METAMODELING FOR THE HLA FEDERATION ARCHITECTURES

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

BY

OKAN TOPÇU

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

DECEMBER 2007

Approval of the thesis:

METAMODELING FOR THE HLA FEDERATION ARCHITECTURES

Submitted by **OKAN TOPÇU** 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. Volkan Atalay
Head of Department, **Computer Engineering** _____

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

Examining Committee Members:

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

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

Prof. Dr. Levent Kandiller
Industrial Engineering Dept., Çankaya University _____

Assoc. Prof. Dr. Ali Doğru
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Cem Bozşahin
Computer Engineering Dept., METU _____

Date: _____

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

Name, Last name : Okan Topçu

Signature :

ABSTRACT

METAMODELING FOR THE HLA FEDERATION ARCHITECTURES

Topçu, Okan
Ph.D., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Halit OĞUZTÜZÜN

December 2007, 194 pages

This study proposes a metamodel, named Federation Architecture Metamodel (FAMM), for describing the architecture of a High Level Architecture (HLA) compliant federation. The metamodel provides a domain specific language and a formal representation for the federation adopting Domain Specific Metamodeling approach to HLA-compliant federations. The metamodel supports the definitions of transformations both as source and as target. Specifically, it supports federate base code generation from a described federate behavior, and it supports transformations from a simulation conceptual model. A salient feature of FAMM is the behavioral description of federates based on live sequence charts (LSCs). It is formulated in metaGME, the meta-metamodel for the Generic Modeling Environment (GME).

This thesis discusses specifically the following points: the approach to building the metamodel, metamodel extension from Message Sequence Chart (MSC) to LSC, support for model-based code generation, and action model and domain-specific data model integration.

Lastly, this thesis presents, through a series of modeling case studies, the Federation Architecture Modeling Environment (FAME), which is a domain-specific model-building environment provided by GME once FAMM is invoked as the base paradigm.

Keywords: Domain Specific Architectures, High Level Architecture, Metamodeling, Generic Modeling Environment, Live Sequence Charts

ÖZ

HLA FEDERASYON MİMARİLERİ İÇİN METAMODELLEME

Topçu, Okan
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Halit OĞUZTÜZÜN

Aralık 2007, 194 sayfa

Bu çalışma, Yüksek Seviye Mimarisi (HLA) uyumlu bir federasyon mimarisinin tanımlanabilmesine olanak sağlayan ve Federasyon Mimari Metamodeli (FAMM) olarak adlandırılan bir metamodel önermektedir. Önerilen metamodel, Alan Özel Metamodelleme yaklaşımının HLA uyumlu federasyonlarına uyarlanmasıyla federasyon için biçimsel bir gösterim ve uygulama alanına yönelik bir dil sağlamaktadır. Metamodel, transformasyonlarıın tanımlanmasını hem kaynak hemde hedef model olarak desteklemektedir. Özellikle, simülasyon kavramsal modelinden transformasyon yapılmasını ve tanımlanmış federe davranışlarından federe temel kodunun üretilmesini desteklemektedir. FAMM'ın öne çıkan özelliği Canlı Sıralama Çizelgelerine (LSC) dayalı olarak federelerin davranışlarının tanımlanabilmesine olanak vermesidir. Jenerik Modelleme Ortamının (GME) meta metamodeli olan MetaGME kullanılarak oluşturulmuştur.

Bu tez özellikle şu noktaları tartışmaktadır: metamodel oluşturulmasında ki yaklaşımlar, Mesaj Sıralama Çizelgelesinden (MSC) LSC'lere metamodelin genişletilmesi, model tabanlı kod üretimi için sağlanan destek ve aksiyon modeli ile alan özel veri modelinin bütünleştirilmesi.

Son olarak, bu tez, FAMM'ın temel model olarak çağrılmasıyla GME tarafından sağlanan alan özel model oluşturma ortamı olan Federasyon Mimarisi Modelleme Ortamını (FAME) bir dizi örnekle desteklenmiş olarak sunmaktadır.

Anahtar Kelimeler: Alan Özel Mimariler, Yüksek Seviye Mimarisi, Metamodelleme, Jenerik Modelleme Ortamı, Canlı Sıralama Çizelgeleri

To My Family

ACKNOWLEDGMENTS

> If I have seen further, it is by standing on
> ye shoulders of Giants.
>
> – Isaac Newton

I express sincere appreciation to Assoc. Prof. Dr. Halit Oğuztüzün for his guidance, unique support, and insight throughout the research.

Thanks go to Assoc.Prof.Dr. Ali Doğru and Prof.Dr. Levent Kandiller for their valuable supervision during my thesis.

I also would like to thank Gürkan Özhan, Ayhan Molla, Kaan Sarıoğlu, Burak Yolaçan, and Deniz Çetinkaya for many fruitful discussions. In particular, collaborative work with the fellow PhD student Mehmet Adak, who developed the HLA federate code generator, has provided invaluable feedback.

Finally, I cannot say enough to express my gratitude to my wife for her endless patience during my thesis study and for her support and assistance in every aspect of my life.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

API	Application Programmer's Interface
BMM	Behavioral Metamodel
BNF	Backus-Naur Form
BOM	Base Object Model
CIOMLib	Console Input Output Model Library
CM	Conceptual Model
DDM	Data Distribution Management
DM	Declaration Management
DMLib	DMSO 1.3 Methods Library
DMSO	Defense Modeling and Simulation Office
EnviFd	Environment Federate
ExPFd	Exercise Planner Federate
FAME	Federation Architecture Modeling Environment
FAM	Federation Architecture Model
FAMM	Federation Architecture Metamodel
FCO	First Class Object
FDD	FOM Document Data
FEDEP	Federation Development and Execution Process
FedMonFd	Federation Monitor Federate
FM	Federation Management
FOM	Federation Object Model
FRG	Federation Rapid Generation
FSMM	Federation Structure Metamodel
GME	Generic Modeling Environment

HDefLib	HLA Defaults Library
HeliFd	Helicopter Federate
HMOMLib	HLA MOM Library
HOMM	HLA Object Metamodel
HFMM	HLA Federation Metamodel
HLA	High Level Architecture
HMSC	High Level MSC
HSMM	HLA Services Metamodel
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineers
IMLib	IEEE 1516 HLA Methods Library
ITU-T	International Telecommunication Union
KnoxFd	Knox Federate
LMM	LSC Metamodel
LSC(s)	Live Sequence Chart(s)
MSC(s)	Message Sequence Chart(s)
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MekoFd	Meko Federate
MetaGME	GME Metamodel
MIC	Model Integrated Computing
MMM	MSC Metamodel
MOM	Management Object Model
M&S	Modeling and Simulation
NSTMSS	Naval Surface Tactical Maneuvering Simulation System
OCL	Object Constraint Language
OM	Object Management
OMG	Object Modeling Group

OMT	Object Model Template
OwM	Ownership Management
PIM	Platform Independent Model
PSM	Platform Specific Model
PSMM	P/S Metamodel
P/S	Publish/Subscribe
pRTI	Pitch RTI
RTI	Runtime Infrastructure
ShipFd	Ship Federate
SOM	Simulation Object Model
STMS	Strait Traffic Monitoring Simulation
TacPicFd	Tactical Picture Federate
TM	Time Management
UML	Unified Modeling Language
V&V	Verification and Validation

CHAPTER 1

INTRODUCTION

This chapter presents the motivation and background for the study, presents the development context and objectives, discusses the related work, and then provides an overview of the relevant technology and tools used during the study.

1.1 Motivation and Background

High Level Architecture (HLA) provides a framework for distributed simulations with special emphasis on interoperability and reusability of simulation components [1, 2, 3]. It became a widely accepted standard in the area of distributed modeling and simulation over the last decade, and it is not surprising to see that the majority of new distributed simulations in both the civilian and military context are being built to be HLA compliant while HLA itself evolves. Although much effort has been spent on developing HLA federations, the state-of-the-art in federation design representation and documentation still does not provide adequate support for full automation of the federation development process with user guidance [4].

The Federation Development and Execution Process (FEDEP) [5] assists and guides the activities of developing an HLA federation. Although it has defined some design activities, it has left design notations and documentation methods to the designers. With respect to the HLA object model, the Object Model Template (OMT) standard [3] is adequate for representing the static view of a federation. OMT, however, does not attempt to capture the dynamic view of a federation or member federates (e.g., creation/deletion of object instances and regions, transfer/accept ownership of instance attributes).

This thesis proposes a metamodel for specifying the architecture of an HLA-compliant federation by adopting the Domain Specific Metamodeling approach to facilitate tool support for federation development. The metamodel treats the structural and dynamic views of a federation on equal footing. The dynamic view of

a federate is tantamount to its interactions with the HLA Runtime Infrastructure (RTI), the middleware implementing the HLA Interface Specification. The dynamic view of the federation emerges as the joined federates interact with each other over the RTI as the federation execution unfolds.

Model Driven Engineering (MDE) is a promising approach in software industry and academia, which views the system development as a series of models and transformations among the models [6, 7]. A known MDE initiative is the Model Driven Architecture (MDA) of Object Management Group (OMG). MDA advocates separating the specification and the implementation of a software-intensive system, in terms of Platform Independent Model (PIM) and Platform Specific Model (PSM), respectively. Most prominently, MDA promotes automated transformations between models. In particular, the PIM of a system to be constructed is to be transformed into a PSM. Automated tools, then, could carry out code generation from a PSM.

An earlier manifestation of MDE is Model Integrated Computing (MIC). As stated in [8], MIC relies on metamodeling to define domain-specific modeling languages and model integrity constraints. The domain-specific language is then used to automatically compose a domain-specific model-building environment for creating, analyzing, and evolving the system through modeling and generation [9].

The proposed metamodel, Federation Architecture Metamodel (FAMM), provides a domain-specific language for the formal representation of the federation architecture. Serving both as a source and a target, the metamodel supports the definitions of transformations. Specifically, it supports federate base code generation from a described federate behavior and transformations from a simulation conceptual model (which could be regarded as a PIM).

1.2 Development Context

To elucidate the purpose and the use of the metamodel, we clarify the development context where this metamodel fits by articulating a methodological view emphasizing models and transformations. Adopting the MDE approach, development steps can be seen as a series of model transformations. In our view, HLA-based distributed simulation development basically is comprised of a conceptual model, federation architecture model, detailed design model, and federation (in some executable form). Figure 1 sketches the roles of the models. Each model layer corresponds to a distinct level of abstraction, for example, the

conceptual model layer pertains to domain entities while the detailed design model layer pertains to software objects.

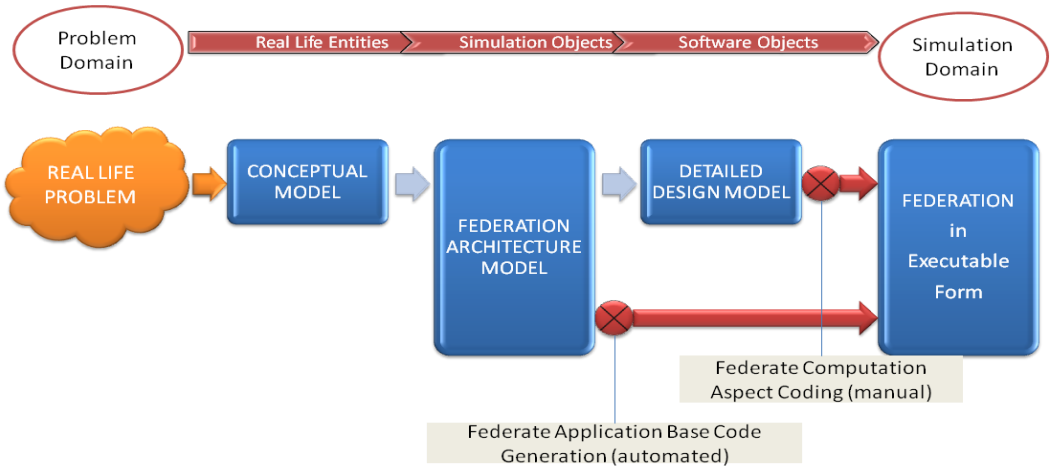


Figure 1. Development Methodology for HLA-Based Distributed Simulations

The *Conceptual Model (CM)* is a PIM of the reality with which the simulation is concerned. A CM can also address simulation capabilities and limitations. It serves as an agreement among project stakeholders about what is to be developed and represents how developers understand the problem domain. Conceptual models and their validation, with respect to the presented methodology, are discussed in [10].

The *Detailed Design Model* outlines the internal structure (computational model) of the federate components. It helps generate the software skeleton for the computational part.

Federation Architecture Model (FAM), which is the main concern of this thesis, is a major portion of the federation design documentation. Federation design for HLA based distributed simulations includes the following activities:

- Forming a federation object model and possibly simulation object models:
  - Designing static information interests of federates (related to declaration management interface),
  - Designing dynamic information interests of federates (related to object management interface),

- Designing dynamic object flows (related to data distribution and ownership management interfaces),
- Designing synchronization scheme (related to time management interface)

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

The Federation Architecture[1] is a PSM where, in our case, the platform is the RTI, and it comprises the Federation Model (Federation Structure, Federation Object Model, and HLA Services) and the Behavioral Models for each participating federate. The model of a particular federation architecture conforms to the Federation Architecture Metamodel.

Both tasks, metamodeling and modeling, are accomplished using the Generic Modeling Environment (GME) developed and maintained by Institute for Software Integrated Systems at Vanderbilt University, as a tool to put the MIC vision into practice. Formalism is due to the conformance of FAMM to the MetaGME, the metamodel (i.e., meta-metamodel) provided by GME. GME is an open source modeling tool that supports domain-specific modeling, where, in our case, the domain is HLA [8, 11]. GME initially serves as a metamodel development environment for domain analysts, and then, based on the metamodel; it provides a domain-specific model-building environment for the developers. GME is chosen in this study both for being open source and academic research tool, and for providing Application Programmer's Interface (API) (i.e., the GME generic BON2 API) for model traversing and manipulation to develop model interpreters[2]. For the complete characteristics of the GME why it is chosen in (meta)modeling, see [12]. It is also worth to note that whenever a minor bug is found in GME, a fix is received immediately.

"Domain specific (meta)modeling is an approach to modeling that emphasizes the terminology and concepts specific to the domain [13], where data types and logic are abstracted beyond programming" [12]. OMG introduces a four-layer metamodel hierarchy for defining modeling, metamodeling, and meta-metamodeling languages and activities in [14]. Table 1 relates the HLA Federation Architecture Metamodel to

[1] "Federation Architecture" is used as replaceable with "Federation Architecture Model".
[2] A model interpreter is a plug-in software component to traverse and interpret a domain-specific model in GME.

OMG's four-layer metamodel hierarchy. Each layer is an instance of the layer above. For example, a particular federation architecture is an instance of FAMM.

Table 1. FAMM Correlated with OMG's Four-layer Metamodel Hierarchy

OMG's Four-layer Metamodel Hierarchy	**Related Model**
Meta-Metamodel (M3 Layer)	GME Metamodel (MetaGME)
Metamodel (M2 Layer)	Federation Architecture Metamodel (referred to as a "paradigm" in GME vernacular)
Model (M1 Layer)	Federation Architecture of a particular federation (e.g., Strait Traffic Monitoring Federation Architecture)
Run-time Instance (M0 Layer)	Federation Runtime Instance (e.g., a particular execution of the Strait Traffic Monitoring Federation. For instance, Bosporus Federation, which is given as an example in this thesis)

1.3 Related Work

Recently, there have been numerous calls to apply MDA to HLA-based distributed simulations, see, for example, [15, 16, 17]. These papers provide an account of the tenets of MDE/MDA and the potential benefits of applying it to simulation development. Some go on discussing how these benefits can be achieved.

In the scope of this thesis, we indeed do not argue for the desirability of applying MDE to HLA, instead, we rely on the cogent arguments made in the relevant literature. We adopt the view that model integrated computing is well suited for HLA development and we go on to actually build a workable domain-specific metamodel to realize the vision, where the domain is HLA. Contrariwise, our contribution is unique in that it is the first metamodel fully accounting for HLA compliant federation architectures from both static and dynamic points of view. Hence, a point-by-point detailed comparison with the earlier literature does not seem fruitful.

Some recent studies [18, 19], albeit limited in scope, represent attempts at building and utilizing metamodels to realize the potential of MDA/MDE. One is the Capsule study which aims to apply the MDE methodology to the simulation study. In this work, a metamodel for HLA related to the other simulation platforms (e.g., LIGASE and ESCADRE) is created, but the metamodel is not intended to be a universal metamodel of HLA, rather it is specific to its intended project and apparently not appropriate for other needs [18]. In the words of the authors,

> Please note that the HLA Meta-PSM presented here is not the unique meta-representation of HLA, but it is the most suitable one to be aligned with the other two simulation platforms for a generic meta-modeling in the COCA study. That's why this representation is not the universal meta-model of HLA, and may not be appropriate for other needs.

The other one [19] mentions a metamodel for HLA without elaboration, but it seems geared toward the needs of a specific project. They discuss, in a paragraph, their "FRG (Federation Rapid Generation) metamodel", which is based on the HLA metamodel. Further, "One of the main components of the FRG is the HLA framework called RAL, which stands for RTI Abstract Layer". These remarks, in connection with other information in the paper, lead us to think that their metamodel is also project specific. Please, note that we make a point of not providing any abstraction layers on top of RTI. This would be effectively redefining the standard interface within the confines of a particular organization. We understand that some developers can find this approach convenient for their specific projects. They can still benefit from our modeling technique for bringing in their own metamodels.

The most important gap in the literature we reviewed is the lack of modeling formalisms specifically addressing the behavioral description of HLA-compliant federates. Sharing the MDE vision put forth in the recent literature, we lay the groundwork to realize it. This thesis propounds a full-fledged metamodel for federation architectures. The metamodel covers not only the static view reflected in the OMT standard, but also the proposed dynamic view, based on Live Sequence Charts (LSCs) and Message Sequence Charts (MSCs).

Providing both static and dynamic views of a federation is a tenet of the Base Object Model (BOM) as well [20]. BOMs are reusable model components and "they provide a mechanism for defining a simulation conceptual model and optionally mapping to the interface elements of a simulation or federation using HLA OMT constructs" [20]. BOM effort aims to support component-based development of simulations, starting with the simulation conceptual model, while our work is concerned with the architectural description of federations, formalized in sufficient detail to allow model-based processing -code generation, in particular. The following phrase is taken from [20].

> While events and BOMs are used to represent pattern actions, variations, and exceptions, the actual behavior modeling required in carrying out a pattern action, variation, or exception by a federate is an implementation focus, which is outside the scope of this specification.

BOM template specification extends the HLA OMT to cover the conceptual entities and events, and contains information on how such entities relate and interact with each other. The BOM can be integrated to our proposed metamodel by replacing the HLA OMT model part of the metamodel with a new BOM template metamodel. The nested structure of our metamodel supports that kind of integrations. Supporting component-based development within our metamodeling framework would be worthwhile track to pursue in later studies.

An earlier use of MSCs in the HLA realm has been reported in [21], where MSCs are used to specify the procedures to test individual federates for HLA Interface Specification compliance.

Some studies propose extending Unified Modeling Language (UML) using its extension mechanisms (a.k.a. profiling). Such a study, carried out by the author of this thesis, is *UML Profile for HLA Federation Design*, which can be seen as development of HLA-specific extensions to UML to support a more formalized and standardized description of the federation, federate design, and documentation issues [22, 23]. The work is not completed because the authors shifted stance on UML profiling approach to a more powerful metamodeling approach. Another similar study [24] presents an extension as stereotyped in Rational Rose for the HLA OMT. The model is simply another rendering of OMT, where stereotyping is a mechanism of UML profiling. This study only sketches the static OMT. For a discussion of metamodeling vs. UML Profiling, see [12].

1.4 Objectives and Scope

The main objective of this thesis is to formalize the federation architectures, so that a federation architecture can be put into a machine processable form, thereby enabling tool support for the code generation and the early verification of the federation architectures.

One of the requirements anticipated is to eliminate the limitations of OMT and FEDEP and thus to bring dynamism into the architectural descriptions.

The benefits of applying MDA to HLA-based distributed simulation area has been discussed in literature but our contribution is unique in that it is the first metamodel fully accounting for HLA compliant federation architectures from both static and dynamic points of view. While realizing this, domain specific metamodeling, which is one of the current research threads, is adopted.

The overall effort is directed towards building an HLA Federation Design and Development Framework.

As a by-product, the MSC and LSC metamodels that cover the entire standard MSC and LSC features are prepared separately from FAMM. Thus, they can be used in other research and development areas.

Moreover, FAMM can be used to extract usable views to support federation designers. Such a usable view is the Publish and Subscribe (P/S) Diagrams, which are design artifacts to focus on the object/interaction interests among the federates. From a federation architecture conforming to FAMM, the P/S models can be automatically generated using the P/S model builder developed in this study.

The metamodel presentation is accompanied by an example: the Strait Traffic Monitoring Simulation (STMS). On a larger scale, the architectural modeling of Naval Surface Tactical Maneuvering Simulation System (NSTMSS) [25], a distributed interactive simulation, is carried out using FAMM and is presented in Appendix B.

1.5 Technology Overview

1.5.1 High Level Architecture

“HLA provides a common framework and approach for distributed simulations and virtual worlds to share information and capabilities, to expand interoperability, and to promote reuse and extensibility” [26]. HLA is a set of specifications which include the HLA Rules, Interface Specification and the Object Model Template.

HLA was developed under leadership of the U.S. Defense Modeling and Simulation Office (DMSO). The HLA was approved as an open standard through the Institute of Electrical and Electronic Engineers (IEEE), namely IEEE Standard 1516, in September 2000. The standard embodies four related standards shown in Table 2.

Table 2. IEEE 1516 Standards

STANDARD	EXPLANATION
IEEE 1516-2000	IEEE Standard for Modeling and Simulation (M&S) HLA Framework and Rules.
IEEE 1516.1-2000 Errata to IEEE 1516.1	IEEE Standard for M&S HLA Federate Interface Specification. Correction Sheet issuued 16 October 2003
IEEE 1516.2-2000	IEEE Standard for M&S HLA Object Model Template Specification.
IEEE 1516.3-2003	IEEE Recommended Practice for HLA Federation Development and Execution Process

The HLA is mainly comprised of three elements:

HLA Rules: "A set of rules that must be followed to achieve proper interaction of simulations (federates) in a federation. These describe the responsibilities of simulations and of RTI in HLA federations" [1].

Interface Specification: "The HLA Interface Specification defines the interface between the simulation and the software that will provide the network and simulation management services. RTI is the software that provides these services" [2].

Object Model Template: "The OMT describes a common method for recording the information that will be produced and communicated by each simulation participating in the distributed exercise" [3].

1.5.2 Tools

1.5.2.1 Generic Modeling Environment

GME is touted as "a domain-specific, model-integrated program synthesis tool for creating and evolving domain-specific, multi-aspect models of large-scale engineering systems" in [11]. GME is an ongoing academic research project at Vanderbilt University, in which the source codes are public.

GME is used as the primary tool for both metamodels and models introduced in this thesis. Metamodels are defined in modeling paradigms using MetaGME, the GME meta-metamodel. After describing the metamodel, the GME creates a design environment for domain models once this metamodel is invoked. Then the generated design environment can be used to design domain specific models (e.g., FAM).

Currently, GME version 7.6.29 is used in this study. Figure 2 depicts a typical GME user interface.

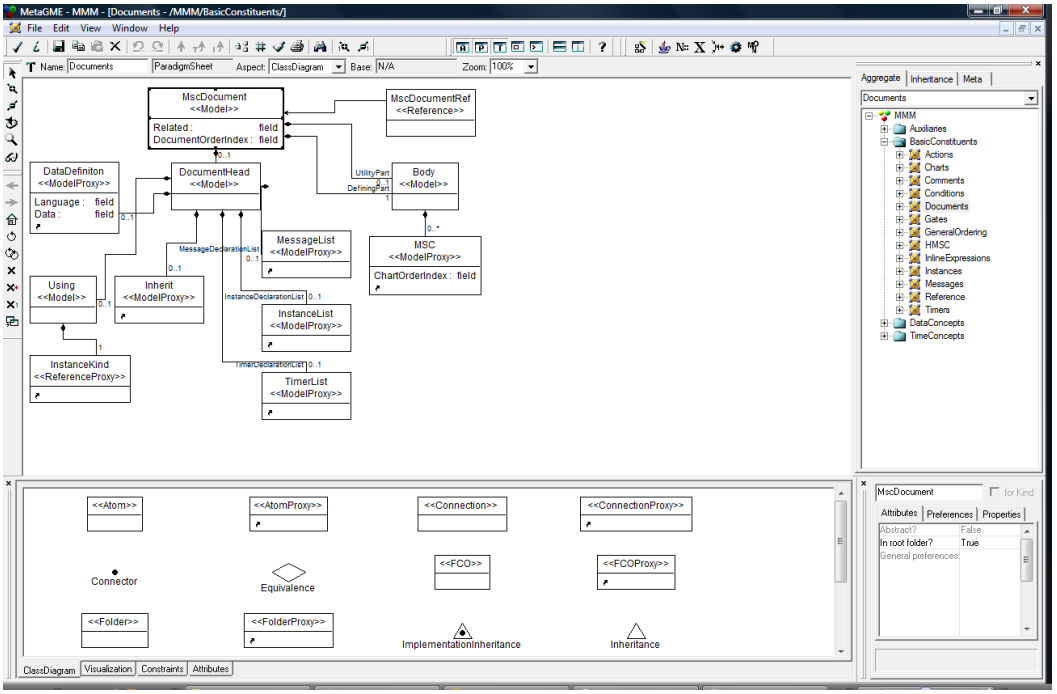


Figure 2. GME Screenshot

A detailed explanation of the GME and its concepts can be found in [11].

1.5.2.2 Microsoft Visual Studio .NET Integrated Development Environment

Microsoft Visual Studio .NET provides a complete set of development tools for building applications. Visual Basic .NET, Visual C++ .NET, Visual C# .NET, and Visual J# .NET all use the same integrated development environment (IDE), which allows them to share tools and facilitates in the creation of mixed-language solutions.

Microsoft Visual Studio .NET is used as the primary IDE tool to develop a model interpreter for the metamodels and the models introduced in this thesis. Currently, version 8.0 (2005) is used.

1.6 Typeface Conventions

This thesis uses the following typeface conventions:

- All code examples/snippets are printed in a Book Antiqua Font.

- At the first introduction or definition of a major term, the term is shown in *italics*.
- All references to classes, attributes, and other elements of a model are shown in `Courier New Font`.
- General emphasis is shown in *italics*.

1.7 Summary of Chapters

The preceding sections of this chapter outline the motivation and background of the study, presents the development context and the objectives, discusses the related work, and then gives a technology overview and tools used during the study. The remaining chapters are broken down as follows:

- Chapter 2 provides at-a-glance information about the HLA development vision, including a methodological and a lifecycle view.
- Chapter 3 through Chapter 5 expounds upon the proposed Federation Architecture Metamodel by giving detailed examples of the metamodel elements.
- Chapter 6 presents the model integration and extensibility capabilities of the proposed metamodel.
- Chapter 7 lays out an assessment for FAMM.
- Chapter 8 outlines the results achieved as a result of this work and points the way ahead.
- Appendix A explains the Federation Architecture Modeling Environment (FAME) and presents a running example: Strait Traffic Monitoring Simulation.
- Appendix B presents a modeling study, a case study with NSTMSS.
- Appendix C gives an analysis for the HLA services and presents the transition tables between the HLA services and the HLA methods library.
- Appendix D presents the details of the HLA arguments for the library developers.

CHAPTER 2

FRAMEWORK FOR HLA FEDERATION DESIGN AND DEVELOPMENT

The material in this chapter is adapted from [10]. A supporting life cycle for the framework is presented in [25].

2.1 Methodology

It is important to explicitly state the development context, where this metamodel fits, in order to clarify the purpose and the use of the metamodel. The development context is put forth by articulating a methodological view emphasizing models and transformations.

Adopting the Model Driven Engineering approach [6], development steps can be seen as a series of model transformations. In our view, HLA-based distributed simulation development basically comprises a conceptual model, federation architecture model, detailed design model, and federation (in executable form).

Figure 3 depicts the basic models. Each layer of models corresponds to a distinct level of abstraction, for example, while the conceptual model layer is related to domain entities, detailed design model layer is related to software objects.

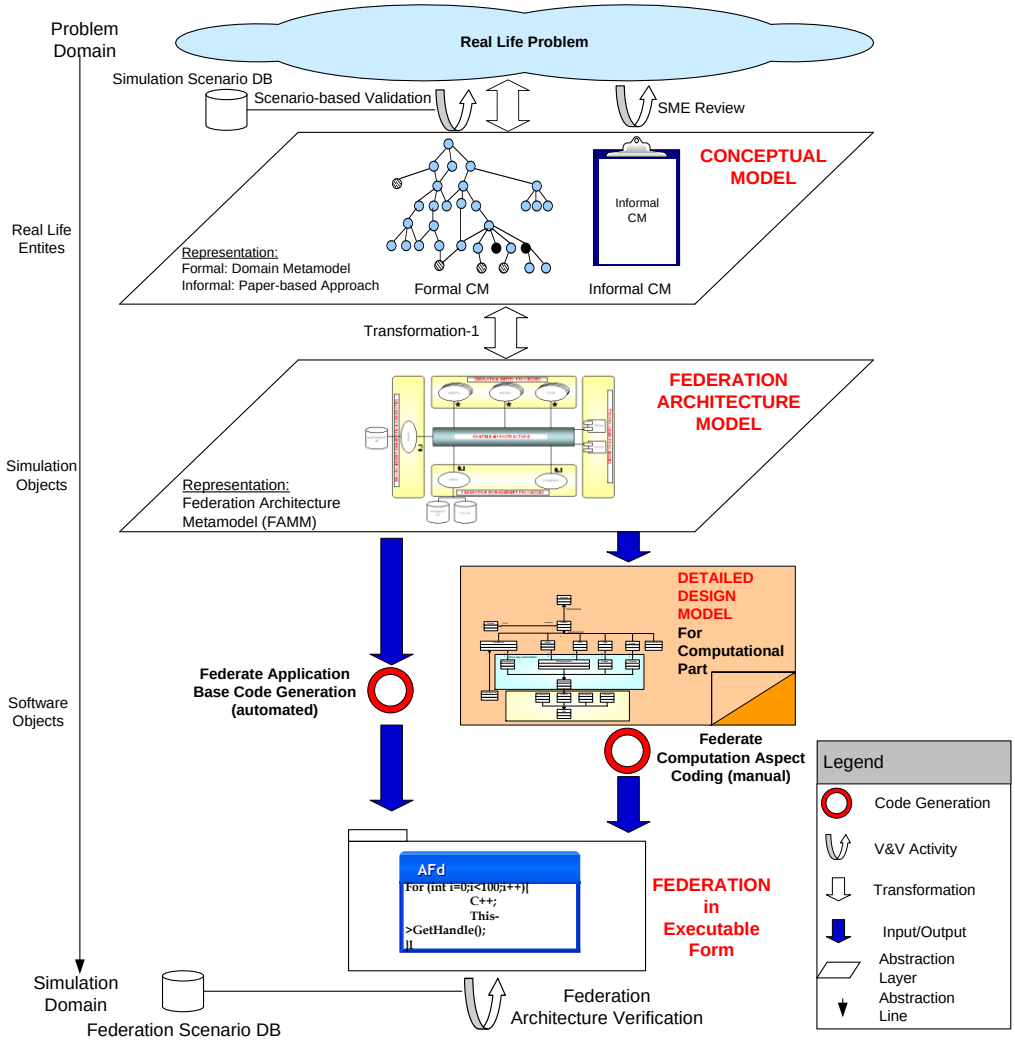


Figure 3. Development Methodology for HLA-Based Distributed Simulations

2.1.1 Conceptual Model

CM is the model of the reality with which the simulation is concerned. Simulation Conceptual Models serve a variety of purposes. From the users perspective, conceptual model provides a documentation to understand the simulation capabilities and limitations. From the developers' perspective, conceptual model serves as an agreement about what is to be developed. It represents how developers understand the problem domain. From a communication perspective, conceptual model serves as a communications link between users and developers.

The methodology suggests two representations of conceptual model; namely, informal conceptual model and formal conceptual model.

Informal CM will be used especially by the sponsor and the user group, which we can call CM users. Informal CM will help CM users to assess the capabilities and the intended focus of the simulation without any technical background. Informal CM can easily be validated by a domain expert, who generally has no technical background of the simulations and the software. We can accept this as the main technique for conceptual model validation. Meanwhile, being informal does not imply being unformatted. The scientific paper-based approach [28, 29], promoted by DMSO, can be used for representation of the informal CM.

On the other hand, formal CM representation will be directly used by the CM developers, who may modify or redevelop the conceptual model. At the same time, formal CM can be applied to solve disputes when there is an uncertainty or a disagreement in the informal CM (e.g., two people can infer different things by reading the same sentence in informal CM). Another main objective of formal CM is to transform a conceptual model into machine-processable form. In this respect, it will be possible to provide the conceptual model to all kinds of software tools (e.g., verification and validation (V&V) tools, HLA federates) and software agents (e.g. web robots). Note that it may not always be practical to formalize the entire conceptual model (e.g., CM may include some photos, charts, etc.).

Conceptual models, in the view of the presented methodology, are elucidated and discussed in [10].

2.1.2 CM Validation Using Scenarios

Scenarios can be used as a supporting validation technique for formal CM validation. Simulation requirements are captured as use cases by using use case requirements analysis techniques [30]. These use cases (a.k.a. use case scenarios) provide the main part of the simulation scenarios. Then, CM will be meaningful according to its level of support for scenarios. The meaning of support should be defined operationally within the overall problem domain. Simply, entities, actions, relationships, states, and parameters implied by scenarios should exist in CM representation.

Scenario-based CM Validation is discussed and explained in [10].

2.1.3 Federation Architecture Model

Federation Architectural Metamodel, which is the main concern of this thesis, will be discussed at length in the subsequent chapters.

2.1.4 Federation Architecture Verification

Federation architecture verification is to check that FAM does what it promises and whether it is consistent within itself.

2.1.4.1 Static Verification

The federation scenarios can be used to verify the federation architecture. The main idea is that if the federation scenarios can be “played” with the current federation architecture, then it can be asserted that the FAM is a reliable model. Playing the scenario in the design phase means static model checking (decomposition of Federation Scenario LSC into the corresponding federate HLA-specific LSCs).

Both Federation scenario(s) and FAM can be represented using LSCs. Therefore, the static model checking can be performed using the model interpretation over both LSCs where Federate LSCs must include the Federation Scenario LSC.

As seen in Figure 4, the FAM in the representation layer is used to model FOM and federate behaviors, domain scenarios in the conceptual layer are used to model the federation scenarios.

The ideas presented here for the static verification of a FAM is noted as a future work.

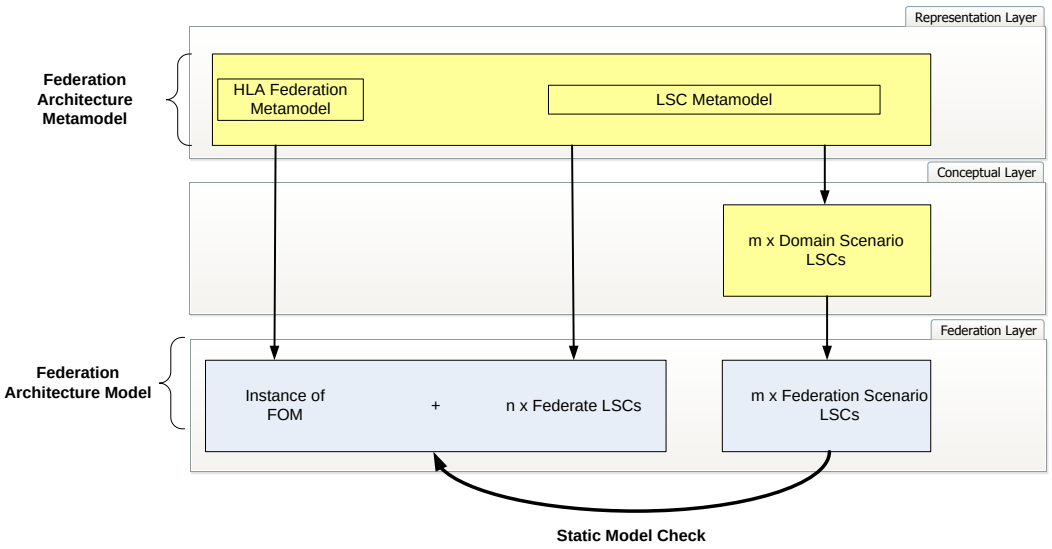


Figure 4. Static Federation Design Verification

2.1.4.2 Dynamic Verification (Runtime Verification, Monitoring)

Verification can be interpreted in the dynamic (federation execution) sense. Dynamic verification is based on the automatic code generation.

Model-based code generation for HLA federates from the given FAM that conforms to FAMM is discussed and explained in [31].

2.1.5 Detailed Design Model

Detailed design briefly depicts the internal structure (computational model) of the federate components in detail and it is the critical design effort before the implementation, and can be seen as the skeleton of the components.

At the end of the federation design activity, if the components, which compose the distributed simulation, are ready at hand, then there is no need for a detailed design. However, if federation design model implies a requirement to develop a new component or to modify an existing component, then a detailed analysis and design that is focused on the component must be conducted.

It will be a complementary approach to use object oriented analysis and design techniques and UML in designing each federate's internal structure.

A typical internal structure of a federate is recommended in [10] and this internal architecture has been applied successfully in the development of some naval federations [25 and 32].

2.1.6 Transformations and Code Generation

"A transformation is the automatic generation of a target model from a source model, according to a transformation definition, which is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language" [33]. The transformations (including code generations) defined in this framework are summarized in Table 3.

Table 3. Transformations

TRANSFORMATION	SOURCE MODEL	TARGET MODEL
Transformation – 1 (T_1)	Domain Model (e.g., a model conforming to Field Artillery Metamodel)	FAM (conforming to FAMM)
Code Generation for HLA	FAM (conforming to FAMM)	Aspect Java Code (by the generator)
Code Generation for Computational Model	FAM (conforming to FAMM)	Java Code (by the user)

FAM is being used to generate RTI related code automatically. For the computational parts, standard programming techniques can be used. Detailed design model is the major source model that helps generate the software skeleton for federate's computational part [31].

CHAPTER 3

FEDERATION ARCHITECTURE METAMODEL

FAMM provides a domain-specific language for the formal representation of the HLA-compliant federation architectures. This chapter outlines its structure, presents the behavioral metamodel and gives a user perspective.

3.1 FAMM Structure

The *Federation Architecture Metamodel* is comprised of two main sub-metamodels: the Behavioral Metamodel (BMM) for specifying the observable behaviors of the federates and the HLA Federation Metamodel (HFMM) for defining both the HLA Federation Object Model (FOM) and the service interface. These two metamodels, included as GME libraries, are connected through a GME paradigm, named `Model Integration`. The structure of FAMM is depicted in Figure 5. BMM is a logical container for the LSC Metamodel (LMM), which is extended from the MSC Metamodel (MMM). HFMM is composed of the HLA Object Metamodel (HOMM), Federation Structure Metamodel (FSMM), and HLA Services Metamodel (HSMM). Lastly, the Publish/Subscribe Metamodel (PSMM) is included as a derivative metamodel in order to illustrate the extraction of utility metamodels from the core FAMM. Once the federation architecture is modeled conforming to FAMM, a model interpreter can traverse this model to extract the federation publish and subscribe view and then display it as P/S diagrams [22, 23].

Metamodels support each other in a way that an element defined in one model can be used in other models. For example, any method parameter that occurs in HSMM is accounted for by HOMM. Nevertheless, each sub-metamodel can be used independently. This was a main concern in devising the structure of FAMM. In particular, the Metamodel for Message Sequence Charts as well as Live Sequence Charts can be used to model the MSCs/LSCs for any system of communicating components, not only for distributed simulation components. In the same vein, the

HOMM stands on its own and can be used to generate useful artifacts, such as the FOM Document Data (FDD) [34].

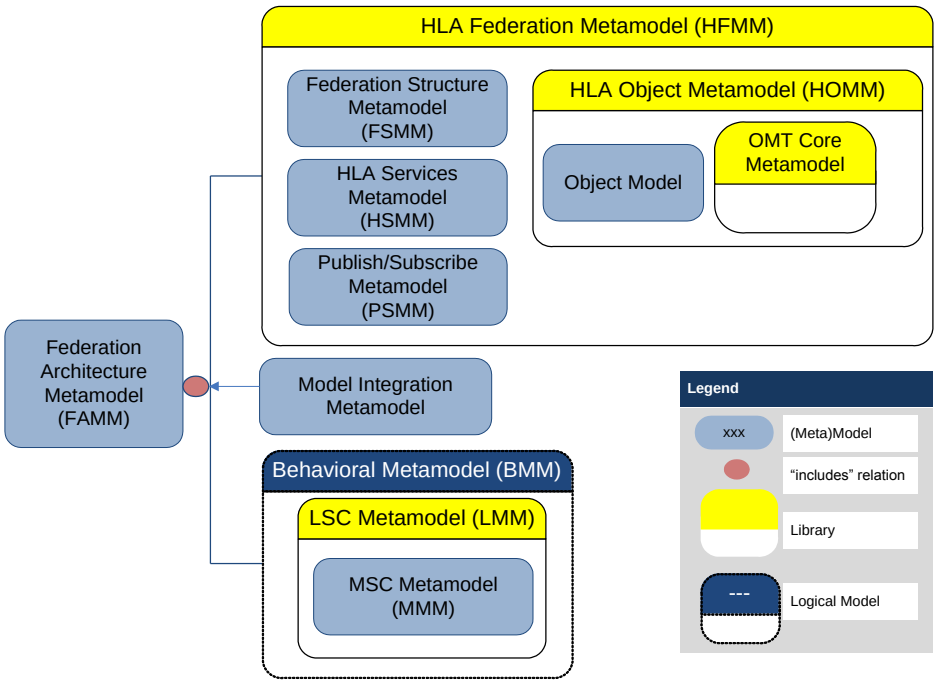


Figure 5. Federation Architecture Metamodel Structure

The implementation of FAMM in GME is depicted in Figure 6. HFMM and LMM are included as GME libraries.

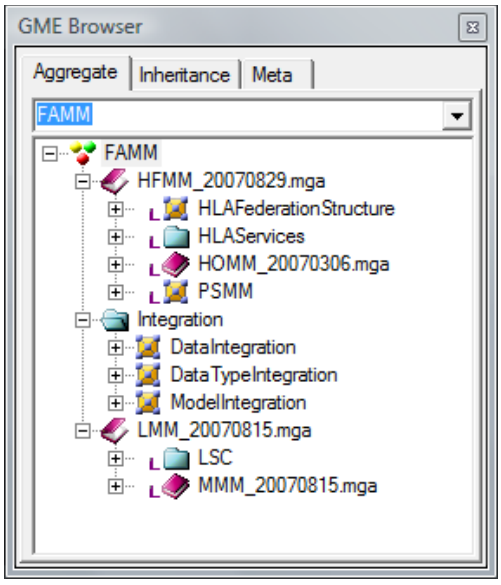


Figure 6. Federation Architecture Metamodel in GME

One may visualize a (meta)model as a graph whose nodes correspond to concepts and edges to relationships. Thus, the following table should give a rough idea about the size of FAMM. Concepts include the GME stereotypes: atom (models, atoms, FCOs, attributes, references, and inheritances), model (paradigm sheet), set (aspects), and folder. Reference stereotype (proxy) is excluded. Relationships include GME connections.

