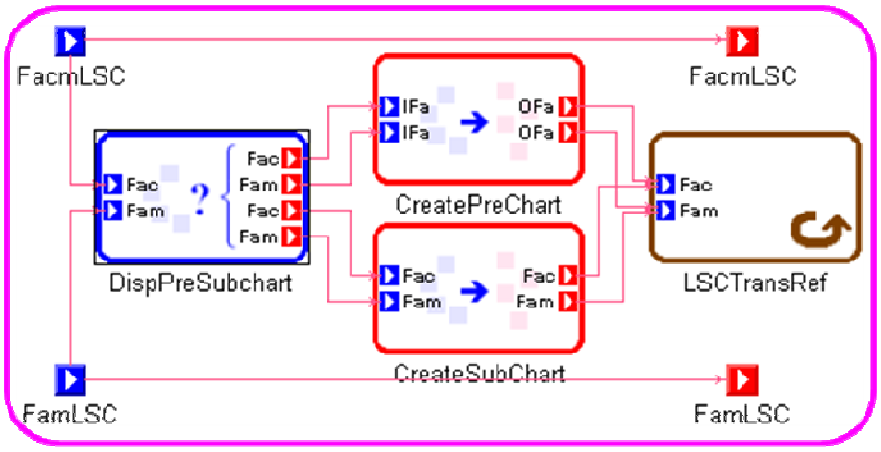


Figure B.62 PreSubChartTr block

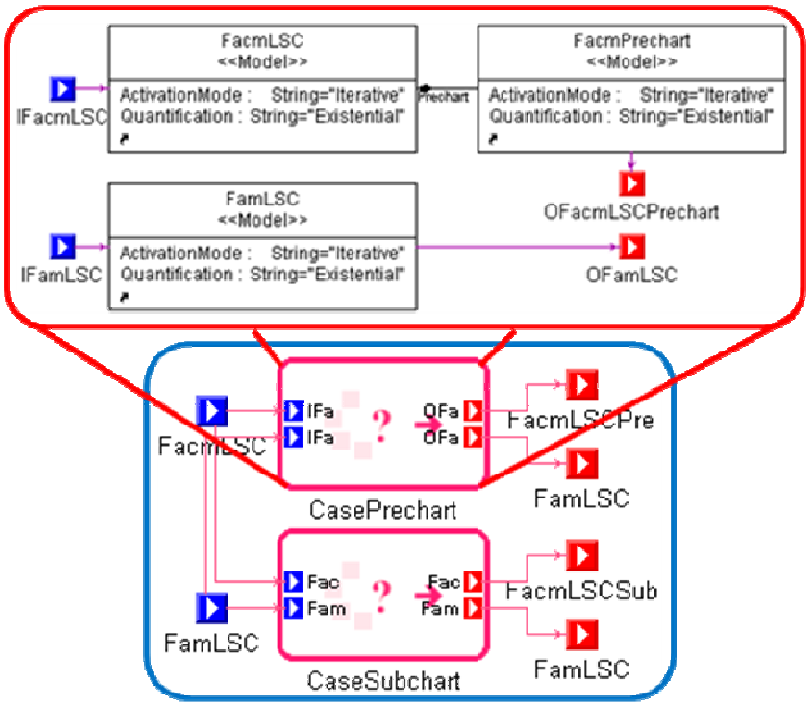


Figure B.63 DispPreSubchart test and CasePrechart case

The `CreatePreChart` rule, which is sketched in Figure B.64, is defined similar to the `CreateSubChart` rule. A notable statement in attribute mapping code (partly shown in the figure) is the call to the `SetInstRefAssocs4LSCChildren` method of the user code library. This method is invoked for all LSC child creations of type LSC (pre/subchart) and multi instance event, including inline expressions, references, conditions, otherwise clauses, and LSC idioms [12]. It handles the routine task of creating associations between an LSC's child elements and the relevant instances in the LSC programmatically.

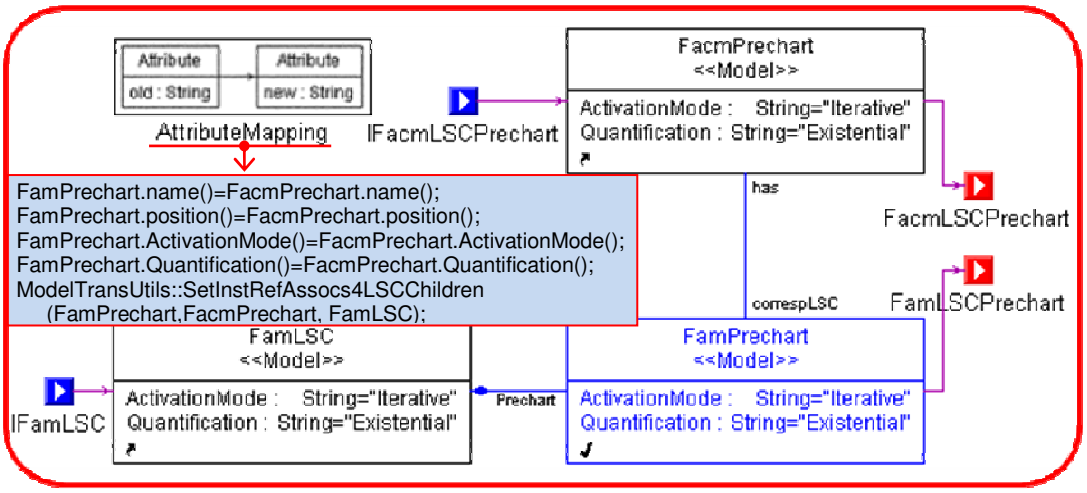


Figure B.64 The CreatePreChart rule

B.3.4.3 Multi Instance Event Transformation

The top-level block, `MultiInstanceEventTr`, is depicted in Figure B.65. Initially, a child multi instance event of the ACM LSC is matched and dispatched to one of the three alternative transformers together with the FAM LSC. The `CreateCondition` (seen in Figure B.67) and `CreateOw` rules perform condition and otherwise transformations, respectively. These rules simply create FAM elements that directly correspond to matched ACM elements. The other types of multi instance events form the family of reference identifications and are handled in the RefIdentTr block. Reference identification types are inline expressions and references. The `CreateReference` and `CreateMSCRef` rules, both shown in Figure B.68, simply create a FAM `Reference` element and a reference to an MSC under that, respectively.

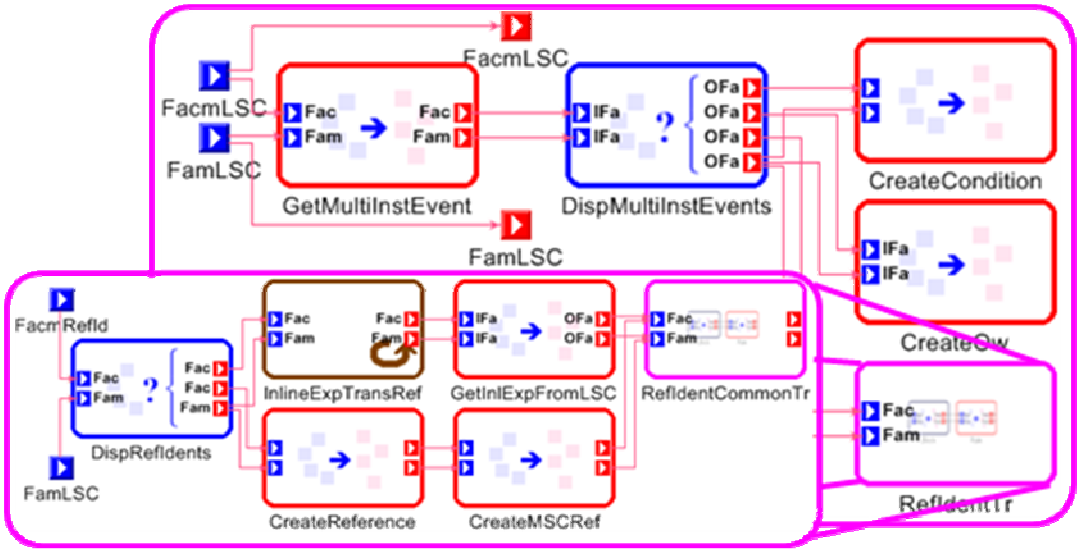


Figure B.65 The MultiInstanceEventTr and RefIdentTr blocks

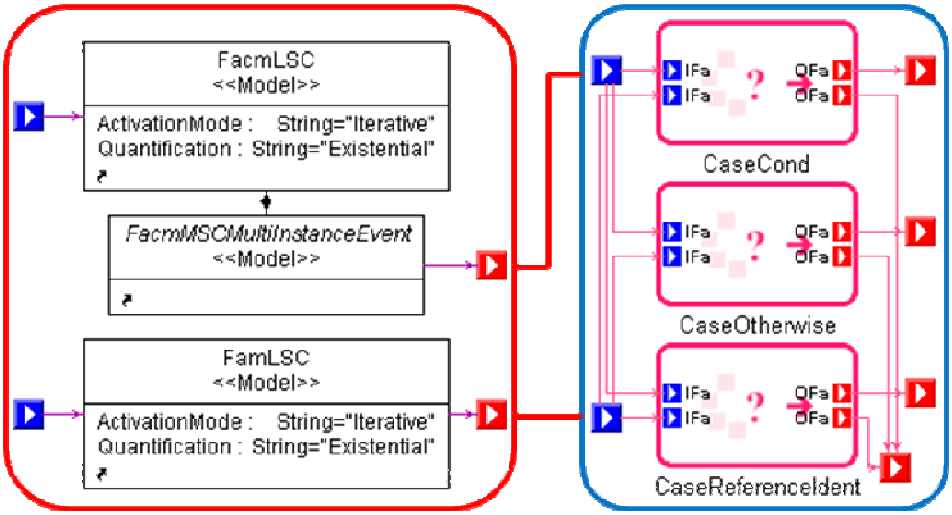


Figure B.66 The GetMultiInstEvent rule and DispMultiInstEvents case

The inline expressions are transformed in the `InlineExpTrans` block. The block initially directs the execution flow to one of the nine inline expression creator rules based on the input ACM inline expression type. Six of these create `alt`, `par`, `opt`, `loop`, `exc` and `seq` elements [15], and three of them create `if-then-else`, `while-do` and `repeat-until` idioms [12]. These rules simply create FAM inline expressions for the given ACM inline expressions and link them together using the `has-correspInlExp` cross-domain association. The attribute mapping codes copy the element properties.