Table 4. Size of FAMM[1] and Its Sub-metamodels

Sub-metamodel	Number of Concepts	Number of Relationships
BMM	326	397
HFMM	454	369
Model Integration	20	35
Total	**800**	**801**

Figure 7 depicts the "conforms to" relationship between the Federation Architecture and FAMM. A *Federation Architecture* encompasses an object model and LSCs for each participating federate. The LSCs of a federate manifest its interaction with the RTI and possibly with other entities (e.g., users and live entities), and so they describe the federate's observable behavior. The Federation Architecture Metamodel provides the underlying language to describe the federation architectures. Each participating federate's behavior is modeled conforming to BMM and HSMM. The FOM is constructed in conformance with the HLA Object Metamodel and the Federation Structure Metamodel.

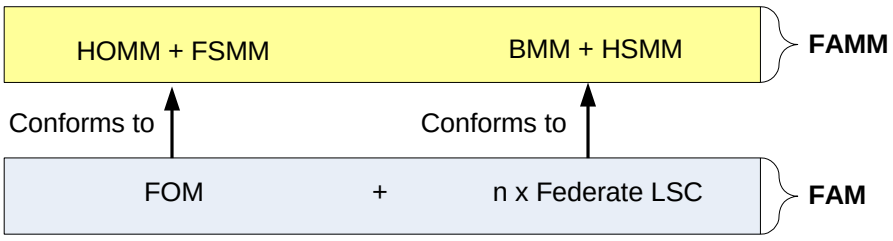


Figure 7. Relationship Between a Federation Architecture and the Metamodel

[1] For FAMM version 20071217.

3.2 User Perspective

The *Federation Architecture Modeling Environment* for users is made available by GME once FAMM is invoked as the base paradigm. The screen shot in Figure 8 shows an example-modeling environment, for FAMM users, who are typically federation designers. GME allows creation of a project for developing a new federation architecture. Figure 8 presents a screen shot of the project for the STMS federation architecture. The root folder (e.g., `StraitTrafficMonitoringSimulation` in the screen shot) serves as a project container for the federation architecture. It includes three major sub-folders, namely, federation structure, behavioral models, and federation models. The federation structure folder contains information about the federation, such as the location of the FOM Document Data file, the link for the related FOM, and the structure of the federation, where the participating federate applications and their corresponding Simulation Object Models are linked. The folder for behavioral models includes an MSC document for each participating federate. The federation model folder includes the FOM, SOMs, and the other Object Model Template related information (e.g., data types, dimensions, etc.). In the example, SOMs for ship and station applications and a FOM for the STMS federation are provided.

There are auxiliary libraries that can be readily attached to a project. Three libraries are currently provided: IEEE 1516.1 Methods Library, IEEE 1516.1 Management Object Model (MOM) Library, and IEEE1516.2 HLA Defaults Library. In the example, the methods library (designated with a book icon) is attached to the project.

Detailed explanation for FAME is provided in Appendix A.

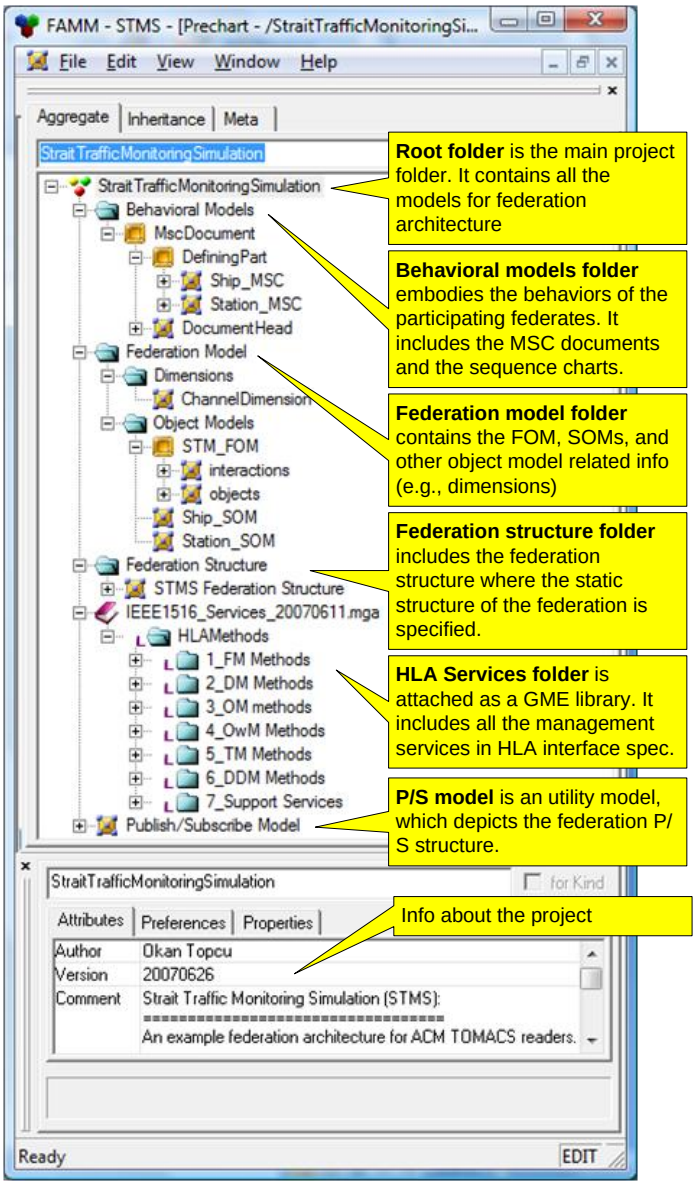


Figure 8. Federation Architecture Modeling Environment (FAME)

3.3 Behavioral Metamodel

The *Behavioral Metamodel* provides an abstract syntax for specifying the observable (primarily, as witnessed by the RTI) behaviors of a federate. Forming precise behavior models of the participating federates along with their object models gives us the ability to exercise a federation architecture. In a fully automated exercise, intra-federation communication will follow the specified patterns; the communicated values, however, will not be correct. Taking a step

towards complete federate application generation, the developer can weave the computation logic onto the generated code.

Modeling the behavior of a federate can involve not only the HLA-specific behavior (e.g., creating regions in runtime, exchanging ownership of the objects, etc.), but also the interactions between the components of the federate and the actors (e.g., interactive users and live entities) in the environment.

A fundamental decision is to adopt Live Sequence Charts, and in turn, Message Sequence Charts as the basis for the behavioral modeling of the federates. LSCs are chosen among the alternatives such as UML sequence charts and activity diagrams, to model the federate's behavior because (1) they are currently active research topic, (2) they are suitable to extend and customize, (3) LSCs allow a distinction to be made between mandatory and possible behavior, which is believed important for the behavior specification for federate and federation.

The observable behaviors of a federate are represented by means of LSCs, specialized for HLA federates. Specialization involves, in essence, formulating the RTI methods as MSC/LSC messages and integrating the HLA Object Model as the data language of MSC/LSC. Initially, MSC is formalized as the basis of the behavioral metamodel, and then LSC extensions are added on top of the MSC metamodel. Note that BMM covers all the standard MSC features [36] and the proposed LSC extensions [37] as long as they do not conflict with the MSC standard (e.g., an MSC loop is used instead of LSC iteration).

As an example, consider the graphical and textual representation of an MSC diagram presented in Figure 9, where the basic MSC elements such as instance, message, action, and condition are depicted. Here, instance `i` creates the instance `j`. Afterwards, `j` performs some initialization action and then if condition `c` is true, `j` sends a message to `i` and terminates.

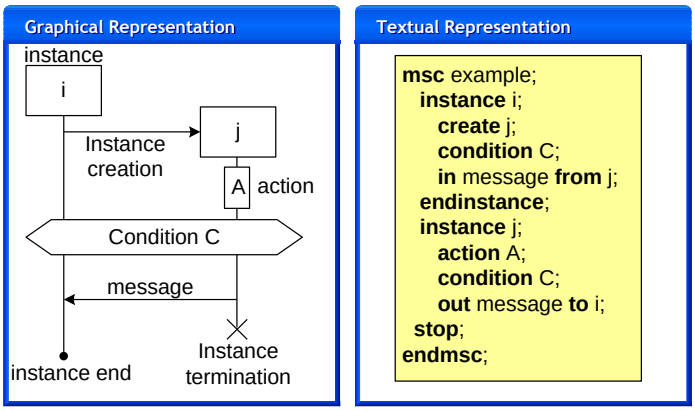


Figure 9. Graphical and Textual Representation of an MSC Diagram

The technical details of the main sub-metamodels of BMM: MSC and LSC Metamodel and the utility of MSC/LSCs in behavioral specifications are presented and discussed in Chapter 4. HLA Federation Metamodel is presented in Chapter 5. Integrating the HLA Federation Metamodel (i.e., data model) and the Behavioral Metamodel (i.e., action model) is presented in Chapter 6.

CHAPTER 4

LSC/MSC METAMODEL

One of the main objectives of visual modeling languages is to provide a representation suitable to specify, design, and analyze systems. The system representation must be precise enough to support automated processing, specifically, generation of useful artifacts, such as the source code. Modeling the observable behavior of a system is considered as an important part of the system's specification. There are some visual modeling languages that are aimed at behavior specification, such as UML Sequence Diagrams, Message Sequence Charts, and Live Sequence Charts.

In this respect, MSC, which is standardized by ITU, is a formal language that enables one to specify the interactions among the components of a system. MSCs are commonly used in the telecommunication area [35, 36] for protocol and service specification. LSCs [39] have been proposed as an extension to MSCs so as to allow distinguishing between the mandatory and the possible behavior of the system.

The proposed metamodel defines a domain-specific language for the formal representation of MSCs and LSCs. The metamodel supports the definitions of transformations. Specifically, it supports base code generation from a described behavior. The metamodel's facility of integration with domain-specific data models plays a critical role to achieve useful transformations.

To the best of our knowledge, a metamodel that would admit particular MSCs/LSCs as models, has not been put forth in the literature.

Code generation from MSC/LSC is still an on going and an open challenge for researchers. Automatic code generation plays an important role in early validation of the model after the behavior of a system is described using the MSC/LSC. Despite the fact that a play engine is proposed in [42] as an implementation

mechanism for LSC, it only provides a simulation of the execution of the LSC diagrams by playing out scenarios and thus helps testing and observing of system behaviors; but it does not attempt to generate code, and more importantly, it is not extendible due to its fixed data model, and not customizable for domain specific modeling. In contrast, our metamodeling approach, due to its data model integration capability, gives power to the user to extend or tailor his[1] application code generator or interpreter in accordance with his data model.

The metamodeling study is conducted by taking textual language for MSCs as a starting point. A feedback loop is established between metamodeling and code generator development activities. Each one proceeds in parallel: metamodeling provide the input to code generation and the latter provides the feedback to the former.

This chapter first presents and discusses the challenges in making the major modeling decisions, then MMM is presented. Extending MMM for LSC Metamodel is discussed afterwards. The material in this chapter has appeared as a technical report [55].

Examples of actual use of the metamodel elements are provided while introducing the metamodel instead of presenting them in a distinct section.

4.1 Metamodeling Approach and Design Principles

This section expounds various design decisions, principles, and metamodeling approach taken concerned with the code generation support.

As a summary, the following metamodeling design decisions and principles are taken:

- For modeling level,
- For constructing a syntax tree for each instance,
- For employing references for the design of the multi-instance elements,
- For supporting cardinality constraints by design,
- For element uniqueness,
- For dispersing abstract syntax trees for each chart.

[1] "He" and "his" is used as replaceable with "she" and "her", respectively.

4.1.1 Modeling Level

In this study, we adopt a syntactical view of metamodeling in that the metamodel serves as a metalanguage (or grammar) for the object language, which, in our case, consists of syntax for MSC/LSC based on their concrete textual syntax, which has a formal standard in case of MSC.

The semantics of a chart is defined in the standard [36] as a partial order of events. Of course, the code generation process must guarantee that any event sequence observed while the generated code is running respects the partial order specified by the chart.

Message Sequence Charts and Live Sequence Charts can be represented, in a standard way, graphically or textually. The textual form is intended to facilitate exchange between tools and to serve as a basis for automated analysis.

The MSC/LSC language definition offers two principal means for the textual description. First, an MSC/LSC can be described by giving the behavior of all instances separately, which is called *instance-oriented textual syntax*. The other one is called *event-oriented textual syntax* where events are listed as they are encountered while scanning the MSC/LSC from top to bottom [35]. As an example, the graphical and textual representation of an MSC diagram is presented in Figure 10.

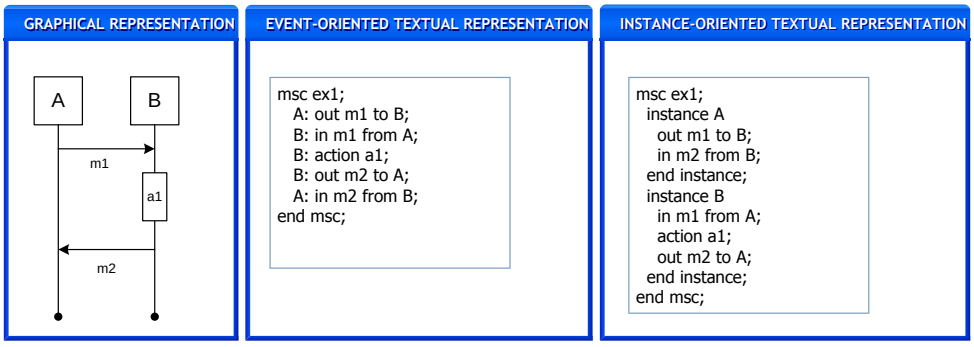


Figure 10. Graphical and Textual Representations of an MSC Diagram

While constructing the metamodel, instance-oriented textual syntax, which is based on the concrete textual grammar and lexical rules defined in Backus-Naur Form (BNF) [36], is assumed as the concrete syntax. In addition, some non-terminals,

defined in the textual grammar and in the lexical rules, are carried over to the metamodel.

One of the modeling issues was to decide which non-terminals would be included. The problem is to select the “essential” versus “nonessential” non-terminals so that only the essential ones are carried over to the metamodel.

As a rule of thumb, the essential non-terminals are identified as those constituting a “building block”, from the syntactical perspective a significant syntactic category and from the semantical perspective, having an intuitive meaning. For example, the non-terminal, `Input Address`, is defined by a production rule expressed below [36]:

```
<input address>::=<instance name>|{env|<reference identification>}[via<gate name>]
```

The input address specifies the MSC elements that can be connected to an MSC message. As it is regarded as an essential non-terminal in the sense described, a metamodel element, `Address Connection`, for representing a connection between a message model element and an input/output address element (e.g., environment), is created. Another example, `Orderable Event`, is defined in the textual grammar. Most events are categorized as orderable according to their connectability with the instance axis. Furthermore, some lexical rules refer to orderable events, for instance, a rule states that the MSC general ordering elements (i.e., before and after) could be connected with the orderable events. Eventually, the orderable event constitutes a building block and a corresponding modeling element is constructed (as an abstract element) in the metamodel. Thus, to express the aforementioned rule, it is sufficient to define a connection between the “general ordering element” and the “orderable event” modeling elements.

In contrast, the nonessential non-terminals can be seen merely as BNF artifacts, serving as stepping stones to define the essential ones. For example, `Action Statement` is regarded as a non-essential non-terminal as it is simply used to define the actions.

Note that those non-terminals included in the metamodel serve as abstract modeling elements, so that the modeler cannot use them in constructing an MSC/LSC model directly. They are used to structure the metamodel and to provide traceability of the MSC metamodel with the BNF grammar for the textual syntax given in the standard.

4.1.2 Using Abstract Syntax Trees for Each MSC/LSC Instance

Constructing an abstract syntax tree for each instance (representing a system component whose behavior is under observation), was a major metamodeling design decision. Instance-oriented representation leads to isolated trees for each instance due to its definition. Although this approach simplifies the code generation, when two or more instances are involved, it adds superfluous modeling work (e.g., modeling the interactions for each instance separately) for the modeler.

4.1.3 Designing Multi-instance Elements

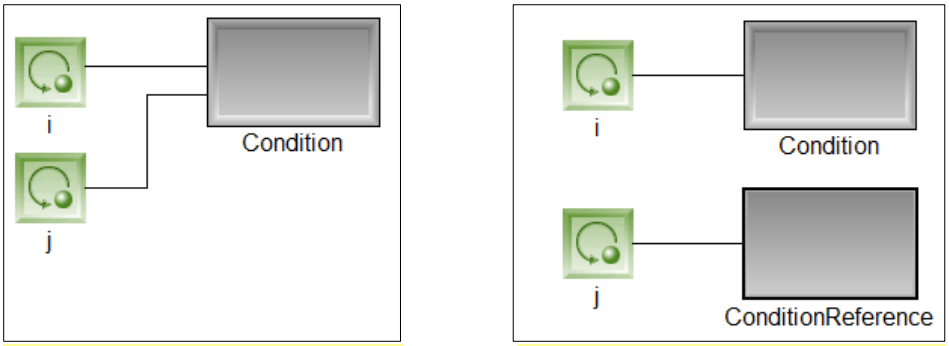
Some MSC/LSC elements may be connected to more than one instance. These are called Multi-instance Elements (a.k.a., shared elements). They are inline expressions, reference expressions, conditions, pre- and sub-charts. Separate instances make it hard to interpret the shared elements between the instances. The metamodel must reflect a shared element as a unique element to both the modeler and the code generator.

The use of copies of the multi-instance elements in each shared instance model as distinct modeling elements is not appropriate. If a multi-instance element is shared among multiple instances, each modeling element corresponding to multi-instance element must be the same. For instance, if an attribute value (e.g., name of the element) is changed in one instance, then the other copies must be changed automatically.

A straightforward design approach for such elements is to connect directly the shared events with the instances that share them. So, the metamodel allows multiple connections for shared events. This approach is called *multiple connection method*. However, for complex charts that have many instances and shared events, this approach will hinder the readability of the chart because of many connections between the shared events and instances. Consequently, to overcome this obstacle, the references (“pointers” to other model elements) between modeling elements are devised in the metamodel. Actually, there must be only one multi-instance event to be shared by instances. This sharing is done via GME built-in references and this mechanism is called *Referencing Mechanism*, which works as follows: the multi-event element must only be used in one of the instances. In the other instances, references to this element are used for sharing. As all the references point to the same element, they share the properties with the

referenced element. If an attribute value is changed (at model building time) in the multi-instance event, then the references reflect this change.

The examples for the usage of multiple connection method and referencing mechanism are presented in (a) and (b) of Figure 11 respectively. In the figure, *condition* is a shared element between *instances* `i` and `j`. In (a), instances are directly connected to the condition, while in (b), a condition reference element is used to substitute the original one.



(a) Using ***Multiple Connections*** to refer a shared element

(b) Using ***Referencing Mechanism*** to refer a shared element

Figure 11. Modeling Multi-instance Elements

The referencing mechanism is also used in event referencing. General ordering elements may point out to the events defined in the other instances found in the chart. This is done connecting the ordering element with the event references. Moreover, the input and output message events between instances are shared by using the event references. To facilitate this, each event has also a corresponding reference (e.g., message output event has a reference to itself).

4.1.4 Cardinality Constraints

The metamodel supports cardinalities. The cardinality constraints for the relationships are preserved by the metamodel structure. For example, Figure 12 depicts two kinds of cardinality constraint for relationships: (1) an instance must have only one instance end, (2) instance end must be connected only once to the instance.

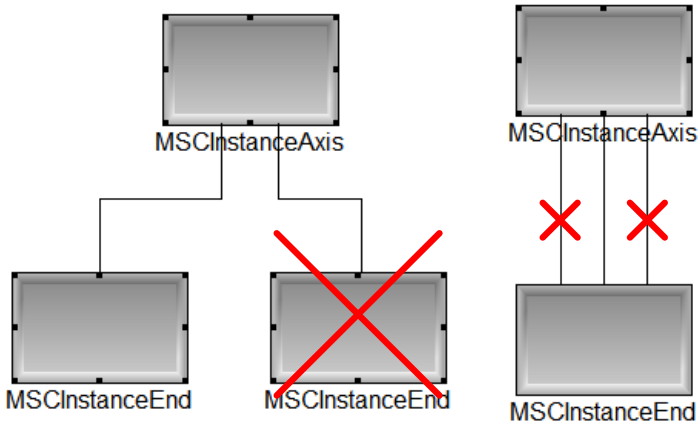


Figure 12. Cardinality Constraint

If the modeler violates a cardinality constraint, then GME Constraint Manager automatically generates a warning as seen in Figure 13.

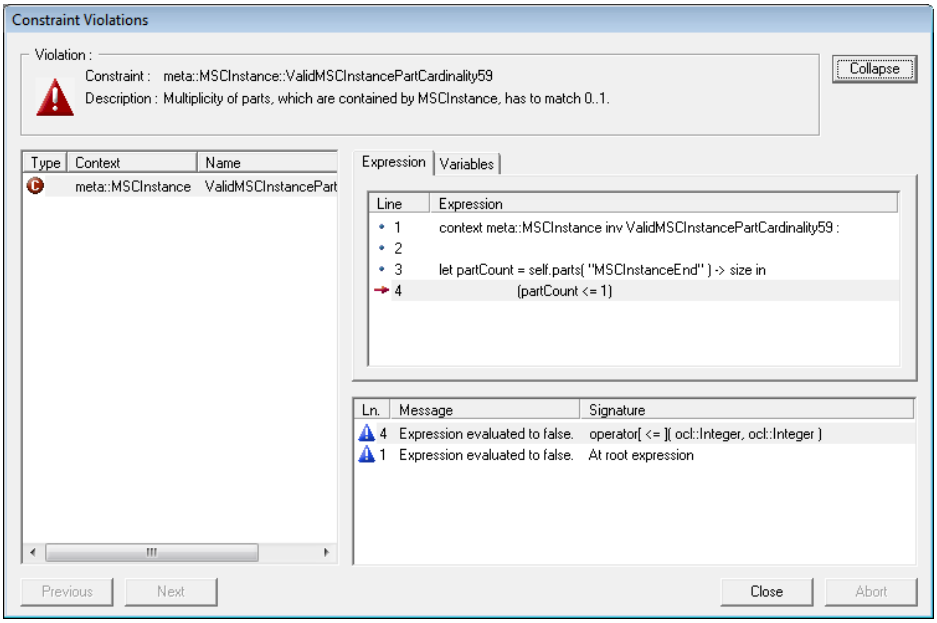


Figure 13. GME Constraint Manager Screenshot

Further constraints that cannot be enforced by metamodel structure, such as semantic/business/domain constraints, can be formulated using the Object Constraint Language (OCL) [27]. GME allows the constraints in OCL to be included

in the metamodel. OCL utilization in the LSC/MSC metamodel is noted as a future work.

4.1.5 Element Uniqueness, Naming, and Naming Scope

In the MSC standard, naming provides element uniqueness to distinguish among the same type of MSC/LSC elements. At the model level, for uniqueness of model elements, using unique names is not necessary. Because, the metamodel does not depend on the naming constraint for uniqueness, but rather it uses stronger constraint mechanisms (i.e., using references and connections). The modeler is free to use the same names for the same kind of elements. This approach also eliminates the message-overtaking problem, which occurs in textual representations of MSCs. The problem is that when two messages with the same name are sent, message instance naming is required for a unique correspondence between message input and output [36].

Note that when a modeling element is created in GME, GME automatically assigns the type of the element as the default name. Therefore, a separate name attribute is not defined for each modeling element. Uniqueness of model element names in the generated code is guaranteed by appending a portion of the GME-provided ID to the name in the model.

On the other hand, naming scope has an impact on code generators. Therefore, the MSC metamodel conforms to the naming scope specified in [36] in terms of declarations of elements. The root folder of the model is the scope for defining the MSC documents. An MSC document is the scope for defining charts, instances, conditions, timers, messages, and variables. An MSC is the scope for gates and MSC formal data parameters. Metamodel only allows legal declarations according to the scoping rules. Declaration of elements is discussed in MSC data concepts section.

4.1.6 Multiple Branches of an Instance in Different Charts

Axis of an instance may be scattered among more than one chart. Figure 14 presents an example. In LSC charts, pre-chart and body are in fact two sub-charts. Thus, the axes of both instances `A` and `B` are dispersed between these sub-charts.

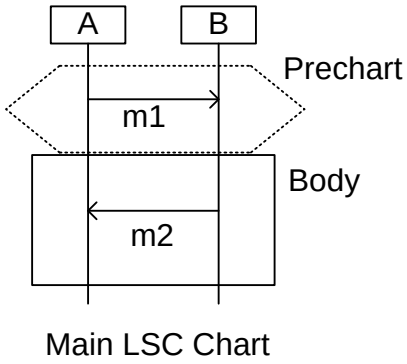


Figure 14. Multiple Braches for Instance Axis

Owing to the events that are dispersed into the sub-charts, each chart must be traversed by the model interpreters (e.g., code generator) to generate a complete abstract syntax tree for an instance.

4.2 MSC Metamodel

The *MSC Metamodel* is the basis for the Behavioral Metamodel where MSCs can be used to model the observable federate behavior. MMM includes all the MSC constituents, time concepts, data concepts, and High-level MSCs (HMSC) specified in [35, 36].

The hierarchical structure of MMM is depicted in Figure 15. MMM is formed by four main containers (i.e., folder in GME parlance), namely, Auxiliaries, Basic Constituents, Data Concepts, and Time Concepts.

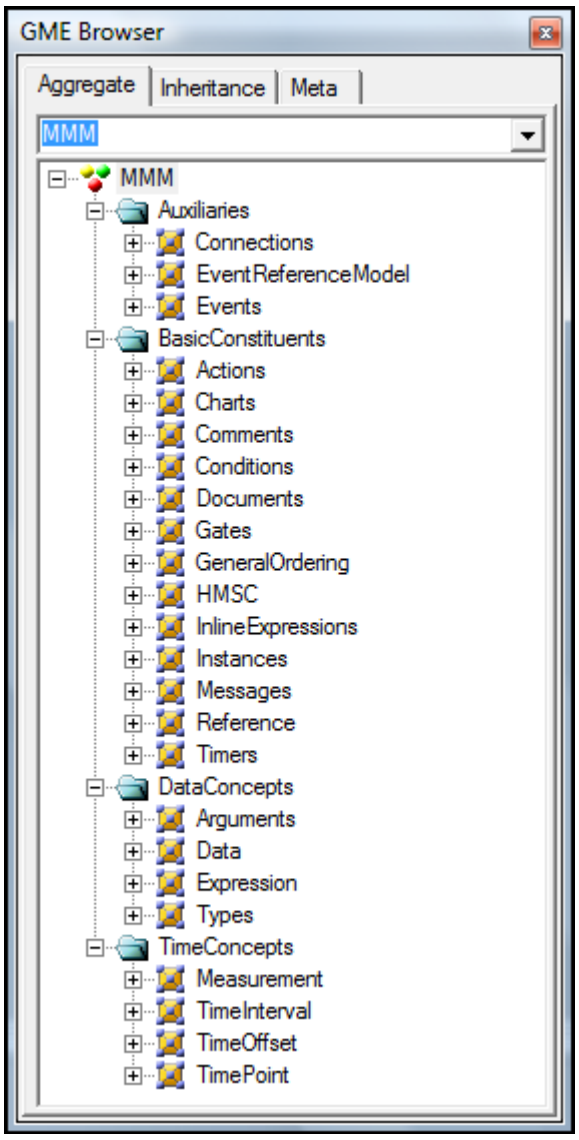


Figure 15. The MMM Implementation View (GME Screenshot)

The MSC documents, diagrams, instances, messages, comments, ordering elements, coregions, actions, references, inline expressions, gates, and timers are all defined in the *Basic Constituents* folder. *Time concepts* folder includes time measurement, time interval, time offset, and time point modeling elements. *Auxiliaries* folder includes the associations and events (e.g., message events: in and out) between the MSC elements. *Data Concepts* folder includes the data related elements such as arguments and expressions.

In the following sections, the basic constituents, data concepts, time concepts, and lastly the auxiliary metamodel elements are introduced. Elementary knowledge on MSCs, as provided in [40 and 41], is required for the succeeding discussion.

4.2.1 Constituents

4.2.1.1 MSC Documents

An MSC document (`mscdocument`) groups a number of MSCs and determines a namespace. An MSC document has two major model elements: the document head and body. As GME allows us to define roles for associations, body acts as both a defining container and a utility container.

The document head contains the declaration lists of messages, instances, and timers, which are used as types for the counter-part elements (instances) in the charts whereas the charts (i.e., MSCs) are defined in the defining or in the utility part. MSC documents define an instance kind for the other MSC elements. A document may use or inherit the other instance kinds by referring to the document by means of the “Using” and “Inherit” model elements. The document head also contains a data definition part, which specifies the data language, data, and the wildcards for referencing an external data model/language.

The charts in the defining part are the public charts while the ones in the utility part behave like private charts, which are used merely by the defining charts.

Model interpreters, particularly the code generator, traverse the MSCs in the defining part for each instance found in the instance list of the MSC document. For the modeler, an attribute, named `Chart Order Index`, is defined to guide the chart execution/interpretation order of the model interpreters. If required, this attribute can be set by the modeler. Similarly, for multiple documents in a model, the order of the documents may be specified by the `Document Order Index`.

Lastly, the MSC document contains an optional attribute `related` to specify the pathname of the document, to which the MSCs refer. The structure of the document model element is depicted in Figure 16. Additionally, the right pane of the screen shot gives a top folder view of the MSC metamodel.

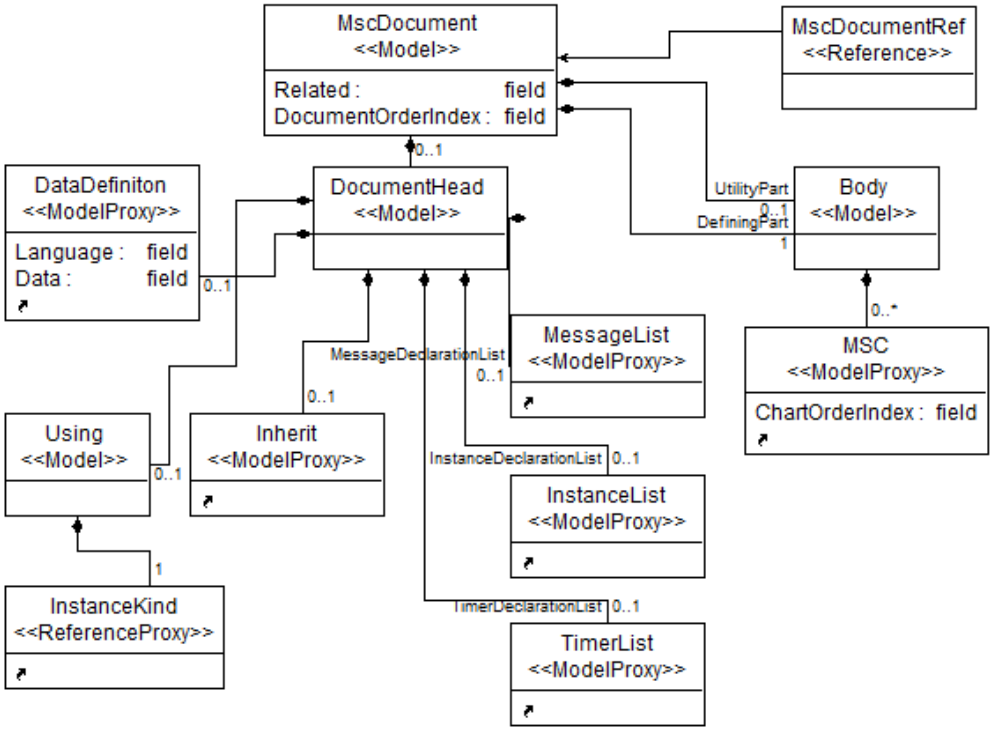


Figure 16. The Structure of MSC Document Model Element

Figure 17 and Figure 18 present an example about the usage of MSC documents. The examples are mostly taken from [35, 36]. Corresponding model for Figure 17 is presented in Figure 18.

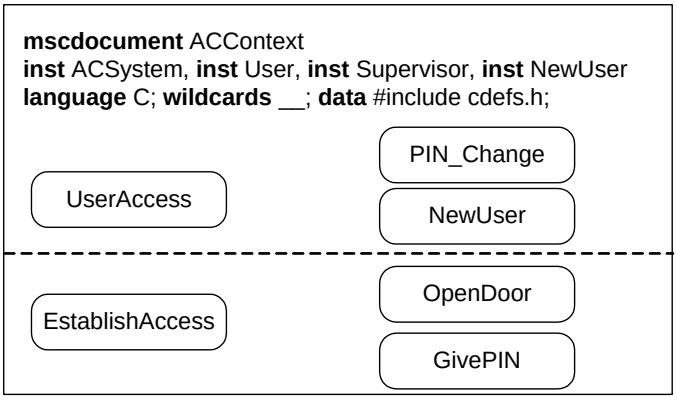


Figure 17. Example from (Figure 23 of) [36].

In Figure 18, the right top pane depicts a tree structure of the MSC document, the left top pane shows only the document head model, the right down pane shows the

data definition attributes, and the left down pane shows the model elements that can be used in document head model.

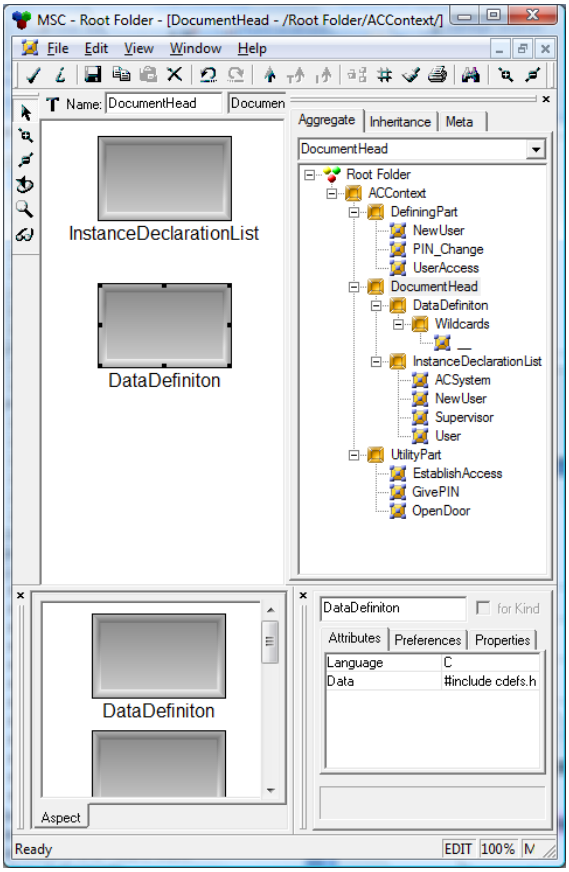


Figure 18. Corresponding Model for Document Example (GME Screenshot)

4.2.1.2 Charts

A Message Sequence Chart determines a partial order on the events that constitute the behavior of a system. The behavioral description given by an MSC, however, may not be complete description. MSC is the main model where most of the model elements (e.g., events, messages, conditions etc.) are contained. MSC model is presented in Figure 19.

MSC has two parts: the head and the body. The head, which is optional, contains the MSC parameters and an offset for time. The body, which is compulsory, contains either an MSC body or a HMSC. MSC body (also can be seen as the MSC itself) contains the abstract syntax trees of each instances where the interactions (a.k.a. events) of an instance are specified. The instances, messages,

and timers that declared in the MSC document are used in the MSC body. The instance (or instance reference) is the root node for the abstract syntax tree.

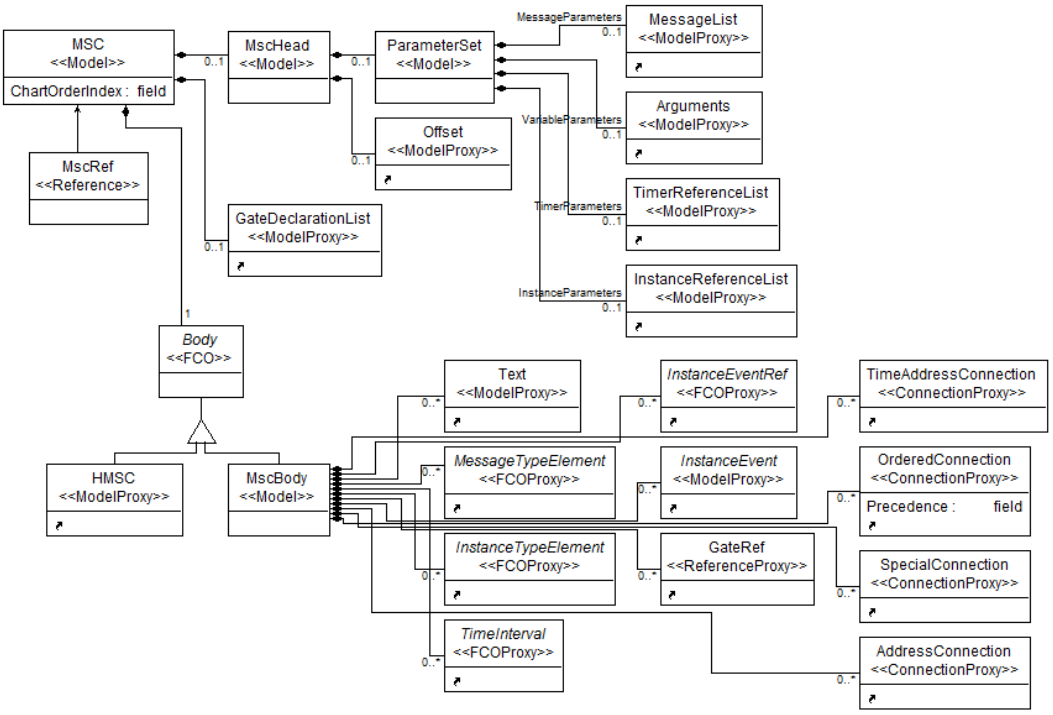


Figure 19. Chart Paradigm Sheet

To enforce the constraint "MSC *must have* a body as *either* an `MSCBody` *or* `HMSC`", first, the cardinality of association between `MSC` and `body` elements is set to one and then, a new element (i.e., `body`) is introduced as a GME First Class Object (FCO) class which is mandatorily abstract, to group the "either/or" elements (i.e., `MSCBody` and `HMSC` respectively).

4.2.1.3 Instances and Instance Decomposition

An instance is the root for the abstract syntax tree ended with one instance end (`endinstance`). Instance is an entity on which events can be specified. They are declared in the enclosing MSC document.

All model elements and events can be connected to the instance via the connection element `OrderedConnection`. Only this type of connection is allowed with the instance. The order of events (connections to the instance) is specified by the help of the attribute `Precedence`. The modeler must manually specify the

order of the events in the instance axis by assigning values to this attribute. For example, the connection between `instance` and `endinstance` must have the last precedence order expected.

The structure of the instance element is depicted in Figure 20. Instance kind, which is an MSC document, may be specified in addition to the instance and an instance may inherit from an instance kind. Finally, instances may define a variable list where instance variables are declared.

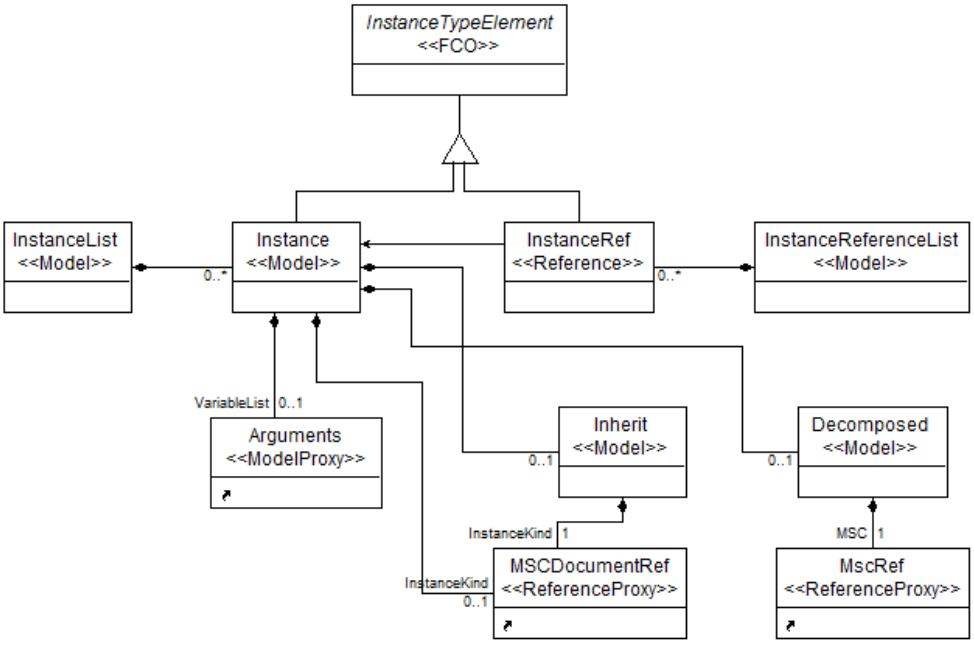


Figure 20. The Structure of Instance Model Element

An instance may be decomposed into another MSC. The `decomposed` modeling element includes a reference that refers to the decomposed MSC. When the decomposed model element is used, then an MSC reference must certainly be included.

`InstanceTypeElement` is an abstract class to group instance and instance reference to simplify the connection structure in metamodel and to ensure that instance references are also behaved as instances. For example, when a connection is defined for an instance element, then generally it will be legal for an instance reference too. So, while metamodeling, the connection is defined between an element and `InstanceTypeElement`, instead of defining it explicitly both for

instance and instance reference. This kind of generalization (grouping element and its reference) is used mostly for the other elements in the metamodel.

4.2.1.4 Comments

Comments are used to enhance documentation in the behavior model. They can be utilized to help with traceability, e.g., traceability between the generated code and the source model.

Two different kinds of comments, videlicet `text` and `comment`, are used. Comment is associated with the most of the MSC elements. Instead of adding the comment as an attribute for each element, an abstract base class (`ElementHasComment`) is created and a GME string attribute for comment is added only to this base class. Finally, the elements that have a comment are inherited from this base class.

The text, for that matter, can be associated merely with the MSC or HMSC diagrams for the purpose of global explanations.

4.2.1.5 Message and Message Events

A message is the main entity, sent or received, between the message events (i.e., message output `out` and message input `in`) and between the method call events (i.e., `call/receive` and `reply out/reply in`). Messages may have arguments. Message types are declared in the message declaration list of the MSC document. The structure of message model is depicted in Figure 21. Arguments are discussed in *Data Concepts* (section 4.2.2).