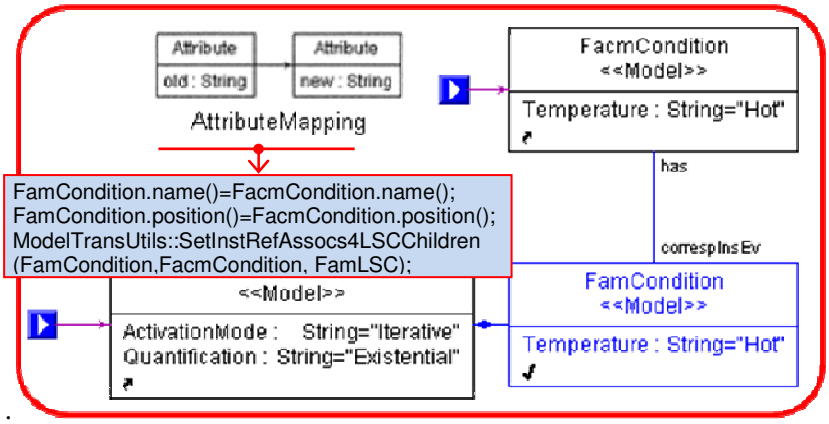


Figure B.67 The CreateCondition rule

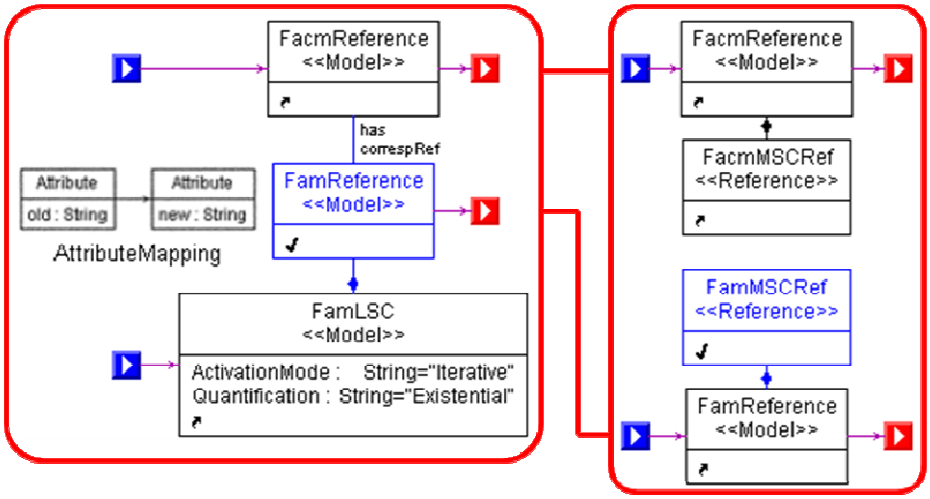


Figure B.68 The CreateReference and CreateMSCRef rules

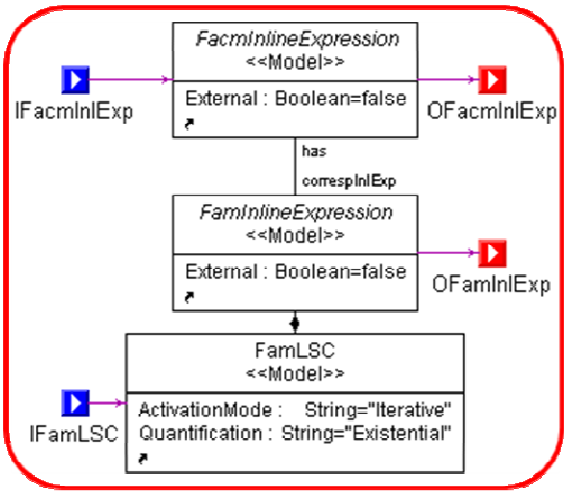


Figure B.69 The GetInlExpFromLSC rule

The `RefIdentCommonTr` is the last, sink block of the `RefIdentTr` block that creates gate, top, bottom and time interval components common for all reference identification type of elements. Time interval transformations further specialize into measurement, singular time and bounded time transformations. All of these rules are quite intuitive and perform ACM to FAM attribute value copying in a straightforward manner.

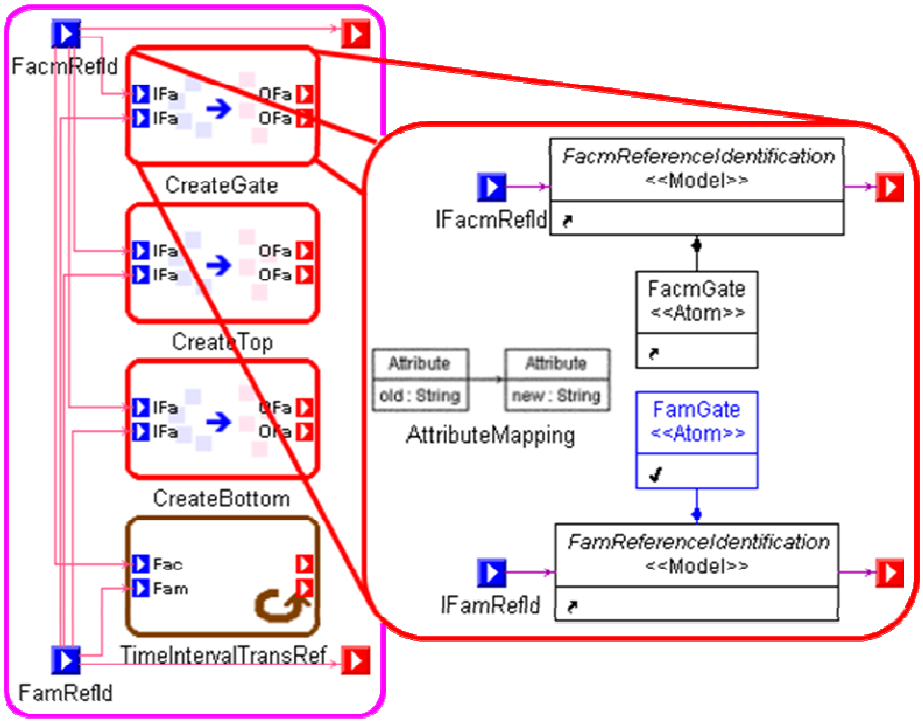


Figure B.70 The RefIdentCommonTr block and CreateGate rule

B.3.4.4 Orderable Event Transformation

The top level `OrderableEventTr` block is shown in Figure B.71. The block starts by matching and dispatching a LSC contained ACM orderable event to the appropriate rule or block to create its FAM counterpart. The kinds of orderable events handled are action, create, timer event, method event, and message event. The `HandleAction` rule is also provided in the figure as an example to explain how a typical orderable event rule works. For any given ACM action, a new FAM action is created in the given parent FAM LSC. From the ACM instance that is in association with the matched action, the corresponding FAM instance reference is obtained using the cross-domain association. Then a similar association is established between the FAM action and the FAM instance reference.

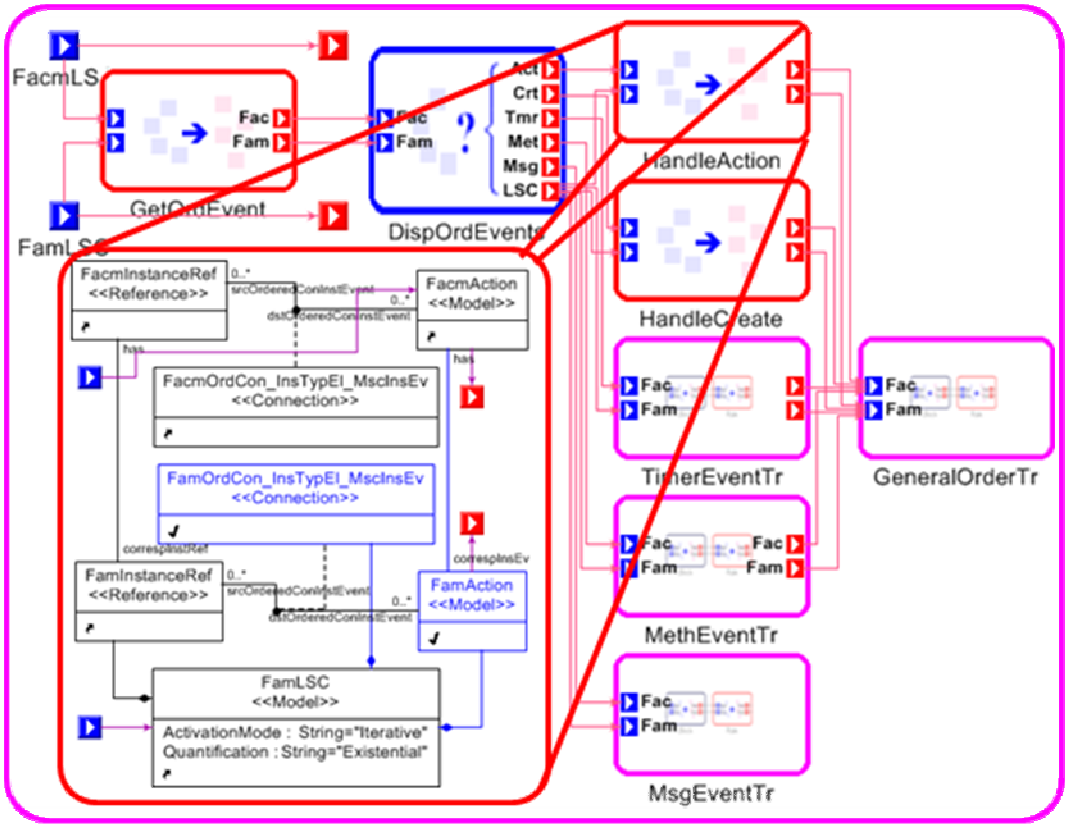


Figure B.71 The OrderableEventTr block and HanleAction rule

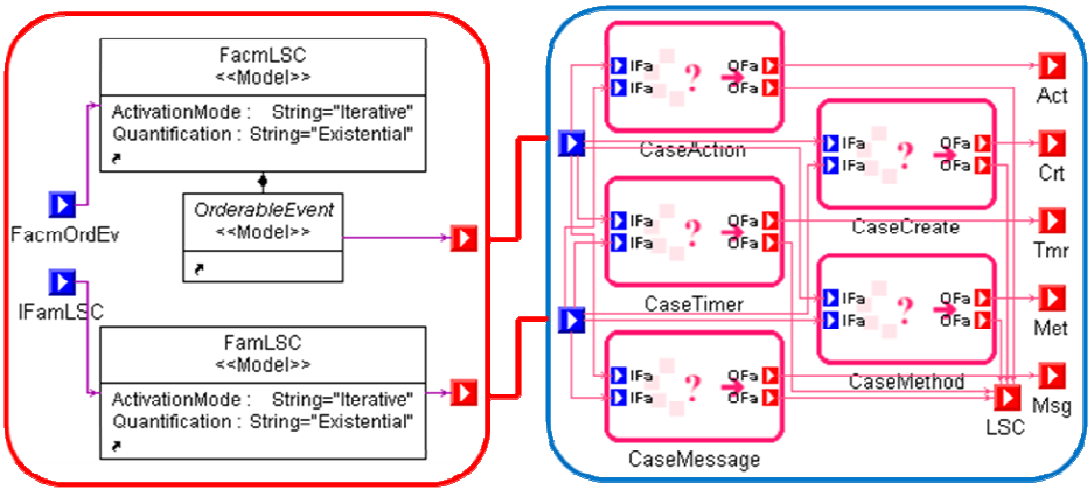


Figure B.72 The GetOrdEvent rule and DispOrdEvents case

The timer events, consisting of start timer, stop timer and timeout, form a sub-category of orderable events. The `TimerEventTr` block, sketched in Figure B.73, performs the

transformation of timer events. The block initially dispatches a matched ACM timer event and a FAM LSC to one of the three timer event creator rules. After the event creations, instance reference - timer event associations are established in the same manner shown in `HandleAction` rule. Timer events contain references to timer elements. Finally, the references to timers are set for the FAM timer events..