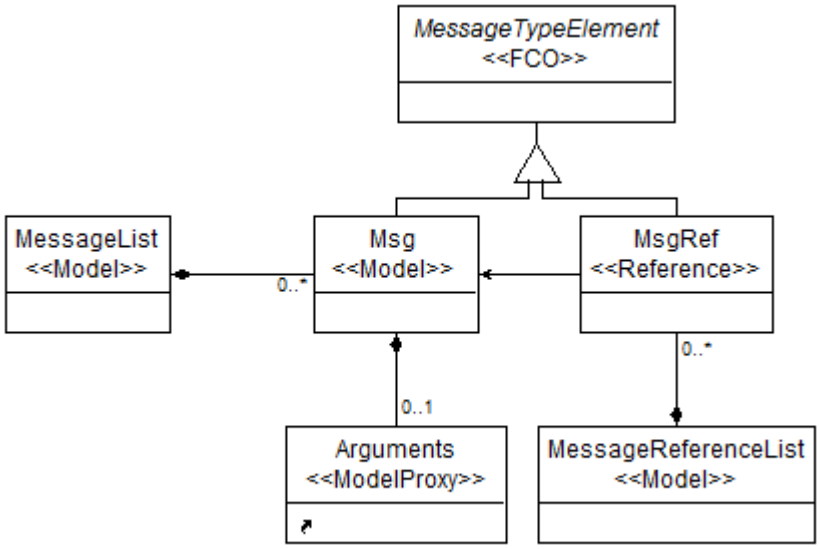
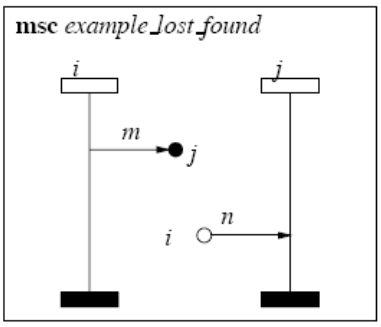


Figure 21. The Structure of Message Model Element

While designing the message exchange, it is taken into consideration that message/method call events must refer to a message and must point out the recipient or sender of the message. Therefore, the metamodel enables this kind of connections.

Message and method call events can be completed or uncompleted. Each event has a Boolean attribute in order to specify an incomplete message, which means that a message can be found or lost. The outgoing message events (i.e., `out`, `call`, and `reply out`) has a Boolean attribute `lost` to indicate the message is lost or not, while the incoming events (i.e., `in`, `receive`, and `reply in`) has `found` to indicate the message is found or not.

An example for incomplete messages, first, the MSC example is presented in Figure 22, and then the corresponding model for instance `i` is presented in Figure 23.




```
msc example_lost_found;
i   : out m to lost j;
j   : in n from found i;
endmsc;
```

Figure 22. Example B.11 from [35].

Notice in Figure 23 that the lost attribute of `out` event of instance *i* is set to true to indicate that the message *m* is lost after sent to instance *j*.

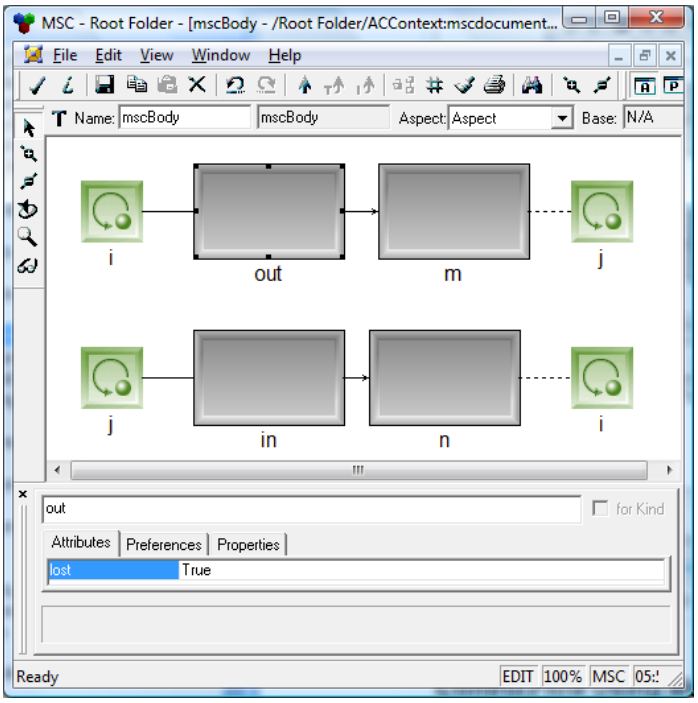


Figure 23. Corresponding Behavioral Model for B.11

4.2.1.6 Control Flow Using Method Call Events

MSC may describe control flows by the means of calls and replies. A method is a named unit of behavior inside an instance. Methods are modeled with `method` and `suspension` elements. Suspension regions indicate the regions where no

events occur till the reply of the call returns (that is to say a synchronizing call). On the other hand, an asynchronous call implies the method regions where the caller may continue without waiting for the reply of the call. Method and Suspension are non-orderable events.

Some elements such as method and suspension define regions; such elements have modeled as a start and an end element to indicate the region. The events connected to the instance axis between a region start (e.g., `Method`) and a region end (`EndMethod`) must be interpreted as the events occurring in the region. For example, a method region can contain any events while a suspension region cannot.

A method may be invoked remotely (`callout` and `receive`) and the results of the calculations of the method may be returned through a reply to the caller (`replyout` and `replyin`). As used in message events, incomplete method calls can be used. An example for the usage of the control flow elements is depicted in Figure 24.

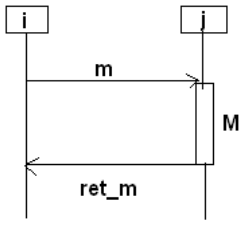


Figure 24. Example for Method Call.

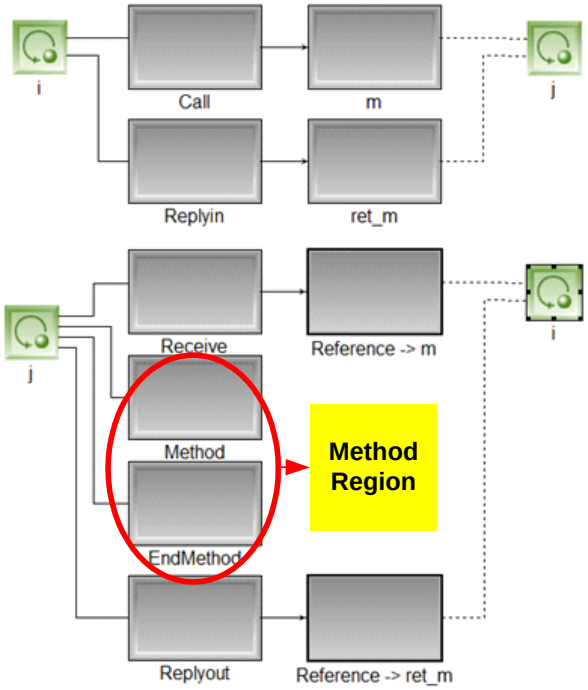


Figure 25. Corresponding Behavioral Model for Method Call in Figure 24.

4.2.1.7 Environment and Gates

The outside region of an MSC frame is called as its environment. Environment is inherited from an MSC instance. While inheriting the environment element from the MSC instance, instead of using the usual inheritance operator, the GME interface inheritance[1] is used, so that the environment becomes a black box. Thus, the environment does not contain any MSC/LSC instance constituents such as a variable list.

As expected, only one environment can exist in an MSC chart.

The gates represent an entry point between the enclosing chart (i.e., MSC) and its environment. The gates are defined in MSC reference or inline expression frame as well as in environment. The message events, the order events, and the process creation event can be connected to the gates. Gates are implemented in the metamodel as GME ports, which behave like an interface (connection points) for the model element.

[1] Interface inheritance is described as "Interface inheritance allows no attribute inheritance but does allow full association inheritance" [37].

4.2.1.8 General Ordering

MSC general ordering elements, namely, `before` and `after`, are modeled as GME ports, which are contained in the orderable events (e.g., timer events, actions etc.).

In the MSC, the preceding/following events are specified via their names, but the metamodel enables us to directly establish a link between the ordering element and the events. Therefore, the general ordering elements can be connected to the preceding/following (according to the ordering type) events (or references of events), environment, or gates.

An example is given in the following figures. First, in Figure 26, the example MSC chart, taken from [35] is presented and then the corresponding model is given.

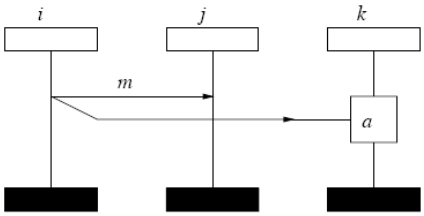


Figure 26. Example B.18 from [35].

The general ordering element `before` embodied as a port in the message `out` event is connected to the `action` local event of instance *k*. So, the interpretation should be that sending message *m* by instance *i* shall occur before the action *a* of instance *k*.

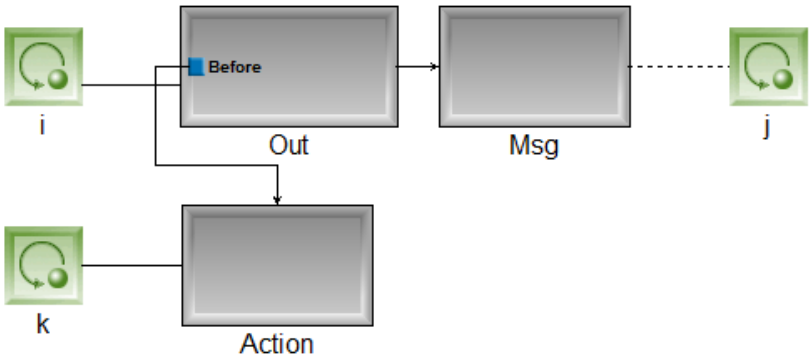
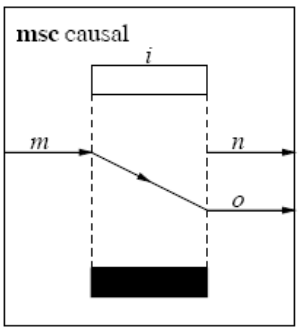


Figure 27. Corresponding Behavioral Model for B.18

Another example with a coregion is provided in the following figures.




```
msc causal;
i    : concurrent;
           in m from env before 12;
           out n to env;
           12 out o to env;
       endconcurrent;
endmsc;
```

Figure 28. Example B.20 from [35].

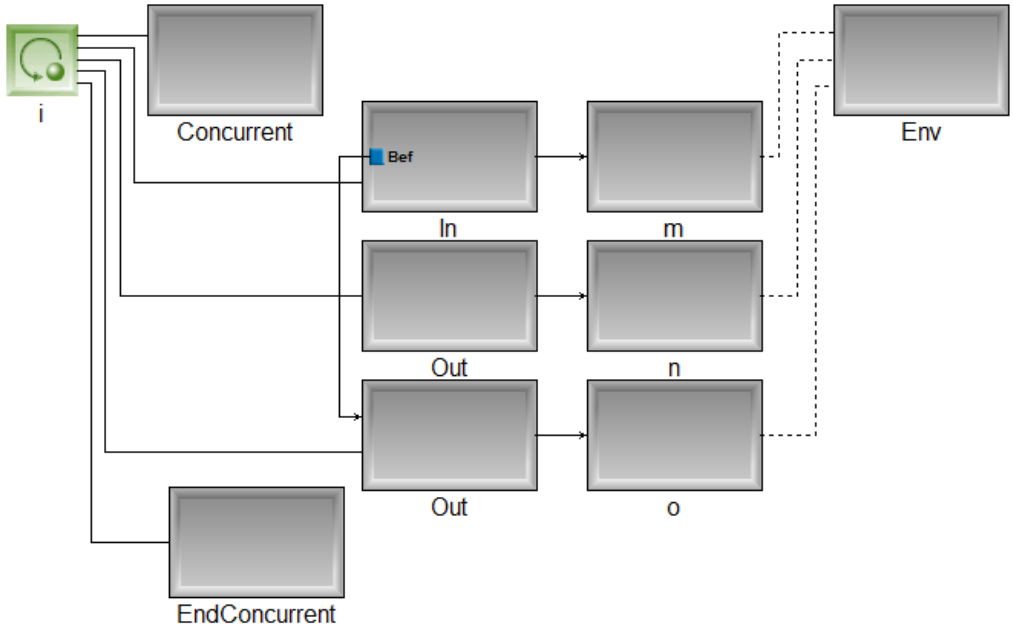


Figure 29. Corresponding Behavioral Model for B.20

4.2.1.9 Conditions

Conditions describe a state that is common to a subset of instances in an MSC. They are multi-instance modeling elements.

Condition has a type, which can be set as *setting condition* or *guarding condition*. Setting condition is the default type for a condition. When condition is a guarding

condition, then an expression field, which evaluates to a Boolean value, is required to be filled. For setting conditions, expression attribute value should be null.

`Otherwise` is a specialized condition used as a guarding condition just in one of the operands of an alternative inline expression. It is evaluated as true if only the other conditions defined in the alternative expression evaluate to false. The condition paradigm sheet is presented in Figure 30.

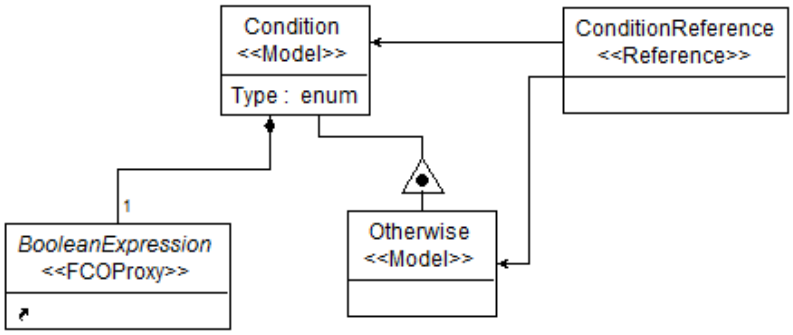


Figure 30. Paradigm Sheet for Condition[1]

An example for the usage of conditions is depicted in the following figures.

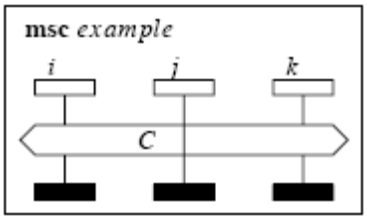


Figure 31. Example B.13 from [35]

Note that instance `i` and `k` uses the same condition where condition type is a setting condition and expression attribute is null. It is also possible to link a condition reference to the instance `k` referring the condition of instance `i` instead of using the same condition.

[1] The triangle with black dot represents a GME implementation inheritance, where "the subclass inherits all of the base class' attributes, but only those containment associations where the base class functions as the container" [37].

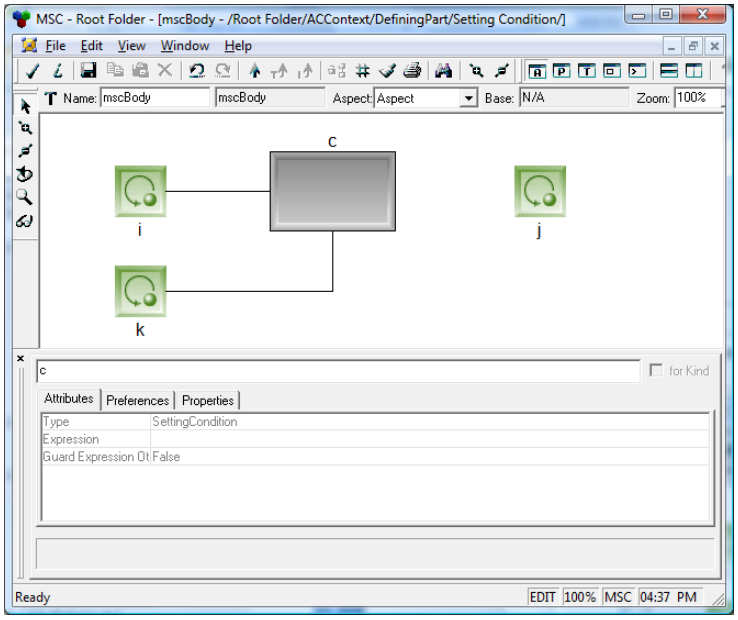


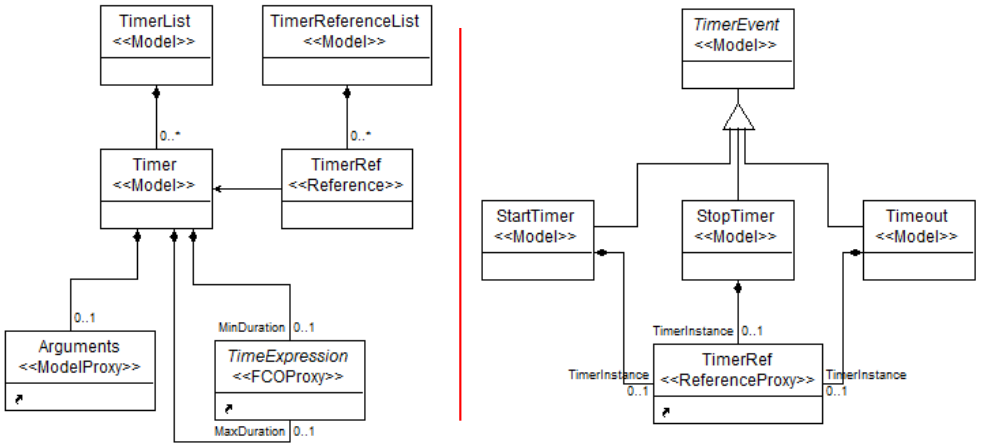
Figure 32. Corresponding Behavioral Model for B.16 (GME Screenshot)

4.2.1.10 Timers and Timer Events

Timers are used in MSC diagrams and they can be controlled using the basic timer events, namely start (`starttimer`), reset/stop (`stoptimer`), and time-out (`timeout`) events. A timer event has a timer identifier (a reference to the declared timer) to specify which timer the event refers. The timer identifier represents a timer instance. Each timer has arguments and furthermore maximum and minimum durations in the form of a time expression. Time expressions are discusses in Time Concepts section.

The timer events are local to an instance and they can be used stand-alone or in combinations, where timer set event is allowed to be connected to timer reset and time-out events.

The modeler may explicitly pair off the timer events at design time, if required. Note that if the timer event pairs (i.e., set-reset, set-timeout, and set-reset-timeout) are contained in co-regions, then unifying those events is compulsory. The timer and timer events model is depicted in Figure 33.



(a) Timer Paradigm Sheet

(b) Timer Events (from Events Paradigm Sheet)

Figure 33. Timer and Timer Events Model.

Examples about timers and timer events are presented in the following figures.

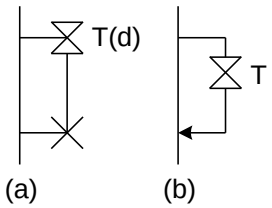


Figure 34. Example B.9 from [35].

In the corresponding model, the timer `T(d)` is declared in the declaration list of the enclosing MSC document. In Figure 35, the right pane shows the timer declaration list. In the MSC diagram, the start timer event includes a reference (aka, timer instance) to the timer `T(d)`.

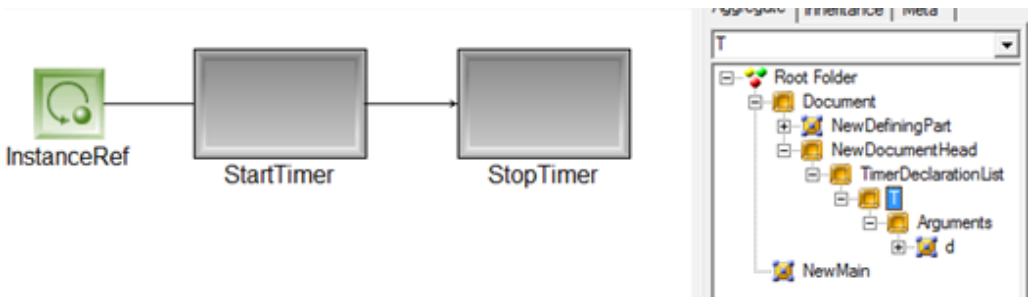


Figure 35. Corresponding Behavioral Model for B.9

4.2.1.11 Actions

Actions are events that are local to an instance where *local* indicates a connection specific to one instance. An action is an atomic event used to specify some computation. Figure 27 presents a simple example for the usage of action model element.

The name of the action model element acts as an informal action string. Moreover, an action may have a data statement list which contains a defined (e.g., def x) or an undefined (e.g., `undef x`) statement with a collection of variables (e.g., the "x" is a variable in `def x` or `undef x`). An undefined statement is used to indicate is used to indicate that a variable has named out of scope and cannot be referenced furthermore. The structure of action model element is given in Figure 36.

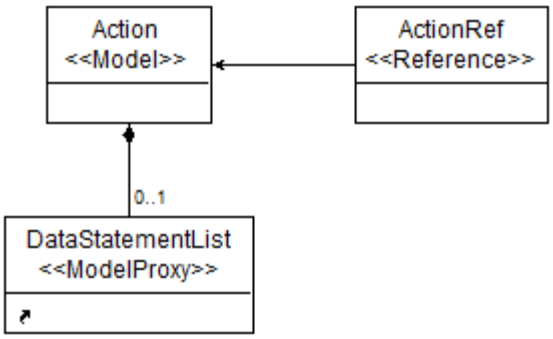


Figure 36. The Structure of Action Model Element

4.2.1.12 Instance Creation and Termination

Instance creation and termination events (i.e., create and stop) are used to handle process lifetime. A create event must be connected to an instance where a termination event behaves like an instance end for the created process. An instance can merely terminate itself whereas an instance is created by another instance.

The created instances are also declared in the instance declaration list of the MSC document. Therefore, the references are used in the MSC.

While creating an instance, a parameter list can be specified in the created instance. An example for the instance creation and deletion is provided in the following figures.

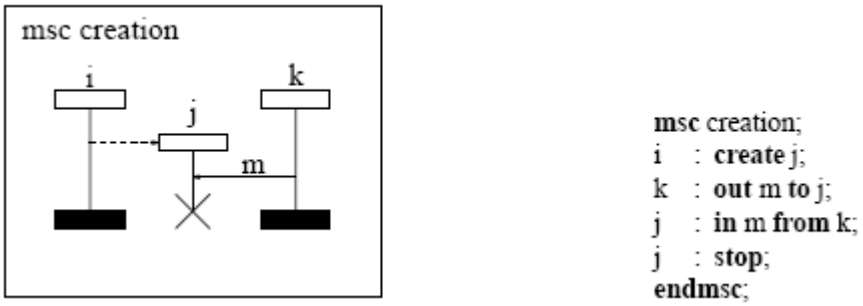


Figure 37. Example B.6 and B.7 from [35].

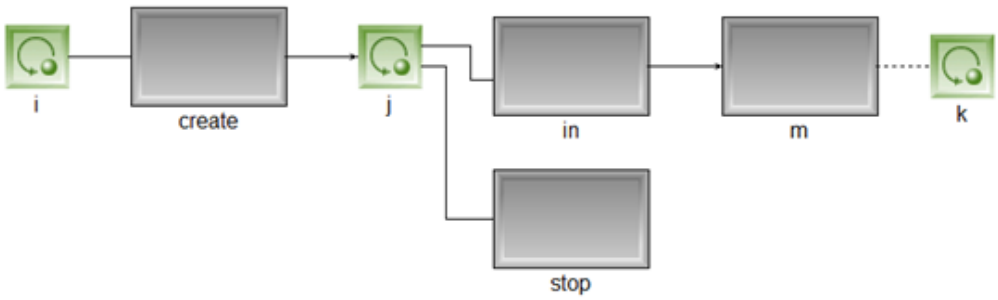
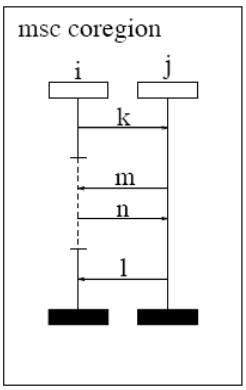


Figure 38. Corresponding Behavioral Model for B.6

4.2.1.13 Coregions

A coregion, specified with the start and the end events (`concurrent` and `endconcurrent` respectively), is a part of the instance axis for which the events connected to that part are assumed unordered. Only orderable events can be connected to a coregion. An example is presented through Figure 39 and Figure 40.




```
msc coregion;
i   : out k to j;
i   : concurrent;
        in m from j;
        out n to j;
      endconcurrent;
i   : in l from j;
j   : in k from i;
j   : out m to i;
j   : in n from i;
j   : out l to i;
endmsc;
```

Figure 39. Example B.16 from [35].

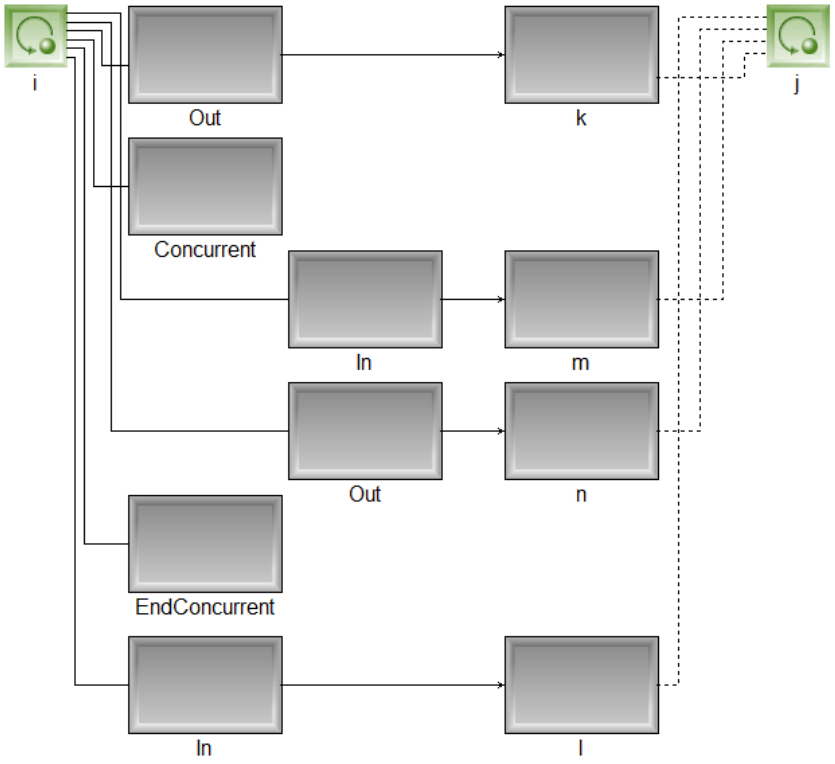


Figure 40. Corresponding Behavioral Model for B. 16

4.2.1.14 Inline Expressions

Inline expressions provide a means for the composition of event structures. The inline operators and their semantics are presented in Table 5. In addition to their usage in the MSC, they are also used with the MSC references and in the High-level MCSs.

Table 5. Inline Operators

OPERATOR	EXPLANATION	OPERANDS
Seq	Sequential operation.	One or more
Par	Parallel execution. It refers to a horizontal composition.	One or more
Alt	For alternative runs for MSC sections. It refers to an alternative composition.	One or more
Loop	The loop structure. It indicates iteration of the events within the inline expression.	One
Exc	It represents an exception.	One
Opt	It represents an optional (zero or one time) execution.	One

Figure 41 depicts the structure of the inline expression model. External attribute indicates the MSC keyword “external” in the textual notation.

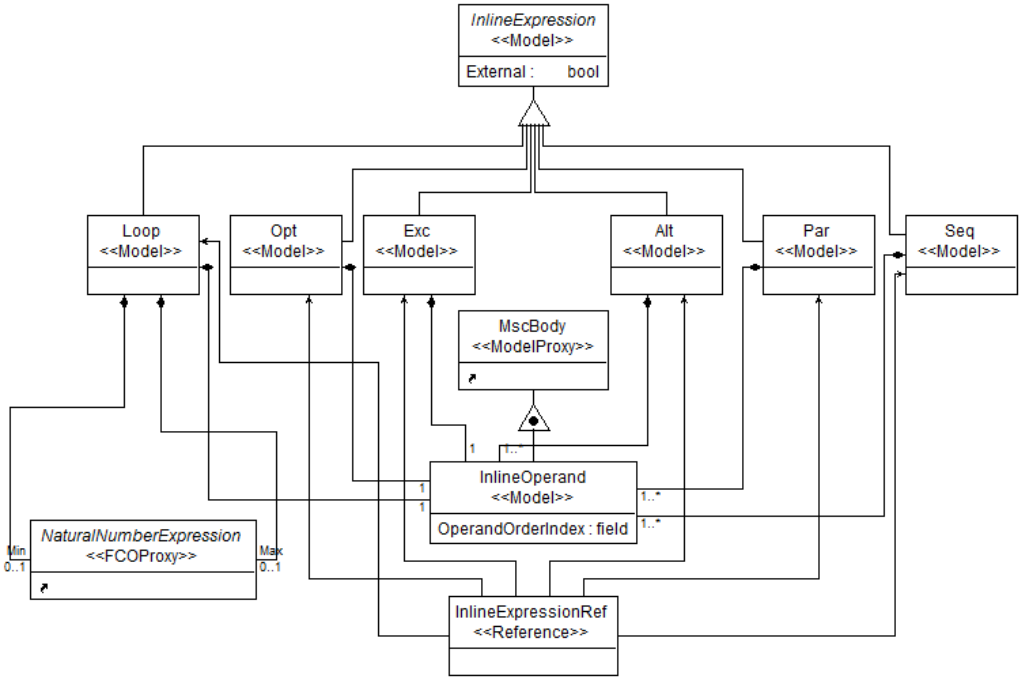


Figure 41. The Structure of the Inline Expressions

Inline operators must have one or more operands according to its model type. Operands are indeed MSC bodies, which contain all the basic MSC events. The metamodel reflects this multiplicity; for example, a loop expression has exactly one

operand. If an inline operator has more than one operand, then it is essential to specify the precedence of its operands. Thus, each operand has an order index to specify the interpretation order.

Inline operators are multi-instance elements, meaning that they can be connected more than one instance axis. In order to share, one inline operator between instances, referencing mechanism is used. Each inline operator has a corresponding reference element (`InlineExpressionRef`).

An example for `alt` inline operator in an MSC is presented through Figure 43.

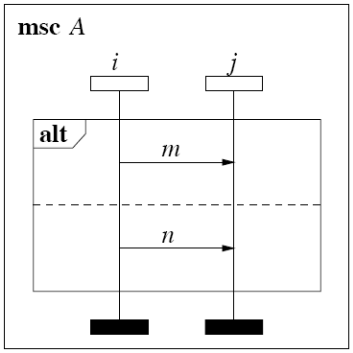


Figure 42. Example B.29 (msc A) from [35].

Corresponding model for Figure 42 is presented in Figure 43. The main chart is seen in (a) where an inline model element can be referenced in two ways. First, a reference element is used (in the left side); second, both instances are directly connected to the inline expression. Inline expression has two operands, where each operand behaves like a sub-chart (or MSC body). The “inside” of each operand is presented in (b) and (c) respectively.

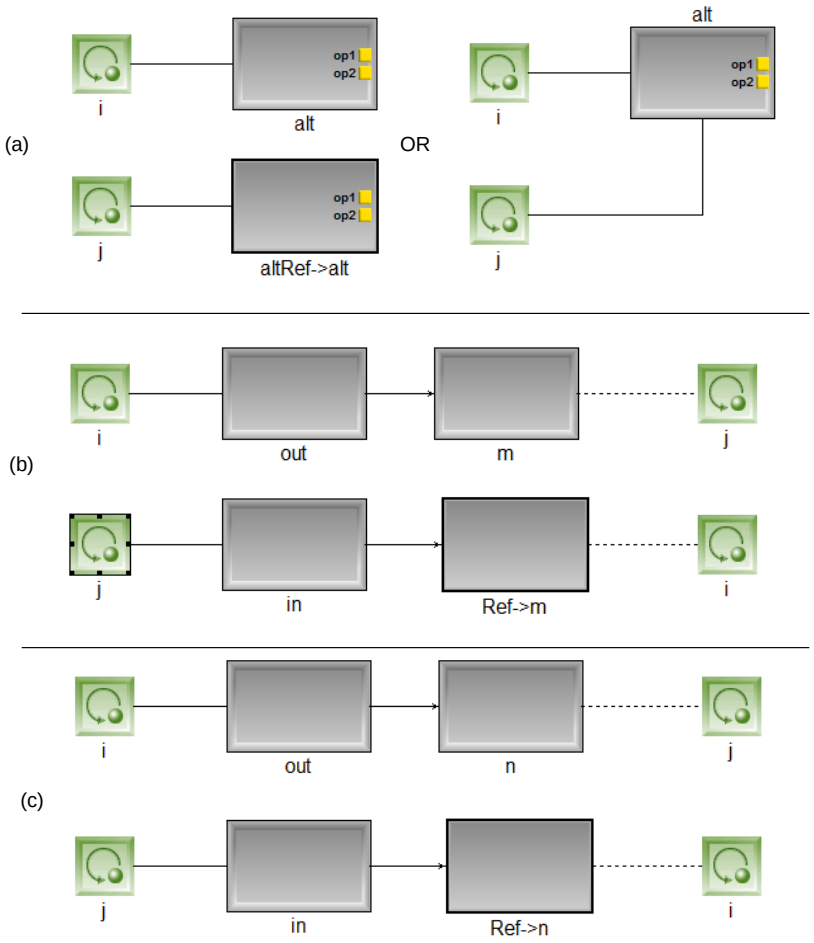


Figure 43. Corresponding Behavioral Model for B.29 "msc A".

Loops have a loop boundary (`NaturalNumberExpression`) in order to indicate the minimum and maximum iteration values for looping behavior. Loop operand is executed at least "minimum" times and at most "maximum" times. The values or minimum and maximum can either be the keyword `inf`, representing infinity, or a sequence of digits. Loop boundary values are initially (`1, inf`). An example for the practice of loop inline operator in HMSCs is given in Figure 64.

Inline expressions define regions. Gates are also used as the entry points for the inline expressions. An example for inline operators with gates appears in Figure 44.

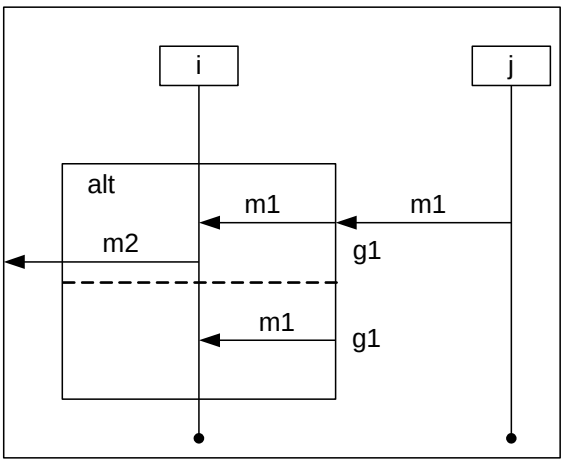


Figure 44. An Example for Inline Operators with Gates.

The instance `j` sends a message to the alt inline expression of instance `i` via gate `g1`. The MSC statement corresponding that is `j: out m1 to inline alt via g1`. The first operand of the alt operator receives `m1` via its gate `g1`, while the second operator receives it via the environment.

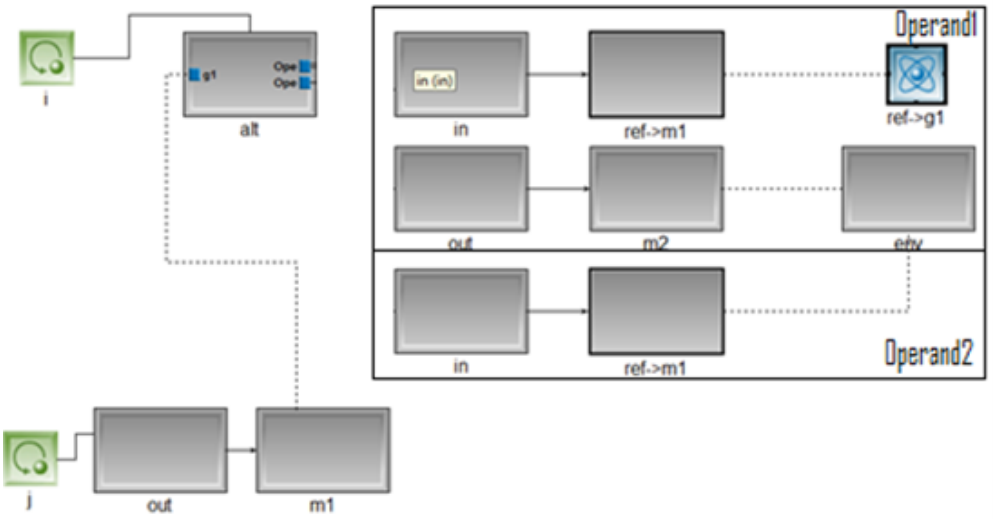


Figure 45. Corresponding Model for Inline Operators with Gates (GME Screenshot)

4.2.1.15 References

An MSC reference expression can be used to refer to other MSCs in an MSC document. `Reference` model structure is depicted in Figure 46. A reference structure can include other MSC diagrams as well as composition operators for

complex referencing mechanisms. Examples of MSC reference expressions are `reference A`, `reference (A alt B) seq C` where *A*, *B*, and *C* are MSC diagrams referenced.

The actual reference parameters are used to call the referred MSC declaration parameters with actual values. The actual reference parameters include the data, instance, message, and timer parameters.

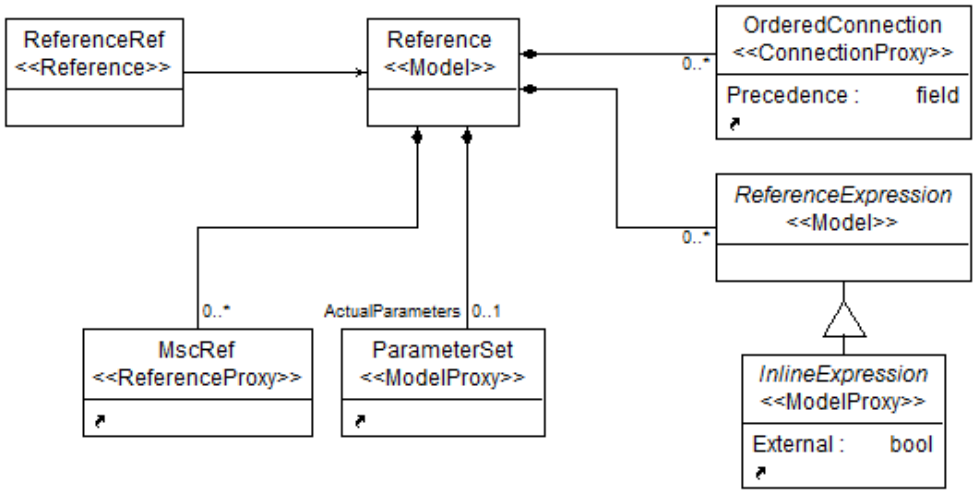


Figure 46. The Structure of the Reference Model Element.

If the MSC reference element does not contain any MSC, then it means that it is a null reference corresponding to the MSC keyword `empty`.

MSC references are also multi instance events. Therefore, a reference element (`ReferenceRef`) for the MSC reference is created.

For the usage of the complex reference expressions, such as `reference (A alt B) seq C`, a conceptual view of the corresponding model is depicted in Figure 47.

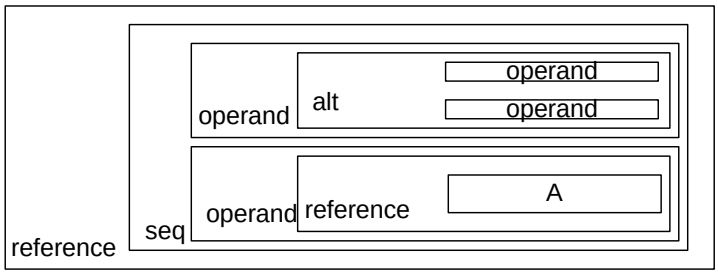


Figure 47. Conceptual View of the Corresponding Model for Referencing.

Gates are also used with references. An example is depicted in the following figures.

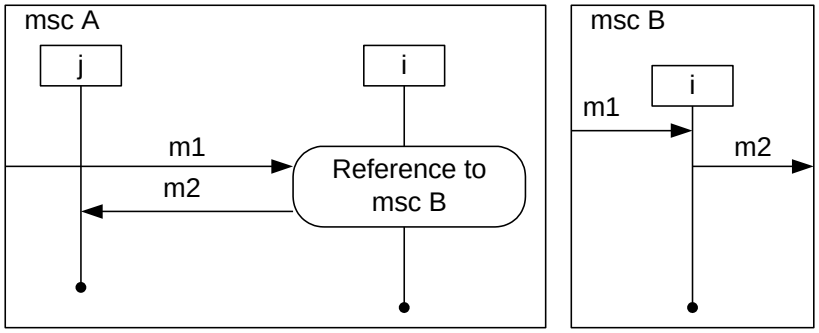


Figure 48. An Example for MSC Reference with Gates.

The message `m1` comes from the environment. In MSC `A`, there is no connection to the gate `g1`. This represents an environment gate.

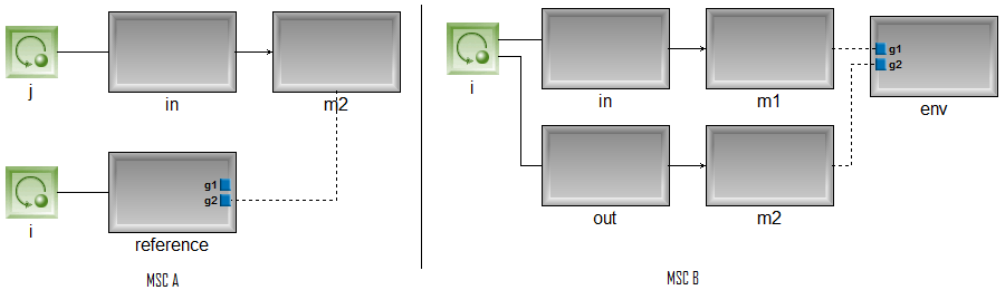


Figure 49. Corresponding Model for MSC Reference with Gates.

4.2.2 Data Concepts

The MSC specification, in addition to an action model, introduces a preliminary data model via some predefined elements, such as messages, actions, and MSC references. Basic data concepts as defined in MSC specification [36] are included in the metamodel to enable the declaration and the use of the static and the dynamic data.

4.2.2.1 Declaring Data and Using Declarations

Messages, timers, instances, and variables are declared via declaration lists as specified in [36]. The declaration lists are message list, timer list, instance list (has variable list), and dynamic variable list. The first three are declared in the MSC document whilst the last one specifies the MSC parameter types. In declaration lists, type, number, and order of the arguments are defined. Declaration lists are used for type checking. For example, messages that have parameters are declared so that the type and number of parameters are defined.

In Figure 50, an example is provided for a message declaration. A message `sum` is declared in the message declaration list of an MSC document header. `Sum` has two arguments; `first` and `second` having the type of natural number.