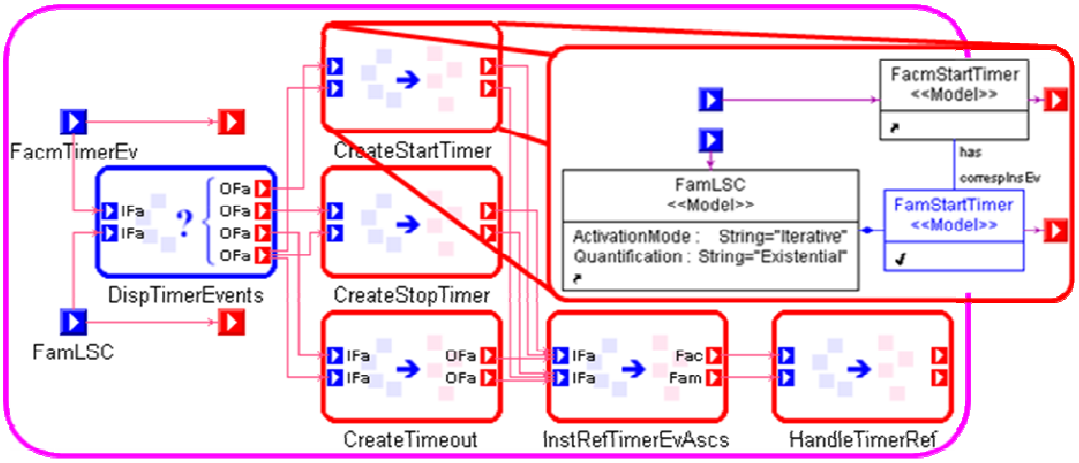


Figure B.73 The TimerEventTr block and CreateStartTimer rule

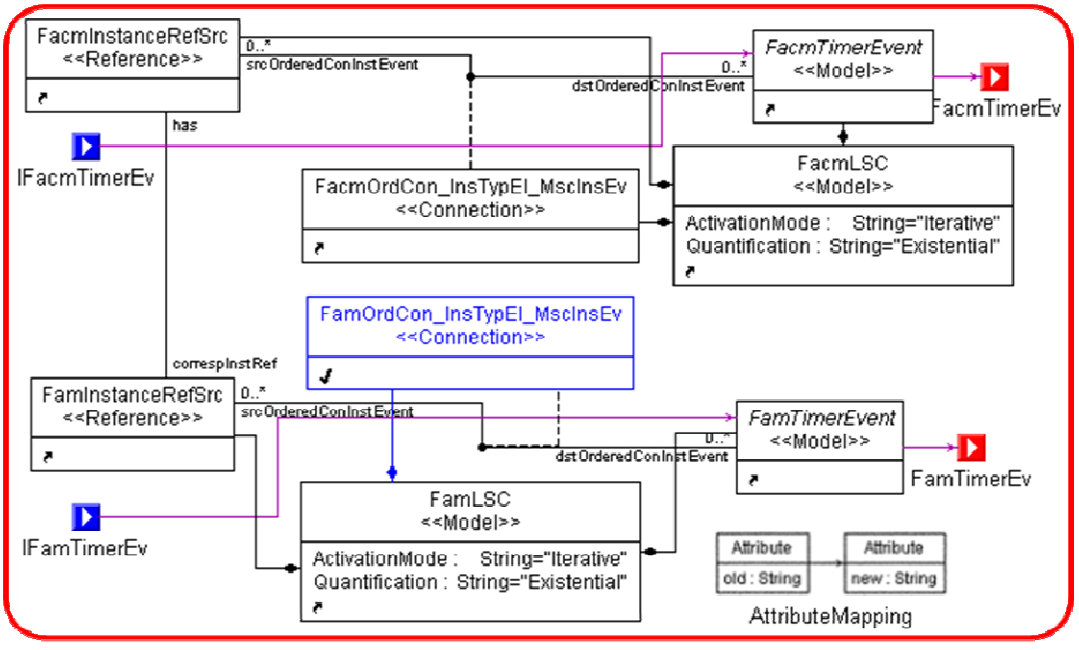


Figure B.74 The InstRefTimerEvAscs rule

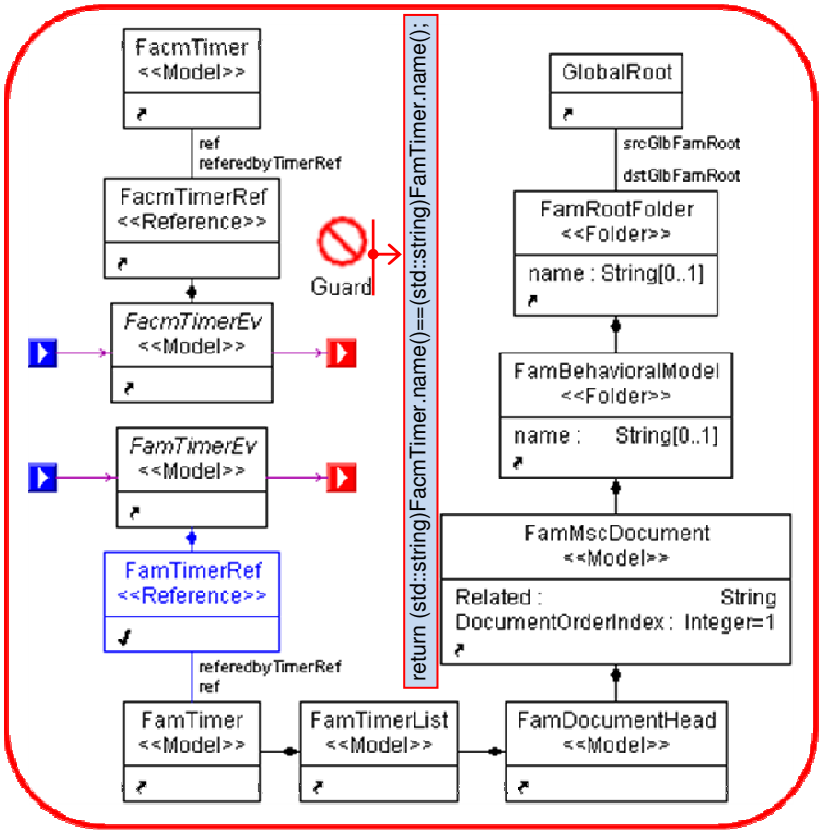


Figure B.75 The HandleTimerRef rule

Once the orderable events are transformed in their specifics, any general orderings (i.e., before and after) imposed on them are finally applied in the `GeneralOrderTr` block, as expounded in Figure B.76.

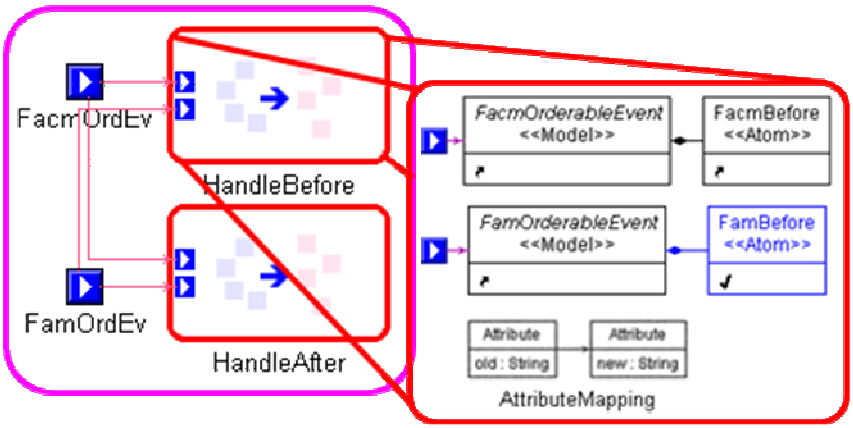


Figure B.76 The GeneralOrderTr block and HandleBefore rule

The `MethEventTr` block is shown in Figure B.77 where it handles the transformation of `call`, `receive`, `replyout` and `replyin` events that constitute the method call event category. These transformations are quite straightforward and handled similar to the `HandleAction` rule explained above.

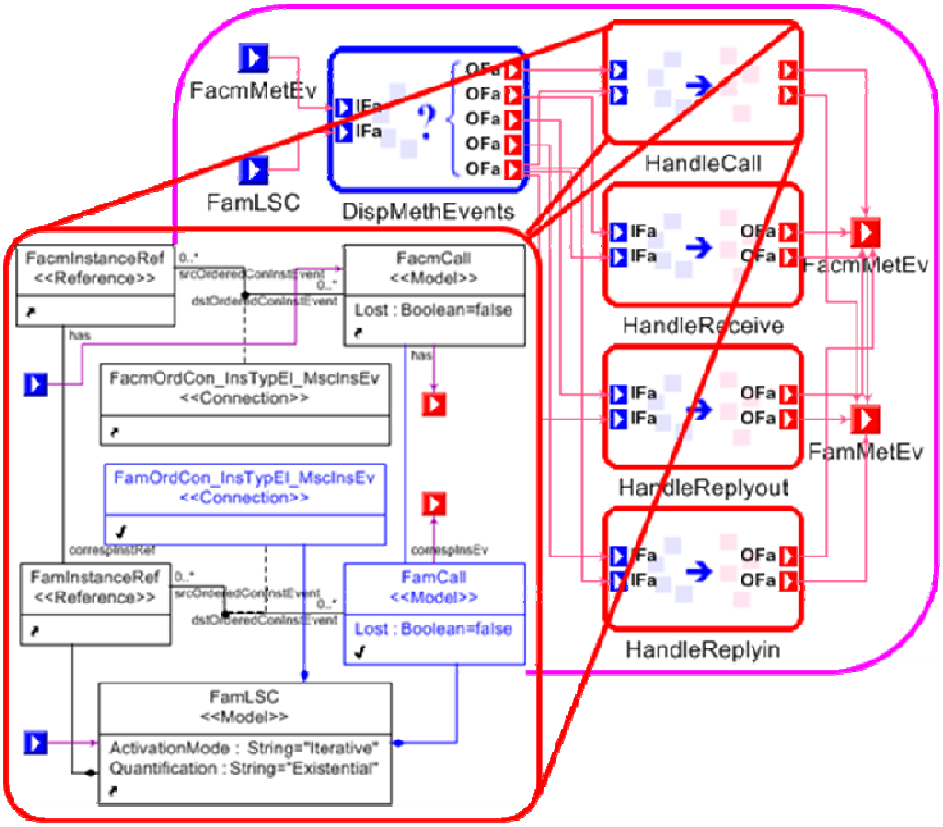


Figure B.77 The MethEventTr block and HandleCall rule

B.3.4.5 Message Event Transformation

The main message event transformation block, MsgEventTr, is displayed in Figure B.78. It distributes the incoming packets according to the type of the matched ACM message event. The OutMsg2HLAMeth block, also shown in the figure, handles the transformation of out events. Within the block, both FACM and FAM input packets are fed to two for-blocks in parallel that are specialized in out event transformations based on the type of the message payload of the FACM out event. Non-durable message transmitting out events are transformed in OutNonDurableMsg2HLAMeth for-block and durable message transmitting out events are transformed in OutDurableMsg2HLAMeth for-block.

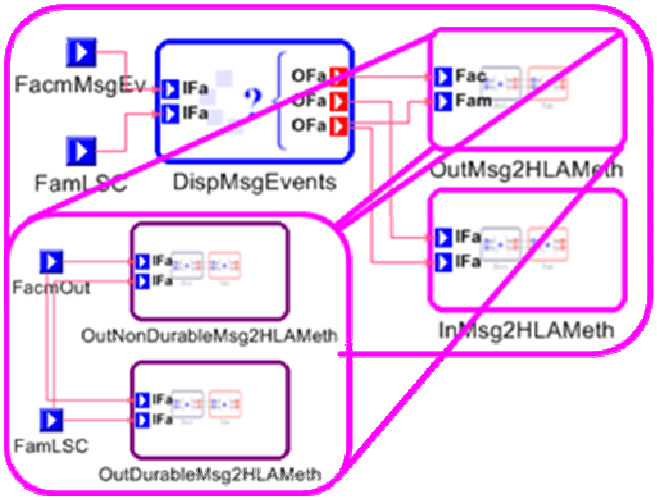


Figure B.78 The MsgEventTr and OutMsg2HLAMeth blocks

B.3.4.5.1 Non-Durable Message Event Transformation

The OutNonDurableMsg2HLAMeth block that handles non-durable ACM out message transformation is rendered in Figure B.79.

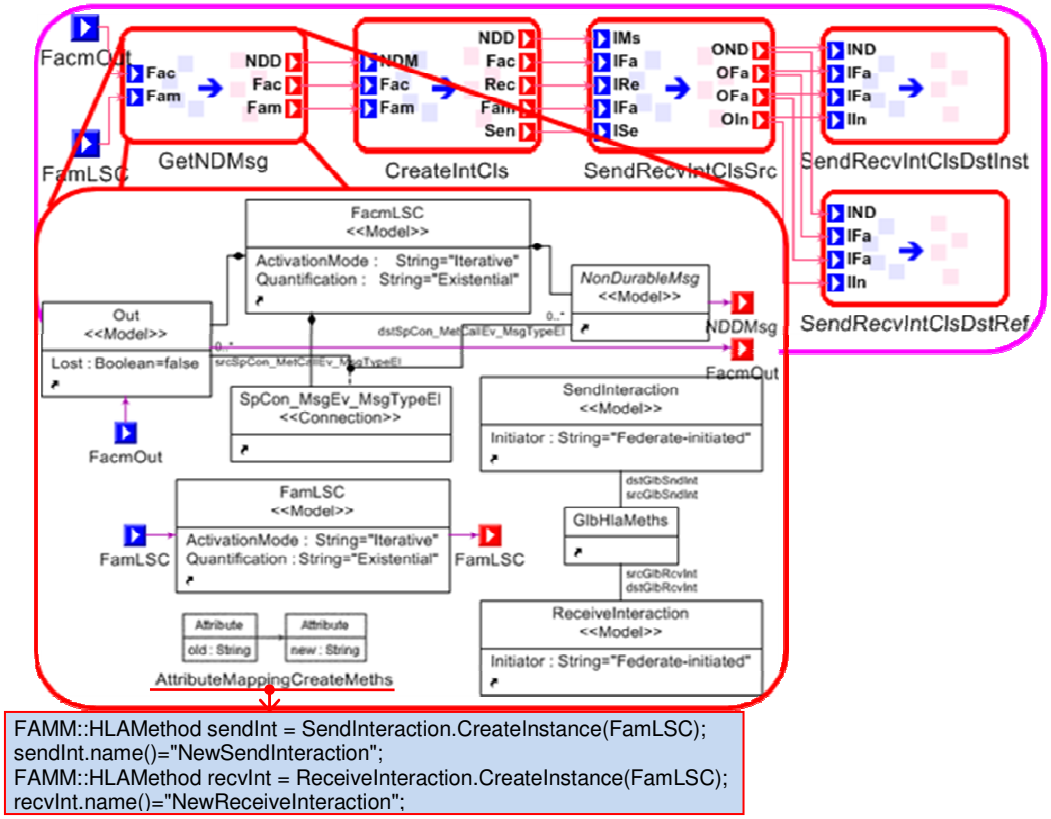


Figure B.79 The OutNonDurableMsg2HLAMeth block and GetNDMsg rule

The initial rule, GetNDMsg, matches and delivers the ACM out event and the non-durable message and FAM LSC to the next rule, as well as programmatically creating a copy of SendInteraction and ReceiveInteraction HLA methods inside FAM LSC. The original methods do not contain any arguments, but their copied instances will have theirs assigned (such as HLA classes and federate references) as the transformation proceeds. The method copies are tagged as "New" to differentiate and match them from the others of the same type in the next rule

In the CreateIntCls rule of Figure B.80, the guard expression is used to make a name comparison to check whether the send and receive interaction methods are prefixed with "New" in their names. Once the match is there, a new interaction class corresponding to the ACM non-durable message is created in the FAM FOM. The interaction class references of the both HLA methods' supplied arguments are made to refer to the new interaction class. The attribute mapping code removes the "New" tags of the HLA methods, sets the name of the new interaction class to the name of the non-durable message suffixed by "IC", and invokes the user library code method to programmatically build the interaction class from the non-durable message

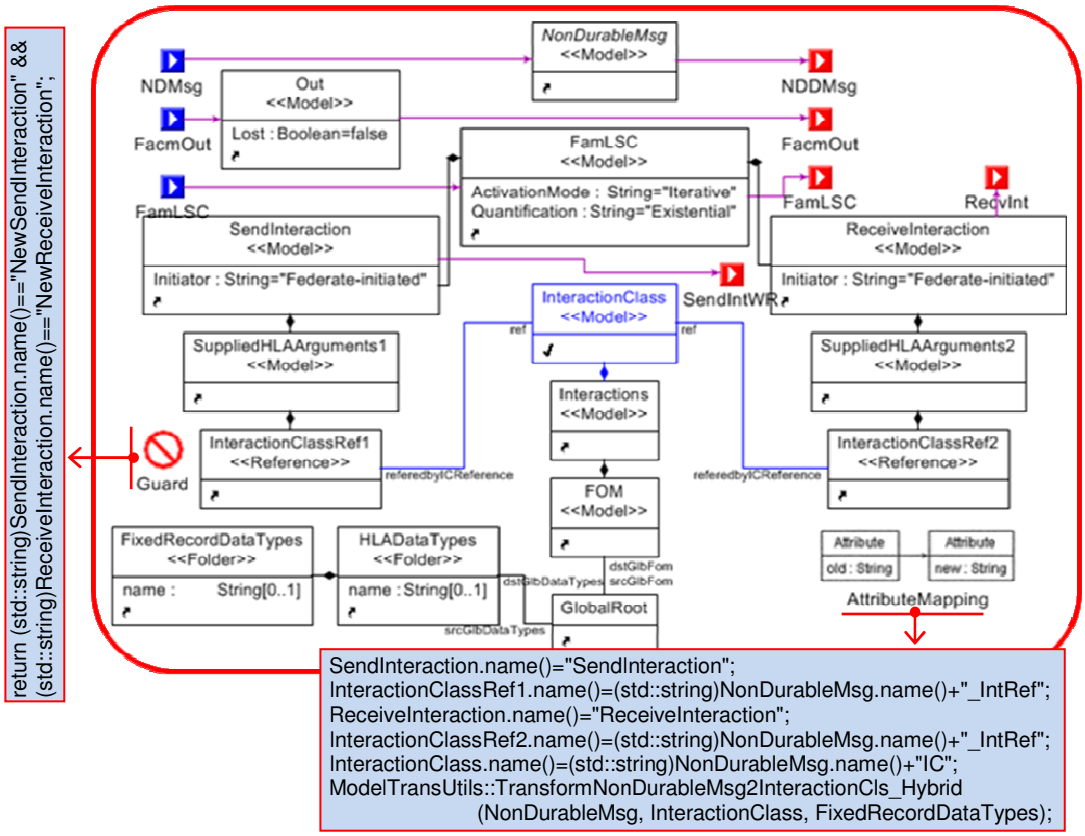


Figure B.80 The CreateIntCls rule

The SendRecvIntClsSrc rule in Figure B.81 first creates an out message event and associates it with the source instance (i.e., federate) using an ordered connection. Then it associates the out event to the send interaction method using a special connection. Finally it associates the send interaction method to the federation instance using an address connection. After this, a similar complementary stage starts for the receive interaction method, but this time from the federation to the target federate. First it associates the receive interaction method to the federation instance using an address connection. Then it creates an in message event and associates it to the receive interaction method using a special connection. The attribute mapping code copies the precedence and temperature values from the ACM ordered connection to the FAM ordered connection.

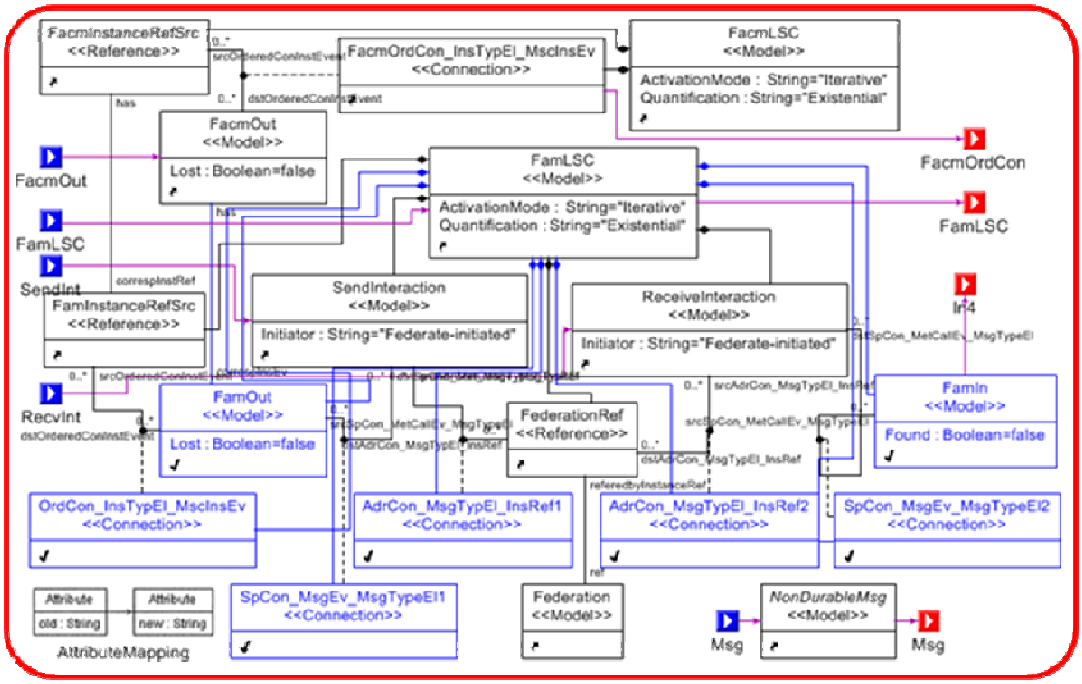


Figure B.81 The SendRecvIntClsSrc rule

The last part of the out event transformation is done by one of the two parallel rules named SendRecvIntClsDstInst and SendRecvIntClsDstRef, addressing the cases of message event target being an instance or an MSC reference, respectively. The SendRecvIntClsDstInst rule is shown in Figure B.82. It matches the pattern that associates the ACM non-durable message to the target instance reference, and creates a corresponding association on the FAM side. The attribute mapping code copies the precedence and

temperature values from the ACM ordered connection to the FAM ordered connection with precedence being increased by one, since that value was already used for the other message event in the previous rule. The other parallel rule, SendRecvIntClsDstRef, is defined similarly with the only difference being the reference to an instance replaced by a reference to an MSC reference.

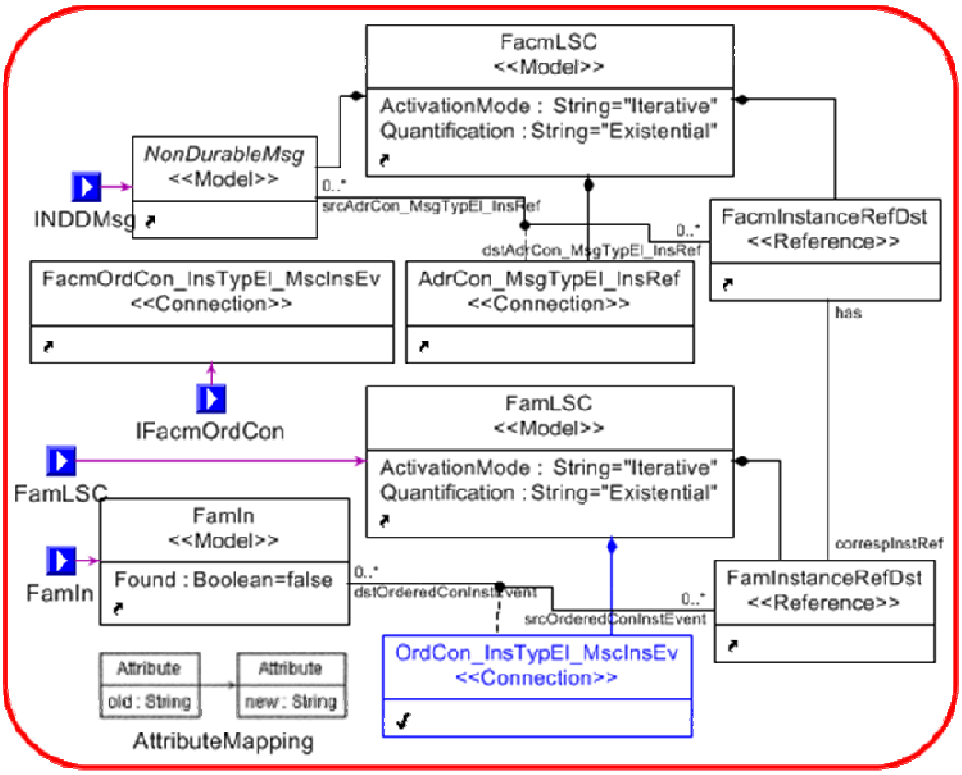


Figure B.82 The SendRecvIntClsDstInst rule

B.3.4.5.2 Durable Message Event Transformation

Durable message transformation is the biggest of the LSC instance event transformations in terms of size and complexity. Figure B.83 displays the OutDurableMsg2HLAMeth. It is defined methodologically similar to the OutNonDurableMsg2HLAMeth block, only being about three times in size. Thus, it is redundant to explain the details of the transformation, but appropriate to provide an overview on the differences.

The durable messages in ACM are defined to be of, instantiation, update and deletion types [11]. There are three parallel courses of transformations that address out message

events of each durable message type. An ACM instantiation message out event maps to six FAM HLA method out events. The mapping cardinalities of an out event for update and delete types are both one to two.