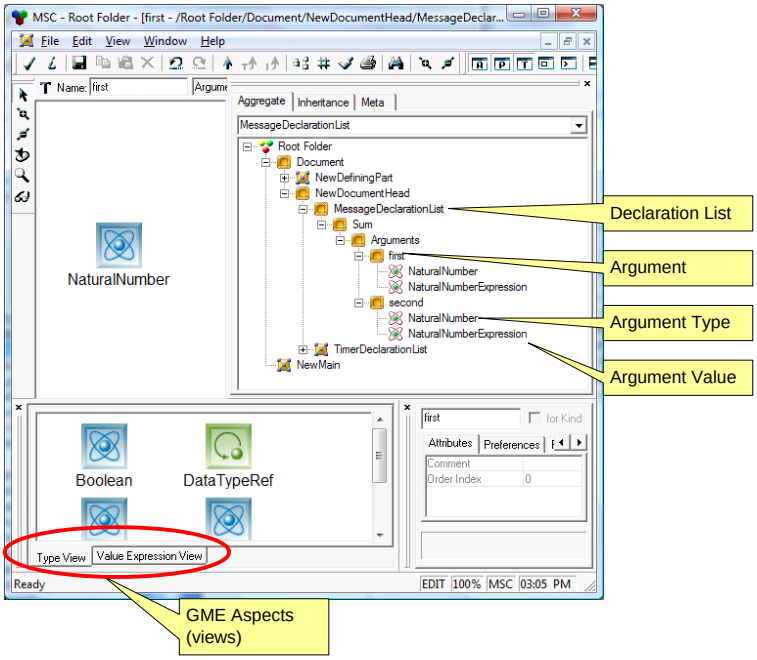


Figure 50. Declaration of a Message (GME Screenshot)

For the representation of both actual and formal parameters, an argument model is devised. The structure of an argument is presented in Figure 51. An argument has a data type and a value. Actually, two GME aspects (i.e., views for the modeler) are defined to separate the data type view and value view as seen in the lower pane of Figure 50. While declaring an element (i.e., message, instance, timer, and variable), the number and the type of arguments are provided. It is also important to specify the order of the arguments. An attribute (`OrderIndex`) is provided for this purpose. Values of arguments may be left unspecified or an initial value may be provided.

Note that for parameterless elements (e.g., a parameterless message), there is no need for a declaration.

A variable is an element that has a specific data type and value. Variables are represented using the arguments. Variable lists are used by the MSC in instances and in wildcards.

Data types and value expressions are discussed in the following sections.

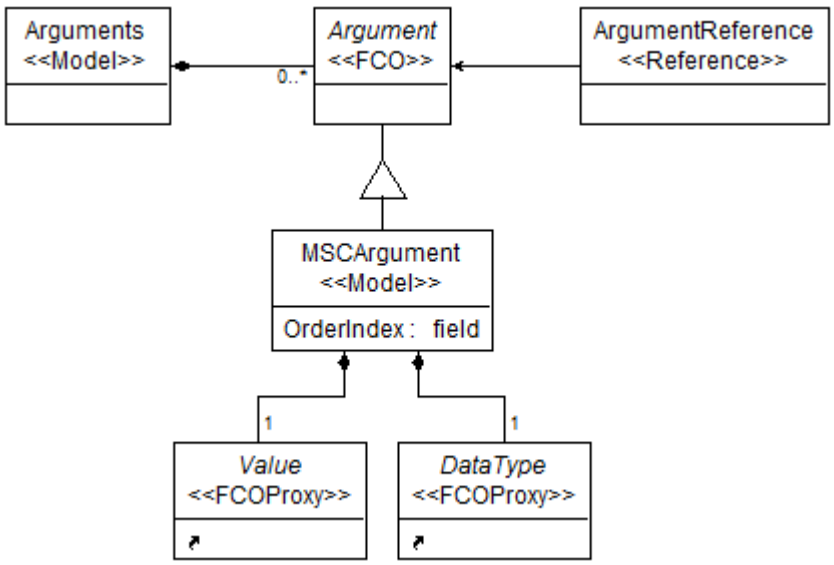


Figure 51. The Structure of an Argument

4.2.2.2 Type Checking

A declaration provides a template (a type model) for the actual use (an instance model). After declaring the element, the instance of the declared element can be used in the MSC diagrams (In GME, an instance model can be created by dragging the type model icon and dropping it to the behavioral model [11]). The modeler cannot change the number and type of the arguments, but can only set the argument values. Figure 52 presents the actual use of message `sum` in a MSC chart. The modeler can assign any natural number expression for the argument values (e.g., Sum (5, 4)). GME marks the `sum` as an instance model and automatically provides the type of the instance (the red oval marking in the figure) as "type of sum" in the example.

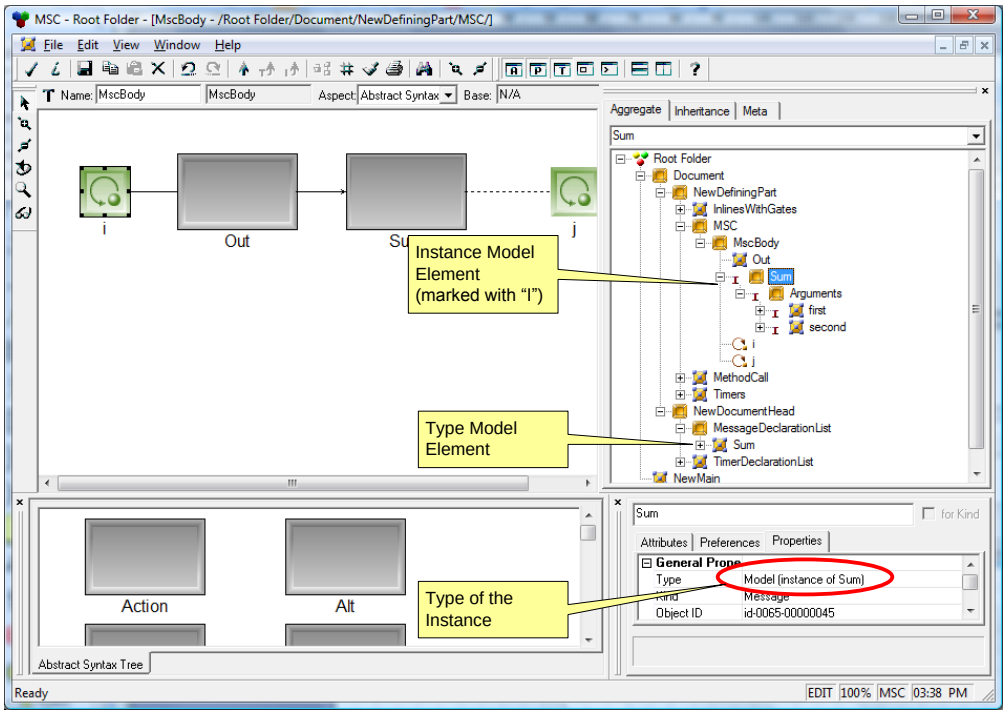


Figure 52. Using a Message Declaration (GME Screenshot)

A clear advantage of this approach is that it eliminates the need for a type checking mechanism to be integrated to the metamodel (e.g., by creating specialized elements for type checking). While creating an instance from a type model element, thanks to GME instantiation mechanism, GME automatically provides the type for the instance. Further type checking is left to the model interpreters.

GME model browser has the `Inheritance` tab used explicitly for visualizing the type inheritance hierarchy. For example, in Figure 53, `Input` message is used (instantiated) two times in the model as `Input_Msg` and `Input_Name`.

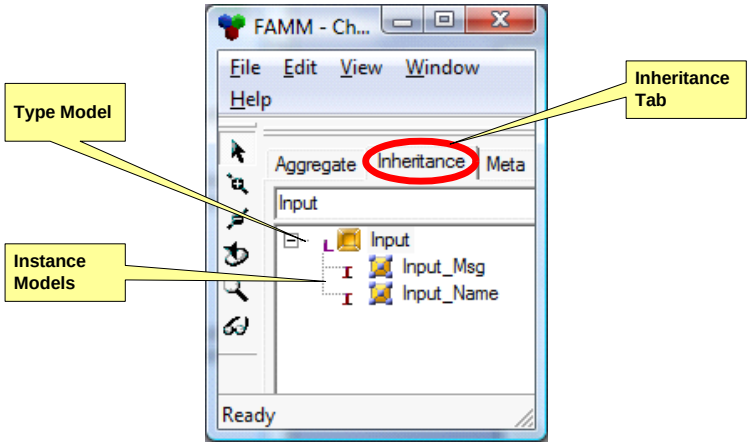


Figure 53. Inheritance Tab of GME Model Browser

4.2.2.3 Data Types

Data types are used to define the kind of model elements. MSC specification defines three predefined basic data types, namely, natural number expressions, Boolean valued expressions, and time expressions [36].

The structure of the data type metamodel is depicted in Figure 54. Data type reference is used to share the declared data types in an MSC. First Class Object (FCO), atom, and reference are all GME built-in stereotypes. The abstract model `DataTypeFromDataModel` is created to support for the integration of the user data models, which will be discussed in Chapter 6.

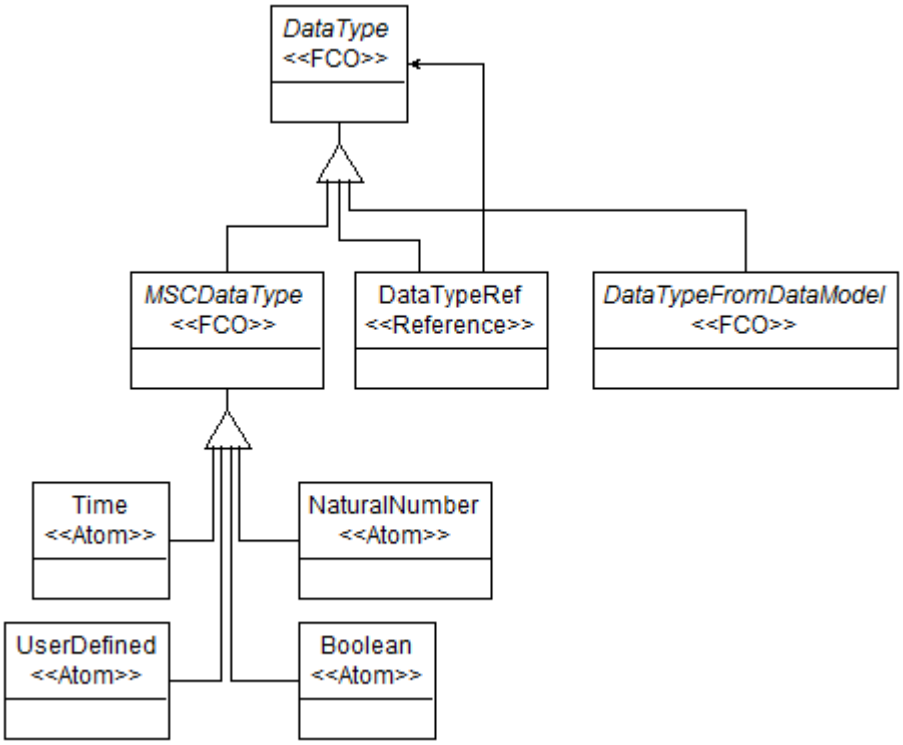


Figure 54. The Structure of Data Type Model Element

4.2.2.4 Expression

Expressions are used in certain MSC elements, such as arguments, conditions, the boundaries of a loop, and the time intervals. Expressions are domain-specific structures that occur in the data model; they might be arbitrarily complex. In the present metamodel, merely the basic expressions (e.g., `MSCBooleanExpression`) are defined. The categorization of the expressions (e.g., `BooleanExpression`) is also defined to support for the user data model. Figure 55 depicts the expressions model structure. Evaluating expressions is up to the model interpreters.

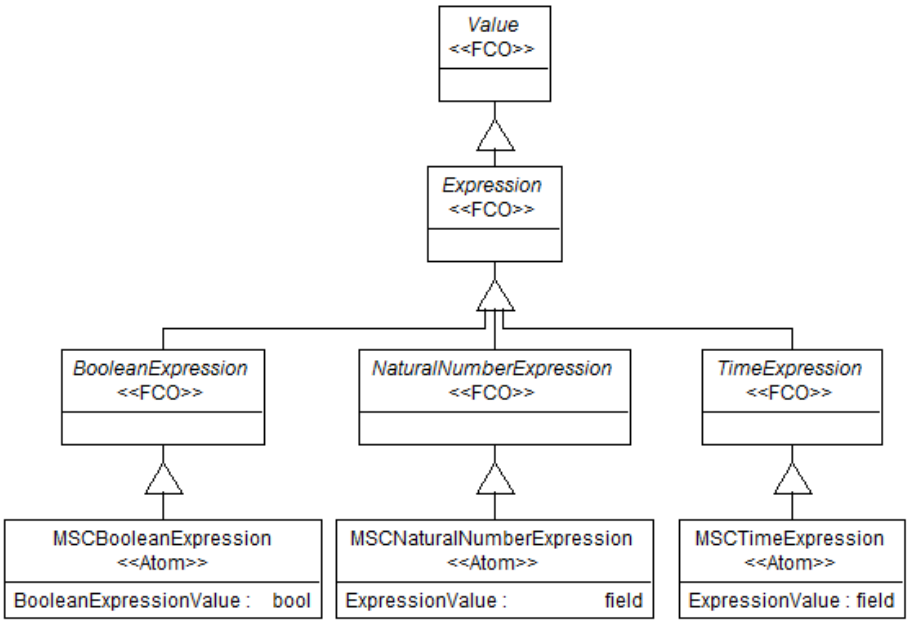


Figure 55. The Expressions

Please, note that an extension is made to the expression model in the *model integration* part of FAMM for string type expressions in order to express the integrability of the model with external data models. Moreover, an example is given to illustrate how to extend the expression model to use a domain-specific Boolean expression model in Chapter 6.

4.2.2.5 Data Definitions

Data definitions include the text for an external data language and a data field to point the external data as well as wildcards. Data definition element is used in the MSC document to declare the external data connections. A wildcard represents a "don't care" value, and it is modeled as an argument. The model is depicted in Figure 56. `DataStatementList` is used in actions (and is explained in section 4.2.1.11).

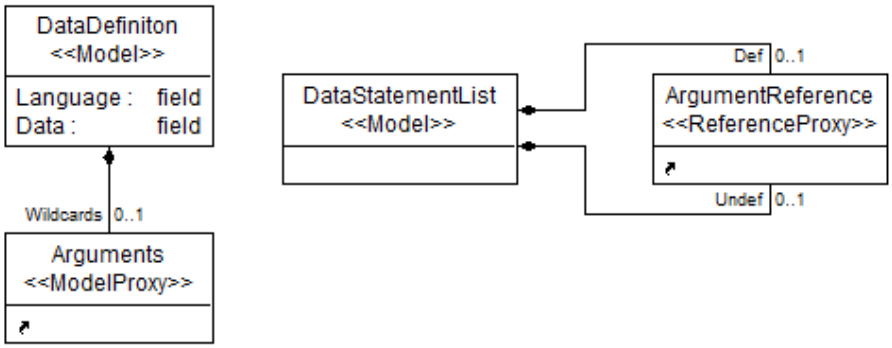


Figure 56. Data Definitions Model

4.2.3 Time Concepts

Time concepts, introduced in MSC, support the notion of quantified time [36]. Absolute or relative timing can be specified by the use of the default type `time`. Static and dynamic time variables are like any other variables except that they are of the type `time`.

Time offset is used as an offset to all absolute time values within that MSC. A time offset is defined in the head of the MSC. The structure is given in Figure 57. If the MSC has no time offset model element, then it means that the offset is equal to zero.

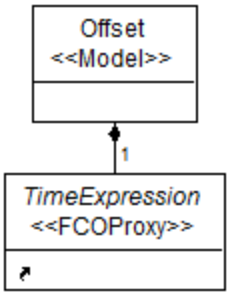


Figure 57. The Structure of Time Offset Model Element

In addition, time constraints can be defined as time points, time intervals, and measurements. Measurements and time points are used in time intervals. The structure of a measurement and a time point is presented in Figure 58 while the structure of a time interval is presented in Figure 59.

Measurement is a time observation that has a measurement type and a reference to a time variable (i.e., a variable with a default type `time`). Measurement type is either absolute or relative where the latter means duration.

Time point represents a concrete time value. The absolute mark, when set to true, indicates absolute timing. Time value is expressed in time expression attribute. Inclusion mode is only used when a time point is used in a time interval.

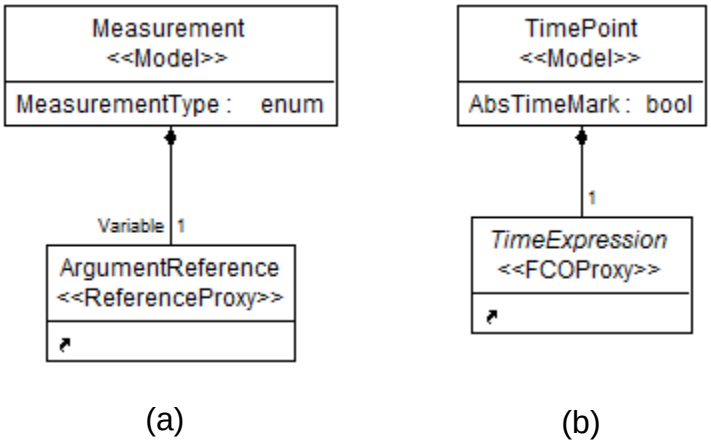


Figure 58. (a) Measurement Model Element (b) Time Point Model Element

Time intervals are used to define constraints on the timing for the occurrence of events. A time interval may include a measurement, a singular time that is either a time point or a measurement, and a bounded time. Minimal or maximal bounds can be defined as time points for the delay between two events. The inclusion mode indicates whether the bound is included or excluded.

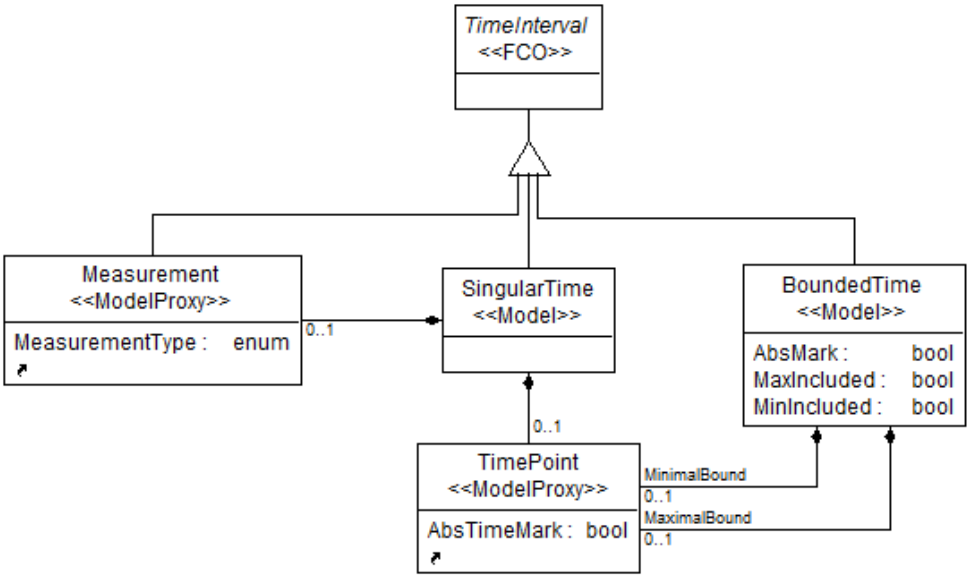


Figure 59. The Structure of Time Interval Model Element

Time intervals can be defined for any two events within an MSC document. Time intervals can be connected to the orderable events, `Top` or `Bottom` of an MSC reference or an inline expression. This connection is called *Time Address Connection* and it specifies the source and the destination events by the connection order. When an event is connected to a time interval, the event becomes a source for the time interval. Conversely, when a time interval is connected to an event, then the event becomes a destination for the time interval. Thus, the origin of the time interval is specified. Therefore, no element is required for the `origin` MSC keyword.

An example is provided in Figure 60 for the use of time constraints for MSC references. The time destination connection that shows the destination (i.e., `reference` element) for the time interval defined in `par` parallel inline operator has a `bottom` attribute set to true indicating the top of the reference region.

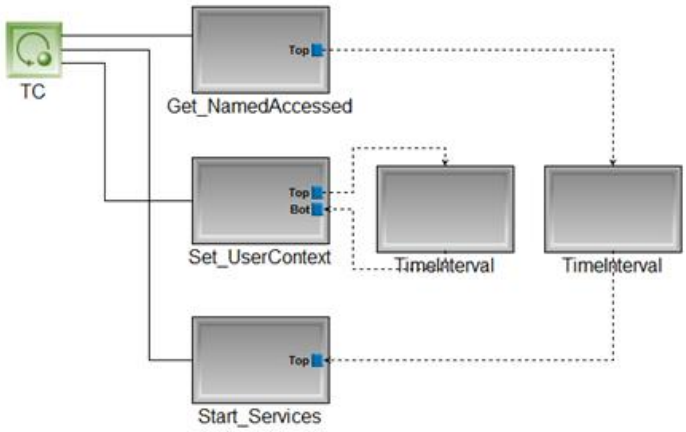


Figure 60. Example for Time Destinations

Time constraints can also be used in the High-level MSCs.

4.2.4 High-level MSC

High-level MSCs provide a way to compose MSCs. The HMSCs can be embodied in the MSC in place of MSC bodies. The structure of the HMSC is depicted in Figure 61.

An HMSC includes a start and an end node (initial and final respectively) as well as MSC references, inline expressions, and conditions. Conceptually, the inline expression and reference nodes are called timeable nodes meaning that those nodes include time intervals.

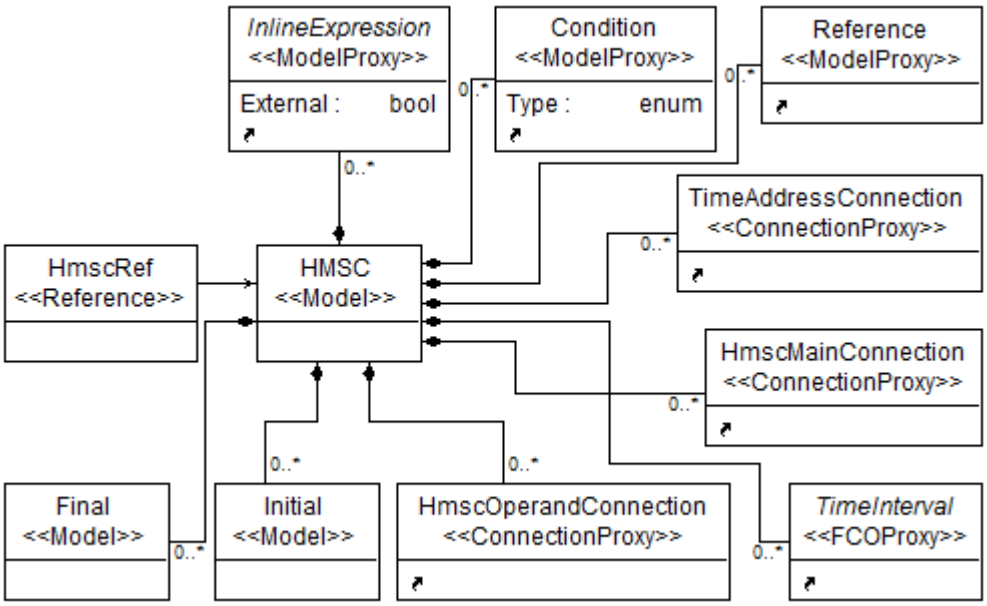


Figure 61. The Structure of the HMSC Nodes.

Each HMSC element can be connected to others to specify the composition order. In order to provide this, a directed connection is defined between the HMSC nodes (i.e., model elements). The inline expressions specify the operands that are also connected to the other HMSC nodes. For this reason, another connection is defined between the operands of the inline expression and the HMSC nodes. The former connection is called “HMSC Main Connection” while the other is called “HMSC Operand Connection”. Connection structure is depicted in Figure 62.

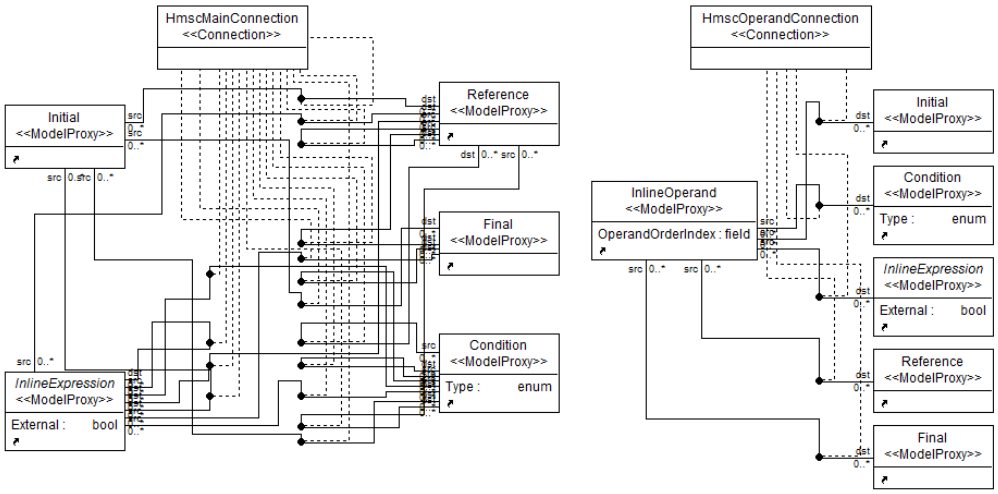


Figure 62. The Structure of the HMSC Connections.

An example is provided in the following figures showing different composition techniques using the HMSCs. First, the HMSC is given, and then the corresponding model in FAMM notation is presented.

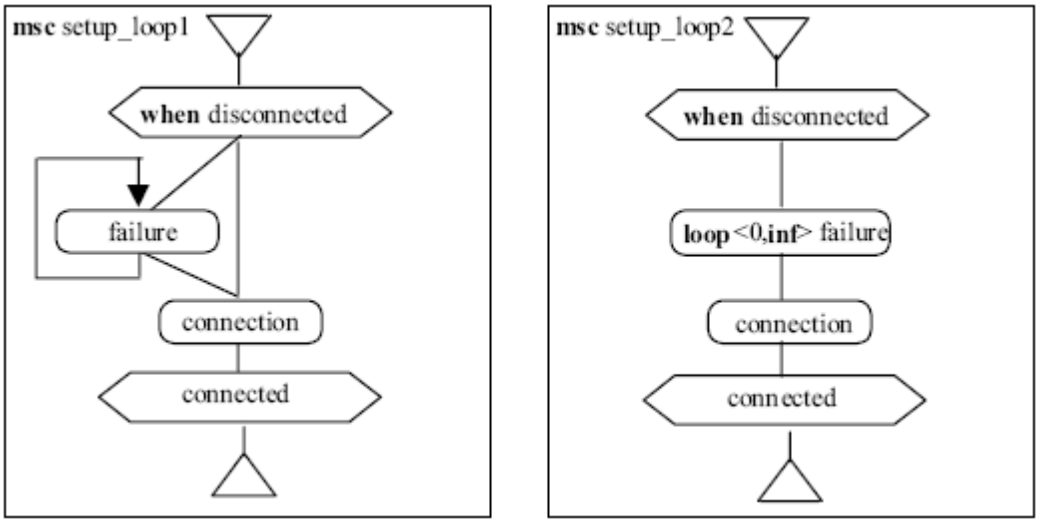


Figure 63. The HMSC Example Fig.59 from [36].

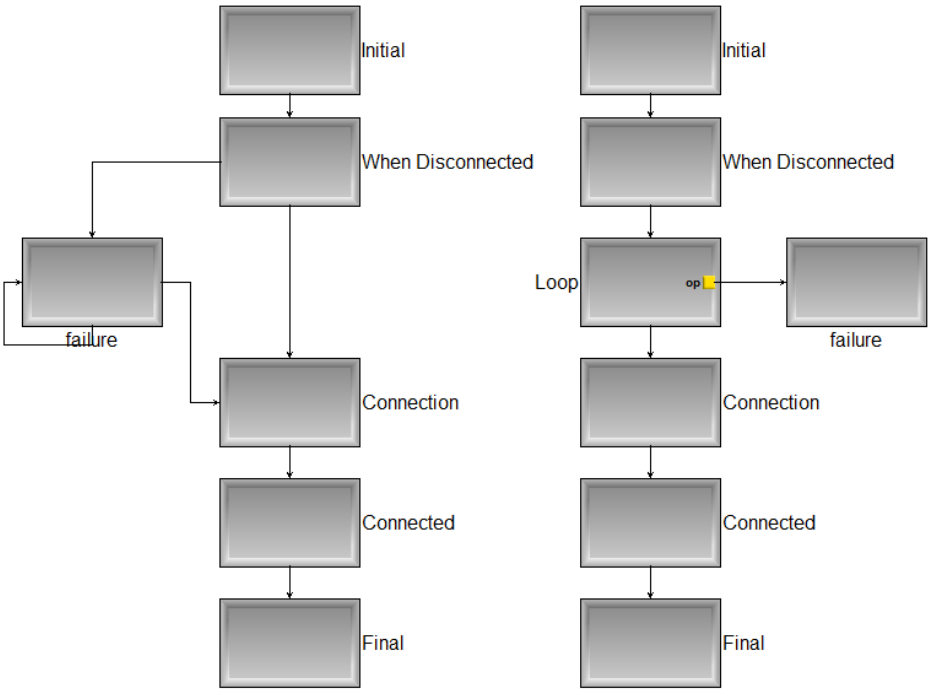


Figure 64. The Corresponding Model for HMSC Example.

4.2.5 Auxiliary Models

4.2.5.1 Events

In the metamodel, the MSC instance events are grouped as orderable events and non-orderable events as classified in the MSC specification in order to simplify the connection between MSC elements.

The classification groups are presented in Table 6.

Table 6. The Event Groups

GROUP	MEMBERS
Orderable Events	Message events, method call events, actions, timer events, and process creation
Non-orderable Events	Multiple Events, method, end method, suspension, end suspension, concurrent, end concurrent, process stop, and instance end
Multi Instance Events	Condition, Inline Expressions, and MSC reference
Message Events	In and out
Method Call Events	Call, receive, reply out, and reply in
Timer Events	Start timer, stop timer, and time-out

4.2.5.2 MSC Connections

Connection elements are used to connect and associate modeling elements with each other. There are mainly four connection types allowed in the current metamodel: *Ordered Connection, Special Connection*, *Address Connection*, and *Time Address Connection*.

An ordered connection has a `Precedence` attribute to specify the order in the connections. For example, all the connections made to the instance are ordered connections. Some modeling elements need special connections with each other, such as the set timer event, which can be connected to the other timer events. The requirements for this connection type are modeled as special connections. The address connection is used to connect the messages to the destination addresses as the time address connection is used to connect the time intervals with the source and destination events. The metamodel is depicted in Figure 65.

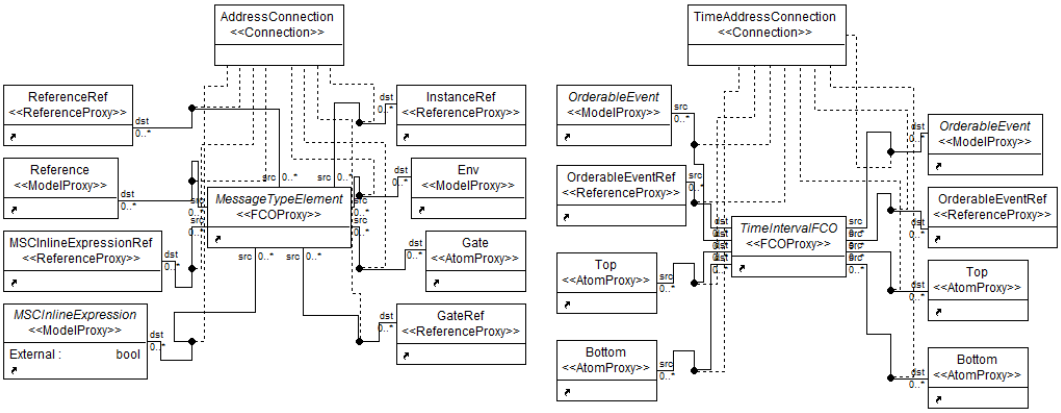


Figure 65. Address Connection and Time Address Connection.

4.3 LSC Metamodel

LSCs are introduced in [39] as an extension to MSCs primarily to enable a distinction to be made between mandatory and possible behaviors in sequence charts. Later studies [37, 42] proposed some extensions to the basic LSC.

4.3.1 Extending MSC to LSC

Before introducing the LMM extensions, we need to clarify how the LSCs are understood in conjunction with MSCs.

The *LSC Metamodel* is constructed on top of the MSC Metamodel. Although LSC is proposed as an extension to the MSC, some incongruity problems were revealed while extending the MSC metamodel to form the LSC metamodel (this can be cited as a side benefit of the metamodeling exercise).

First, there are minor disparities among the LSC papers. As there is no official standard for LSCs to date, the references [37, 49, 42] are taken as the principal specification documents. When a disparity arises, the latest dated reference is given priority. Some disparities are as follows:

- The activation modes for charts are defined as “Preactive and Active” in [42], and changed as “Initial, Invariant, and Iterative” in [37].
- A pre-chart seems mandatory in universal charts in [42], but in [37], the activation conditions are introduced for simple conditions to express an activation point for an LSC.

Furthermore, LSC is originally extended from MSC-96 specification. In our study, we base our metamodel on the MSC-2004 specification [36], the current recommendation for MSC, which improves MSC in many ways such as data and time concepts. It is unclear how LSC incorporates all the MSC artifacts such as gates, inline operators, and MSC composition techniques (MSC references and HMSC).

Such data concepts as symbolic messages, variables, assignments, classes, and symbolic instances, which are incorporated to the basic LSCs by [42] for play-out mechanism, are not included in the metamodel as the data model is inherently domain specific and can be supplied separately by the modeler. We thus achieve a separation of behavioral and data-related concerns in the metamodel. Additionally, the modeler can use the MSC data and time concepts with LSCs.

4.3.2 Extending MMM for LMM

In order to construct a new metamodel as an extension of an existing one (e.g. extending the MSC metamodel to the LSC metamodel), one could copy the existing metamodel and afterwards make modifications and additions to it. Alternatively, one may attach the existing metamodel as a library and then build the new model on top of it without any modifications to the attached library elements. The latter method, using the nested libraries feature of GME, yields better model encapsulation.

The structure of LMM is depicted in Figure 66. As seen in the figure, MMM is attached as a library (indicated with a book icon).

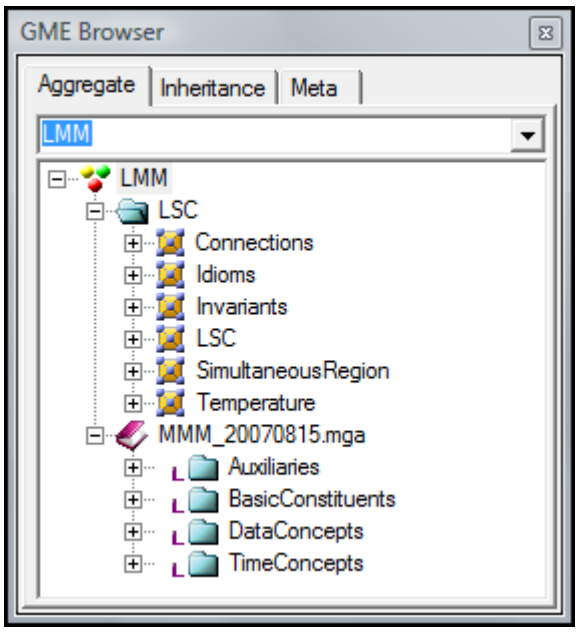


Figure 66. The LMM Implementation View (GME Screenshot)

LSC metamodel uses MMM library to extend its own model elements. GME inheritance is used as the mechanism for extending the MSC. The following sections explain how the extension is done and introduce the structure of LMM.

4.3.3 Live Sequence Charts

A live sequence chart is the main containment for the instance interactions. It matches the MSC body where all the events are defined. MSC Body, a model proxy, is an element from MMM, where events of an instance are specified. Thus, we allowed the modeler to use all the MSC elements. For example, the modeler may declare message types (templates) as he does in MSCs. The other proxy elements, *condition, inline operand,* and *reference* are also the members of MMM.

LSC has an enumerated attribute where it represents the distinction between mandatory (*universal*) and possible (*existential*) behavior on the chart level. The default value is “existential”. LSC also has an attribute to specify the activation mode defined in [37].

LSCs (for universal charts) may include a simple activation condition or a pre-chart or both wherein they behave like a precondition so that if evaluates to true, it activates the body of the LSC [37]. Activation condition for a sub-chart acts as a top-level condition [39]. A pre-chart is always existential by definition.

LSC may also include one or more sub-charts to support composition of charts. The sub-chart and the pre-chart is indeed an LSC, but its semantics is different. The structure of LSC is depicted in Figure 67.

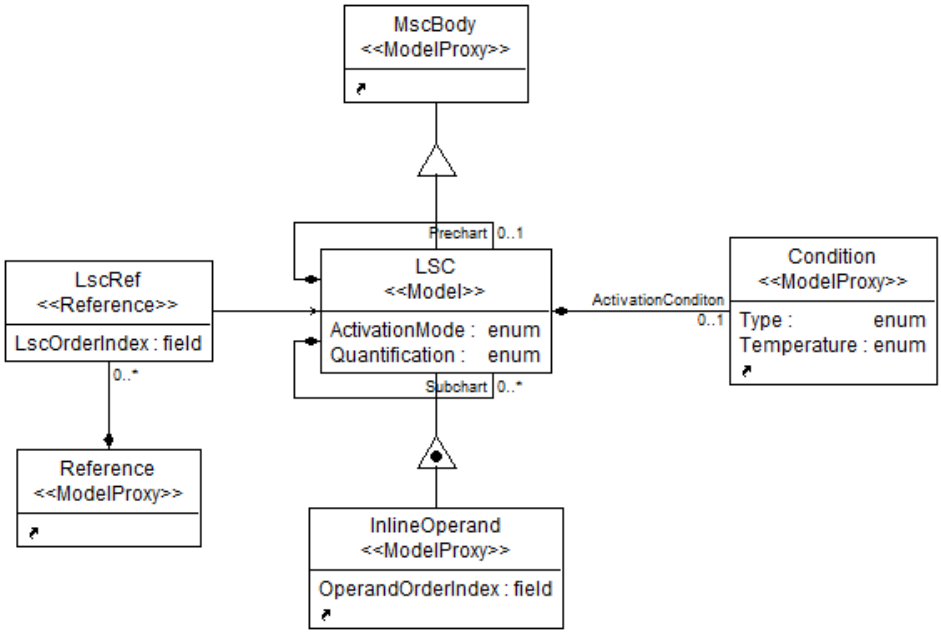


Figure 67. Extending MSC Body for LSC

4.3.4 Temperatures

Temperature of an element shows its liveliness. It can be *hot* meaning mandatory or *cold* meaning possible behavior. Charts, locations, conditions, messages, and local invariants have temperature, which is added as an enumerated attribute. The meaning of temperature has to be tailored to the element for which it is specified.

4.3.5 Locations

Temperature represents the mandatory or possible behavior. Charts, conditions, messages, and local invariants have temperature. Moreover, when an event is connected to an instance, this connection represents a location and it can also be marked as cold or hot. Thus, some interesting combinations of temperatures can occur. For example, a hot message can be connected to a cold location.

The combinations between the temperature of the location and the temperature of the event that is connected to that location need some clarifications. We adopt the

fairness assumption that an enabled event cannot be delayed indefinitely. Table 7 explains the interpretations for such combinations with an emphasis on code generation.

Table 7. Interpretation of Location and Event Temperature Pairs

	HOT LOCATION	**COLD LOCATION**
HOT EVENT (MESSAGE)	Event must occur (e.g., a message is sent or a message is received) and progress is forced to the next event.	Event must occur (e.g., a message is sent or a message is received) and progress is forced to the next event.
COLD EVENT (MESSAGE)	Wait for a pre-defined (as a configuration parameter) duration/number-of-trials for the event to occur. When the event has occurred (e.g., a message is received) or the trial period is over without the event occurrence, continue to the next event. For sending a message, code generator randomly decides to send or not. Not sending amounts to message getting lost. For receiving messages, generated instance code waits for a pre-defined period of time, if the message has not received till then, it is assumed to be lost.	Wait for a pre-defined duration/number-of-try for the event to occur. If the event has occurred, continue to the next event. Else, abort (break) the chart. For sending a message, generated instance code randomly decides to send or not. For receiving messages, generated instance code waits for a pre-defined period of time. If the message has not been received till then, it is assumed that progress beyond this event is not possible, so the chart is aborted.

The combination of a cold event and a cold location is especially useful for pre-charts to characterize the pre-conditional events. As an example, see the LSC in Figure 112.

4.3.6 Conditions

Conditions can be selected as hot or cold, where a hot condition must be satisfied (violation is an error) and a cold condition simply means an exit from the enclosing chart. A condition also acts as a synchronizing point if and only if it is connected to more than one instance.

4.3.7 Messages

Messages are the same as MSC messages except for two extra attributes. One is for specifying the temperature of the message, and the other is for specifying whether the message is time delayed (i.e., asynchronous) or instantaneous (i.e., synchronous).

4.3.8 Simultaneous Regions

Simultaneous regions are used to group several elements, which should be observed at the same time. As far as the simultaneous regions are reference points, they are represented with one modeling element rather than as a region with start and end nodes such as in the coregion element.

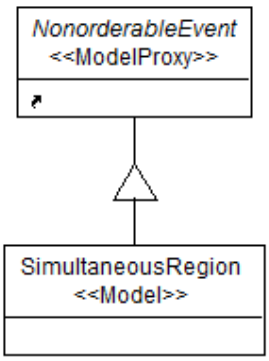


Figure 68. LSC Simultaneous Region Metamodel

An example for the usage of the simultaneous regions is given in Figure 71.

4.3.9 Local Invariants

A local invariant expresses the satisfaction of a condition over a period. They can have temperature, with the same interpretation as for conditions. Since they cover a period, they need reference points for the start (`Invariant`) and the end (`EndInvariant`) [37]. The start element has an expression just as conditions do. It can be possible or mandatory, with the same interpretation as for a condition. When the start element marked as cold or hot, then the same is assumed for the end element. Furthermore, start and end elements can be marked as included or excluded using the *inclusion mode* attribute.

The structure of the invariants is presented in Figure 69.

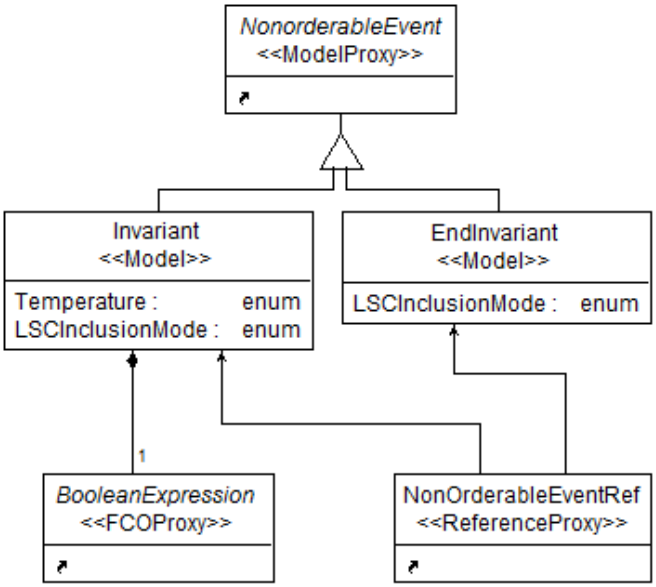


Figure 69. LSC Invariant Metamodel

An example is presented in Figure 70. In the figure, instance `A` has an invariant through its instance lifeline connected with the simultaneous regions denoted as black dots.