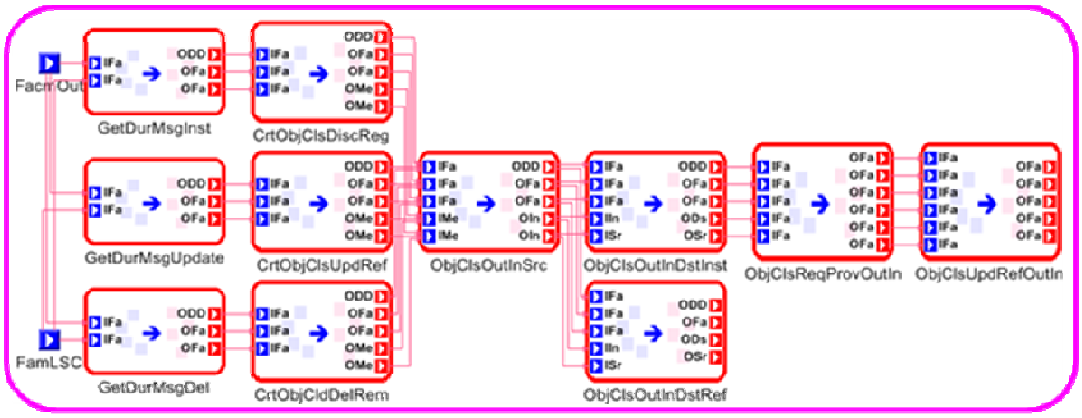


Figure B.83 The OutDurableMsg2HLAMeth block

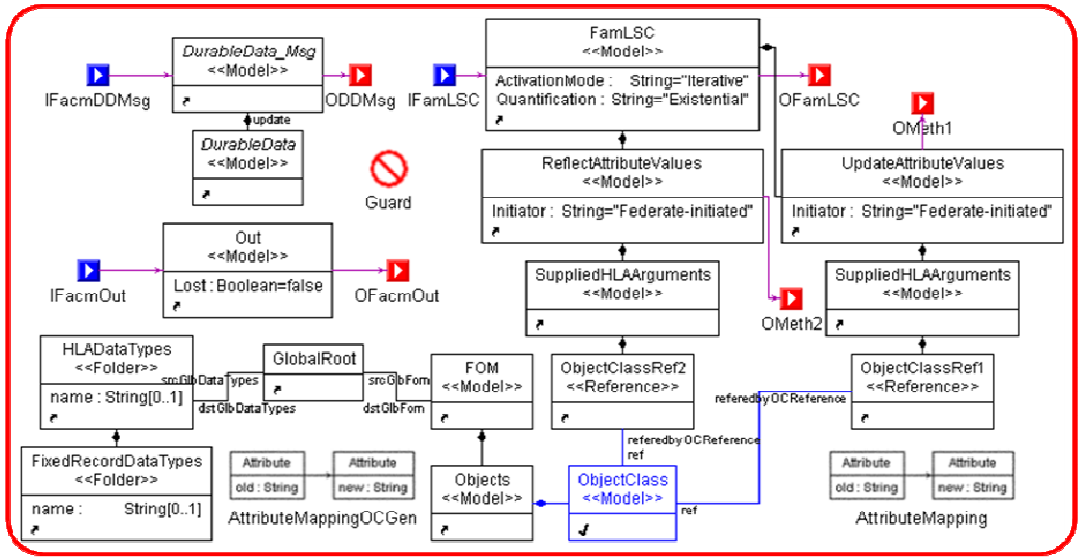


Figure B.84 The CrtObjClsUpdRef rule

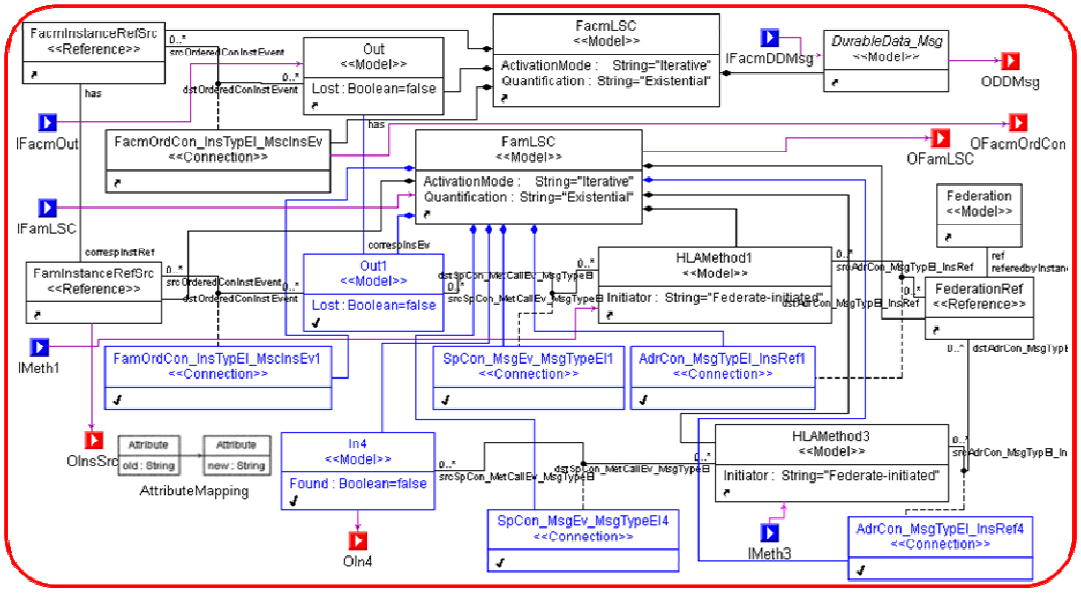


Figure B.85 The ObjClsOutInSrc rule

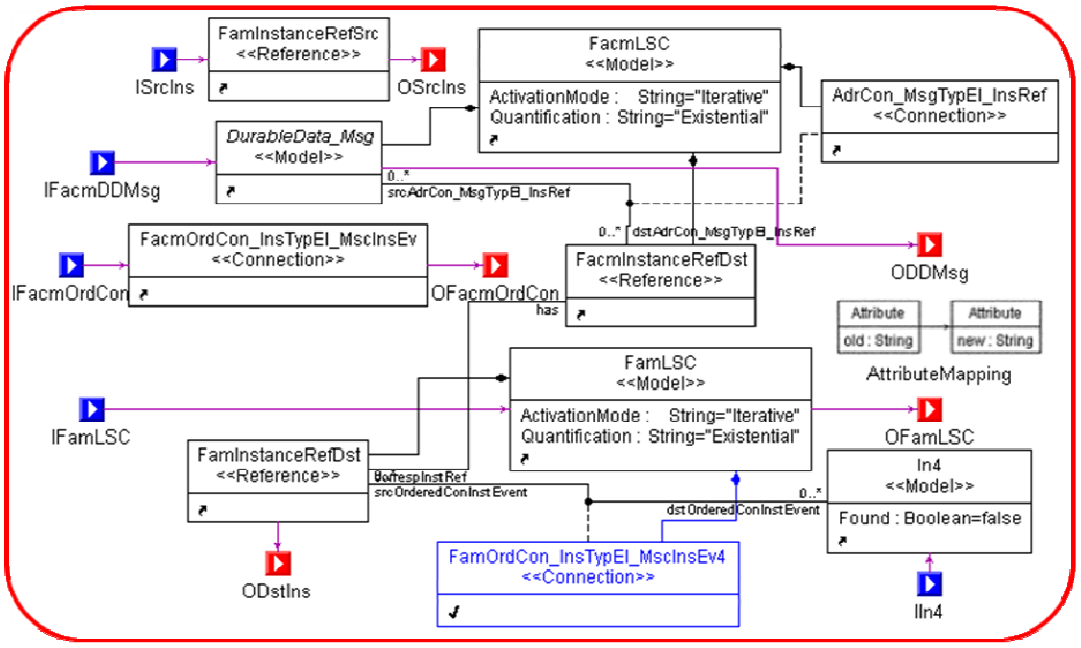


Figure B.86 The ObjClsOutInDstInst rule

.

B.3.4.6 Non-Orderable Event Transformation

The non-orderable events constitute the set of instance events that do not require an explicit ordering of execution. Figure B.87 depicts the NonorderableEventTr block that

handles the transformation of non-orderable events. The block initially matches and dispatches the input packets to one of the handler rules according to the type of the ACM non-orderable event. The handler rules perform the transformation of method, end method, concurrent, end concurrent, suspension, end suspension, stop, end instance, invariant, end invariant and simultaneous region.

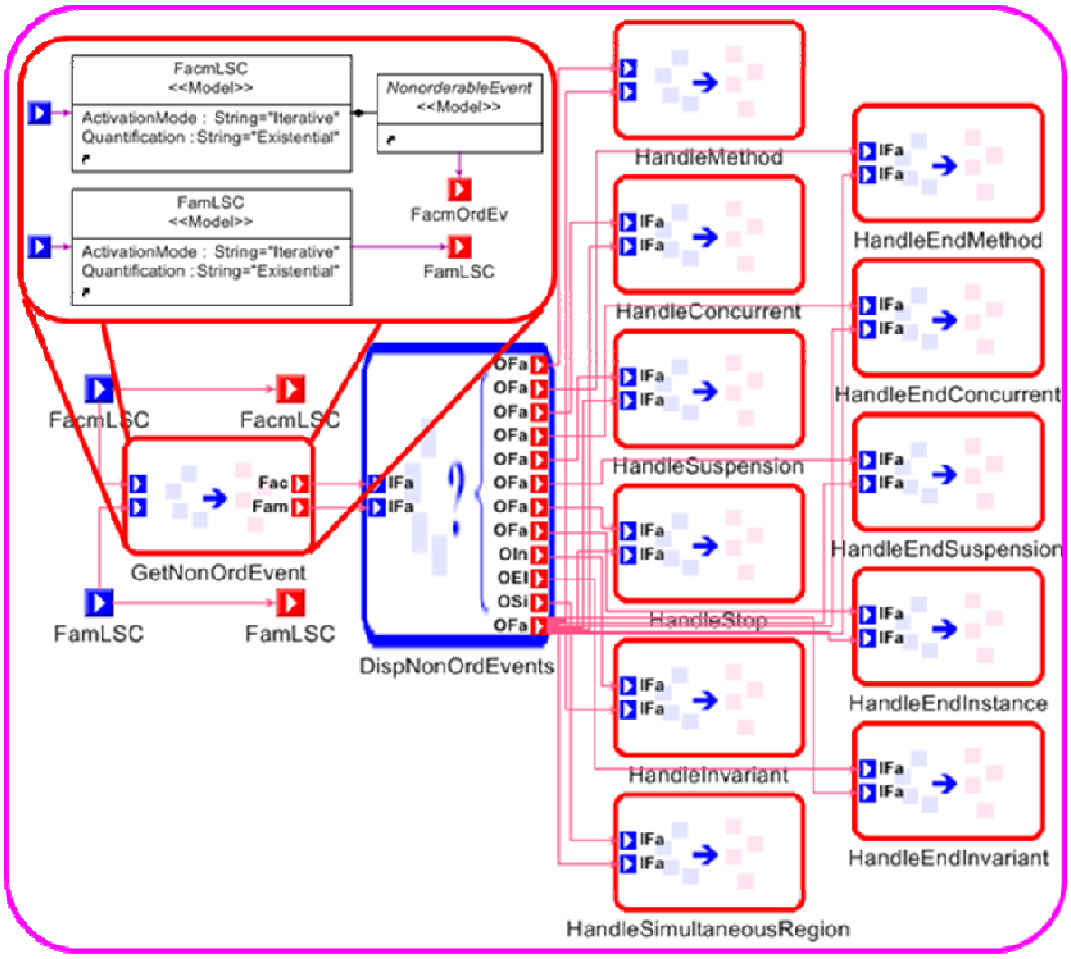


Figure B.87 The NonorderableEventTr block and GetNonOrdEvent rule

The HandleMethod rule is provided as an example to explain how a typical non-orderable event rule works, in Figure B.88. For any given ACM method, a new FAM method is created in the given parent FAM LSC. From the ACM instance that is in association with the matched method, the corresponding FAM instance reference is obtained using the cross-domain association, “has-correspInstRef”. Then a similar association is established between the new FAM method and the FAM instance reference.

Finally, a “has-correspInEv” association is established between the new FAM method and the ACM method. All the other non-orderable event rules are similarly defined in a straightforward manner.

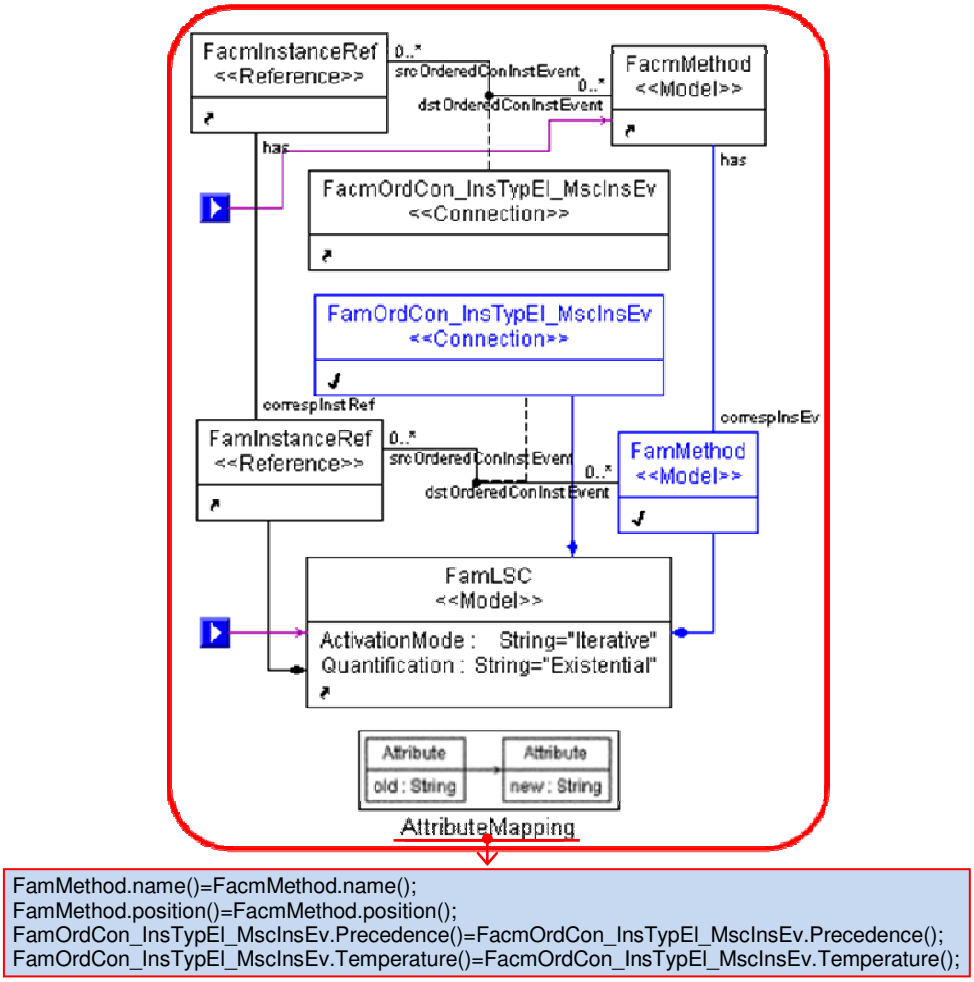


Figure B.88 The HandleMethod rule

B.3.4.7 Special Associations Formation

The SpecialConnsTr block, placed at the end of the LSC transformation path, is responsible for the transformation of those parts that do not involve instances. It is deliberately positioned as the last LSC transformation block because it requires all of the FAM LSC entities to be already created and available by the time it starts execution.

Figure B.89 shows the SpecialConnsTr block, which is the transformer for special associations. There are three kinds of special connections used in this work that associate

simultaneous regions to instance events, timer starts to timer events and general order elements to ordered events. The figure additionally shows the AscSimRegToInstEv rule as an explicatory example. For any ACM simultaneous region that is specially associated with an instance event, the rule matches their corresponding FAM simultaneous region and the instance event by utilizing their cross-links to FAM. Then a similar kind of special association is established between the two FAM elements. The other two special connection transformations are defined with the same approach.

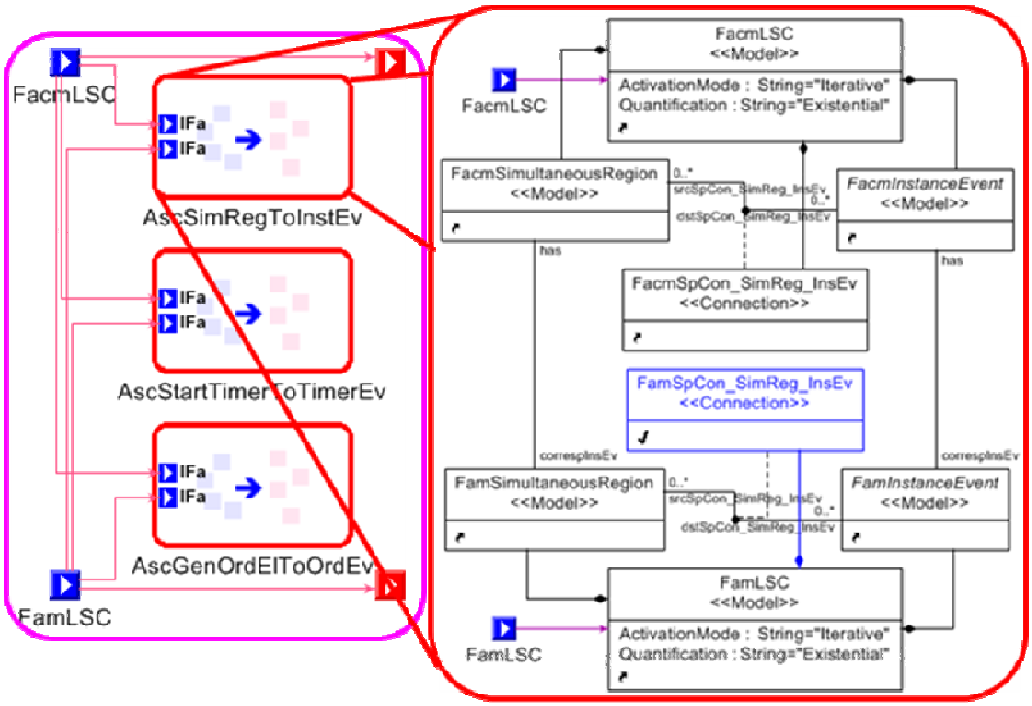


Figure B.89 The SpecialConnsTr block and AscSimRegToInstEv rule

message, start timer event, etc.

B.3.5 Bind Decomposed Instance MSC References

The last block of the start rule, `AssocDecompAsRefs`, is illustrated in Figure B.90. In the previous `DecomposeInst` block, new decomposed FAM MSCs were created corresponding to the FAM MSCs that contained decomposed-labeled instances. Such instances contain references to their decomposed MSCs. At the end of the transformations, the reference associations between the new FAM decomposed instances and MSCs are still not bound. The role of `AssocDecompAsRefs`, is to establish these bindings.

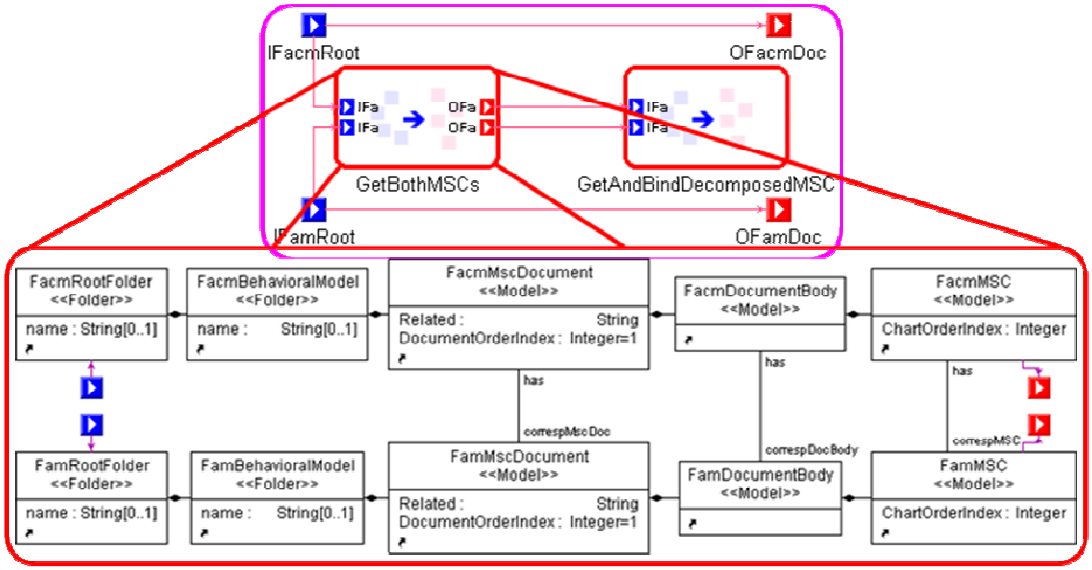


Figure B.90 AssocDecompAsRefs block and GetBothMSCs rule

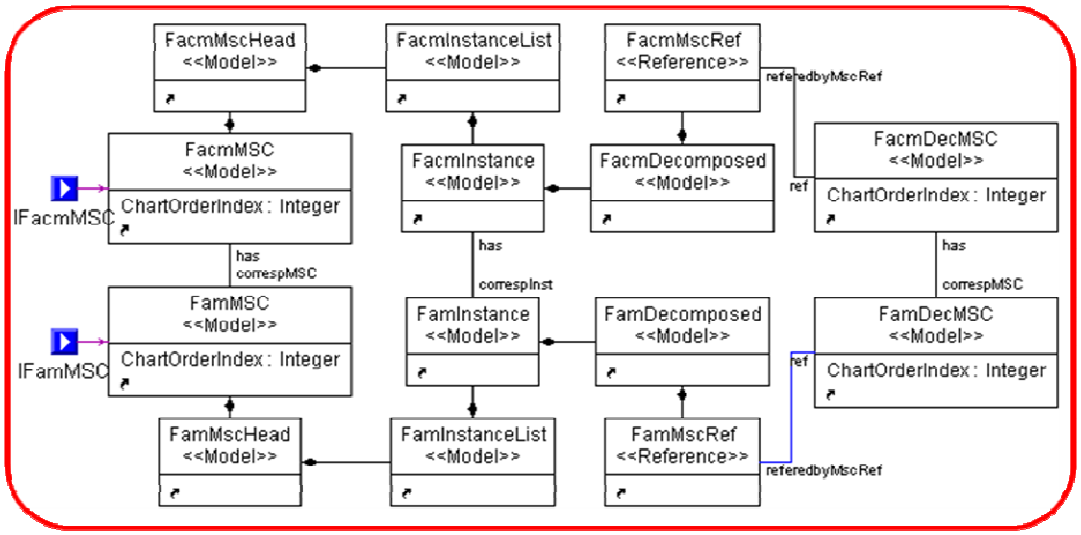


Figure B.91 GetAndBindDecomposedMSC rule

APPENDIX C

CHANGES MADE IN METAMODELS AND CODE GENERATOR

This appendix summarizes the changes made in ACMM, FAMM and the simulation Code Generator (CodeGen) in the course of developing ACM2FAM transformations. The change log for FAMM is especially important because it was previously developed in another study [12]. Although FAMM functions smoothly as a domain metamodel in GME, it causes some errors and issues when used as a target model for transformation rules in GReAT. The reason for most of these problems is that transformation definitions in GReAT are first interpreted into C++ code and then this code is executed to actually perform the transformation. C++ is a strongly typed language and has strict syntax rules. The part of the generated model transformation code from FAMM in its original state is not error free.