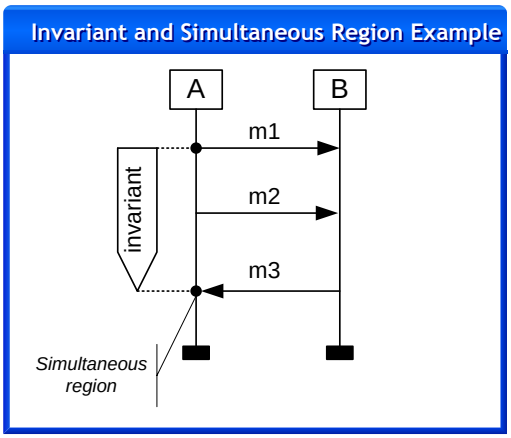


Figure 70. LSC Invariant and Simultaneous Region Example

The corresponding model for instance `A` is presented in Figure 71.

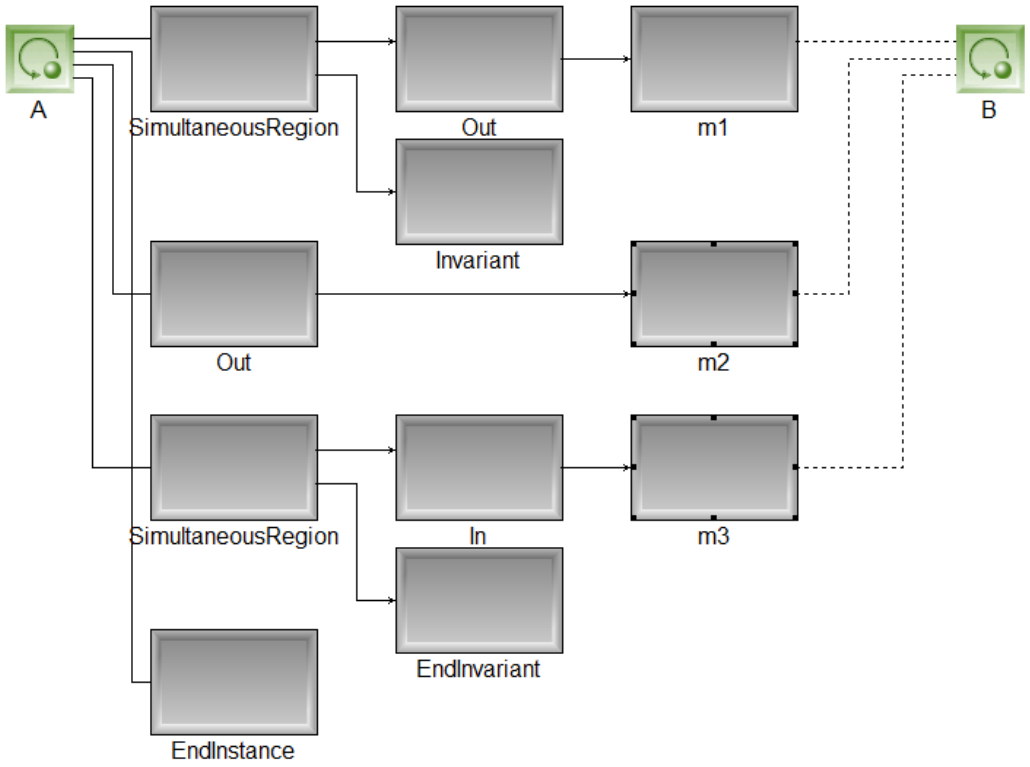


Figure 71. LSC Invariant and Simultaneous Region Corresponding Model

4.3.10 Time (Timing Intervals)

For the specification of time concepts of LSC, reference [37] is taken into consideration while reference [42] also offers another time concept.

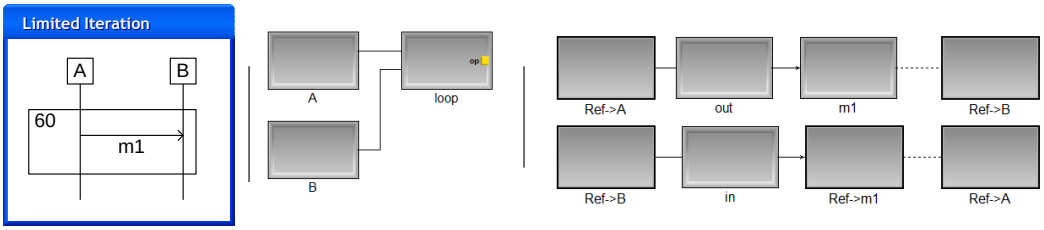
In LSCs, MSC timers and time concepts can be used as they are. In addition, time constraints can be specified with a timing interval. Timing intervals with lower and upper bound are used to give both a minimum and a maximum delay between two directly consecutive events (i.e., message out, message in, instance axis, and instance end).

4.3.11 Iteration and Conditional Execution

Interestingly, LSC iterations are not based on MSC loops. We have chosen to provide iteration in LSC as the loop construct in MSC. Thus, another iteration model element is not needed; instead the MSC loop is extended to cover the dynamic loops (for which the loop count is to be entered by the user at execution time).

In [42], three kinds of loops are defined, namely, fixed loops, unbounded loops, and dynamic loops. The MSC loops cover the first two kinds where iteration can be fixed by defining the min and max iteration bounds or can be unlimited using `inf` keyword as opposed to using an asterisk specified in [42]. For the third kind of loops, the MSC loop is extended by creating a Boolean attribute to specify that the loop is dynamic or not. If this attribute is set to true, then the iteration of the loop will be defined in run time and the iteration bounds become “don’t care” attributes. The value is false by default.

An LSC chart for fixed iteration taken from [38] is modeled and presented in Figure 72.



(b) Chart view where loop has one operand and one boundary element

(c) Inside of the loop operand

Figure 72. Fixed Iteration Example

4.3.12 Additional LMM Elements

While constructing LMM, two minor extensions are done to LSC. One is to extend the usage of LSC pre-charts (e.g., using pre-charts in inline operands) and the other is to package some well-known constructs as idioms (e.g., while-do iteration).

4.3.12.1 Extended Pre-charts

Pre-charts, in their original manner, can only be used with the universal LSCs wherein they behave as a precondition in that if a pre-chart completes its specified behavior; the body of the LSC is activated [37]. In some behavioral patterns; however, it is essential to use pre-charts within the inline operands. Thus, one can specify a conditional behavior pattern in inline operands.

For instance, a bank client can interact with an Automatic Teller Machine (ATM) by selecting money-withdraw operation or exit operation. According to each type of action, the ATM responds with a series of events. When the client selects the

withdrawal, then the ATM asks the amount of the withdrawal request. The behavior model for processing the menu is seen in Figure 73.

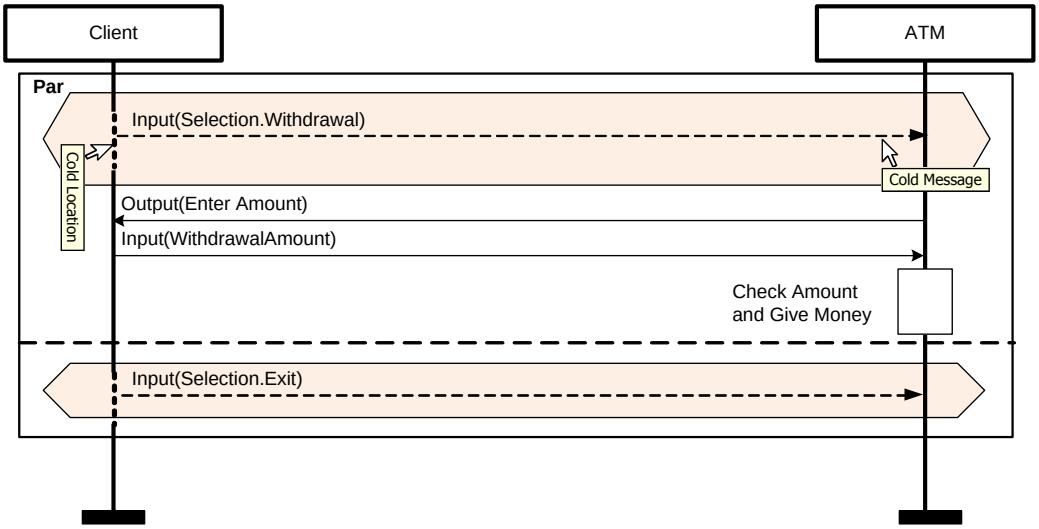


Figure 73. LSC for Process Menu Selection

Consequently, the pre-charts are extended so that they can be nested in the metamodel and can be used within the inline operands. Moreover, the pre-charts can be nested in the pre-charts.

The inline operands are instantiated from the `MSCBody` and `LSC` so that they become charts where many MSC and LSC elements can be defined. The structure of the inline operand element and the pre-chart is depicted in Figure 74.

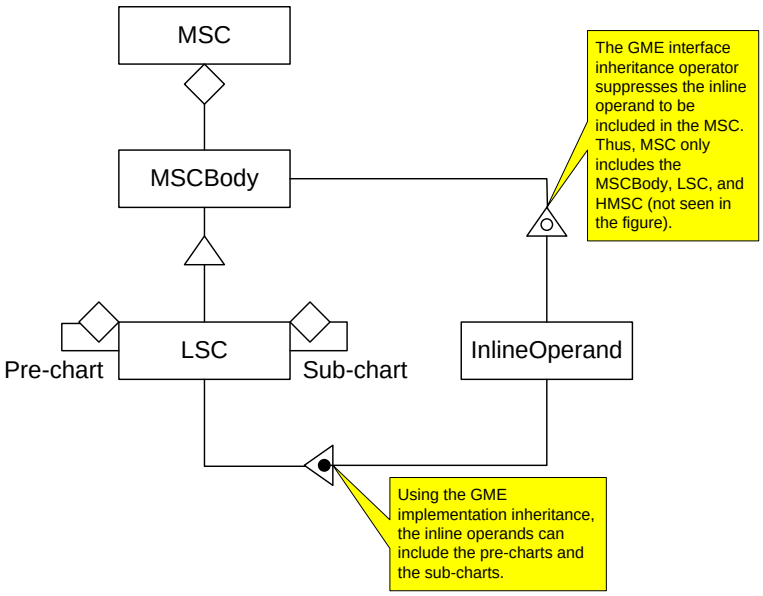


Figure 74. The Structure of Inline Operands and LSC Pre-Charts

4.3.12.2 Idioms

By combining iterations and cold conditions, conditional and repetitive execution (i.e., Repeat-Until (a.k.a., Do-While), While-Do, If-Then, If-Then-Else) can be created [38]. In order to simplify these combinations, an idiom for each is constructed as a new modeling element for conditional and repetitive execution.

The set of idioms is expected to enlarge as more experience is gained with the FAMM usage.

The following example illustrates the `Repeat_Until` construct. This sample chart is taken from [38].

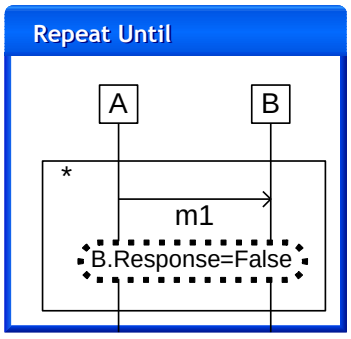


Figure 75. Repeat-Until LSC

The corresponding model is given both by using the idioms and by using the loop construction. The main difference is the location of the condition. When using the idiom element (i.e., `repeat_until`), the condition is provided in the element body while when using the loop, the condition is attached to the instances.

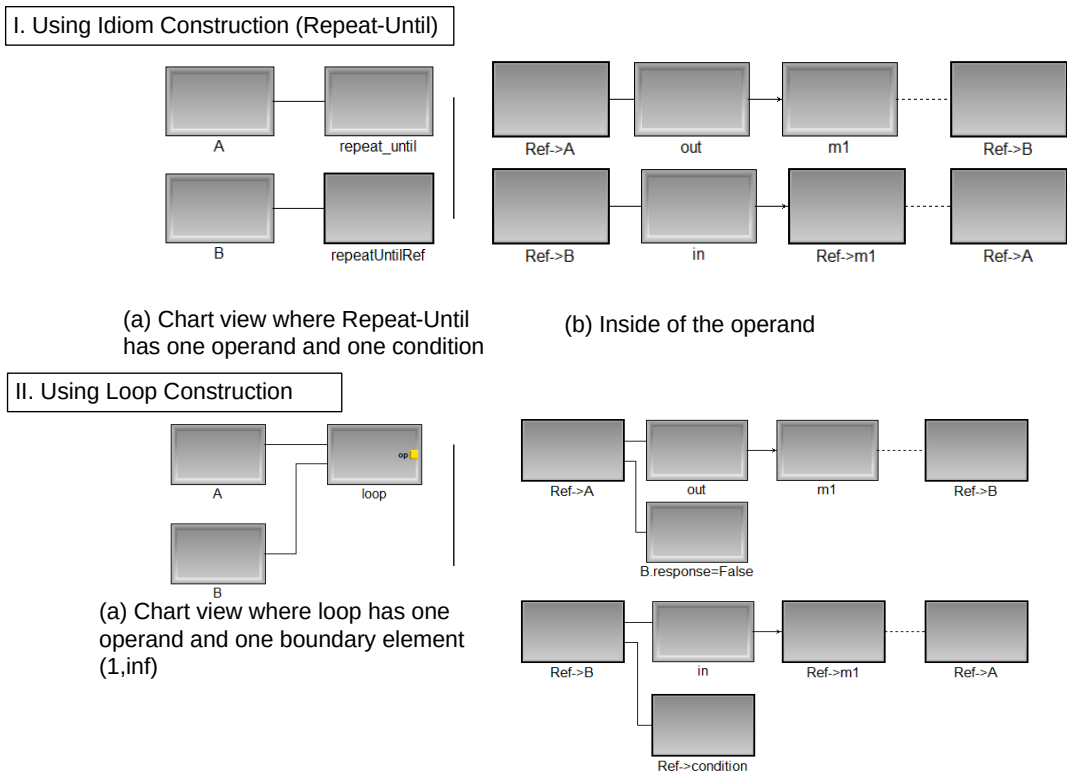


Figure 76. Corresponding models for Repeat-Until

CHAPTER 5

HLA FEDERATION METAMODEL

This chapter introduces the HLA federation metamodel and its sub-metamodels. The material in this chapter is based on [53].

5.1 HLA Object Metamodel

The *HLA Object Metamodel* is constructed to provide a domain specific (i.e., HLA in our case) data model for the behavioral models. HOMM fully accounts for [3] and can be regarded as an alternative rendering of the HLA OMT [34].

The HLA OM paradigm includes the Object Model paradigm sheet and the OMT Core folder. Paradigm sheets are separate portions of metamodels. The OM paradigm sheet includes the main diagram for object models. The OMT Core folder includes the table contents specified in the HLA Object Model Template. Top view is presented in Figure 77.

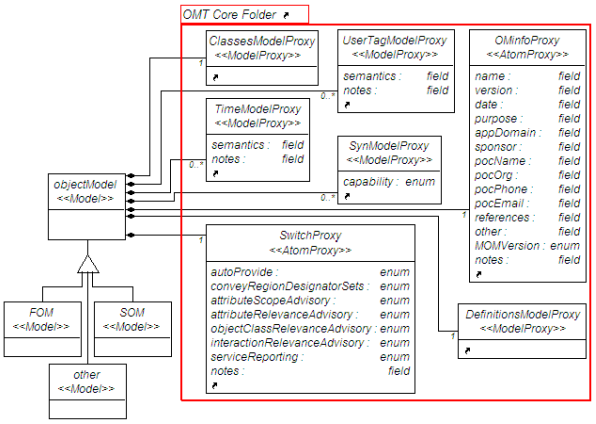


Figure 77. Object Model Top View [34]

Note that based on this metamodel, the IEEE HLA Defaults library (HDefLib) and HLA Management Object Model library (HMOMLib) are modeled as helper libraries. For further details, please refer to [34].

5.2 Federation Structure Metamodel

The *Federation Structure Metamodel* represents the structural aspect of the federation. This metamodel is created for the developer to define a federation and its participating federate applications, and to easily connect them to their respective FOM and SOMs. In this metamodel, the participating federate applications are emphasized and their corresponding SOM's can be specified in addition to the FOM. The FOM and SOMs that are referred by this model are prepared based on HOMM.

Each federation structure can include only one federation and one FOM, while there may be any number of federates and SOMs. Figure 78 shows the GME paradigm sheet of FSMM. There is a `MemberOf` association between the federation and federates, indicating potential federation execution members.

Federate application is discussed in Section 5.4.5.4 and an example for the use of the FSMM is provided in Figure 103.

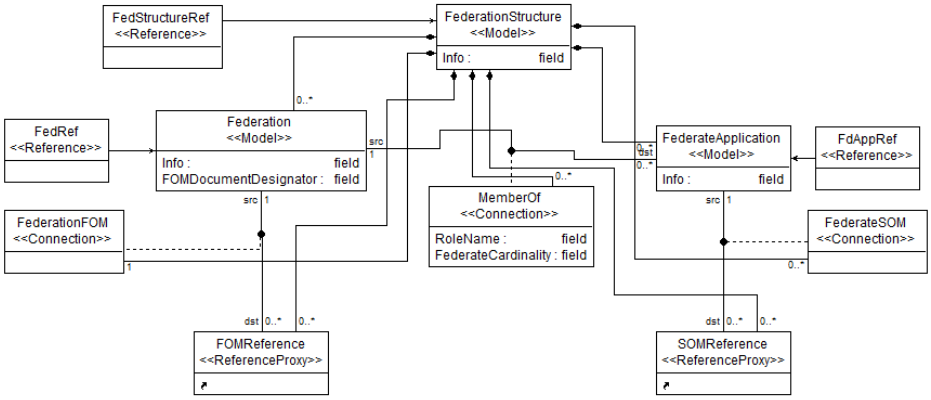


Figure 78. Federation Structure Paradigm Sheet (modified from [34])

5.3 Publish/Subscribe Metamodel

Publish and Subscribe diagrams are introduced in [23] as design artifacts to focus on the object/interaction interests among the federates. In the current metamodel, some minor modifications were made to connect it with the HLA object model.

PSMM is not a core metamodel of FAMM, instead, it is devised to illustrate that FAMM can be used to generate useful views for the federation designers. After the federation architecture is modeled using FAMM, an interpreter, called P/S Model Generator, traverses the FAM to extract the federation publish and subscribe interests of the federates, and then builds the P/S models of the federation.

Publish/Subscribe metamodel is depicted in Figure 79. Metamodel allows interpreter to generate the Federate-based P/S Diagrams as well as the Class-based P/S Diagrams.

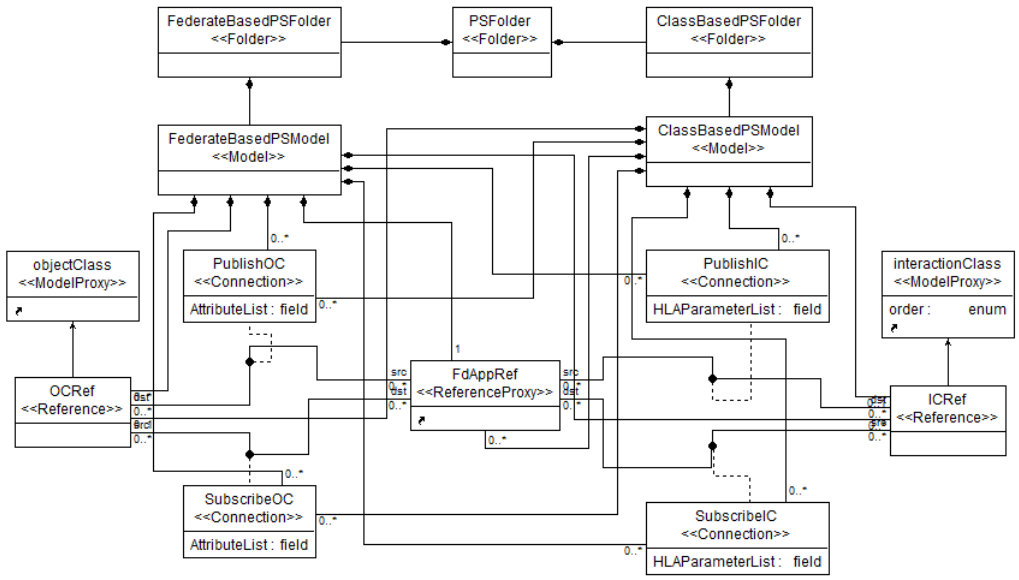


Figure 79. PSMM

5.4 HLA Services Metamodel

The HLA Federate Interface Specification defines the standard services of and interfaces to the RTI. These management services provide a functional interface between federates and the RTI. They are arranged into seven groups: federation management (FM), declaration management (DM), object management (OM), ownership management (OwM), time management (TM), data distribution management (DDM), and support services [2].

The *HLA Services Metamodel* includes the necessary modeling elements to model the HLA services interface. The primary modeling approach for constructing HSMM is to separate the HLA method specifications and the concepts that constitute the

HLA services. An HLA service defines an interface, where an HLA method is a realization of that service. For example, the “Reflect Attribute Values” service can be mapped to more than one “Reflect Attribute Values” methods with different argument sets due to the optional arguments in the service specification. The HLA Services Metamodel merely provides the constructs, namely, methods, arguments, and exceptions to model the HLA services specified in [2]. Afterwards, the HLA methods are defined as a GME library based on this metamodel. This approach offers the following advantages:

- First, it supports modeling different HLA interface specifications in forms of GME libraries such as IEEE 1516 and 1.3 along with the DoD interpretations to both, without changing the metamodel. The IEEE 1516 Methods and DMSO RTI NG 1.3 library are included in the present work.
- More specifically, it enables construction of the “method” libraries for particular RTI implementations such as Pitch RTI (pRTI). For instance, the pRTI library has methods `attributeIsNotOwned` and `attributeIsOwnedByRTI`.
- It also has the potential to support project-specific RTI abstraction layers.
- As the HLA interface specification evolves, it will be easy to modify the metamodel to support the new libraries.

Regarding another metamodeling issue, it is worth noting that cardinality constraints in the standard are preserved in the metamodel. For instance, as seen in Figure 81, an HLA method shall have exactly one “exceptions” model that accounts for the method exceptions.

5.4.1 Connection to the Other Metamodels

Some RTI methods require a connection to the HLA Object Model. HSMM allows the required connections to be made between the behavioral model and the HLA Object Model by offering reference elements that refer to the other metamodel elements. This referencing mechanism links the method parameters to the related model elements.

For example, the “publish interaction class” method requires an “interaction class handle” to specify which interaction class, defined in the FOM, will be published. In

modeling, this argument is provided to the user as a reference to the interaction class element in the FOM. Thus, the interaction class to be published is specified.

When generating code, thanks to this referencing mechanism, it is easy to traverse the model. GME provides the necessary API for model traversing.

5.4.2 Methods

The labels of the messages in a generic MSC/LSC are un-interpreted; they are just symbols. In the context of a federation architecture, though, the labels must be interpreted. That is, the messages must correspond to the RTI method calls (including callbacks), and refer to the HLA objects and interactions defined in the FOM. HLA methods are simply modeled as interpreted LSC messages which can be exchanged with simple message events (i.e., `in` and `out`) or method call events (i.e., `call`, `receive`, `reply out`, and `reply in`).

In the metamodel, the HLA services are indicated as being RTI-initiated or federate-initiated by means of the enumerated attribute `initiator` of `HLAMethod` element as sketched in Figure 80. This attribute is devised to create verification points to help consistency checking in model verification. For example, if a federate initiates an RTI-initiated method, which indeed it should not; an interpreter may catch this design error by merely checking the initiator attribute.

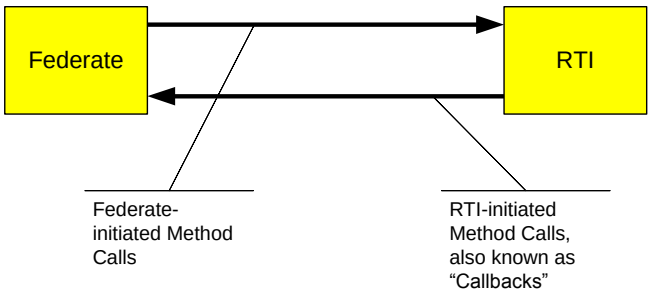


Figure 80. RTI/Federate-initiated Methods

The top view of HSMM is presented in Figure 81.

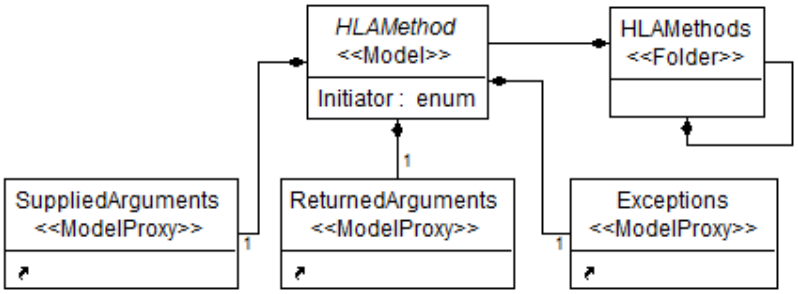


Figure 81. HLA Method of HSMM

5.4.3 Arguments

Each HLA method has arguments, either in "Supplied Arguments" form or in "Returned Arguments" form. Accordingly, a container modeling element is introduced for each of them, called `SuppliedArguments` and `ReturnedArguments`, respectively. Arguments model contain all the arguments of the services specified in [2] as well as the additional arguments for DMSO RTI NG 1.3.The structure of the arguments model is presented in Figure 82.

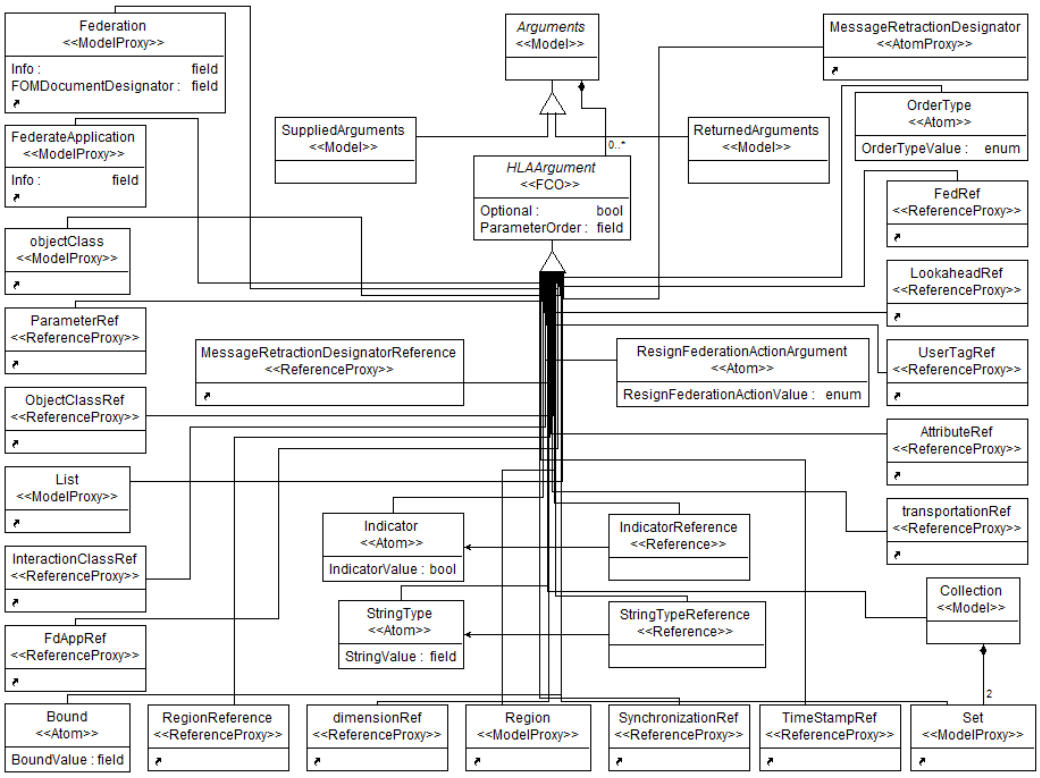


Figure 82. Method Arguments Model

Many of the arguments are references to the other modeling elements (e.g., lookahead reference, synchronization reference, transportation reference, etc.), but some are made up by us, namely, indicator, order type, numeric type, resign federation action, and string type arguments. The `Indicator` element is created for the representation of Boolean arguments such as registration-success indicator. The `OrderType` element is created for the representation of the order type arguments such as the sent message order type. The `StringType` element is created for the representations of the label type arguments such as the federation save label. The `NumericType` element is created for the representation of numeric arguments such as the region bounds and normalized values. Resign federation action argument is an enumerated element that represents the actions taken by the federate when resigning federation. The details of arguments for the library developers are given in Appendix D.

Each argument element has an `Optional` attribute to indicate whether the argument is optional or not in the method call and a `ParameterOrder` attribute to specify the order of the parameter in the method call.

For null arguments in an HLA method call, it is enough to leave the returned or the supplied argument containers empty.

Set, List, and Collection models correspond to their counterparts in the HLA interface specification. Sets and lists include pairs such as a federate and restore status pair while the collections include only the sets. The pair structure is depicted in Figure 83.

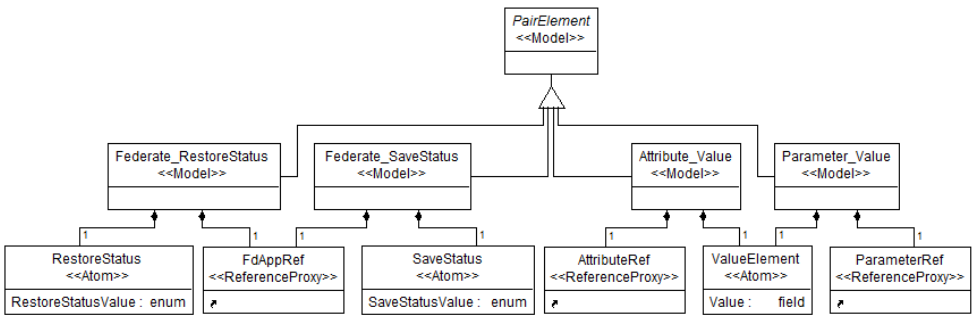


Figure 83. Pairs Model

Synchronization references are used to make the connection between HLA interface method parameters and HLA object model element. For example, one

can introduce static synchronization points in the object model, and then design the dynamic synchronization behavior of the federation in connection with federation object model.

5.4.4 Exceptions

For each exception of HLA services, there is only one modeling element, called `Exception`, in the `Exceptions` container model. In the HLA Methods library, the exception modeling element is named according to the exception name in the standard. For instance, the “RTI internal error” and “restore in progress” exceptions of method “join federation execution” are the same type of exception, the only difference lies in their names. The structure for exceptions is depicted in Figure 84. The Common Exceptions folder is used to group common exceptions, which can be thrown by more than one method, for instance, the “RTI internal error” exception. Therefore, only one exception is created and located in the “Exceptions” folder and then the reference to this exception is used in the HLA methods.

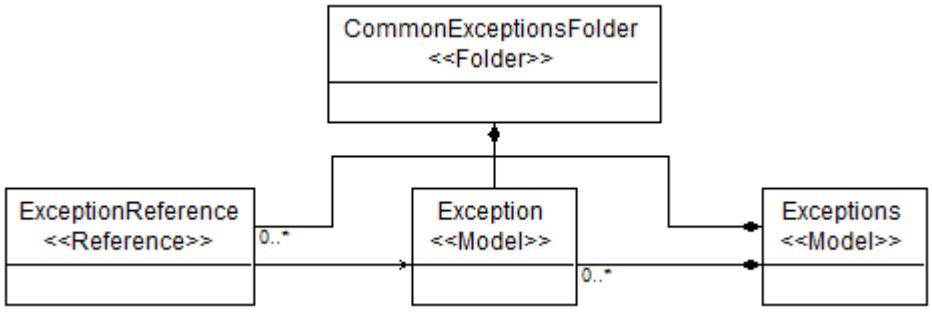


Figure 84. Exceptions Model

Exception modeling element may contain an MSC reference. Thus, the modeler can define a behavior (in terms of an MSC chart) handling the exception.

5.4.5 Runtime HLA Instances

The method arguments can be classified as static or dynamic according to their creation time. A static argument generally refers to the static objects in the object model such as an HLA object class (defined at design time) while a dynamic object (i.e., instance) is created at runtime to refer to a runtime instance such as an object instance, message retraction designator, federate, or region.

Dynamic arguments that are used by other methods during design and judged as being primary modeling elements for the federate's behavior are logically classified into two groups: those with static counterparts in the metamodel and those without.

The former ones can be instantiated directly from the counterpart model using the GME built-in instantiation mechanism. The static counterparts act as template classes for instantiation. These arguments are object instances and federate instances. An object instance is an instance of an HLA object class found in the object model and this element can only be created by instantiating the `ObjectClass`. Other method calls can reference this object instance via the `ObjectClassReference` modeling element as the object instance designator. The federate instance is discussed in Section 5.4.5.3.

For those that do not have static counterparts in the object model (i.e., message retraction designators and regions), a new modeling element is devised in the auxiliary metamodel. These dynamic elements are modeled as symbolic instances that represent the actual values produced by the RTI. The rationale for modeling message retraction designators and regions is presented in the following paragraphs, along with some examples.

5.4.5.1 Message Retraction Designator

Some OM methods use message retraction designators to keep track of the messages sent to the other federates. During runtime, the RTI automatically assigns a message retraction designator after "Update Attribute Values", "Send Interaction", "Send Interaction with Regions", and "Delete Object Instance" calls. This runtime instance must be referred to symbolically in the design time to keep track of the message calls made. Thereupon, a `MessageRetractionDesignator` modeling element is added to the metamodel. This element corresponds to the assigned designators; however, the object management and time management methods that need this designator use "the designator references" to point the assigned designator. For example, as seen in Figure 85, the `Retract` and `RequestRetraction` methods use the "Message Retraction Designator" references resulting from some object management calls (e.g., send interaction and receive interaction).

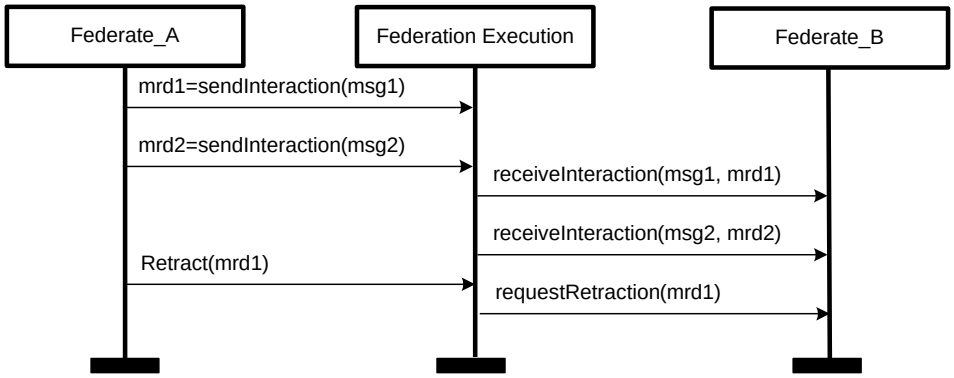


Figure 85. Using Message Retraction Designator

5.4.5.2 Region

Some HLA data distribution management methods define and use regions during run time. The HLA Object Model allows defining dimensions, which are static elements and used by federates to define regions. To use the regions at federation design time, a `Region` modeling element is introduced. The “Create Region” method call creates the region designator while other DDM method calls (e.g., “Delete Region” call, which deletes a created region) refer to it by using the `RegionReference` modeling element.

5.4.5.3 Federate and Federate Application

The HLA standard makes a distinction between federate and federate application. Federate application is akin to component type (in Architecture Description Languages parlance, see [43]) and is associated with an HLA SOM, while a federate is akin to a component, which exists at federation execution time. When a federate joins a federation execution, the RTI generates a federate handle for it. All federate handles are unique in a federation execution. As an example, a virtual “Ship” which is capable of joining a federation is a federate application. This federate application may join a federation (more than one federation, for that matter) as many federates, say ship A, ship B, and so on, during federation execution.

Hence, it seems appropriate to maintain this distinction in the metamodel. The `FederateApplication` represents the software elements, which are connected to a SOM in the federation architecture and it helps to describe the federation structure in the static view. A federate is created by instantiating the

`FederateApplication` and it is referred to in HLA methods such as `JoinFederationExecution` method. This modeling element has a reference (named `FederateApplicationReference)` to be used by the FM calls that need the federate handle. For example, `RegisterSynchronizationPoint` requires knowledge of which federates are currently joined to the federation, as seen in Figure 86.

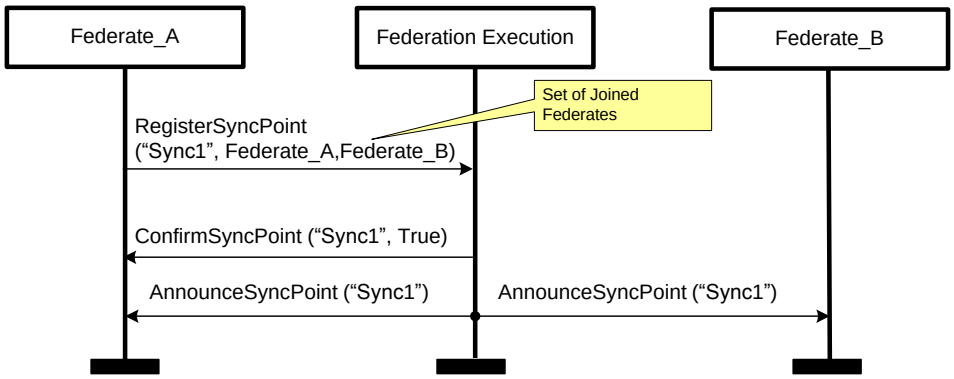


Figure 86. Using Federates

There is a difference between the concepts of federation and federation execution in parallel with federate application and federate, respectively. Federation executions are instantiated from the `Federation` model element, which is created in the Federation Structure Model. Consult [2] for further clarifications of the terminology.

Federation and federate application are specialized from LSC instances using GME inheritance (denoted by a triangle) [11].

5.4.6 HLA Runtime Infrastructure and Federation Executions

HLA Runtime Infrastructure is not explicitly referred in the metamodel. Federation execution, though, is an instance model element of Federation. A federation execution is the primary instance the federate interacts with (e.g., joining the federation, receiving an interaction etc.). A federation is modeled as an LSC instance rather than as the MSC/LSC environment. This decision leaves the environment concept at the disposal of the modeler and allows multiple federation executions in a model. While inheriting the federation element from the LSC instance, instead of the usual inheritance operator, the GME interface inheritance

is used, so that the federation execution becomes a black box. Thus, the federation execution does not contain any MSC/LSC instance constituents such as a variable list.

5.4.7 Live Entities

In interactive and live simulations, the users (players) and live entities, such as a real ship, play an essential role in the federation and federate's behavior. Therefore, a modeling element is created as an LSC instance using GME interface inheritance to represent live entities.

5.4.8 Libraries for HLA Methods

The HSMM supports modeling of the HLA methods as GME libraries. In this respect, two libraries are constructed for representing the HLA methods, namely IEEE 1516 HLA Methods library and DMSO 1.3 Methods library, using the HSMM in order to test the metamodel and to prepare the required methods to model the federate behavior both for IEEE 1516 and DMSO 1.3 compatible federations.

As seen in Figure 87, DMSO 1.3 Methods library is a programming language specific library whereas IEEE 1516 HLA Methods library is an abstract library, which is language unspecific. Both are provided in this study providing the evidence of HSMM support for HLA specifications. Furthermore, the modelers can also model the programming language specific libraries that conform to the IEEE 1516 specification, such as pRTI library.

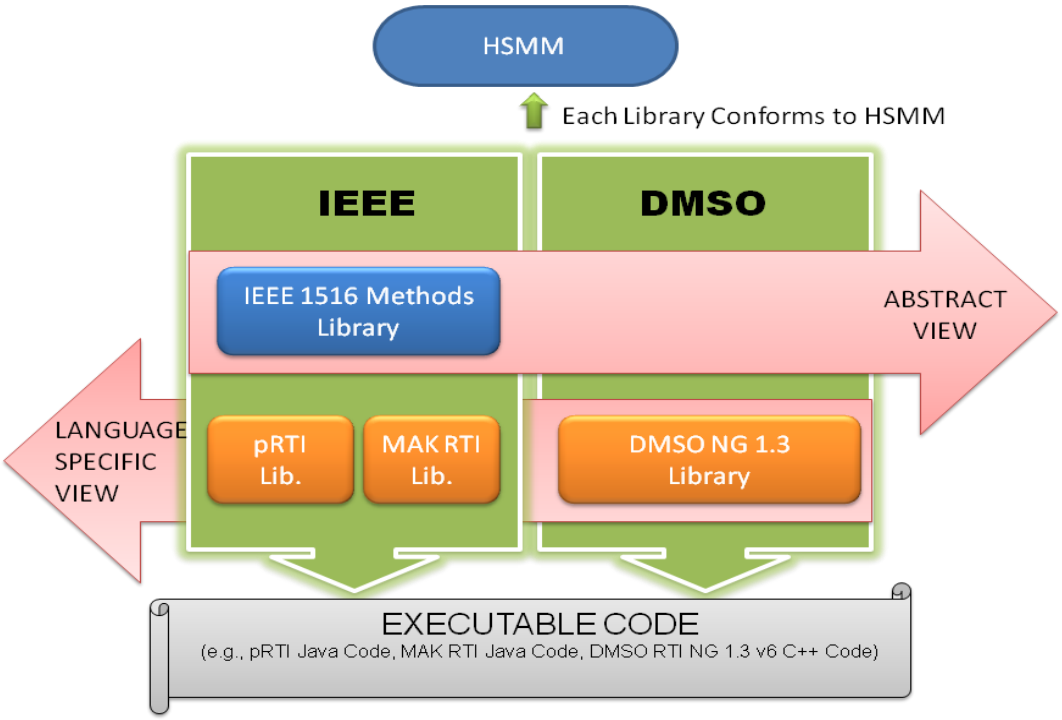


Figure 87. HLA Methods Libraries

5.4.8.1 IEEE 1516 HLA Methods Library

An IEEE 1516.1 HLA Methods library (IMLib) is providing an abstract view (i.e., programming language unspecific) of HLA services defined in [2] and it is the library required to model the federate behavior for IEEE 1516 federations. A screen shot of the library is presented in Figure 88. There are 811 concepts[1] (elements with a kind of model, atom, and folder stereotypes in GME parlance) defined in the library. The mapping of the HSMM elements to the method arguments of the IMLib is presented in Appendix C.

[1] For IMLib version 20071217_01.

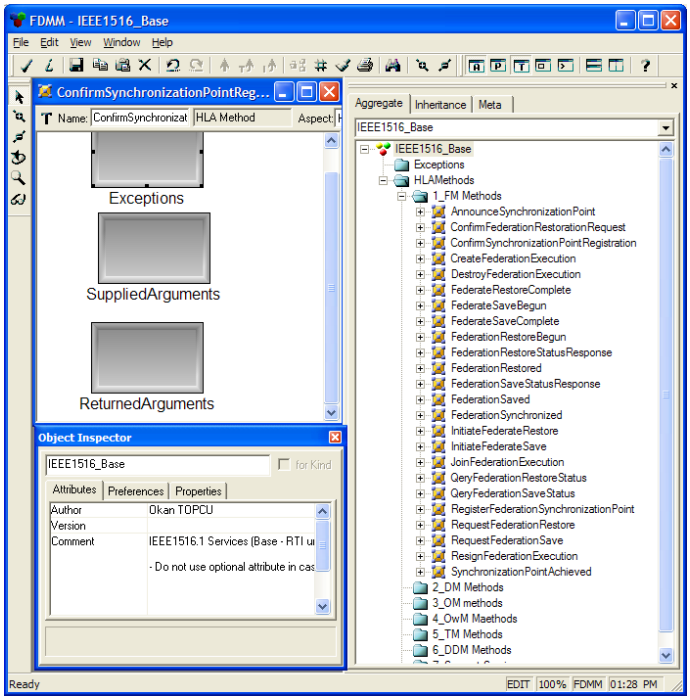


Figure 88. IEEE 1516.1 HLA Methods Base Library (GME Screenshot)

In order to use this library, we simply attach it to the model concerned and then instantiate the required method in the model (in a drag-and-drop fashion). It is also possible to refer to a method by the GME referencing mechanism.

Although the HLA management services are defined in full detail in [2]; it might make sense to at least recap what they are.

HLA Federation Management Interface refers to the creation, dynamic control, modification, and deletion of a federation execution. HLA Declaration Management Interface is used to coordinate data exchange between federates, to specify the data a federate will send and receive, and to control where data are sent. HLA Object Management Interface Methods deal with the registration, modification, and deletion of object instances and the sending and receipt of interactions. HLA Ownership Management Interface is used by joined federates and the RTI to transfer ownership of instance attributes among joined federates [2].

The OwM method “Inform Attribute Ownership” has an attribute `FederateReference` type to inform the ownership status of the queried attribute. This status can be a “Federate”, RTI, or un-owned. If the ownership is a federate, then a federate instance reference must be included in the model to point to the

owner, or a null pointer refers to un-owned or a federation execution reference to point to the RTI.

The Time Management services and associated mechanisms permit messages sent by joined federates to be delivered in a consistent order to any joined federate in the federation execution that is to receive those messages. Data Distribution Management services serve to filter the data communicated between federates at the class attribute level and the interaction level. HLA Support Services are defined in HLA interface specification to be utilized by joined federates for performing such actions as name-to-handle transformation (and vice versa), controlling advisory switches, manipulating regions, and RTI start-up/shutdown [2].

The set argument has only one element (i.e., a reference to an argument) by default. If this reference is null, then it is assumed an empty set. The user may add new elements to the set. For example, `RegisterFederationSynchronizationPoint` method has a set argument, called “set of joined federate designators”, the HLA methods library provide only one reference inside the set (i.e., a federate application reference). The user can subtype this method and then can add any number of federate application references.

5.4.8.2 DMSO 1.3 Methods Library

DMSO 1.3 Methods library (DMLib) provides the methods for the DMSO RTI API[1] to model the federate behavior for DMSO 1.3 federations. Its structure is very similar to IMLib. There are 815 concepts[2] defined in it.

[1] For DMSO RTI NG 1.3 v6 API.
[2] For DMLib version 20071217_01.

CHAPTER 6

MODEL INTEGRATION AND EXTENSIBILITY

The MSC specification, in addition to an action model, introduces a rudimentary data model via some predefined elements, such as messages, actions, and MSC references. Basic data concepts as defined in MSC specification [36] are included in the metamodel to enable the declaration and the use of the static and the dynamic data.

However, it is certainly needed to build a domain-specific data model for many real-life applications and to integrate it with the MSC. Therefore, the LSC/MSC metamodel must provide by design a flexible structure for such future integrations.

While designing the data concepts in BMM, the facility for model integration was a main design principle. Therefore, as explained later, some explicit integration model elements are devised. This approach enables the user to supply a domain-specific data model.

6.1 Integration by Extension

FAMM is a typical example of how to integrate the domain-specific data model (HFMM) and action model (LMM) by using extensions as depicted in Figure 89.

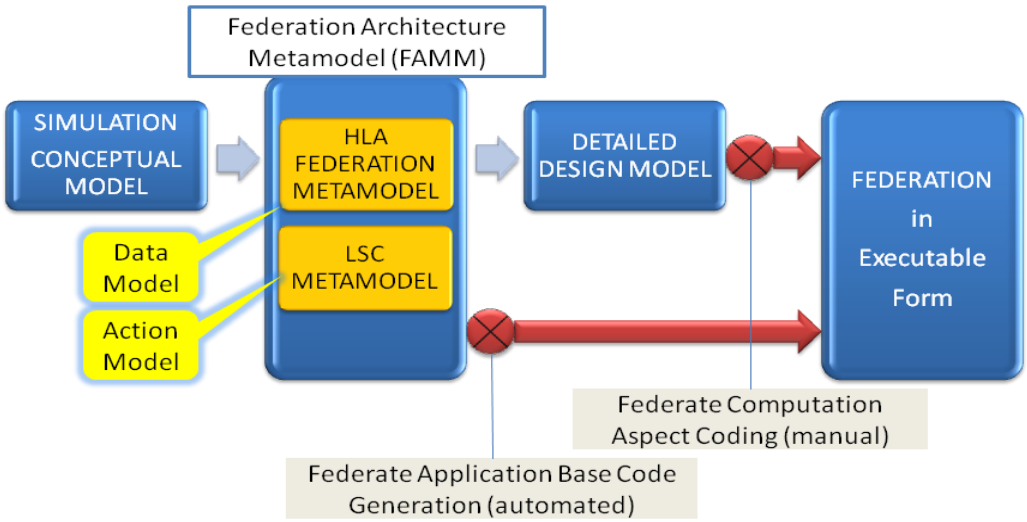


Figure 89. Development Methodology for HLA -Based Distributed Simulations

A model can be integrated by extending any element defined in the MSC/LSC metamodel. For example, the MSC/LSC message can be extended to provide a domain-specific meaning to it. The HLA interface methods are a specialization of the MSC/LSC message as seen in Figure 90. Thus, HLA methods are allowed to be used in MSC/LSC charts in place of the MSC messages.

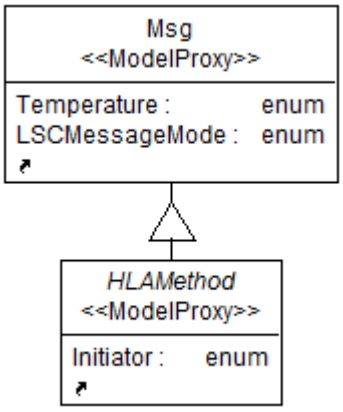


Figure 90. Extending MSC Message as HLA methods

Using the same approach, the HLA model elements, federate application, federation, and live entity, are all instantiated from the MSC instance model element. The integration is depicted in Figure 91. Federate application is inherited using normal inheritance operator because some properties of MSC instance element such as “decomposed” can also be used for federates. The others are

inherited using the interface inheritance operator. Therefore, they behave as a black box.

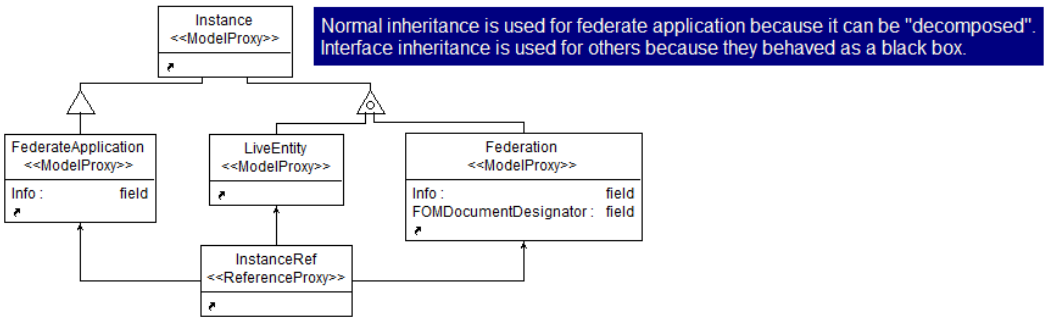


Figure 91. Extending MSC Instance to Integrate Some HLA Model Elements

6.2 Accommodating Future Integrations

Explicit integration points in LMM are created with the purpose of easing the (meta)model integration. These integration points are defined in expressions, in data types, and in arguments. Integration points are intentionally designed as abstract GME FCO (First Class Object) classes so that any kind of GME class (e.g., atom, reference, etc.) can easily be inherited.

A data model may define a probabilistic kind of Boolean expression, to account for the occurrence probability of the truth, to be used in the conditions. After modeling this special Boolean expression, it is enough to inherit it from the `BooleanExpression` FCO class defined in the MSC metamodel. As a result, while modeling a condition, modeler can select either the default Boolean expression or the new probabilistic Boolean expression.

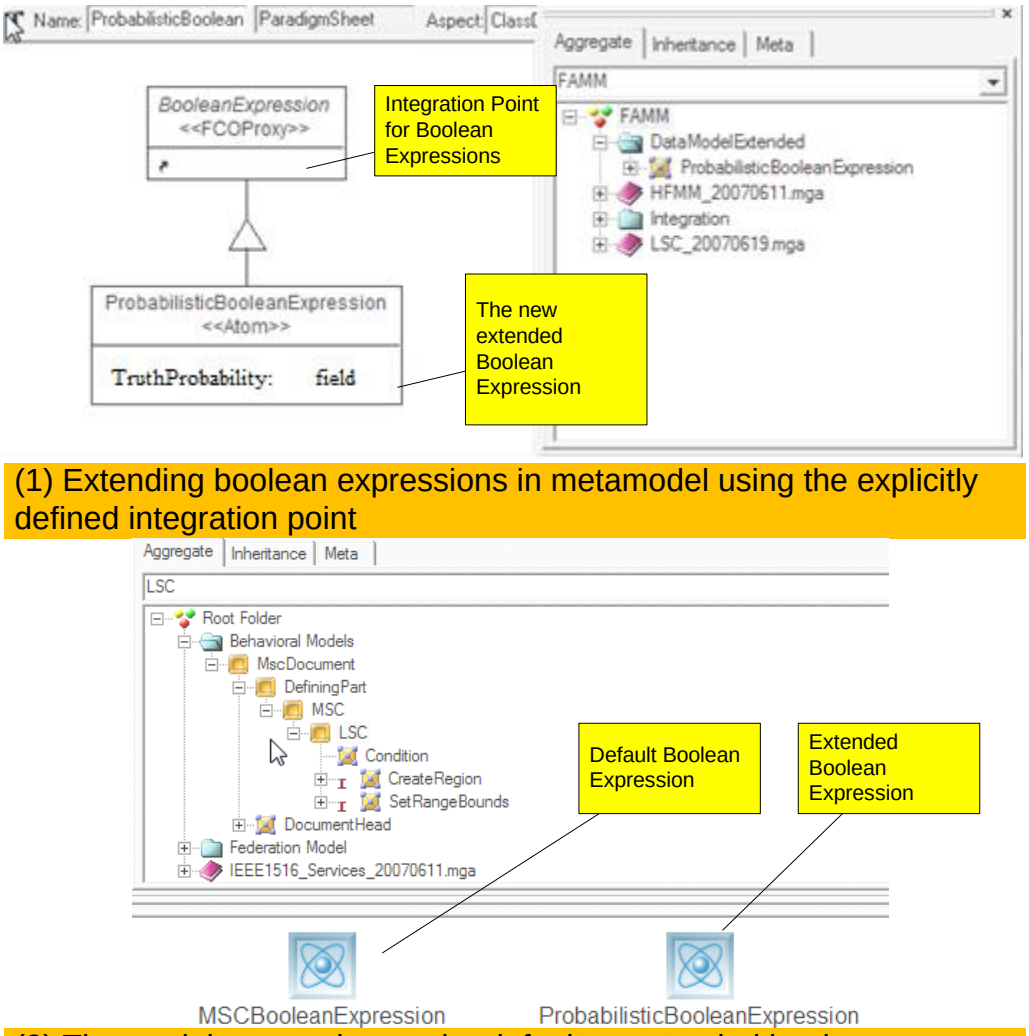


Figure 92. Integration of a Probabilistic Boolean Expression

The same approach is also used for message arguments and data types. For arguments the `Argument` FCO class, and for data types the `DataTypeFromDataModel` FCO classes are created as integration points. As seen in Figure 93, the domain data types (HLA) are instantiated as MSC data types. Thus, the modeler can use the HLA data types in MSC/LSC charts as well.

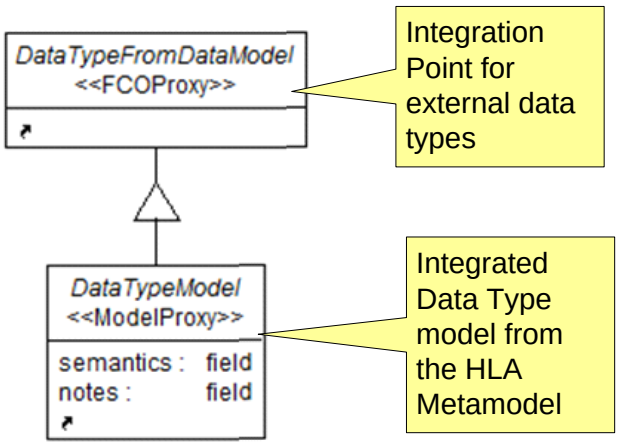


Figure 93. Integration of MSC Data Types and HLA Data Types

6.3 Console Input Output Library

To illustrate the integration of the external data models, a basic Console Input Output Model Library (CIOMLib) is created using the default MSC data concepts. This library is used to model the basic user input output via a simple console. There are two interactions; Input and Output, and two arguments, bearing the names, `InputString` and `OutputString` respectively. The interactions are modeled as MSC messages. The arguments can be easily used as arguments of HLA methods. For instance, in Figure 94, the user inputs a name using Input message and then its argument, `InputString`, is used in `ReserveObjectInstanceName` method call.

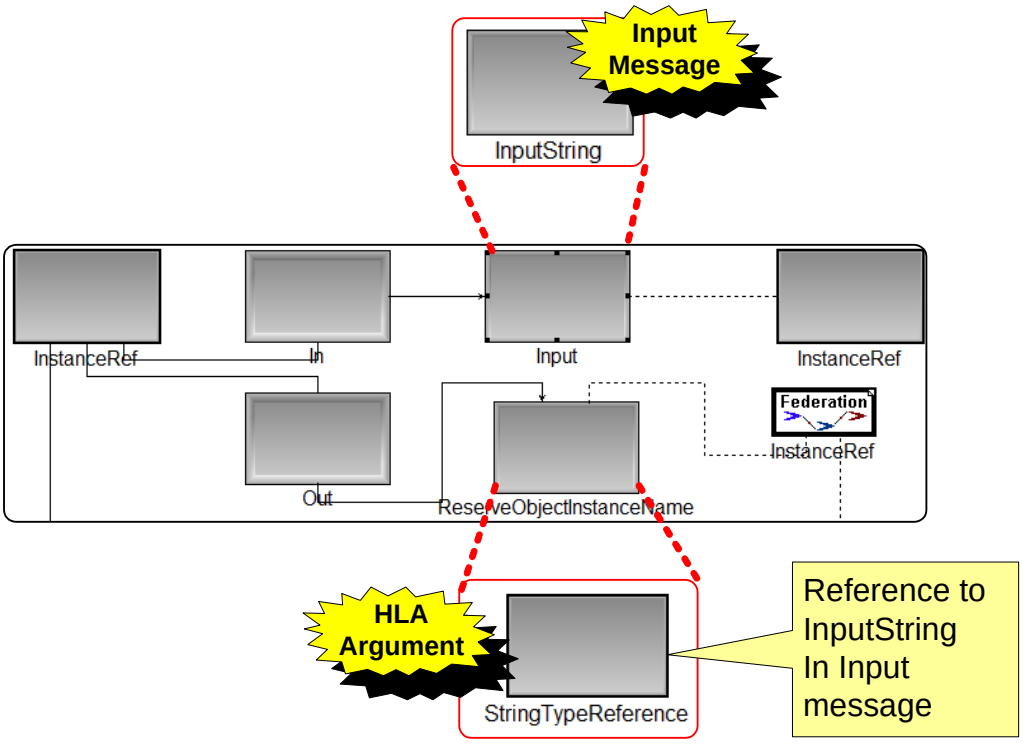


Figure 94. Example for the integration of an External Data Model and HLA Methods

The use of this library demonstrates that a data modeling element defined with the MSC model can be used in the HLA model.

CHAPTER 7

FAMM ASSESSMENT

FAMM assessment can be carried out, in qualitative terms, based on the criteria of *completeness, traceability, modularity, layering, partioning, extensibility*, *reusability* and *usability* (adapted from Section 7.1 of [14] and from [44]). It is believed that all the criteria defined here determines the metamodel quality.

The assessment is based on the case studies and architecturing a real world federation, NSTMSS, which is presented in Appendix B. Though this assessment can be seen as a self-assesment and can be judged as being subjective, the assessment is open to discussion. Of course, architecting and modeling more real world federations using FAMM will bring out a more objective assessment of FAMM.

7.1 Completeness (Scope)

Completeness criterion answers whether the metamodel includes all the relevant concepts and entities of the intended domain [44, 45, 46]. The completeness of the metamodel can be checked using the specifications and the standards of the intended domain.

Completeness refers to the scope of the FAMM. FAMM offers a formalization of a significant portion of the IEEE 1516 standard (the HLA Framework and Rules specification and the pre/post conditions of the interface services are excluded). In particular, HOMM formalizes the HLA OMT (IEEE 1516.2), and HSMM formalizes HLA Federate Interface Specification (IEEE 1516.1). Another benefit of formalizing the HLA standard is that vendors' deviations from the official standard become manifest from library. For example, in Pitch RTI implementation, the indicator argument is absent in `SubscribeInteractionClass` method call. This should also help, for example, in porting a federate to a different RTI. Further, FAMM

covers the complete MSC/LSC specification in its Behavioral Metamodel. FAMM is checked with each of the examples found in [35 and 38].

7.2 Traceability

Traceability between the domain specific concepts and the metamodel elements is important for the proposed metamodels. Because the modelers generally tend to expect to see the same concepts, which they are familiar from the domain. For example, in LSC domain, the modelers would like to see the coregion construct instead of a new devised construct doing the same work. This situation also alleviates the learning and adaptation period. Therefore, keeping traceability between the standardization documents and the metamodel elements straightforward was a (meta)modeling guideline.

Another traceability issue is between the model and the generated code. Traceability via comments is significant because the application developer works over the generated code (e.g., he/she weaves the application logic code after automatic base code generation).

7.3 Modularity

The modularity (Section 7.1 of [14]) principle addresses high coherence and low coupling between the modules. It should be evident from the FAMM presentation, as seen in Figure 5, that modularity principle is adhered to so that each concern area is addressed by a self-functional (high coherence) sub-metamodel (e.g., LMM and HFMM) as these sub-metamodels are connected loosely through *Integration Metamodel*. Moreover, each sub-metamodel is also separated into sub-metamodels (e.g., LMM is separated to MMM). By separating FAMM into sub-metamodels provides the modularity. In terms of GME, this modularity is provided by using libraries where each sub-metamodel (i.e., module) is a GME library.

7.4 Layering

Layering is defined as (1) separating the core constructs from the higher-level constructs that use them, (2) separating concerns by a four-layer metamodel architectural pattern [14].

The correlation of FAMM with OMG's four-layer metamodel hierarchy has already been presented in Table 1.

FAMM structure separates the core constructs and higher level constructs by using the GME folder and the GME paradigm sheet structures. For example, the HOMM encompasses the OMT Core folder and the Object Model paradigm sheet where the former includes the core OMT elements and the latter includes the elements, which use the core elements. Another example for layering is the structure of the LSC idioms, which are created using the LSC/MSC core constructs such as the inline expressions and the conditions.

Layering property of FAMM becomes more pronounced in a Federation Architecture Model, which conforms to FAMM. In a federation architecture, two levels become visible by separating the model specific (i.e., federation/federate specific) and non-specific (i.e., HLA specific) concerns. The base layer is the HLA-specific layer. The top layer is the federation/federate specific layer. The top layer uses the constructs found in the base layer.

This layering is done via the GME libraries. The libraries, provided with FAMM such as IMLib, are all specific to HLA standard rather than to a specific model. These libraries provide the core constructs and form the base layer in a federation architecture project. The top layer, which is formed by the behavior models, the federation structure model, and the federation model in the project use the core constructs provided with the libraries. The layers are depicted in Figure 95.

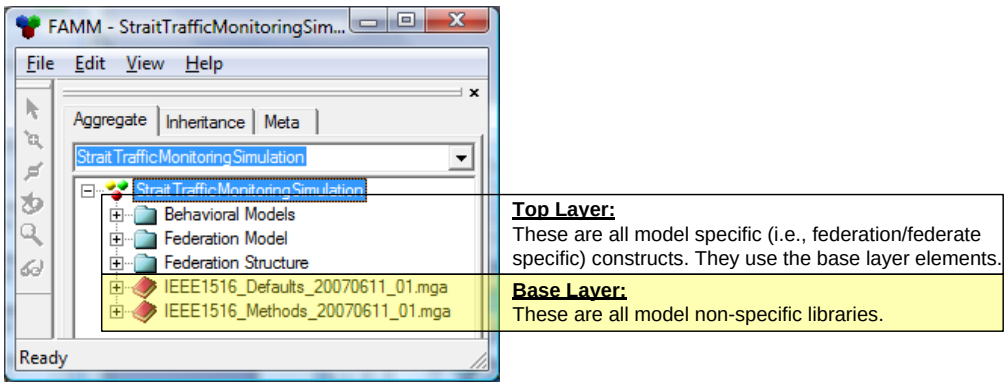


Figure 95. Layers in a Federation Architecture Model

7.5 Partioning

As specified in [14], *partioning* is used to organize the conceptual areas within the same layer. In the case of FAMM, partioning is provided by grouping constructs

into folders for each sub-metamodel. For example, considering MMM, it organizes the MSC constructions in four folders, namely, Auxiliaries, Basic Constituents, Data Concepts, and Time Concepts. In each folder, by using the GME paradigm sheets, the constructs are grouped. For instance, in Basic Constituents folder, there are actions, charts, comments, gates, etc paradigm sheets and in Time Concepts folder, there is measurement, time interval, time offset, etc. paradigm sheets. Each paradigm sheets include the model constructs and the structure of these constructs.

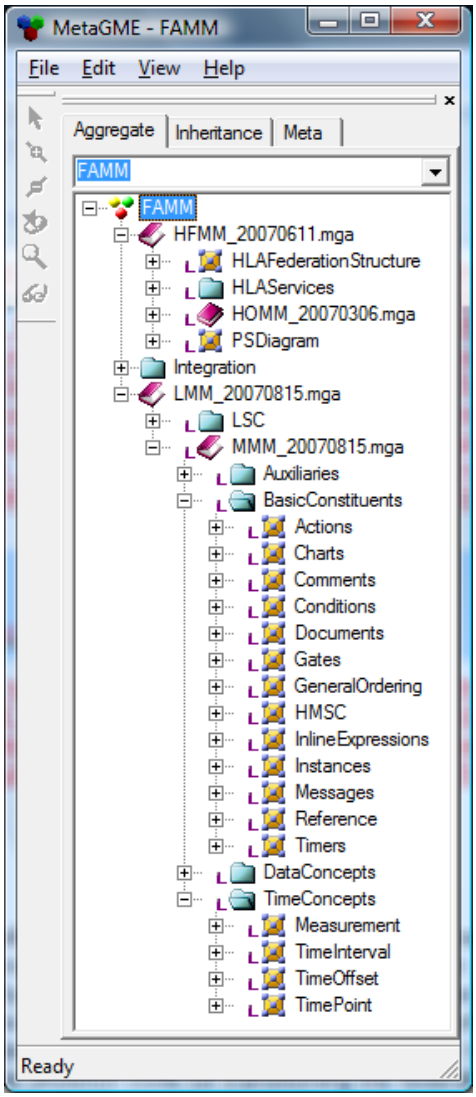


Figure 96. FAMM Partioning

Partioning is conducted according to the specifications and standards of the intended domain (e.g., for MMM, the MSC specification is taken into consideration)

to support the traceability between FAMM and the domain specific concepts. Thus, the conceptual areas are organized according to the domain authoritative documentation.

7.6 Extensibility

In order to construct a new metamodel as an extension of an existing one (e.g. extending the MSC metamodel to the LSC metamodel), one could copy the existing metamodel and then make modifications and additions to it. Alternatively, one may attach the existing metamodel as a library and then build the new model on top of it without any modifications to the attached library elements. The latter method, using the nested libraries feature of GME, yields better model encapsulation. An example is provided in Section 4.3.2.

Attaching a metamodel into another metamodel as a library can be seen as an analogy to the class inheritance in object-oriented languages. Moreover, read-only metamodel elements resemble the protected attributes in a class where, model elements to be extended resemble the public attributes.

Extensibility [14 and 44] emphasizes modifiable metamodels. Extension to a metamodel is inevitable because the requirements and the expectations from a metamodel will change from time to time. In case of FAMM, for specific domains, it is essential to provide a domain-specific data model, which extends the basic MSC data model. The proposed metamodel's facility of integration with domain-specific data models plays a critical role to achieve code generation.

UML specification dictates two kinds of extension mechanism: (1) using profiling mechanism (i.e., profiles are used to customize the language for specific domains) and (2) reusing part of the infrastructure package and augmenting it [14].

FAMM does not provide a profiling mechanism; instead, it has built-in explicit integration points for model extensions. Reusing infrastructure package strategy is also followed for instance, in extending MMM for LMM.

Extensibility support of FAMM and model integration are explained in detail in Chapter 6.

7.7 Reusability

Reusability (Section 7.1 of [14]) will take a longer period of time to assess. Presently, we note that the LSC/MSC metamodel is already being utilized in

performance modeling of network-based information systems. The MSC/LSC metamodel is used to model the traffic generation behavior of the software and is extended to include performance-related concepts, such as probability distributions over the messages.

Another study employs the HOMM to attack the FOM independence problem. The existing FOM and the new FOM are both modeled in HOMM, and a third model, called the Correspondance Model, specifies the mappings from the former to the latter. From this trio of models, the skeleton code for transitioning the federate to the new FOM is automatically generated [47].

7.8 Usability

The *usability* of FAMM is evaluated in re-modeling of NSTMSS and some of its federates, namely, Federation Monitor Federate (FedMonFd) and Exercise Planner Federate (ExPFd) [48]. Both are fairly common types of federates in HLA federations, where FedMonFd serves as a stealth observer by using the HLA MOM, and ExPFd serves as federation scenario manager. Modeling the architectures of each federate took one person-month. Most of the effort was spent on learning the MSC/LSC and the FAMM basics, and reverse engineering of the existing federate application.

In the matter of usability, the granularity of modeling matters. At present, the user must model all the RTI interactions in full detail. For example, the number of concepts (elements with a kind of model, atom, and folder stereotypes in GME parlance) and connections (association in GME parlance) that constitute the STMS model is about 634 and 243, respectively.

One means of alleviating this situation is to reduce the size of the handcrafted model with the help of model transformation. The idea is that the designer (or an automated conceptual model transformer) will need to specify only the essential interactions in the behavior model, and then an auxiliary model transformer will fill in the implied message exchanges taking the method pre- and post-conditions into account. Static analysis of the behavior model is, of course, a prerequisite for such model manipulations.

A complementary approach is to isolate the users as much as possible from FAMM details. A graphical front-end that supports the LSC graphical syntax would

facilitate more intuitive behavior specification as well as prevent easy mistakes in modeling.

7.9 Other Criteria

The criteria that are often applied as the quality criteria for the conceptual models can also be applied to assess the metamodels. Especially, two criteria among them: being *analyzable* and *executable* [46], which is an expected effect of Model Integrated Computing, is important for models that conform to FAMM. The code generation study over FAMM proves that the federation architecture models (that conform to FAMM) are executable [31].

The *quality of definitions of the documentation* [44] criterion leads us to give importance to the documentation for a metamodel. In this sense, FAMM is well documented both in defining the metamodel (i.e., FAMM) constructs and in explaining the domain-specific modeling environment (i.e., FAME), which use FAMM.

The *correctness* criterion [14, 45, and 46] is an indispensable characteristic of a metamodel. This criterion must be evaluated by objective studies and it will take longer time to assess.

Consistency [45 and 46] criterion emphasizes that the metamodel constructs are not in conflict with any other constructs. Due to compound structure of FAMM, which integrates specifications from interdisciplinary domains (i.e., HLA, MSC, and LSC), consistency was a major design principle. Especially, while (1) extending the MSC metamodel to form the LSC metamodel and (2) while integrating HLA and behavioral sub-metamodels, eliminating the conflicts was a design principle. As a result, the constructs in the sub-metamodels of FAMM are not in conflict with any other constructs.

Comprehension criterion [46] addresses the need for understandable models. It is clear that if a metamodel is not understandable, then no one will use it.

CHAPTER 8

RESULTS, DISCUSSIONS, AND FUTURE WORK

8.1 Accomplishments and Discussions

This study proposes a metamodel, designated FAMM, for federation architectures to enable a broad range of tool support for the HLA federation development process. A significant part of this proposal is adoption of Live Sequence Charts for the behavioral specification of federates. FAMM can be regarded as a domain specific architecture description language for HLA federations. A federation architecture model comforming to FAMM is in a machine-processable form, thus enabling tool support.

Specifically, FAMM offers the following benefits to federation developers:

- FAMM serves both as a basis for source models for code generation [31] and as a basis for target models for transformation from the domain-related models (e.g., conceptual models of mission space) [9]. An interpreter, called *Code Generator*, for automatic code generation is supplied with FAMM. The interpreter takes a model (i.e., a FAM) including a federate behavior specification as input and produces the federate application base code as output [31].
- FAMM brings forth the expressive power to represent not only the static view of the federation but also the behavior of the federates. It relates behavior with the structure. This power comes from the Behavioral Metamodel, which is integrated with the HLA Object Model and HLA Services Metamodel. Thus, it eliminates a significant limitation of the OMT and FEDEP.
- FAMM lays the groundwork for implementing model interpreters in order to generate useful artifacts, such as FDD files and to extract interesting views,

such as publish/subscribe diagrams. An interpreter, called *P/S Model Generator*, for the automatic generation of P/S models is implemented and an interpreter, called *FDD Generator*, for automatic FDD file generation is supplied.

- FAMM provides support for the verification and validation activities due to the increased precision in the description of the federation.
    - Constraints support early verification (in the sense of consistency checking) in the architectural design phase. Cardinality constraints are supported by design. Further constraints that cannot be enforced by metamodel structure can be formulated using the OCL [27]. Note that currently the HLA Object Model contraints are formulated in the OCL. Formulating the constraints for the MSC/LSC and the HLA interface specification is left as a future work.
    - Generating codes for member federate applications and executing the federation serve as a test for validity of the federation architecture. Thus, it supports a dynamic verification of the federation design.
- FAMM enables static analysis of the federation architecture. This can be helpful, for example, in collecting metrics for assessing the complexity of federation architectures. An interpreter, called *Model Metrics Collector*, for collecting the metrics over FAM is supplied with FAMM.
- FAMM can help improve the communication among simulation engineers, software engineers and programmers, again due to increased precision.

The metamodel, along with the libraries, the interpreters, and documentation for sample case studies, including the federation architecture and the automatically generated code of the Strait Traffic Monitoring Simulation, and NSTMSS models are available from the FAMM web page: http://www.ceng.metu.edu.tr/~otopcu/famm/index.html, last accessed December 25, 2007.

8.2 Future Work

8.2.1 From a User-friendly Trimmed Model to a RTI-friendly Full Model

It is also aimed to minimize the behavioral modeling effort without loosing any details in federate's behavior in order to simplify the designer's work. Work is aimed at reducing the initial model size with the help of model transformation. The idea is that, the designer or the conceptual model transformer will specify only the essential interactions in the behavior model, and then an auxiliary model transformer will fill in the implied message exchanges taking method pre- and post-conditions into account. Static analysis of the behavioral model is, of course, essential to lay the groundwork for such model manipulations. The designer can specify the minimal and basic behaviors in model (behavioral model in trimmed form – user friendly), and then the model transformer can fill the standard model elements to generate an RTI-friendly full model by the help of pre-defined transformation rules as exemplified in Figure 97.

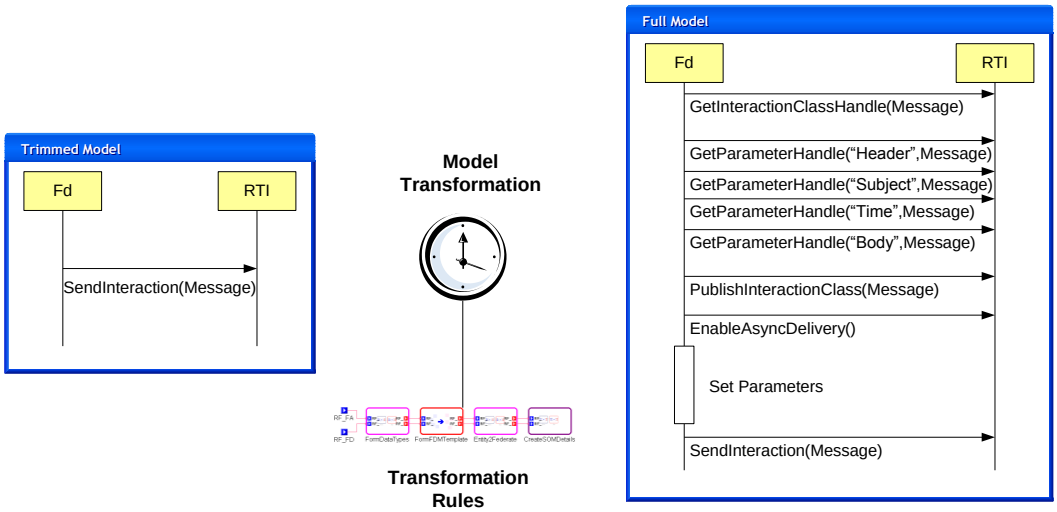


Figure 97. Transformation

Some potential objectives/issues to be addressed and covered are:

- The static analysis of a federate to determine the observed state of the federate from RTI's viewpoint.
- The transformation rules for RTI methods.
- The determination of the correct position for the generated elements in the model.

8.2.2 Graphical Front-end (from LSC to BMM)

Future work will address improving (meta)model usability. A graphical front-end, supporting, in particular, the LSC graphical syntax, is certainly desirable to improve the usability of the metamodel.

Abstract syntax is, clearly, not for human consumption. Understandability of the models is harder than that of graphical representation. Moreover, from our experience, the metamodel usage is error-prone. Each visual element in LSC corresponds to one (sometimes two) modeling element(s), which ostensibly results in cluttered models. This drawback can be circumvented with a front-end that converts the LSC in concrete syntax (in graphical notation) to the LSC in abstract syntax (in our metamodel).

8.2.3 Federation Scenario

The scenarios and federation executions are closely interrelated. Developing a scenario specification related to the Federation Architecture Metamodel will ease the transformation of the conceptual models into federation architectures. Moreover, the federation scenarios can be used to generate test cases for both integration testing and operational testing of federations.

In a simplistic approach, federate developers may not care about federation scenarios but the federation developers have a concern to enforce a specific scenario during federation executions. The solution for enforcing a scenario will affect the federate design. Consequently, the computational aspect may be weaved according to the scenario.

8.2.4 Extraction of Usable Views of Federation Architecture

Another application area of the proposed metamodel will be to extract some usable views such as filters (e.g., filtering the architecture to show only the interactions) from the federation architecture.

8.2.5 Defining Metrics for Metamodel Quality Assessment

A research to specify generic assessment parameters for metamodel quality can be conducted. The assessment parameters may be discussed with respect to the proposed metamodels.

8.2.6 Constraint Checking Over FAMM (Through the OCL Constraints)

There are many constraints defined in the MSC/LSC and the HLA interface specifications by prose. FAMM maintains the structural constraints of these by design. For example, “a loop element has exactly one operand” is a structural constraint. The others rather than the structural ones can be expressed using the OCL and can be added to the metamodel. For example, name of the elements must be unique in an MSC document.

The constraints for the HLA interface specification and MSC/LSC specification must be formulated. The constraints provide robust models. The more the constraints are added to FAMM, the fewer mistakes the modeler does.

8.2.7 Decomposition of a Federate Application

MSC decomposition can be automatically done via model transformations. Both the source and target model are based on the same MSC metamodel. Model transformation rules defined in this scope may also support the transformations from scenario to federate LSCs. To explain decomposition more clearly, an example for the decomposition of federation scenario LSC into federate RTI-specific LSCs is given in Figure 98. The scenario is very simple where entity `A` is sending an interaction `m1` to entity `B`. When decomposition is applied, federate `AFd` LSC and federate `BFd` LSC can be generated as shown in the figure. Message `m1` is decomposed into RTI specific interface method calls where `AFd` sends an interaction and `BFd` receives an interaction.

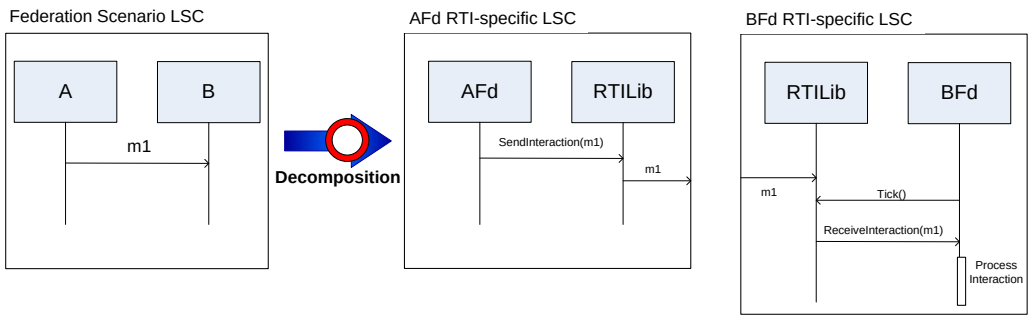


Figure 98. LSC Decomposition Example

REFERENCES

1. IEEE 1516 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules, 21 September 2000.

2. IEEE 1516.1 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification, 21 September 2000.

3. IEEE 1516.2 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template Specification, 21 September 2000.

4. Stytz M. R. and Banks S. B. “Enhancing The Design and Documentation of High Level Architecture Simulations Using the Unified Modeling Language”, In the Proceedings of 2001 Spring Simulation Interoperability Workshop (SIW), 2001.

5. IEEE 1516.3 Standard for IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP), Apr 23, 2003.

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

7. Schmidt D.C., “Model-Driven Engineering”, IEEE Computer, vol.39 no.2, pp. 25-32, 2006.

8. Ledezci A., Bakay A., Maroti M., Volgvesi P., Nordstorm G., Sprinkle J., and Karsai G., “Composing Domain-Specific Design Environments”, in IEEE Computer, vol.34 no.11, pp. 44-51, 2001.

9. Özhan Gürkan and Oğuztüzün Halit, "Model-Integrated Development of HLA-Based Field Artillery Simulation", Proceedings of 2006 European Simulation Interoperability Workshop, Stockholm, Sweden, June19-22, 2006.

10. Topçu Okan, “Development, Representation, and Validation of Conceptual Models in Distributed Simulation”, Defence R&D Canada – Atlantic (DRDC Atlantic) Technical Memorandum (TM 2003-142), Halifax, NS, Canada, February 2004.

11. ISIS, “A Generic Modeling Environment GME 7 User’s Manual v7.0”. Institute for Software Integrated Systems (ISIS) Vanderbilt University, 2007.

12. Çetinkaya Deniz, “A Metamodel for the High Level Architecture Object Model”, Master’s Thesis, The Graduate School of Natural and Applied Sciences, Middle East Technical University, August 2005.

13. Gray J., Bapty T., Neema S., and Tuck J., “Handling Crosscutting Constraints in Domain-specific Modeling”, in Communications of the ACM Journal, vol.44 no.10, pp.87-93, 2001.

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

15. Tolk A., “Avoiding another Green Elephant – A Proposal for the Next generation HLA based on the Model Driven Architecture”, in the Proceedings of 2002 Fall Simulation Interoperability Workshop (SIW), 2002.

16. Tolk A., “Metamodels and Mappings – Ending the Interoperability War”, In the Proceedings of 2004 Fall Simulation Interoperability Workshop (SIW), 2004.

17. Parr S. and Keith-Magee R., “Making the Case for MDA”, in the Proceedings of 2003 Fall Simulation Interoperability Workshop (SIW), 2003.

18. Guiffard E., Kadi D., Mochet J., and Mauget R., “CAPSULE: Application of the MDA Methodology to the Simulation Domain”, In the Proceedings of 2006 European Simulation Interoperability Workshop (SIW), 2006.

19. Etienne S., Xavier L., and Olivier V., “Applying MDE for HLA Federation Rapid Generation”, in Proceedings of 2006 European Simulation Interoperability Workshop (SIW), 2006.

20. SISO, “Base Object Model (BOM) Template Specification”, SISO-STD-003-2006, 2006.

21. Loper, M.L., “Test Procedures for High Level Architecture Interface Specification”, Technical Report, Georgia Tech Research Institute, Georgia Institute of Technology, 1998.

22. Topçu Okan and Oğuztüzün Halit, “Towards a UML Extension for HLA Federation Design”, in the Proceedings of 2nd Conference on Simulation Methods and Applications (CSMA-2000), pp 204-213, Orlando, FL, USA, Oct. 29-31, 2000.

23. Topçu Okan, Oğuztüzün H., and Hazen G. M., “Towards a UML Profile for HLA Federation Design, Part II”, in the Proceedings of Summer Computer Simulation Conference (SCSC-2003), pp. 874-879, Montreal, Canada, July 19-24, 2003.

24. Dobbs V., "Managing a federation Object Model with rational Rose: Bridging the Gap between Specification and Implementation", In Proceedings of the 2000 Fall Simulation Interoperability Workshop, 00F-SIW-010, 2000.

25. Topçu Okan and Oğuztüzün Halit, “Developing an HLA Based Naval Maneuvering Simulation”, in Naval Engineers Journal by American Society of Naval Engineers (ASNE), vol.117 no.1, pp. 23-40, winter 2005.

26. Dahmann J., “High Level Architecture for Simulation”, Defense Modeling and Simulation Office, 1998.

27. OMG. Object Constraint Language (OCL), Object Management Group (OMG), May 01, 2006.

28. DoD VV&A Recommended Practices Guide (DOD VVA RPG Build 2) in http://vva.dmso.mil, last accessed November 04, 2007.

29. Pace Dale K., "Conceptual Model Descriptions", Simulation Interoperability Workshop (SIW) Spring (99S-SIW-025), 1999.

30. Jacobson I., Christerson M., Jonsson M., and Overgaard G., "Object-Oriented Software Engineering: A Use Case Driven Approach", Addison-Wesley, ACM Press, 1993.