The first part of this appendix categorically summarizes the issues revealed in the metamodels and the generated transformation code. The second part outlines the issues with the simulation code generator, which were either inherent or introduced indirectly due to the changes made in FAMM. The change log for the metamodels and the CodeGen are provided in the thesis documentation CD.

C.1 Issues with the Metamodels

This section summarizes the issues caused by ACMM, FAMM or their sub-metamodels during the ACM2FAM development and transformation code generation processes and describes the solutions, and sometimes, the workarounds applied to mitigate the problems. The changes made in between the metamodel versions together with the dependent sub-model versions, if there are any, are documented in the accompanying thesis CD.

C.1.1 Eliminating One-to-Many Connections

If a connection modeling element is used more than one time as source-to-connector or connector-to-destination, then the code generator generates duplicate method definitions for those connection parts, which result in compile errors. In other words, if a model element `E1` (e.g., `InlineOperandInterfaceBase`) is associated as source with more than one elements, say, `E2` and `E3` (e.g., `Reference` and `Final`) as destinations using the same

connection element (e.g., `HmscOperandConnection`), then the generated code is erroneous with duplicate type definitions. The specific place of the error is inside the class definition of `E1` where duplicate association role types are defined for `E2` and `E3` with the same name as the connection element. As a result, the GR-engine or GR-debugger gives error. This case is simply exercised and verified in the `House2OrderTest` sample. The same problem also arises if the associations were destinations.

Note that if the association source (or destination) role names are manually changed to unique names in the umt.mga file, then the above problem of method redefinition is resolved, but this time the metamodel header file produces an error.

Therefore, one to many connections of the same `Connection` element must be avoided. This is achieved by building a connection hierarchy so that ambiguities in code generation are eliminated.

C.1.2 Name Clashes with Reserved Words

GReAT's transformation code generator generates a class or method for every modeling element, role or attribute having either the exact element name or prefixed/suffixed by some tag word. In the exact name usage, the C++ compiler produces syntax errors for those names that are C++ keywords. These names must be altered to non-keyword forms within the metamodels.

Also, there are some auto-generated utility or management methods for every generated First Class Object (FCO) class, such as `Create()`. Modeling element names must not also be the same with such internally used artifact names for the same reason.

C.1.3 Non-unique FCO Names

Some FCOs in different paradigm sheets within the same metamodel or in the unified ACM2FAM transformation model obtained by exporting ACMM and FAMM into the transformation model, may occasionally have exactly the same name. This does not cause any problem as far as GME modeling is concerned. However, when those modeling elements are used in GReAT, the transformation code generator produces duplicate class names, which result in syntax errors at compile time.

Therefore, modeling element name uniqueness must be established all across the components in a GReAT transformation model.

C.1.4 References Pointing to Multiple Items

Any GME `Reference` that pointed to all of a super class's child classes was modified to refer to the super class only, in order to reduce redundancy. This resulted in an extra inheritance hierarchy in the model transformation definition file to pack all the child classes

under a generated common parent, and the reference pointing to that super class. This is just for the reader's information. This modification is just an internal optimization and does not have any negative effect on the models or transformation definitions.

C.2 Issues with the Simulation Code Generator

As in the case of FAMM usage, the simulation Code Generator (CodeGen) gave errors when invoked on the FAMs that were generated from AdjFFE mission ACMs. The problems generally had to do with the imperfection of the CodeGen. The reason for most of the issues, we think, is that the development of CodeGen was majorly steered by the samples that were manually created during FAM testing. The scope and representative power of those samples were not as far-reaching as AdjFFE FAMs. In addition to these, the CodeGen simply had some syntactic and semantic flaws in its code generation logic and shortcomings in FAM coverage that we have discovered during our exercises. Finally, a portion of the problems were introduced after changes were made to FAMM due to GReAT and C++ restrictions, because the CodeGen is strictly coupled to FAMM in terms of model element names and structures.

The details of the changes made to the CodeGen are provided in a separate document inside the accompanying thesis CD. The changes are presented in two-column tables per Java source file, where the first column shows the original code part and the second column shows the changed code part.

APPENDIX D

TIPS AND PITFALLS IN DEVELOPMENT WITH GREAT

This appendix provides hints and recommendations derived from our experience in realizing ACM2FAM transformation for future model transformation developers of GReAT. The GReAT version used in this thesis is 1.7.1. Although GReAT documentation contains a fair amount of information and samples on how to use GReAT in defining transformations we have found out that it was not clear enough on some crucial points or contained missing information. In addition to that, GReAT's error messages are often not very informative and even worse, the system occasionally crashes after encountering errors. Thus we expect that the tips and the explanations on the pitfalls presented in here would be very valueble for the prospective GReAT developers.

D.1 Defining Cross-Links

It is often the case in model transformations that maintaining references between the different models is necessary. Moreover, it is usually required to maintain temporary information that may correspond to both source and target paradigms. Such problems are tackled in GReAT by using an additional domain to represent all the cross-domain links and temporary links. In GReAT users can create a `Package` for describing the cross-links. In the package the users can drag references to classes in other packages and create new association types

Cross-links can be defined not only between different domains but can also be used to extend a domain to provide some extra functionality required by the transformation. By using a different domain/package for cross-links we are able to specify a larger, heterogeneous domain that encompasses all the domains and cross-references. This model extension capability can be very handy in defining the ACM2FAM transformation, but care must be spent during its usage.

We tried to utilize the cross-links mechanism to annotate the metamodels with extra model elements for facilitating the transformations, but unexpected errors thrown at run-time later proved it useless. An example to this from our case study was the introduction of the `NonDurableMsg` type to represents the whole family of durable messages in the model.

It was decided in ACM2FAM transformations to transform every `DurableData_Msg` messages into HLA object classes and all the other messages into interaction classes. Every ACMM message extends from the `FAMessage` abstract super type, including the `DurableData_Msg`. However, there was initially no super type to represent "Non-Durable" messages. The existence of `DurableData_Msg` enabled us to produce all of the object classes with a simple, straightforward rule definition. On the other hand, without a "Non-Durable" super type, pattern matching for the production of interaction classes would be cumbersome, with many similar, but distinct rules. An alternative to that was a single rule with a fairly sophisticated guarding mechanism to distinguish among the FAMessages – not a better solution either.

We introduced the `NonDurableMsg` abstract element, which is not part of the ACMM, in a new cross-links package in the ACM2FAM model to gather all the messages having non-durable nature under a single super type. A single transformation rule that catched any non-durable message sufficed to create the stubs of all the interaction classes in the target model, just as in the case of object class generation. Things went fine during transformation rule definition until it came for testing.

When we introduced the new message type inside a class diagram that is not under the ACMM package (e.g., a cross-links package), then the GReAT master interpreter threw a "Buffer overrun error", leading to the crash of the execution and corruption of the transformation file, with an abnormal exit. The sequence of error messages are presented in Figure D.1.

Only after defining the element inside the ACMM package, the transformation worked. The new hierarchy definition then could be inside cross-links, or ACMM, it did not matter. What is important is that the new element must be defined under ACMM package. (We believe that this provides a namespace for the new element, which is mandatory for all of the patterns that are used in transformation definitions.).

Our lesson learned was to spare cross-links usage for only defining associations between the source and target metamodels, not introducing new model elements. After this incident we modified ACMM to accommodate the new `NonDurableMsg` element there.

D.2 Role Names and Cardinalities in Cross-Links

If new associations are defined as cross-links between the source and target metamodel elements and roles and cardinalities are given to both association ends, as seen in Figure D.2, then the associations have to be used with exactly the same role names (and cardinalities) later in rule definitions. Otherwise, the transformation crashes at runtime,

giving a “FACM2FAM-gr.dll don’t exist” error. This error never goes away unless the association usage is corrected and the GReAT Master Interpreter is re-run (to regenerate everything). Note that whenever we modify something in CrossLinks, we have to re-run GReAT Master Interpreter, since this is counted as part of metamodel.

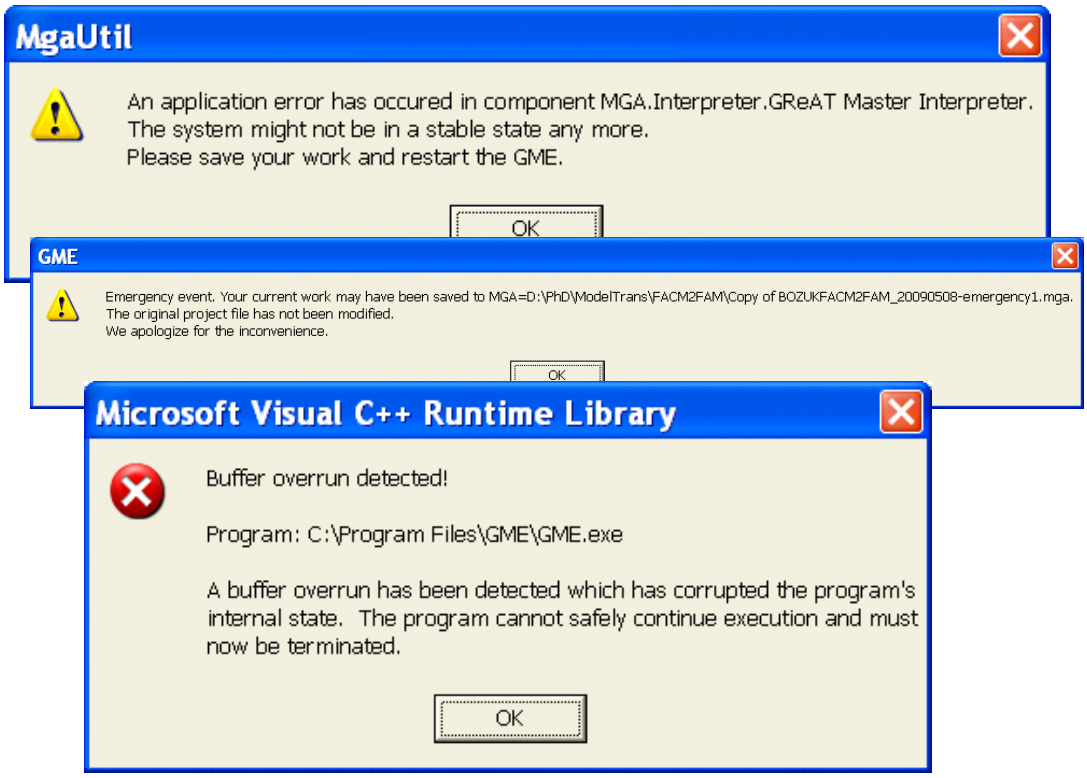


Figure D.1 Errors thrown by GR-engine when using cross-links to define model elements

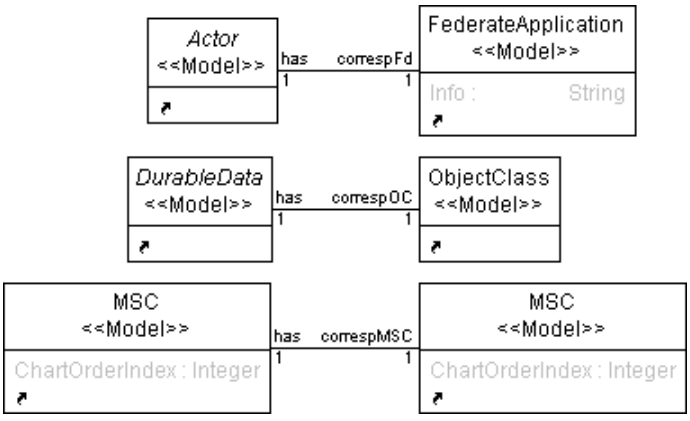


Figure D.2 Sample cross-links

D.3 Working with Globals

In order to easily access an element not found in a rule pattern, a "Global" object is defined and an association between the element and the global object is created. A global object can directly be accessed anywhere within transformation rules. Through a global object other model elements can be accessed, provided that they are defined to be associated with the global object in a separate package, and that a `CreateNew` binding is established between them in a previous rule, before being accessed. Please refer to Figure D.3 and Figure D.4 for a sample global object definition and usage.