31. Adak M., Topçu O., and Oğuztüzün H., "Model-based Code Generation for HLA Federates", submitted, 2007.

32. Savaşan Hakan and Oğuztüzün Halit, "Distributed Simulation of Helicopter Recovery Operations At Sea", Proceedings of Military, Government, and Aerospace Simulation (MGA02), Advanced Simulation Technologies Conference Simulation Series Volume 34 number 3 pp.120-125, April 2002.

33. Kleppe A., Warmer S., and Bast W., "MDA Explained: The Model Driven Architecture, Practice and Promise", Addison-Wesley, 2003.

34. Çetinkaya Deniz and Oğuztüzün Halit, "A Metamodel for the HLA Object Model", in the Proceedings of the 20th European Conference on Modeling and Simulation (ECMS), pp. 207-213, Bonn, Germany, May 28-31, 2006.

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

36. ITU-T Recommendation Z.120, "Formal Description Techniques (FDT) - Message Sequence Charts. Pre-published Recommendation Telecommunication Standardization Sector of International Telecommunication Union (ITU-T), 2004.

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

38. Madsen C.K., "Integration of Specification Techniques", Master of Science Thesis, Computer Science and Engineering Division of Department of Informatics and Mathemetical Modeling (IMM) at the Technical University of Denmark (DTU), Nov 2003.

39. Damm, W. and Harel, D., "LSCs: Breathing Life Into Message Sequence Charts", in Formal Methods in System Design, 19, 45-80, 2001.

40. Haugen Ø., "MSC-2000 Interaction Diagrams for the New Millennium", Computer Networks. 35(6): 721-732, May 2001.

41. Rudolph E., Grabowski J., and Graubmann P., "Tutorial on Message Sequence Charts", in Computer Networks and ISDN Systems, 28(12):1629-1641, 1996.

42. Harel D. and Marelly R., “Come, Let’s Play: Scenario-Based Programming Using LSCs and the Play-Engine”, Springer-Verlag, 2003.

43. Medvidovic N. and Taylor R.N. A Classification and Comparison Framework for Software Architecture Description Languages. In IEEE Transactions on Software Engineering Journal, vol.26 no.1, pp. 70-93, 2000.

44. Community Site for Metamodeling, “http://www.metamodel.com/”, last accessed November 04, 2007.

45. Pace Dale K., “Ideas About Simulation Conceptual Model Development”, John Hopkins APL Technical Digest, Volume 21, Number 3, pp.327-336, 2000.

46. Lindland, O.I., Sindre, G., and Solvberg A., “Understanding Quality in Conceptual Modeling”, IEEE Software 11(2), 42-49, Mar 1994.

47. Uluat, M.F., “Model-based Approach to the Federation Object Model Independence Problem”, MS thesis, Department of Computer Engineering, Middle East Technical University, Ankara, Turkey, August 2007.

48. Molla A., Sarıoğlu K., Topçu O., Adak M., and Oğuztüzün H., “Federation Architecture Modeling: A Case Study with NSTMSS”, in the Proceedings of 2007 Fall Simulation Interoperability Workshop (SIW), 2007.

49. Molla Ayhan, “Modelling of Exercise Planner Federate with Federation Architecture Metamodel”, Graduation Project Report, Department of Computer Engineering of Middle East Technical University, June, 2007.

50. Sarıoğlu Kaan, “Modeling Federation Monitor Federate”, Graduation Project Report, Department of Computer Engineering of Middle East Technical University, June, 2007.

51. Elrad T., Aksit M., Kiczales G., Lieberherr K., and Ossher H., “Discussing Aspects of AOP”, Communications of the ACM, vol. 44 no.10, pp. 33-38, 2001.

52. AspectJ, Project web site, “http://www.eclipse.org/aspectj”, last accessed at November 04, 2007.

53. Topçu Okan, Adak Mehmet, and Oğuztüzün Halit, “A Metamodel for Federation Architectures”, will appear in ACM TOMACS, September 12, 2007.

54. Sarıoğlu, K., Adak, M. and Oğuztüzün, H., “Modeling and Code Generation for a Federation Monitor Federate”, Technical Report METU-CENG-TR-07-07, Department of Computer Engineering, Middle East Technical University, 2007.

55. Topçu Okan and Oğuztüzün Halit, “A Metamodel for Live Sequence Charts and Message Sequence Charts”. Technical Report (METU-CENG-TR-2007-3), Middle East Technical University, May 2007.

APPENDIX A

FEDERATION ARCHITECTURE MODELING ENVIRONMENT

A.1 Overview

The model of a particular federation architecture is an instance of the Federation Architecture Metamodel (FAMM). Both tasks, metamodeling and modeling, are accomplished using Generic Modeling Environment (GME) developed and maintained by Vanderbilt University. GME is an open source, meta-programmable modeling tool that supports domain-specific modeling where domain is HLA in our case [8, 11].

GME initially serves as a metamodel development environment for domain analysts, and then it provides a domain-specific model-building environment, called Federation Architecture Modeling Environment (FAME), for the developers and the modelers.

The screen shot in Figure 101 shows an example modeling environment for FAMM users, who are typically federation designers.

Please refer to GME Manual and User Guide [11] for an explicit understanding of GME tool and to FAMM website (http://www.ceng.metu.edu.tr/~otopcu/famm/index.html, last accessed at December 25, 2007) for the example architectures introduced here.

A.2 Introduction to Example

The subsequent sections introduce the metamodel in detail, accompanied by a simple example: the Strait Traffic Monitoring Simulation (STMS). On a larger scale, the architectural modeling of Naval Surface Tactical Maneuvering Simulation System (NSTMSS) [25], a distributed interactive simulation, is carried out using the metamodel and is presented in Appendix B and in [48, 49, 50].

A traffic monitoring station tracks the ships passing through the strait. Any ship entering the strait announces her name and then periodically reports her position to the station and to the other ships in the strait using the radio channels. Channel-1 is used for ship-to-ship and channel-2 is used for ship-to-shore communication. The traffic monitoring station tracks ships and ships track each other through these communication channels. All radio messages are time-stamped to preserve the transmission order.

The traffic monitoring station and the ships are represented with two types of applications[1]: a station application and a ship application, respectively. The ship application is an interactive federate allowing the player to pick up a unique ship name, a direction (eastward or westward), and a constant speed by means of a textual interface. Joining a federation corresponds to entering the strait, and resigning from the federation corresponds to leaving the strait. The station application is a monitoring federate, which merely displays the ships (in the strait) and their positions. The federation has a time management policy where each ship application is both time regulating and time constrained and station application is only time constrained.

The conceptual view of the application is presented in Figure 99.

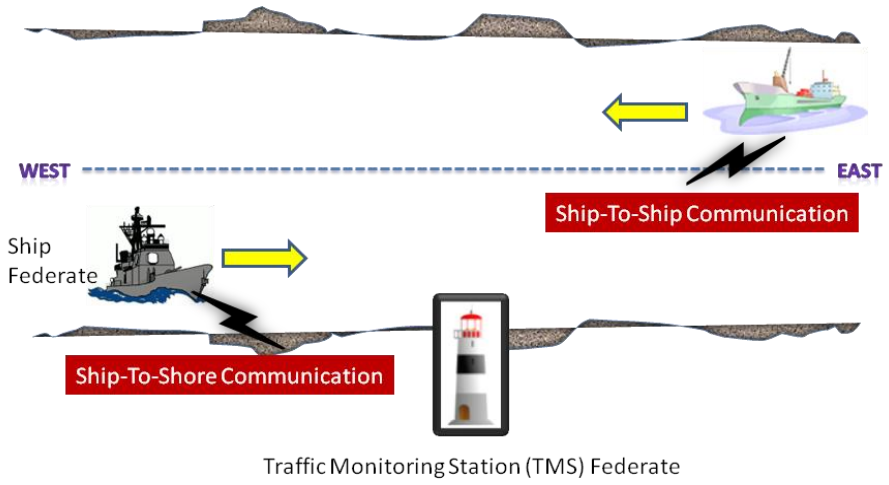


Figure 99. Strait Traffic Monitoring Simulation Conceptual View

While selecting this example, the following highlights were in mind:

[1] Application is used as a substitute for "federate application".

- Clearly, the essence of this simple federation is an example of a set of objects tracking each other making it a common scenario/interaction for most distributed simulations.
- It is believed that this example has a simple conceptual model, which will make it easily understandable and capture the reader's attention immediately. Thus, it will force the user focus on the modeling part than the example itself.
- Moreover, the sample federation naturally includes time management, ownership management, and data distribution management services in addition to the base services (e.g., federation management services).
- The sample federation involves two distinct federate applications and it has a potential to support multiple federations.
- It is an interactive simulation. Thus, it presents how to model the user interactions in FAME.

For the complete federation architecture and metamodels, along with other supporting material, the reader is referred to the FAMM web site (http://www.ceng.metu.edu.tr/~otopcu/famm/index.html, last accessed at December 25, 2007).

A.3 Registering the FAMM

FAME is provided by GME once FAMM is invoked as the base paradigm. First, it is required to register the FAMM paradigm to configure the GME as the FAME.

Run the GME. Click `File` and then select `Register Paradigms`. A dialog box will be opened as seen in Figure 100. Click `Add from File`; select the `FAMM.xmp` file obtained from the FAMM web site. Please, take care to check the version.

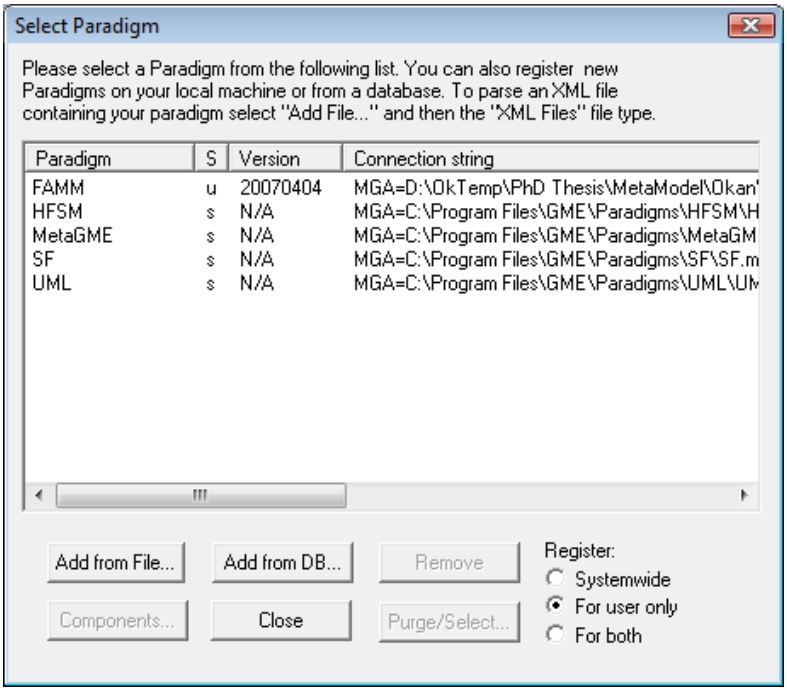


Figure 100. Registering the FAMM

A.4 Creating a New Project

GME allows creation of a project for developing a new federation architecture. Start GME, and select “File/New Project”. A dialog box asks you to choose the paradigm that the new project will be based on. Select FAMM and press the `Create New` button. The next dialog asks you to specify the data storage. Simple models are usually stored in project files. Click `Next` and name your project file. The standard extension is “.mga”. GME has now created and opened an empty project that is named and associated with the FAMM paradigm. The FAME is ready to use now.

Figure 101 presents a screen shot of the project for the STMS federation architecture. The root folder (e.g., “Strait Traffic Monitoring Simulation” in the screen shot) serves as a project container for the federation architecture. It includes three major sub-folders, namely, federation structure, behavioral models, and federation models. The federation structure folder contains information about the federation, such as the location of the FOM Document Data file, the link for the related FOM, and the structure of the federation, where the participating federate applications and their corresponding Simulation Object Models (SOMs) are linked. The folder for behavioral models includes an MSC document for each participating federate. The federation model folder includes the FOM, SOMs, and the other

Object Model Template related information (e.g., data types, dimensions, etc.). In the example, SOMs for ship and station applications and a FOM for the STMS federation are provided.

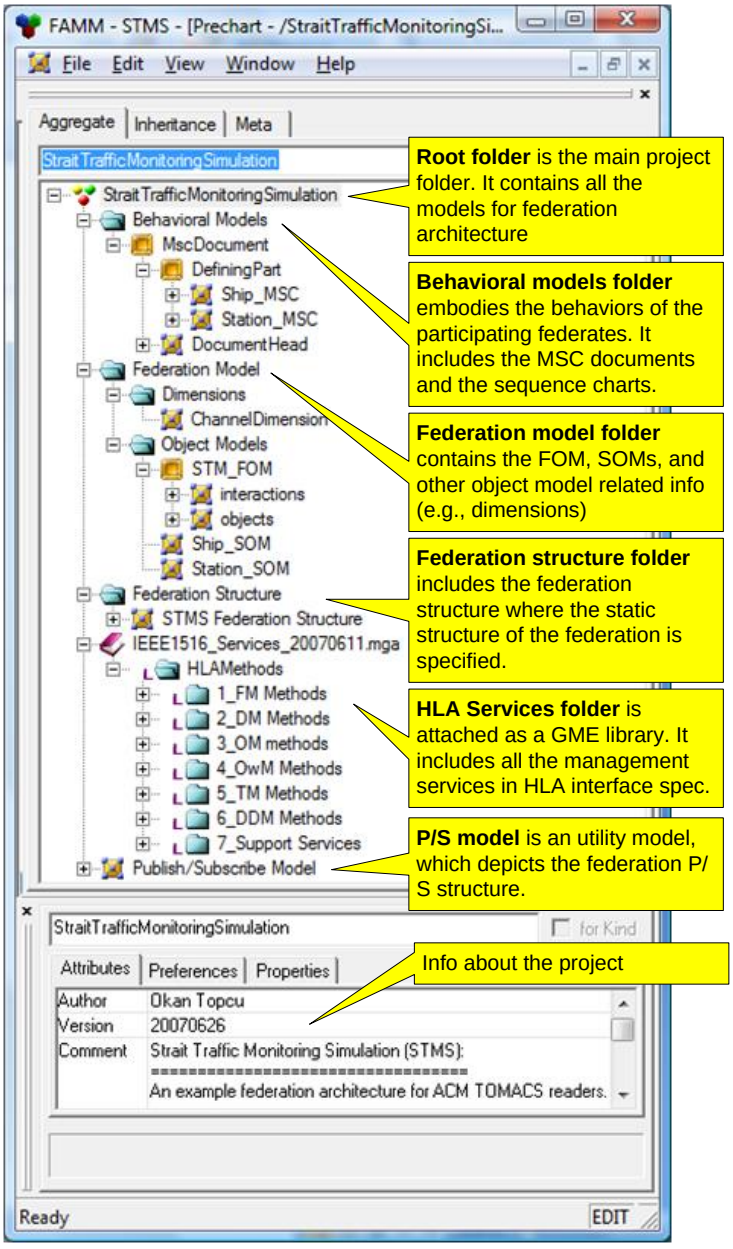


Figure 101. Federation Architecture Modeling Environment (FAME)

A.5 Creating the Federation Model

Federation Model is a folder that includes all the federation specific models and related data (e.g., data types). First, create a federation model folder.

A.5.1 Creating the Object Models

Creating the object models are explained in [12]..

An appropriate FOM for the STMS federation is prepared conforming to the HOMM. The object class and interaction class hierarchies of the object model are presented in Figure 102. The FOM involves two object classes, namely “ship” and “station”, and one interaction class, namely “radio message”. The ship object has four attributes, namely “name”, “course”, “speed”, and “position”, and the station object has two attributes, namely “name” and “location” as the radio message interaction class has two parameters, namely, the “call sign” and “message” parameters, indicating “the name of the entity that sent the message” and “the content of the message (i.e., the position data)”, respectively.

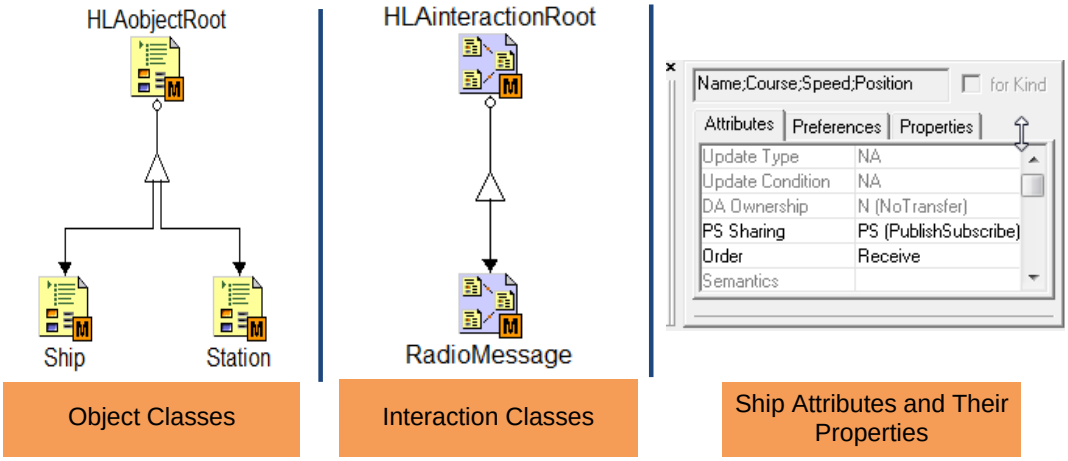


Figure 102. A Part of the STMS FOM (GME Screenshot)

A.5.2 Creating Other Elements

Right click on the federation model folder in the browser window within the “Insert Folder” option you can select Dimensions, Transportations, Data Types or Notes. After the selection, a new folder is created under the root. By right clicking on the new folder, you can select the related model elements. In addition, in the same way

with object models and federation design model, you can define the model elements.

For adding HLA notes, define notes under the Notes folder and give references to them; or directly add notes to the notes attribute of each modeling element. For adding design notes, use annotation facility of GME.

Lastly, we can add a few points. References used in the model shall not be null, if you want to denote “NA”, then simply use no reference element. If you want to check the validity of your model, you can use Check facility of GME, by selecting “File-> Check” option.

A.6 Creating Federation Structure Model

Right click on the root folder in the Browser window (the one usually positioned at the right side), and select the option “Federation Structure” within the “Insert Folder” option. Then create a new model named “New Federation Structure” is created under the federation structure folder; you may change the name from Attributes browser. Double click on the model to open it. An empty window appears in the user-area.

The Part Browser, a small window in the lower left portion of the program, displays the model elements that can be inserted into the model in its current aspect. The elements in this browser are `Federation`, `FederateApplication`, `FOMReference` and `SOMReference`. You can use them by dragging from the Part Browser onto the main window. You can connect federation to federate applications to denote the members of the federation; federation to `FOMReference`; and `FederateApplication` to `SOMReference`. When using references, you drag the referred element over the reference and drop it when the mouse icon changes. But before referring elements you should first define FOM and SOM object models. Copy and paste operations on elements are supported by GME and all elements can be created, moved, or copied by drag and drop as usual.

In the federation structure model of STMS, the connection is made for the federation and federate applications with the FOM and SOMs, respectively. The model is depicted in Figure 103. The federation is named “Traffic Monitoring Federation”. The multiplicity information is also supplied while connecting the applications to the federation. The ship application may join the federation multiple

times while the station application is limited to one in this specific scenario. The lower pane of the screen shot shows this multiplicity.

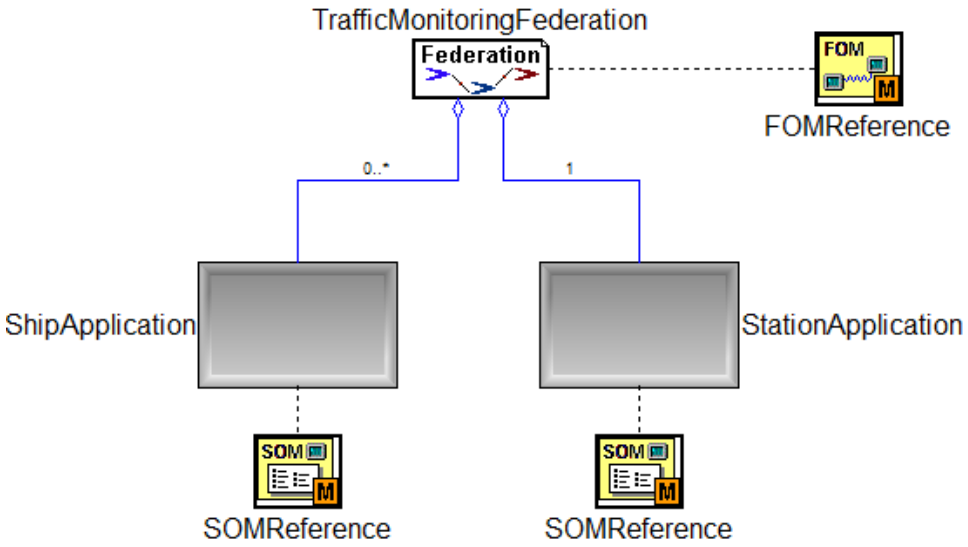


Figure 103. The Strait Traffic Monitoring Federation Structure Model

A.7 Creating Behavioral Models

You can create a `behavioral models folder` under the root folder. This folder groups all the behavioral models of the federates in the federation.

The folder for behavioral models defines the whole system of MSC documents and includes MSC documents for each participating federate. The behavior model folder includes a detailed structure; a screen shot for MSC/LSC building environment is presented in Figure 104. A *document* consists of head, utility, and defining parts where defining and utility parts include the (MSC) charts. A *chart* includes an MSC body, an HMSC, or an LSC. Charts also have precedence order indexes to specify the interpretation order. The document head includes the declaration lists.

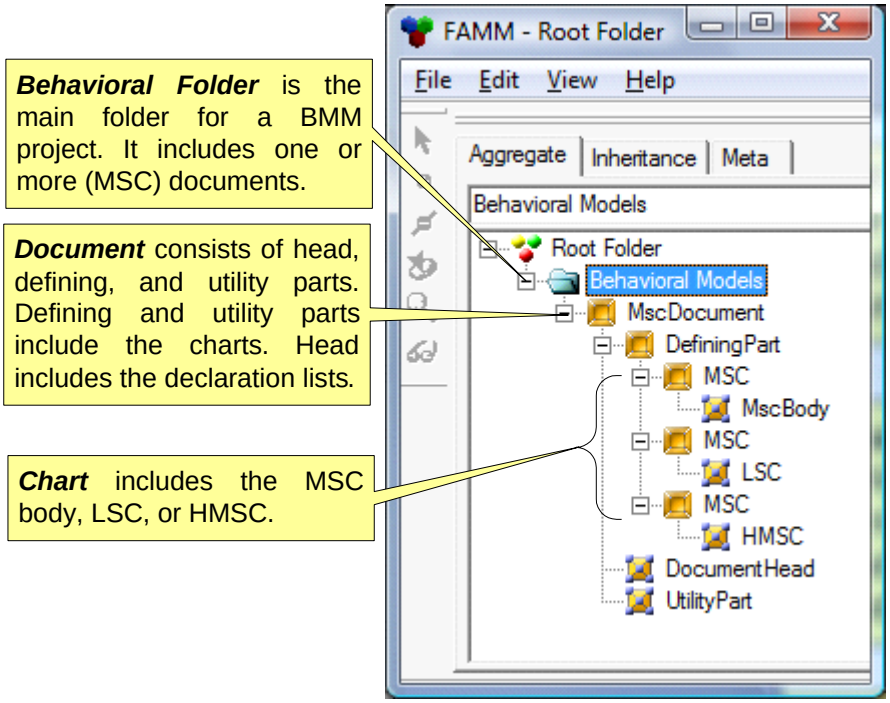


Figure 104. MSC/LSC Model Building Environment

To illustrate the usage of MMM, referring back to the STMS federation, there are four actors that contribute to the overall behavior of the federation. These are the ship and station applications, the user, and the federation execution (with the RTI "behind the scene"). The ship application can join the federation execution multiple times as distinct federates. Hence, it will be the focal point for the code generation process. The behavior model of the ship application will be presented first in LSC graphical form (Figure 105), and then in FAMM form (Figure 106).

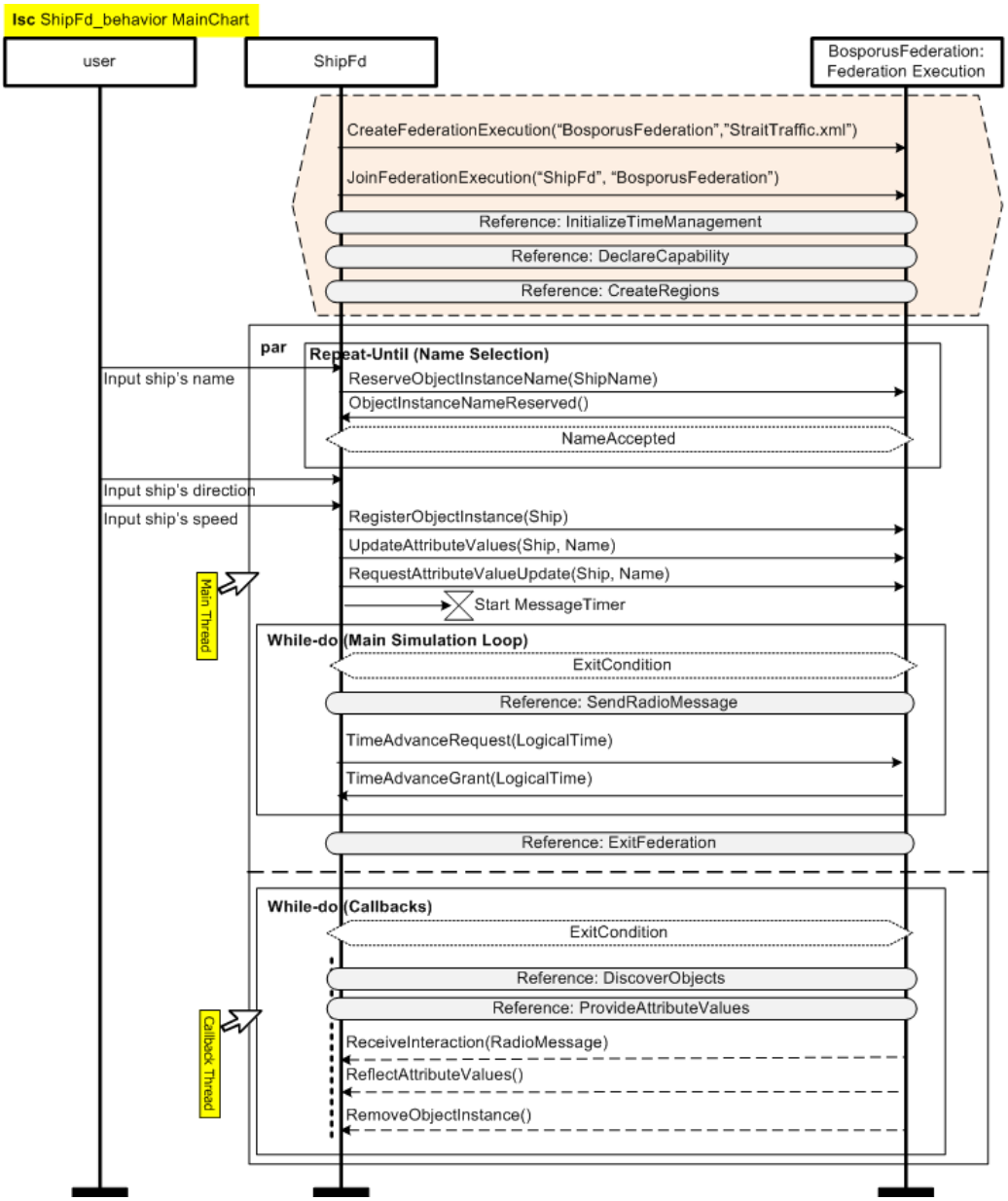


Figure 105. Behavior Model for the Ship Federate in LSC' Graphical Notation

The behavior model for the Ship Federate (ShipFd), which is an instance of ship application, includes the federation execution and user interactions as well as the application logic. There are three instances (represented as rectangles): User, ShipFd, and Bosporus Federation (i.e., an instance of strait monitoring federation). The code generator only generates code for the federate instance (i.e., ShipFd). The vertical lines represent the lifelines for the instances. A typical LSC includes mainly two charts: a pre-chart (the diamond-shaped area on top) and a main chart (the rest of the chart). The pre-chart behaves like a conditional. If it is satisfied, the

main chart is executed. In the pre-chart of the diagram, the ship federate first creates the Bosporus federation and then keeps interacting with it as it initializes its time management policy, declares its data interests, and creates data distribution regions. The behavior for these interactions is defined in separate LSC diagrams (not shown), namely, `InitializeTimeManagement`, `DeclareCapability`, and `CreateRegions`, and is referred to by references (the oval shapes) within the pre-chart. If the ship federate successfully completes the pre-chart, then the diagram proceeds with a parallel execution structure covering the rest of interactions with the user and the federation execution. This structure includes two operands that run in parallel: the main thread and the callbacks thread. A condition (`ExitCondition`) synchronizes the exit for these threads. In the callbacks part, the callbacks can occur in any order, and therefore they are connected to a coregion (designated by the vertical dotted line).

Figure 106 depicts the corresponding model of the pre-chart part of the diagram in abstract syntax. The right pane shows the structure of the project while the left pane depicts the behavior model of the ShipFd corresponding to the pre-chart. The abstract syntax is in a one-to-one correspondence with the LSC. Therefore, the traceability is straightforward. As seen, the message-out events are connected to the HLA methods (specified in the methods library). The reference modeling elements are used to point to other LSCs.

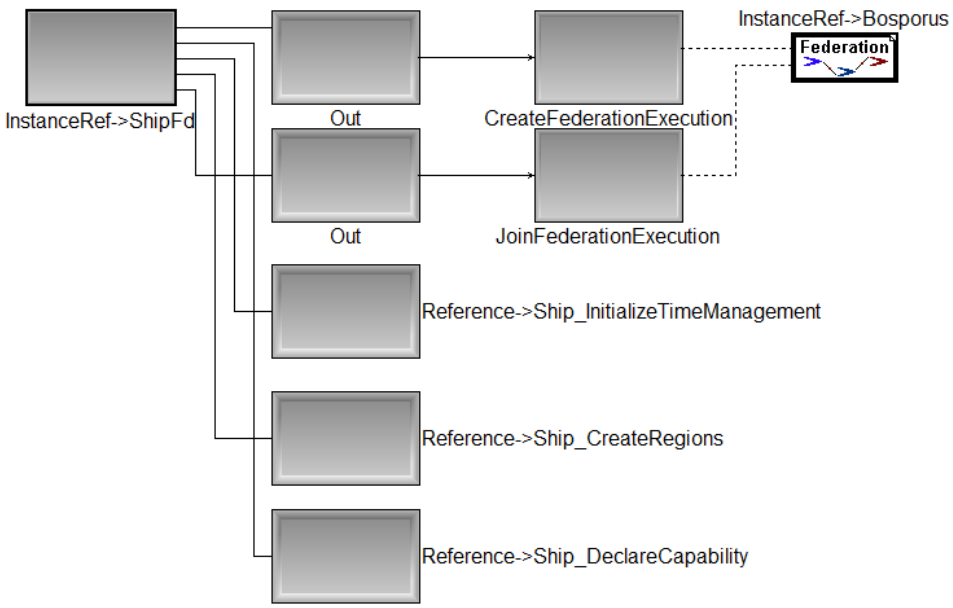


Figure 106. Pre-chart Part of Ship Federate's Behavior Model in Abstract Syntax

The HLA methods can only be connected to the MSC message events (out and in) and method call events (call out, receive, reply in, and reply out).

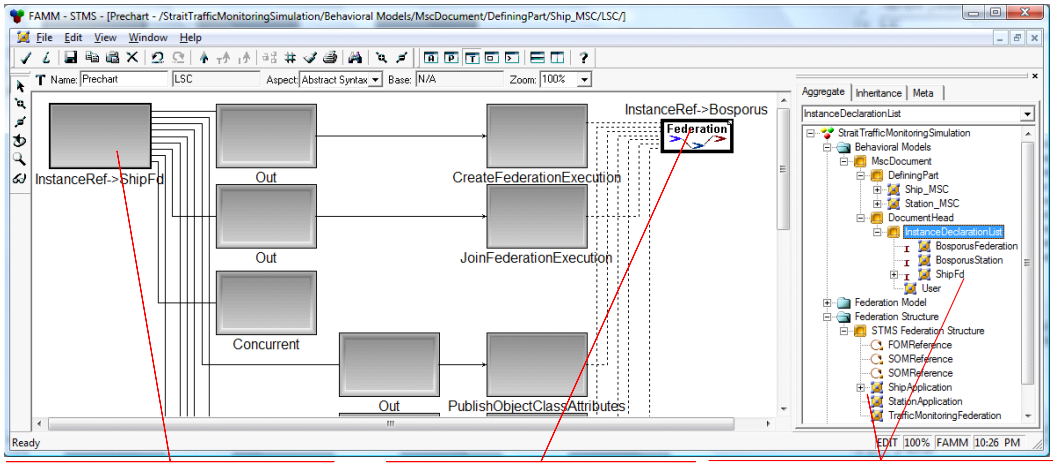
A.7.1 Creating Federates and Federation Executions

Federates and federation executions are the main instances that interacts in a behavioral model. Federate corresponds to the joined federates. It is created by instantiating the `FederateApplication` element. Federation execution corresponds to the federation executions created by RTI. A federation execution is the primary instance the federate interacts with (e.g., joining the federation, receiving an interaction etc.). It is created by instantiating the `Federation` element. For example, Figure 107 presents how to create federates and federation executions. ShipFd and Bosporus Station, which represent two different types of joined federates, are instantiated from the federate applications: ship and station applications, respectively; while Bosporus Federation, a federation execution, is instantiated from the Traffic Monitoring Federation

`FederateApplication` and `Federation` modeling elements in Federation Structure Model provide a template (type model) for the federate and federation execution, respectively. To create a federate and federation execution;

- First, design the Federation Structure Model as described in the previous sections,
- Create an Instance Declaration List under the Document Head of the MSC Document,
- Instantiate the federate and federation execution elements by dragging the type models (i.e., FederateApplication and Federation) while pressing the [Alt] key, and dropping them into the Instance Declaration List,
- Rename their names as appropriate, for example, in STMS federation, federate, named `ShipFd` is created.
- Now, they are ready to be used in the behavior charts. In the behavior charts (e.g., LSC), use only the instance reference elements that refer to the instantiated models. They are also used in HLA method calls such as `JoinFederationExecution` and `CreateFederationExecution` method calls.

It is also possible to instantiate federation multiple times and thus to create multiple federation executions.



Federate: This element is a reference to the ShipFd which is instantiated from the Federate Application, ShipApplication.

FederateExecution: This element is a reference to the BosporusFederation which is instantiated from the Traffic Monitoring Federation.

ShipFd is instantiated from Ship Application where BosporusFederation is instantiated from TrafficMonitoringFederation

Figure 107. Creating Federates and Federation Executions

A.7.2 Reserving Object Instance Names

After joining the federation, a federate may reserve object instance names with RTI. Although this step is not a compulsory step for a federate, sometimes, it is important to reserve unique names through federation executions. For example, in our running example, it is important to have unique ship names in the federation. Therefore, after joining the federation, first we seek a unique ship name and then try to reserve it (using a repeat-until block).

This part of the federate's behavior also presents an example for how to connect a repeat-until block condition (i.e., until condition) with a Boolean indicator of an HLA method.

The LSC portion, extracted from Figure 105, for reserving the object instance names is depicted in Figure 108. If `ObjectInstanceNameReserved` callback returns a true indicator, then the repeat-until loop will be exited reserving the name successfully. Else, the loop will start over (i.e., the user will input a new name and the federate will try to reserve it again).

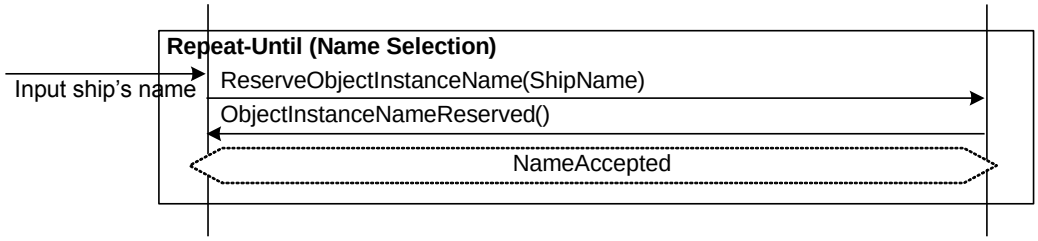


Figure 108. LSC for Reserving The Object Instance Names

To preserve this semantic, the condition `NameAccepted` for the repeat-until loop must be connected to the indicator provided by the `ObjectInstanceNameReserved` callback. To do this, the modeler must define an indicator variable in the federate's variable list. For example, in Figure 109, an indicator (`Indicator_True`) is defined to represent the true valued indicators. As seen in Figure 109, both the callback method argument and the repeat-until construction condition refers to the same indicator. Thus, if callback returns true, then the loop is exited successfully.

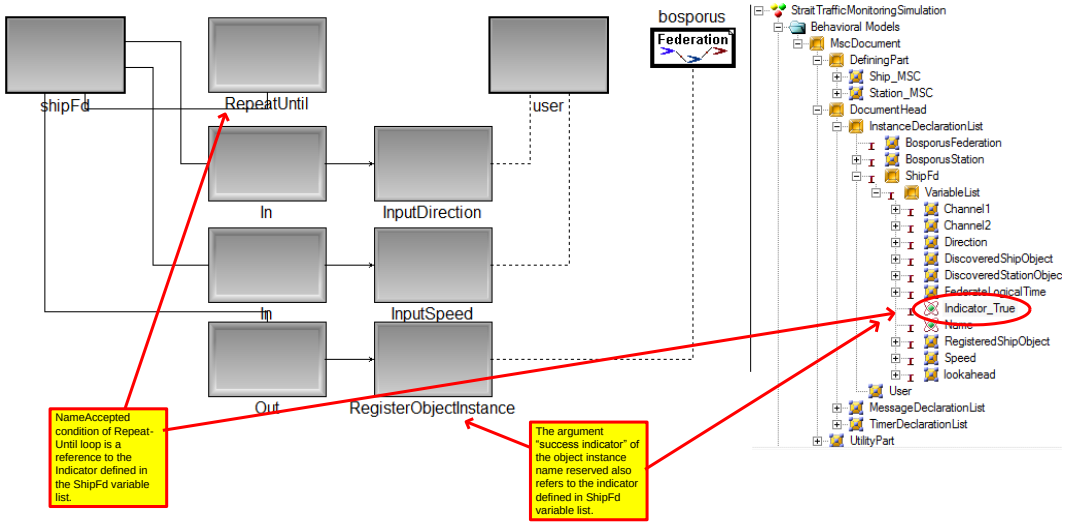


Figure 109. Model for Reserving the Object Instance Names

A.7.1 Creating Elements for the Variable List

FAMM allows declaring the elements: message retraction designator, region, timestamp, lookahead, and object instance in the variable list of the federate Instance. Message retraction designator and region are directly created inhere

while others are instantiated from the templates (e.g., object instances are instantiated from the object classes).

A.7.1.1 Creating Object Instances

Object instances are created by instantiating the ObjectClass elements declared in FOM. They are placed in the variable list of a federate application. The method calls that use or refer to the object instances have a reference to point to the instance declared in the variable list.

A.7.1.2 Creating Timestamp and Lookahead

All federations shall document their use of time stamp and lookahead via the time representation table in OMT [3]. When using FAMM; timestamp, lookahead, and their data types are created in FOM folder. These provide a static template. The instance of these static templates can be created in the `Variable List` folder of the federate application (i.e., MSC Instance). After creating instance of timestamp or lookahead, one can assign a value to it.

The instances declared in variable lists can be used in the method calls that include “timestamp/lookahead” references as arguments, such as `EvokeCallback`.

A.7.1.3 Creating Regions and Dimensions

First, the dimensions, which constitute a region, must be defined in the federation object model. To do this, under the `Federation Model Folder`, create a `Dimensions` folder. Herein, we can create the dimensions. Each dimension has a type and a normalization function as described in [3]. `Type` is a reference to a pre-defined type in `Data Types` folder. In the normalization function element, one can specify the upper and lower limits.

Second, in the `Variable List` of the federate under concern, the `Region` element can be created. Each region element has a reference to the dimension elements. Now, regions are ready to be used in the behavioral charts.

Regions are handled via the HLA DDM methods such as `CreateRegion` and `CommitRegionModifications` calls. All these methods have a reference to the regions defined in the variable lists. `CreateRegion` method call creates the regions while `SetRangeBounds` method call sets the boundaries for the regions.

An example, a one dimensional communication space for radios, is created to illustrate a distribution region. In our example, radio communication is carried out using the radio channels: channel-1 for ship-to-ship communication and channel-2 for ship-to-shore communication. Channels are regions over the channel dimension (there are two channels from zero to 3). Channels are defined by the dimension numbers as shown in Figure 110.

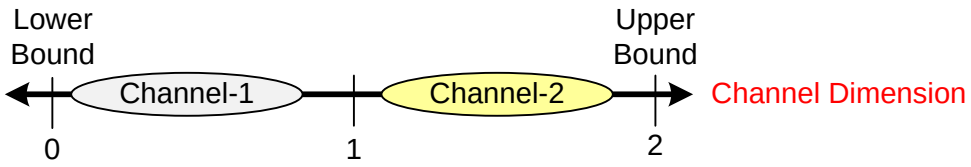


Figure 110. DDM Example

Figure 111 presents the model.

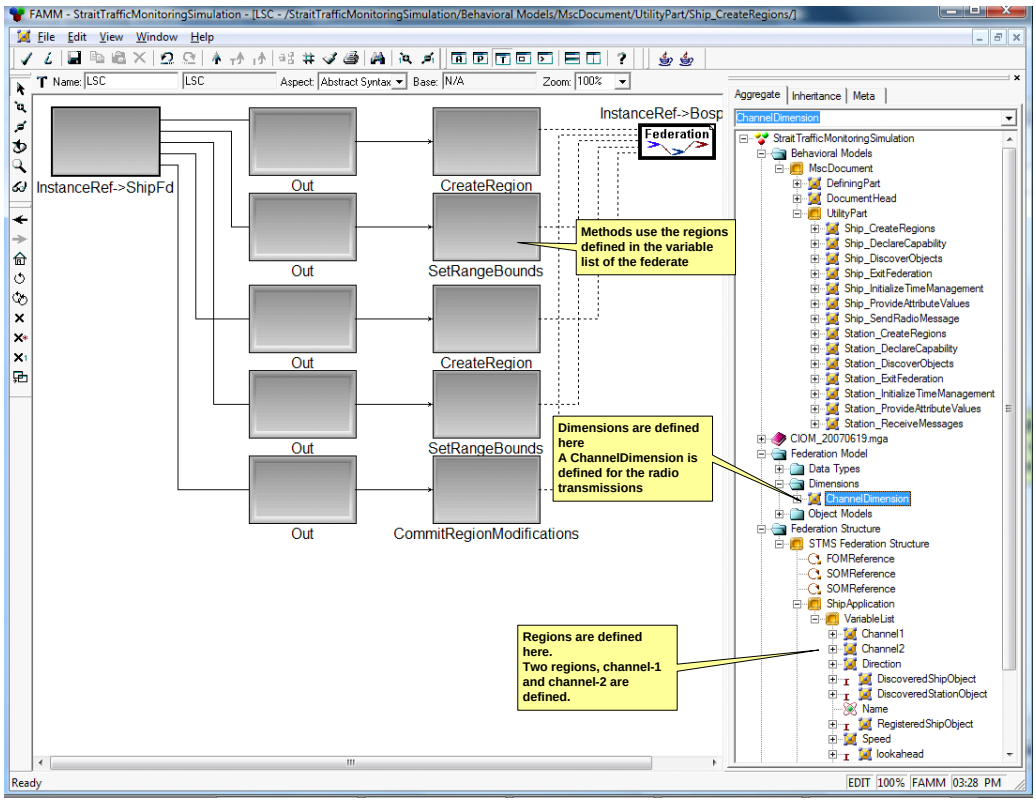


Figure 111. Creating Regions and Dimensions

A.7.2 Modeling Callbacks

A.7.2.1 Discovering Objects

Discovering objects are done via the `DiscoverObjectInstance` method. A federate may discover many objects; it is the user responsibility to specify what the federate will do after object discovery. There are two modeling approaches for object discovery:

A loosely modeling approach is not to model the behavior after the federate discovers an object. If modeling each object discovery has no impact over the design, then it is sufficient to put only one `DiscoverObjectInstance` call to the model and to leave the arguments empty. After code generation, the modeler can weave the object discovery codes.