D.3.1 Rules for Defining Global Objects

The rules listed below should be followed in defining and using global objects:

1. Define a package under the root folder. Set the `Temporary` attribute of the package to `True`. (This is mandatory for globals to work!)
2. Define a class diagram under the package created in 1.
3. Drag and drop an object of kind `Class` to the class diagram from the part browser.
4. Create as many `ClassCopy` objects into the class diagram as needed. Make those copies refer to the necessary model elements in the source or target metamodels.
5. Establish associations between the global class and the class copies. Make sure to give valid and unique role names to association ends. Also set both `src` and `dst` cardinalities, where `0..1` is usually what is needed.

D.3.2 Defining Multiple Global Objects

It is possible to define as many global objects as wished; however, there is a crucial point to take into consideration in doing so: Instead of defining extra class diagrams into the previously defined package, define a separate package and define the new class diagram with the new global object under the new package.

Otherwise, GReAT mistakenly disregards at least one of the global objects that are under the same package, but in different class diagrams. This could be observed as an `ERROR` in the `Translator.log` file if the `CodeGen` interpreter is run, indicating that one of the global objects that is used in rules are unbound. (Actually, in our exercise, it was the first `GlobalRoot`, not the newly defined `GlbHlaMeths`). Consequently, all of the associated objects with the global object in question are also reported to be unbound. Although in this situation, GReAT execution completes, desired results are not obtained.

We have not tried to define extra global objects in the same package and in the same class diagram, but have the feeling that it would most probably work correctly. However,

limiting oneself to a single package and class diagram would quickly clutter the diagram sheet and it would be impractical to define many items there.

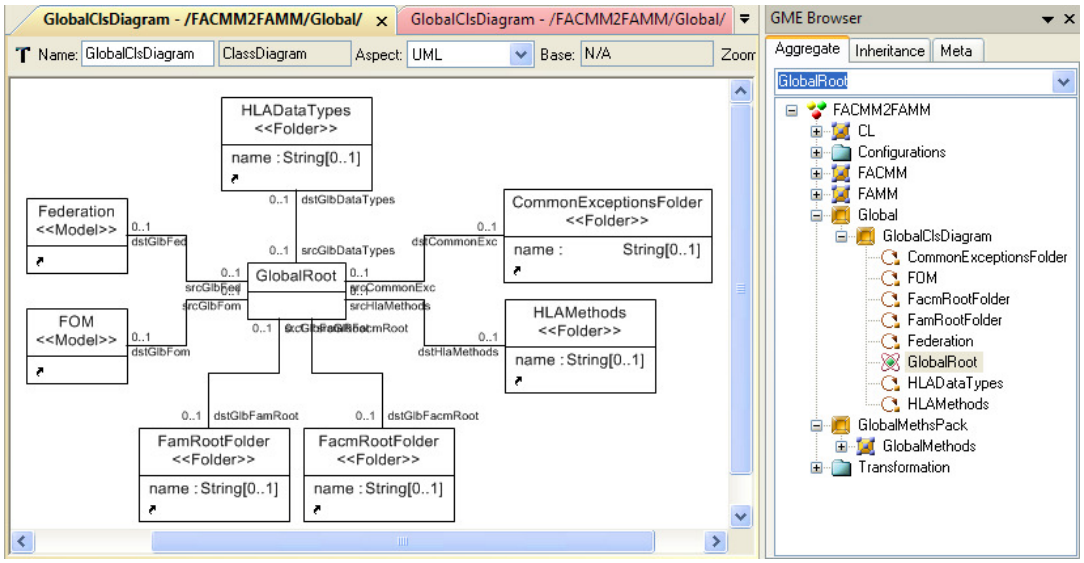


Figure D.3 Global object definition in GReAT/GME

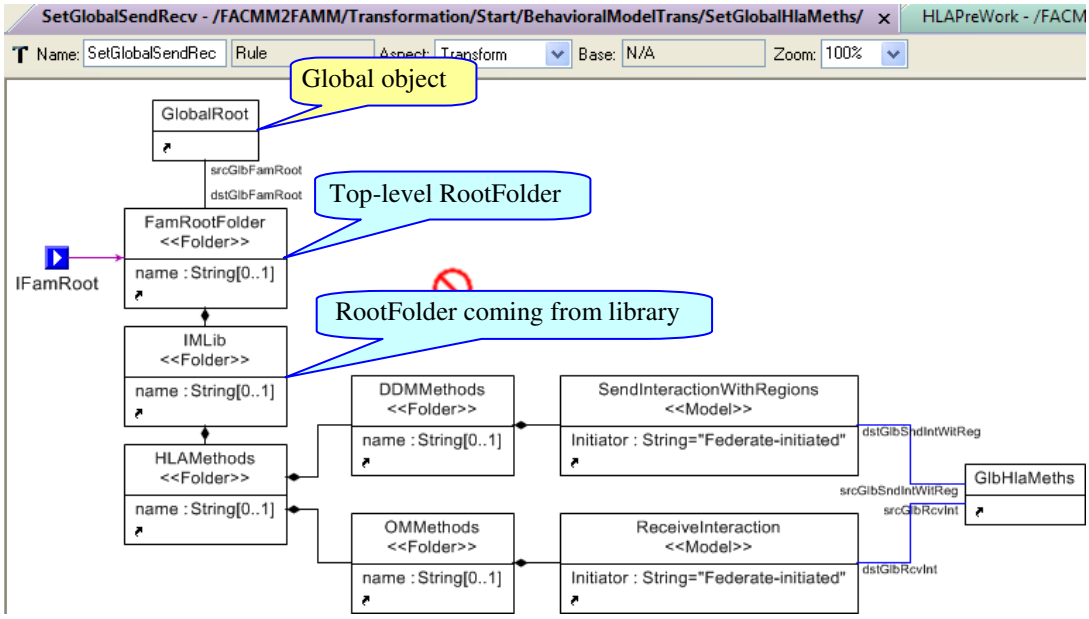


Figure D.4 GReAT rule showing two global objects and a library usage

D.4 Library Usage in Models

A previously defined model could be imported and used as a library in another newly defined model, provided that the latter model's metamodel is a superset of the former model's metamodel. For example a model of HOMM can be attached as a library in a HFMM or FAMM model. Please refer to Figure D.4 and Figure D.5 for sample library usages in a transformation definition and in a FAM, respectively.

D.4.1 Rules for Attaching a Library

The rules listed below should be followed in attaching a library to a model:

1. Open a model in GME editor. Right click the root folder and select "Attach Library" menu item from the context menu.
2. Select the .mga file of the library model from the opened file selector, and press OK.
3. The model is seen attached, as is, under the root folder, marked with a booklet icon.
4. Note that the library item indicated with the icon is also a (subordinate) `RootFolder` type, such as the already existing, system provided, top level `RootFolder`.
5. In accessing the library's child objects during model transformations, make sure to indicate (as a pattern) the root folder coming from the library under the top level root folder (see Figure D.4).

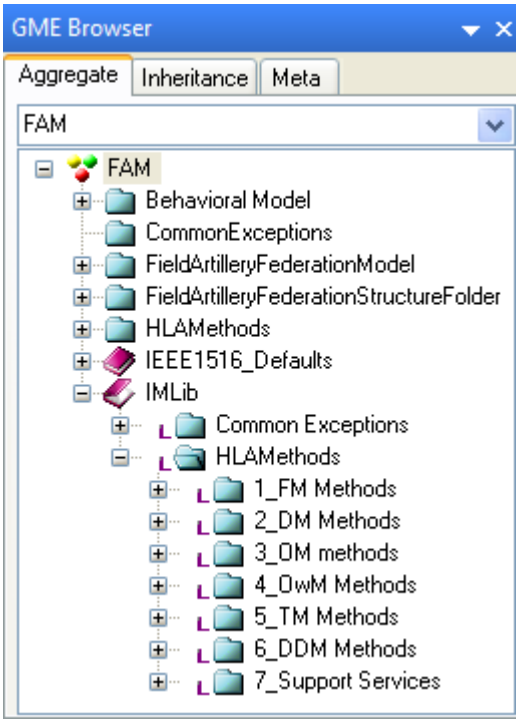


Figure D.5 IMLib and IEEE1516_Defaults used as libraries in a FAM model

D.4.2 Crashing of GReAT during Library Import

In the default GReAT/GME configuration, we receive a "Buffer Overrun" exception similar to the one shown in Figure D.1 if we try to attach large libraries such as LMM into, for example, ACMM or FAMM. We occasionally need such library updates after we modify the underlying libraries in the models. The solution to this problem was provided by the GReAT development team when we had reported the issue to them.

There is an add-on in GME that runs in the background when a `MetaGME` project, such as ACMM or FAMM, is edited. Its role is to turn the abstract attribute of `FCO`s to `true`, anytime an `FCO` is added. This add-on called `MetaMAid` is causing the problem. We are advised to turn off that add-on while a project is open and then attach the library. The add-on is turned off by selecting `MetaMAid` from the list provided by the `File`→`Register Components` menu item and then pressing the `Toggle` (`Disable` since GME 11) button. Also, the '`Systemwide`' radio button must be selected ('`For user only`' is selected by default) in the `Register` radio group in order for the operation be effective. After the library is attached, we can turn back the add-on if we want to. Future releases of GME might solve this problem.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Özhan, Gürkan
Nationality: Turkish (TC)
Date and Place of Birth: 18 August 1976, Tarsus
Marital Status: Married, two daughters.
Phone: +90 532 3102067
Email: gurkanozhan@gmail.com

EDUCATION

Degree	Institution	Year of Graduation
MS	METU Computer Engineering	2001
BS	METU Computer Engineering	1998
High School	Tarsus American School	1994

WORK EXPERIENCE

Year	Place	Enrollment
2008-Present	NATO C3 Agency	Senior Scientist
2004-2008	STM A.Ş.	Software Team Leader
2002-2004	Havelsan A.Ş.	Software Engineer
2001-2002	Cybersoft A.Ş.	Software Engineer
1998-2001	METU Computer Engineering	Research Assistant

FOREIGN LANGUAGES

English, Dutch, German

PUBLICATIONS

Journals

1 **G. Özhan** and H. Oguztüzün, Graph-based transformation of conceptual models to executable high level architecture federation models, (under review).

2 **G. Özhan**, H. Oguztüzün, P. Evrensel, Modeling of field artillery tasks with Live Sequence Charts, The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology (JDMS), vol. 5, no. 4, pp. 219–252, October, 2008. DOI: 10.1177/875647930800500402.

International Conferences

1. **G. Özhan**, H. Oguztüzün, Generating Simulation Code From Federation Models: A Field Artillery Case Study, in: European Simulation Interoperability Workshop, 11E-SIW-007, The Hague, Netherlands, June, 2011.

2. **G. Özhan**, A.C. Dinç, H. Oguztüzün, Model-integrated development of field artillery Federation Object Model, in: Second International Conference on Advances in System Simulation (SIMUL), pp.109–114, Nice, France, August, 2010.

3. **G. Özhan**, H. Oguztüzün, Model-Integrated Development of HLA-Based Field Artillery Simulation, in: European Simulation Interoperability Workshop, 06E-SIW-027, pp. 187-196, Stockholm, Sweden, June, 2006.

National Conferences

1. **G. Özhan,** H. Oguztüzün, Topçu Bataryası İçin Kavramsal Modelleme Ve Uygulamaları (Conceptual Modeling for Field Artillery Battery and Its Applications), in: 2. Ulusal Savunma Uygulamaları Modelleme ve Simülasyon Konferansı – USMOS'07 (2nd National Conference on Defense Applications of Modeling and Simulation), pp. 437-446, Ankara, Turkey, April, 2007.

2. **G. Özhan**, H. Oguztüzün, N. E. Özdemirel, Olay Çizgeleriyle Simülasyon Modellemesi (Simulation Modeling with Event Graphs), in: YA/EM'01 Yöneylem Araştırması ve Endüstri Mühendisliği XXII. Ulusal Kongresi, p. 55, Ankara, Turkey, July, 2001.

Thesis

1. **G. Özhan**, Developing a Discrete Event Simulation Engine with Concurrent Constraint Programming, MSc Thesis, Department of Computer Engineering, Middle East Technical University (METU), Ankara, Turkey, January, 2001.

HOBBIES

Reading, traveling, swimming