If the modeler wants to model each object-discovery and what-to-do-afterwards, then the modeler can use PAR operator for each object discovery. In Figure 112, object discovery for two different object classes: `Ship` and `Station` are modeled. After discovering the objects, the federate requests object updates for each of them. `DiscoverObjectInstance` calls are marked as cold messages as well as their locations in order to indicate that the call “may” be received.

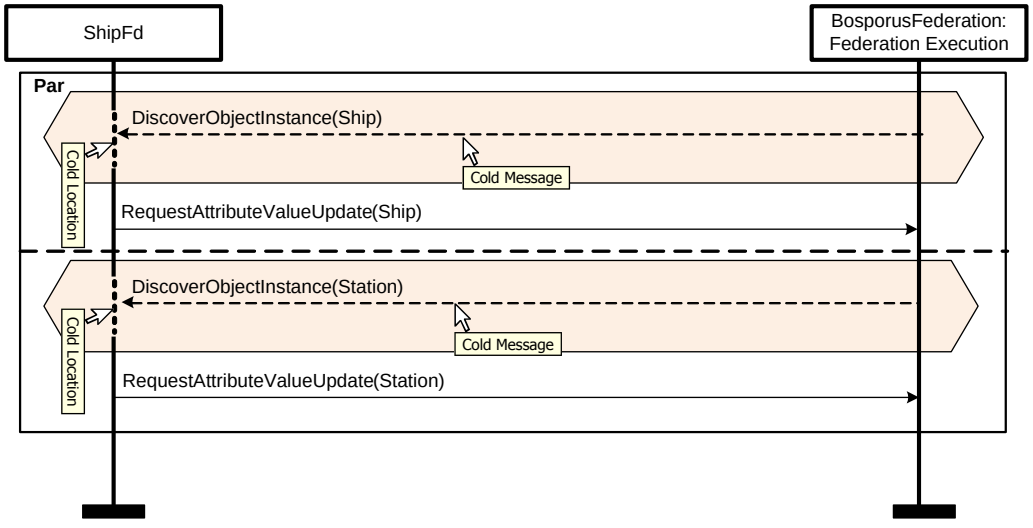


Figure 112. Object Discovery

A.7.3 Setting the Temperatures

The temperature of a location can be specified by setting the temperature attribute of the event connection. For example, as seen in Figure 113, to set the location temperature of the discover object instance message input, first click the connection between the input event and the instance reference, and then set the temperature attribute found in the attributes pane. In the same way, to specify a message/condition temperature, first select the message, and then set the attribute.

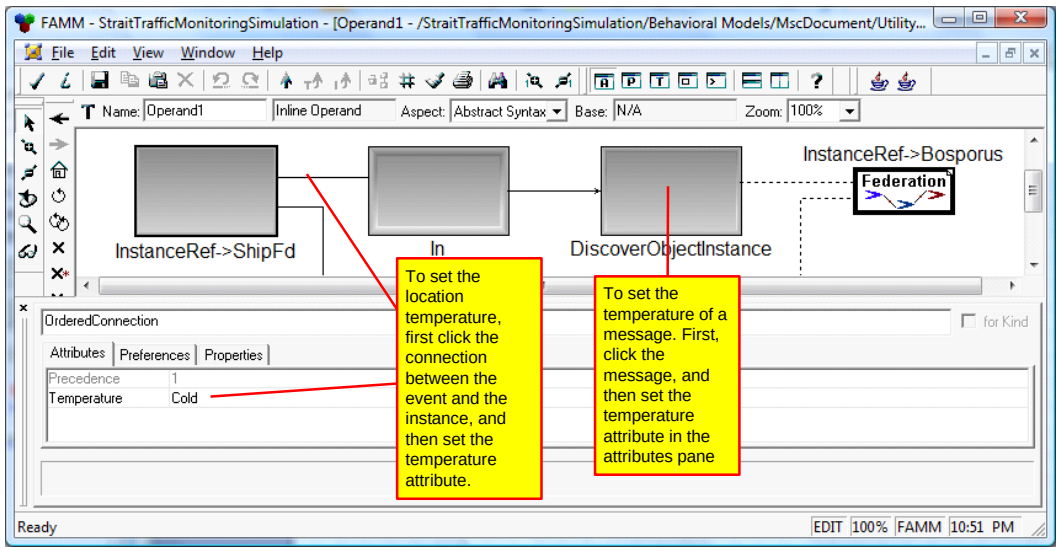


Figure 113. Setting the Temperature of a Location and a Message (GME Screenshot)

A.8 Using the Libraries

GME supports model libraries, which are ordinary GME projects. Each GME project can be used as a library if both the library and the target project are based on the same version metamodel (i.e., FAMM). The primary ways of using libraries is to create sub-types and instances from the library objects. It is also possible to refer library objects through references. Apart from being read-only, objects imported through the library are equivalent to objects created from scratch [11].

There are auxiliary libraries that can be readily attached to a project. Three libraries are currently provided: IEEE 1516.1 Services Library, IEEE 1516.1 Management Object Model Library, and IEEE1516.2 HLA Defaults Library.

Libraries are provided in "`.mga`" format. In order to attach a library into a project, right click to the root folder, and select `Attach Library` option. Choose the library file in the opened dialog box.

Please, note that before attaching a library, first register the base paradigm of the library that will be attached to the project. The base paradigm for IEEE 1516.1 Services Library, IEEE 1516.1 Management Object Model Library, and IEEE1516.2 HLA Defaults Library is HLA Federation Metamodel (HFMM).

When a library version is changed, it is sufficient to refresh the library in the project, do not re-attach or delete the library. To do this, select the library in the project, right click, and select `Refresh Library` option.

A.8.1 IEEE 1516.1 Methods Library

An IEEE 1516.1 HLA Methods library serves the template methods (a type model) of the HLA services specified in [2] for the actual use (an instance model). This library is required to model the federate behavior. In order to use this library, we simply attach it to the model concerned.

A.8.1.1 Using Template Methods Defined in the Library

There are three ways to use the template methods found in the library.

First and the common way to use the library is the *Instantiation Approach*. In this approach, whenever the modeler wants to use a template method in the library, he/she instantiates the template method (in other words, he/she creates an instance model from the type model). In GME, instance models can be created by dragging the type model and dropping it while pressing `alt` key. The modeler cannot change the number and type of the arguments, but only the argument values. This approach is the common way to use the HLA methods in the library and must be preferred to the other approaches in general. An example for the usage of this library is already presented in Figure 106 for STMS federation. The method calls, such as `CreateFederationExecution` and `JoinFederationExecution`, are instantiated from the template models specified in the HLA Services Library attached to the project.

Another way is to use a *Method Reference* to refer to a template method in the library. This approach can be preferred only if the argument values are not specific for each method call and where the template method is appropriate for each call.

The non-argument HLA methods are typical examples for such a use. For instance, `QueryFederationRestoreStatus` method has no supplied and returned arguments. So, when calling it, instead of instantiating it each time, the modeler may choose to use a reference pointed to it.

The last way to use a template method is to sub-type it first, and then instantiating it. As presented in the first approach, the instantiation of a template method does not allow changing the number of arguments in a call. But, some HLA methods use the sets, collections and lists as supplied or returned arguments. These containers may include a number of elements. Since the template method provides only one element for these containers, the modeler may need to modify the number of the elements in the template method. Therefore, first a sub-type of the template method must be created in the declaration list of the MSC document head. Then, the modeler can modify the number and type of the arguments as needed. Lastly, the sub-type can be instantiated as described in instantiation approach. For example, `PublishObjectClassAttributes` method has a "set of attributes" supplied argument to specify the attributes to be published. The library provides a template method for it, but the set argument of the method has only one attribute reference. To add new attribute references, the modeler must sub-type it to add needed attribute references and then use (instantiate) it in the model.

A.8.1.2 Using Arguments of the Template Methods

The modeler must be familiar with the HLA methods and their arguments. Arguments are provided to the modeler inside the template methods.

Most of the arguments are provided as null references for the object model such as federation reference, federate application reference, and object class reference. While modeling the federation architecture, the modeler must (re)direct the reference at any time by dropping a new target modeling element on top of them. For instance, `CreateFederationExecution` method has a federation reference, as a supplied argument. This argument is provided as a null reference in the template method. The modeler must manually direct which federation execution this null reference refers to.

Some arguments are provided in form of the Boolean and string type values. These are the indicator and string type arguments. When a method has this type of argument, the library provides both the element itself and its reference in the arguments of the template method. Only one must be utilized in modeling where

the modeler may prefer to use the element itself in case specifying the value directly or its reference in case referring to other arguments defined in another method. A code interpreter must check the reference first, if it is null, then it must interpret the value assigned to the element itself. Example in Figure 94 provides a sample usage for `StringTypeReference` arguments.

The order type element provides an enumerated list for the representation of the order type arguments such as the sent message order type. Message retraction designator, object instance, and region are new model elements used to represent the counterpart arguments. The modeler can use these arguments as described in chapter 5.

Appendix B provides a quick reference document for mapping the HLA services and their arguments in the library.

A.8.1.3 Exception Handling

Each exception of an HLA method found in the library is empty by default. The code generator only generates the skeleton for the exception (i.e., catch block), and the user introduces the advice code by hand.

If the modeler wants to specify a behavior for handling the exception instead of coding, then he can add an MSC reference to the exception in order to point a behavior chart.

A.8.2 IEEE 1516.1 Management Object Model Library

This library provides the required object models, specified in [2], to model HLA MOM.

Whenever HMOMLib is attached to a FAM project, MOM automatically becomes a part of the FOM. There is no need an additional association among federation, MOM, and FOM. Which federates are using MOM is easily understood from the behavioral model.

After attaching this library, predefined object and interaction classes are loaded, and then they can be used just as the federation object models do.

If there is no need for a FOM, for example, in case of modeling the architecture of a federation monitor, it is just sufficient to attach the MOM library as the object model. Federation element can be attached to the MOM instead of FOM.

A.8.3 IEEE1516.2 HLA Defaults Library

This library provides the predefined object model elements specified in [3].

A.9 Creating a New Library

Using FAMM, one can also create a new library for future use. As an example, a basic Console Input Output Model (CIOM) Library is created. This library is used to model the basic user input output via a simple console. There are two interactions; Input and Output, and two arguments; `InputString` and `OutputString`. The interactions are modeled as MSC messages.

First and foremost, as described before, create a new FAMM project, and model the interactions as message declarations having seen in the following figure. After saving this project, you can attach it as a library in your other modeling projects. For example, it is used in STMS Federation for modeling the user interactions.

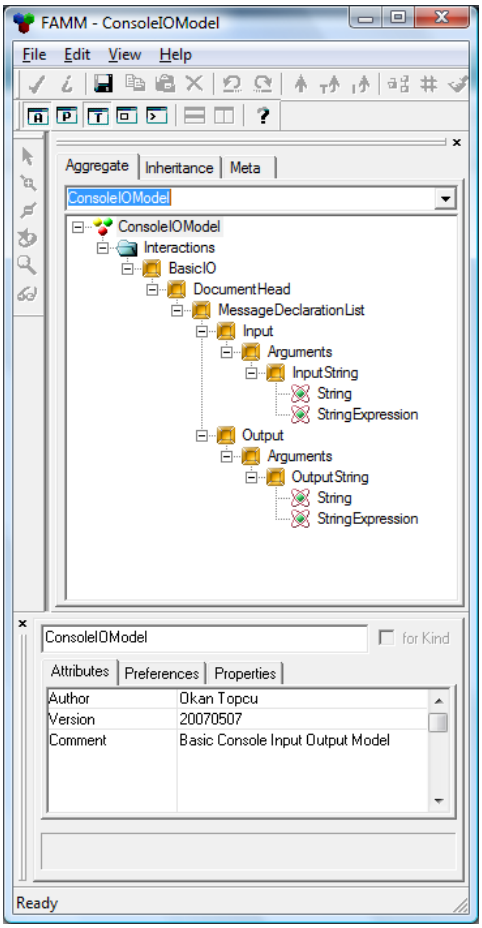


Figure 114. Console Input Output Model Library (GME Screenshot)

A.10 How to Upgrade Models

When the version of a base metamodel or a library is changed, GME allows the upgrade of the related models. The upgrade methods are threefold:

A.10.1 First Method – Default Upgrade Mechanism

The first method is to use the GME default upgrade mechanism. After installing the newer version of the base metamodel, click your project (model in `mga` format) to open it, a dialog box will ask you to upgrade the model or not. Choose `Yes`. GME will attempt to upgrade the model. If this operation does not succeed, follow the other methods.

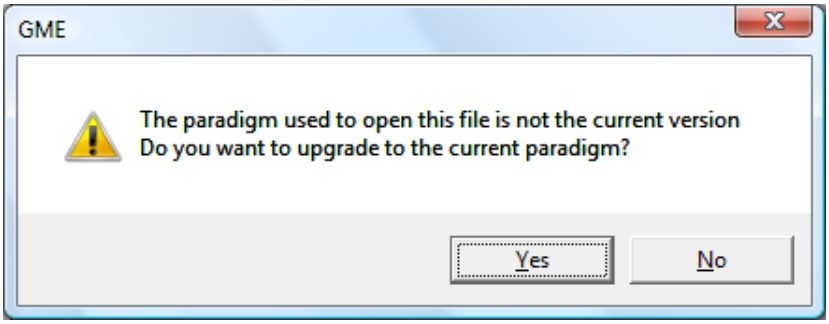


Figure 115. Upgrading the Models – Method I (GME Screenshot)

A.10.2 Second Method – Using Update through XML

If the first upgrade method does not succeed and if you do not have an export (in XME format – special XML format for GME) of your model, then you can use this method. Try the first method, when the upgrade dialog appears, say `No` this time. The model will be opened as usual without upgrading (using its old paradigm – therefore, do not delete the base paradigm). Then, select `Update through XML` method in `File` menu as depicted in Figure 116. This command allows updating the models by automatically exporting to XML and importing from it.

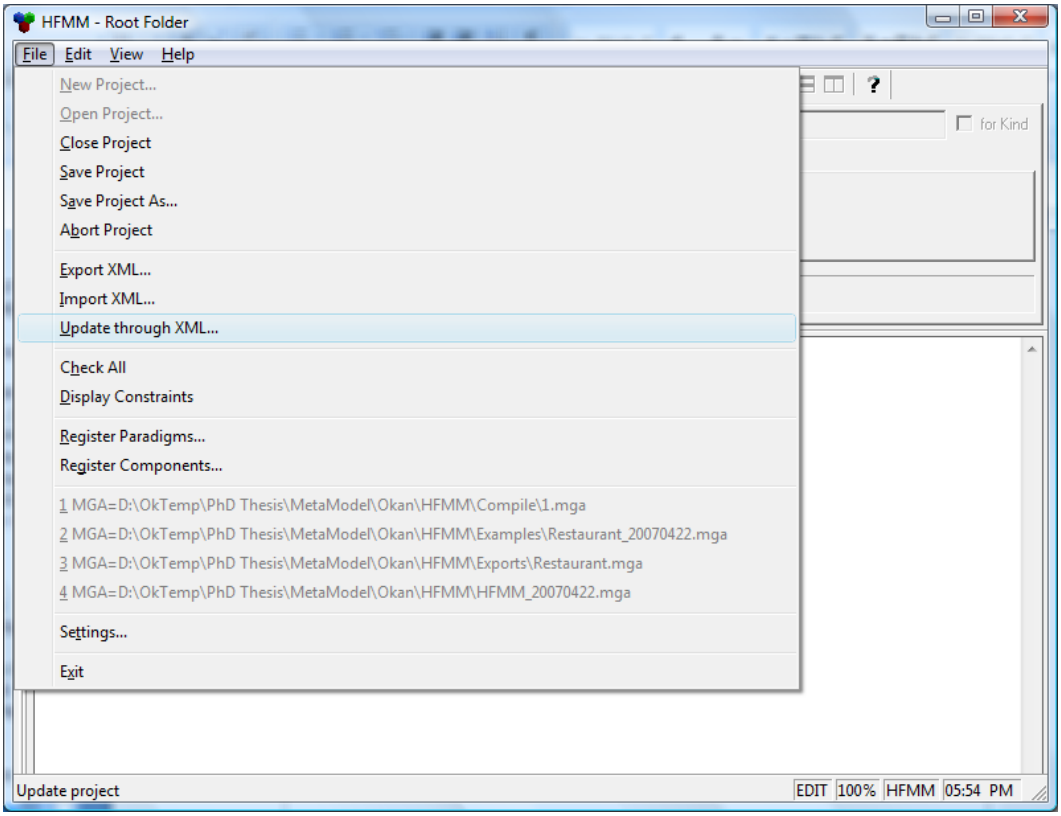


Figure 116. Upgrading the Models – Method II (GME Screenshot)

Please note what the GME Manual [11] commends on upgrading:

> New paradigm versions are not always compatible with existing binary models. If a model is reopened, GME offers the option to upgrade it to the new paradigm. If the upgrade fails, XML export and re-import is needed (the previous generation of the paradigm is to be used for export). XML is usually the more robust technique for model migration; it only fails if the changes in the paradigm make the model invalid. In such a situation the paradigm should be temporarily reverted to support the existing model, edited to eliminate the inconsistencies, and then reopened with the final version of the paradigm.

A.10.3 Third Method – Using Import/Export Mechanism

Before registering the new base paradigm, export your model, then register the newer paradigm, and import your model into a new project based on the newer paradigm.

A.11 Practical Matters

A.11.1 Setting Port Label Length

In FAMM, some modeling elements are designed as GME ports. By default, only three letters of the port names are appeared inside the model. Port label length can be changed by selecting the preferences of the model that contains the ports and then setting the `PortLabelLength` to zero.

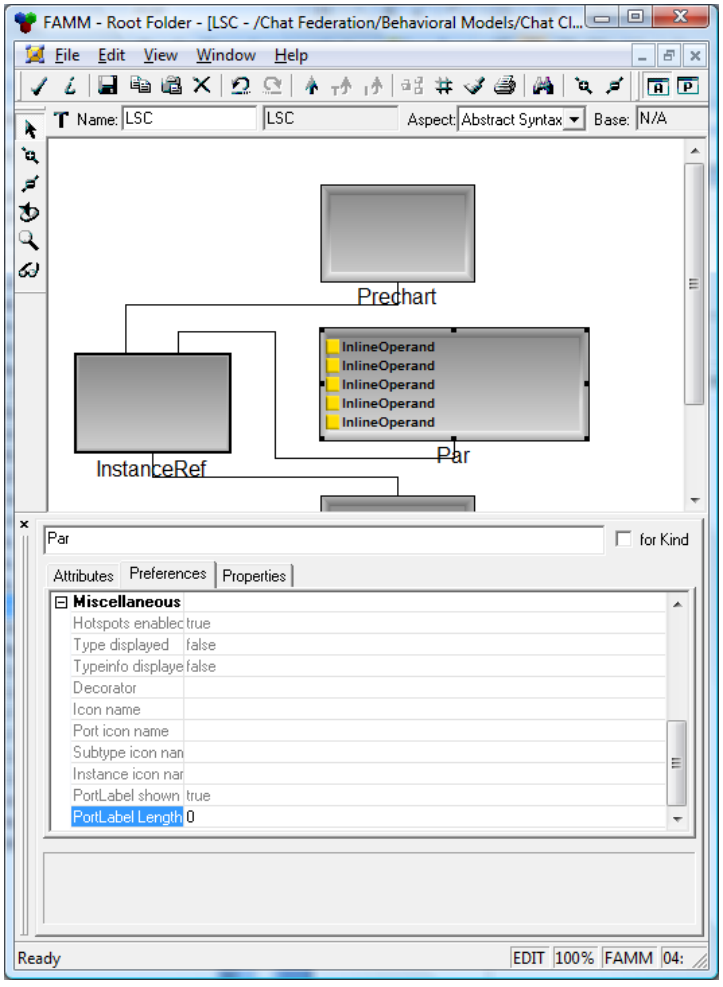


Figure 117. Setting Port Label Lengths (GME Screenshot)

A.11.2 Using the Icons

In order to use the icons provided with the FAMM for model elements instead of the standard GME icons, first create “icons” folder in project folder (this is where the

GME project file resides), and then copy the icon library into this folder. When the project is reopened, the icons will be changed automatically by the GME.

Please not that if the icons are not used in the project, then the name of the elements under the FOM folder will be blank.

A.12 Creating Publish/Subscribe Models

The P/S model visualizes the capabilities and interests of the ship application and PSMM models are generated from the behavioral part of the federation architecture by analyzing the HLA declaration management services.

After completing the federation architecture, the modeler can use the P/S Model Generator, which is an interpreter supplied with FAMM, to generate automatically the P/S models for the federation.

A.13 P/S Model Generator

A.13.1 Registering the P/S Model Generator

In order to use the P/S Model Generator (i.e., an interpreter), first the interpreter must be registered in GME. To register, open the federation architecture in GME, select `File` and `Register Components`. In `Components` panel, press `Install New` button, and select the interpreter ".dll" file.

A.13.2 Using the P/S Model Generator

After registration, an interpreter icon appears in the toolbar. Clicking this icon executes the P/S Model Generator. The first interface panel seen is the panel of the configuration utility (in Figure 118). Here, the modeler can choose the types of the P/S models (i.e., Federate-based or Class-based). Pressing `Generate` button generates the chosen type models.

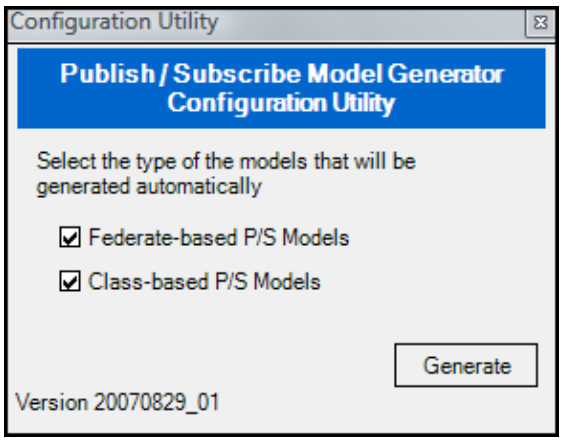


Figure 118. P/S Model Generator Configuration Utility

An example P/S model for ship application is presented in Figure 119. In the right pane, the generated P/S models according to their categories can be seen. The left pane depicts the inside of the selected P/S model (i.e., `ShipFd_PSModel`). Here, the publish and subscribe interests of the selected ship application federate are seen in a neatly formatted view. One can easily interpret from the figure that ship federate has the ability to generate radio messages and ships as it has interest in stations, other radio messages, and other ships.

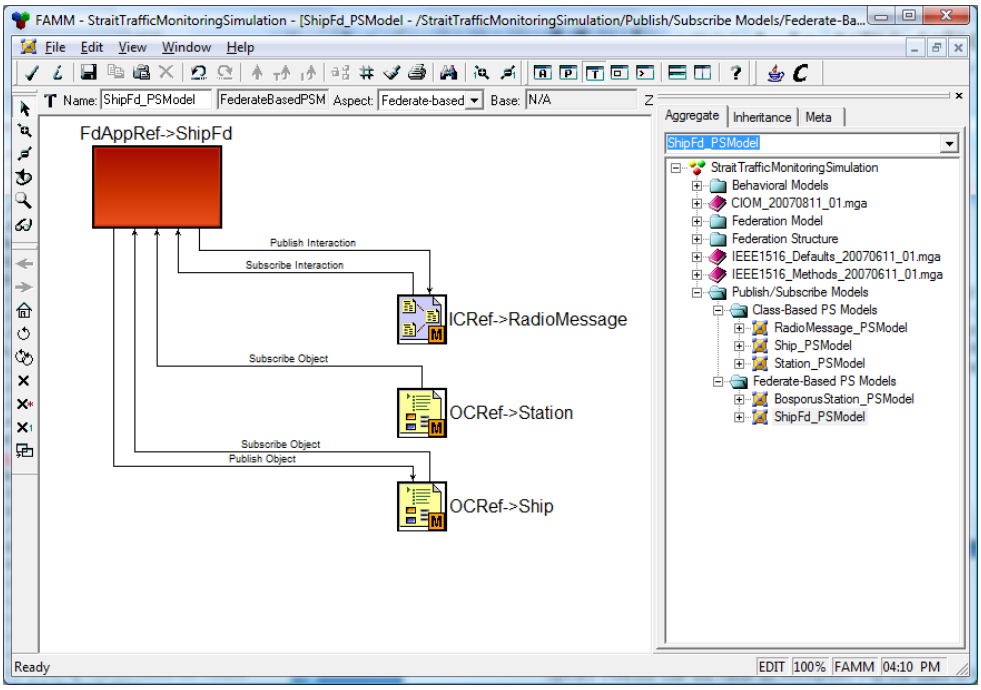


Figure 119. Ship Federate Application P/S Model (GME Screenshot)

A.13.3 Features

Some prominent features of the P/S Model Generator are as follows:

- It allows the modeler to choose the category of the P/S Model via its configuration utility written in .NET 2.0.
- It checks the HFMM and LMM are included as GME libraries. Publish and subscribe method structures used in the federation architecture where each P/S method has a class reference and connected to a federate reference. If one of these references is null, then it generates a warning and depicts the path of the null reference to the modeler.

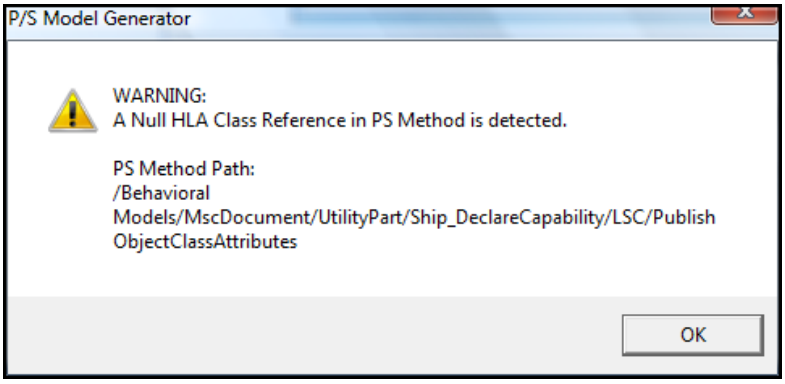


Figure 120. P/S Model Generator Warning

- As creating the model elements, the interpreter colors and positions the model elements to obtain a formatted view.
- It formats the name of the references according to their referred elements (e.g., `OCRef->Station`). Thus, it provides views that are more readable.
- It names the associations between elements (e.g., Publish Interaction) for a comfortable review.
- It uses the GME generic BON2 API for model (i.e., a federation architecture) traversing and manipulation.
- It is written in C++ using Microsoft Visual Studio 2005.
- The size of the handcrafted code is approximately about 324 lines of code (loc).

A.14 Code Generation

Based on the proposed metamodel, ongoing experimental work with the federate base code generator tests the utility of RTI-related code generation in the early prototyping of federation designs. Moreover, code generation facilitates the dynamic verification of the architecture. For instance, a retraction designator may become invalid during federation execution due to some design error. Then a retract method call will raise an exception. For the sake of not shifting the focus of the present paper, only the strategy for code generation is outlined. For further details, please refer to [31, 53].

In principal, the code generation strategy is based on the Aspect Oriented Programming approach [21], which allows us to generate code to exercise LSCs in a computation-free manner. Then the user can weave, using AspectJ [52], application-specific computational (and other non-communication) aspects onto the generated base code. In fact, the HLA-specific portions of the code are automatically weaved onto the base code generated from the LSC, but the federate programmer does not need to be aware of this process.

The LSC instance is a critical element in code generation. Interactive users, federation execution, the environment, federates, and, if desired, the components of a federate are all instantiated from the LSC instance element. In case of multiple federations on the same RTI, we have one instance per federation execution. The code generator generates code only for the federate application. Note that code generation for the environment, live entities, interactive users, and federate application components (e.g. federate graphical user interface) would require special data models to be integrated with BMM. All LSC instances are generated in separated class files and they are declared and used in the diagram code generated from the LSC diagram.

An example of the generated code for the strait monitoring federation is presented in Figure 121. For the sake of brevity, only the first operand of the parallel structure (in Figure 105) is presented.

```
public static void ShipFdMainMethod (){
(…) // Pre-chart code for federation management, initialization time management, declaration management,
and region creation
class MainThread_02ee extends Thread { //Thread for the first operand of the parallel structure.
 MainThread_02ee() {}
 public void run() {
 do { // Loop is repeated until the ReserveObjectInstanceName (ROIN) is succeeded.
  condRecvMessageInput_03e0User(); // Input of ship's name from the user
  // Reserve object instance name
  SendReserveObjectInstanceNameROINBosporusFederation("s0"); // "s0" is to be overridden by aspect
  // Object instance name Rreserved
  condRecvObjectInstanceNameReservedOINRBosporusFederation(); // Get the ROIN result
  (…) //If ROIN succeeded, while condition is satisfied
 } while (!((Boolean)Ship_MSC.coldChoices.get("until_0300")).booleanValue());
 (…)// Other inputs, direction and speed, from the user
 SendRegisterObjectInstanceRegisteredShipObjectBosporusFederation(...);// Register Object Instance
 SendUpdateAttributeValuesRegisteredShipObjectBosporusFederation(…);// Update Attributes
 SendRequestAttributeValueUpdateDiscoveredShipObjectBosporusFederation(…); // Request Attribute
Update
 doLaterMessageTimer_03c6(100); //Start timer
       // While-Do (Main simulation loop)
       while (((Boolean)Ship_MSC.coldChoices.get("ExitCondition_040f")).booleanValue()) {
  (…) // The code generated for SendRadioMessage chart is inserted here.
  condLSC_02ec=condLSC_02ec&& // Condition of the pre-chart is established
  condRecvProvideAttributeValueUpdateRegisteredShipObjectBosporusFederation();//Received cold
message
  (…) // Detailed code for receiving a cold message
  if (condLSC_02ec) {// If pre-part is satisfied, body part of the pre-chart is executed.
   SendUpdateAttributeValuesRegisteredShipObjectBosporusFederation(…);// Update attribute values
  }// End of pre-chart condition
  // Time management
  SendTimeAdvanceRequestTARBosporusFederation(new Double(55.0)); // Request time advance
  condRecvTimeAdvanceGrantTAGBosporusFederation(); // Time advance granted
 }// End of loop.
 // Exit federation reference
  SendDeleteObjectInstanceRegisteredShipObjectBosporusFederation(…); // Delete local objects
  SendResignFederationExecutionRFEBosporusFederation(0); // Resign federation
  SendDestroyFederationExecutionDFEBosporusFederation("s0"); // Destroy federation
}// End of main thread
```

Figure 121. Excerpts from the Generated Java Code of Ship Application [53]

APPENDIX B

CASE STUDY: NAVAL SURFACE TACTICAL MANEUVERING SIMULATION SYSTEM

This case study presents the work of modeling the architecture of a real life case study, Naval Surface Tactical Maneuvering Simulation System conforming to FAMM.

The introduction section introduces a case study while presenting its components. The following section, federation architecture model, presents the federation architecture and the behaviors of the federates. The last section presents the P/S models generated automatically from the NSTMSS FAM.

This appendix is mostly summarized from [25, 48, 49, 50].

B.1 Introduction

Conceptually, Naval Surface Tactical Maneuvering Simulation System (NSTMSS, pronounced “NiSTMiSS”) is a distributed virtual environment, where a group of players controls the virtual frigates (either Meko or Knox class) in real time and some players behave as tactical players that command the groups of the frigates. All shares a common virtual environment, which its environment characteristics (e.g., time of day) and parameters (e.g., the wind direction) are forced by an environment application, obeying a common scenario that is distributed (e.g., role casting), controlled (e.g., injection messages), and monitored by an exercise planner.

Technically, NSTMSS is a High Level Architecture (HLA) based distributed simulation system that is composed of 3-Dimensional ship-handling simulators, a tactical level simulation of operational area, a virtual environment manager, and simulation management processes. It has been developed by using the concepts of HLA, which provides a structural basis for interoperability and reusability.

NSTMSS provides a Networked Virtual Environment testbed for naval surface actions in which new formations can be evaluated and tested as well as the present ones can be practiced and analyzed. A potential application is training, where naval college cadets can practice formations [25].

B.1.1 System Components

Software components can be classified into three groups according their functionality:

- Simulation Entity Components
- Federation Management Components
- Environment Generation Components

During federation execution a process is created (by the host operating system) corresponding to each of the components mentioned.

Simulation Entity Group consists of the counterparts of real life entities in the virtual environment. There are three different kinds of federates in Simulation Entity Group:

- *Meko Federate (MekoFd)*: The ships are brought into federation by the Meko and Knox Class Frigate Federates (KnoxFd), which are platform level simulations allowing a person to steer the ship in (nearly) real-time. The frigate federates implement a three-dimensional ship handling interactive simulator of a frigate with a single user interface.
- *Helicopter federate (HeliFd)*: HeliFd is an interactive simulation that simulates a helicopter operated in the ships [32]. Helicopter federate is a six degree-of-freedom flight federate, which is controlled by a user.
- *Tactical Picture Federate (TacPicFd)*: Tactical Picture Federate (TacPicFd) is a tactical level interactive simulation that maintains the tactical picture of the operational area, including task group formations and maneuvers. TacPicFd provides interfaces to the user (i.e., Officer in Tactical Command (OTC)) to control and order the formations of the surface task group to achieve a given operational objective.

Federation Management Group provide facilities for controlling and monitoring the federation activity as well as distributing roles and scenarios to players. They are:

- *Exercise Planner Federate (ExPFd)*: Exercise Planner Federate, also called Scenario Manager, in short, selects the training scenario and distributes it to the participants (i.e., TacPicFd and ship federates), injects events defined in the scenario into the federation execution; collects data and generates a report about the federation execution. ExPFd simulates the Officer Scheduling the Exercise (OSE) functionality and operates as the orchestra conductor.
- *Federation Monitor Federate (FedMonFd)*: Federation Monitor Federate enables generic data collection and reporting on HLA federates about their usage of underlying Run-time Infrastructure (RTI) services by using HLA Management Object Model interface. FedMonFd is a stealth federate that also controls the federation reporting behaviors. FedMonFd provides a basis for implementation of an observer federate and provides user interfaces to monitor the status of the federation and the federates. FedMonFd collects the federate specific RTI data and presents them in tables. FedMonFd also provides detailed reports for review of the monitoring activity.

Environment Generation Group consists of one federate:

- *Environment Generation Federate (EnviFd)*: Environment Federate is designed to control the atmospheric and sea state of the virtual environment. Environment Federate enables for the user to control the virtual environment atmospheric effects (e.g., fog, time of day, sea state), and publishes the weather reports to the entities in the virtual environment at scheduled intervals specified in scenario file.
- Figure 122 graphically depicts the software components of NSTMSS. Multiplicity information in the figure indicates that while many Meko, Knox, Helicopter, and OTC federates can participate in the environment, at most one ExPFd, FedMonFd, and EnviFd can participate.

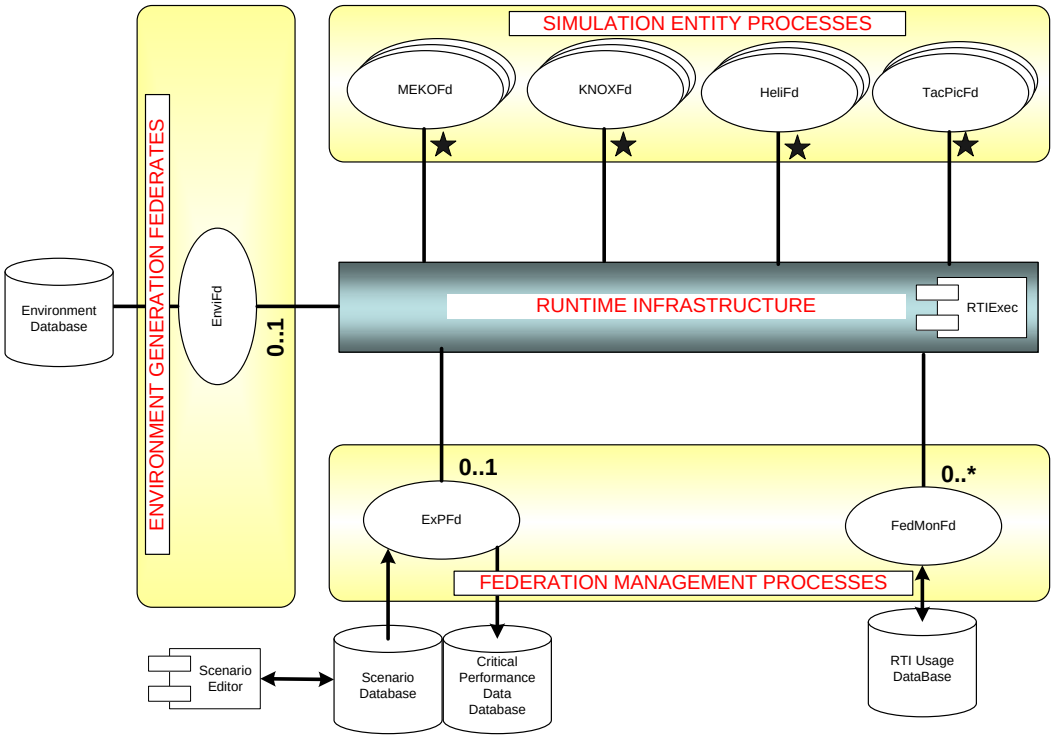


Figure 122. Federation Structure [25]

B.1.2 Scope of the Case Study

Due to large numbers of federates in NSTMSS federation; this case study does not attempt to cover all the NSTMSS federates, but it selects the interesting federates which differentiate in terms of behavior and architecture. Eventually, all the federates in the federation management group and only the Meko Federate in the simulation entity group are covered. The federates in the federation management group are good candidates for a re-modeling study because they represent common federates in a typical HLA federation. FedMonFd is a generic federate that represents the stealth monitoring federates and is independent from any specific federation. ExPFd is a simulation planner federate that is typical in most of the distributed interactive simulations. On the other hand, MekoFd represents platform federates and has the same pattern (in terms of behavior and architecture) with the other federates in the simulation entity group.

B.2 Federation Architecture Model

In an HLA based distributed simulation, federation design, which aims at having federates cooperate to achieve the federation objectives, involves two major

activities. The first one is forming the federation model (i.e., the structure and the object model of the federation), presumably by reviewing the simulation object models. This part covers designing information interests of federates as well as object flows such as data distribution and ownership management of objects. The second activity is specifying the behavior of the participating federates so that they can fulfill their responsibilities in the federation scenario [48].

Federation structure establishes the links among the federates, the federation, FOM, SOM, and MOM. Definitions of FOM, SOM, MOM, federation-wide data types, and HLA specific objects are included in Federation Model. Eventually, Behavioral Models folder contains the behavioral descriptions of federates. Libraries based on FAMM can be attached to the project as well. For example, Figure 132 shows three such libraries, namely the CIOMLib, IMLib, HDefLib, and HMOMLib.

B.2.1 Federation Model

NSTMSS federation model includes the federation structure model and federation object model.

B.2.1.1 Federation Structure

The federation structure model represents the federation overall structure in terms of federate applications. It shows the software components of NSTMSS. It also presents a multiplicity information how many federates of a certain type can join in the federation.

Relationships between the federation, federate, SOM, and FOM are described using the FAMM notation.

The model in Figure 123 corresponds to the architectural view in Figure 122.

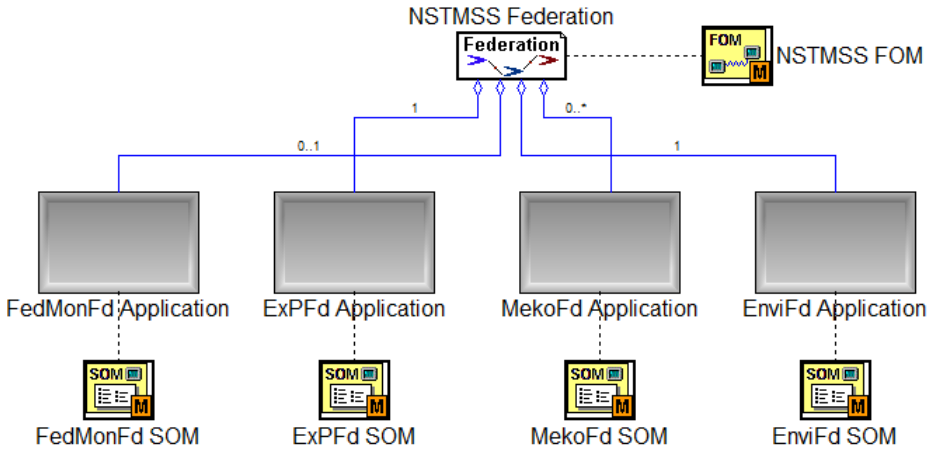


Figure 123. NSTMSS Federation Structure

B.2.1.2 Federation Model

NSTMSS Federation Object Model (FOM) includes the all the information needed in HLA Object Model Template Specification [3]. In this study, no separate SOM model for the federates is provided as the same set of object classes and interactions in FOM are assumed by all federates in the NSTMSS federation.

The modeler only models the application specific HLA classes. For a monitoring federate such as FedMonFd, which additionally uses the MOM as the object model, the MOM classes are provided by adding the HMOMLib to the project. Thus, no extra effort is needed.

NSTMSS federation object class hierarchy is depicted in Figure 124, and interaction class hierarchy is depicted in Figure 125.

`OTC` class represents the tactical officers in the virtual environment. `Knox` and `Meko` classes are inherited from the `Frigate` class. They represent the Knox-class and Meko-class frigates respectively. `Environment` class represents the characteristics of the virtual environment. It includes time of day, sea state, wind state, and fog state data. `Scenario` class includes the federation scenario information such as scenario start time and scenario location. `Helicopter` class represents a generic chopper in the virtual environment. `HLAObjectRoot` is the root class by default for all the HLA classes.

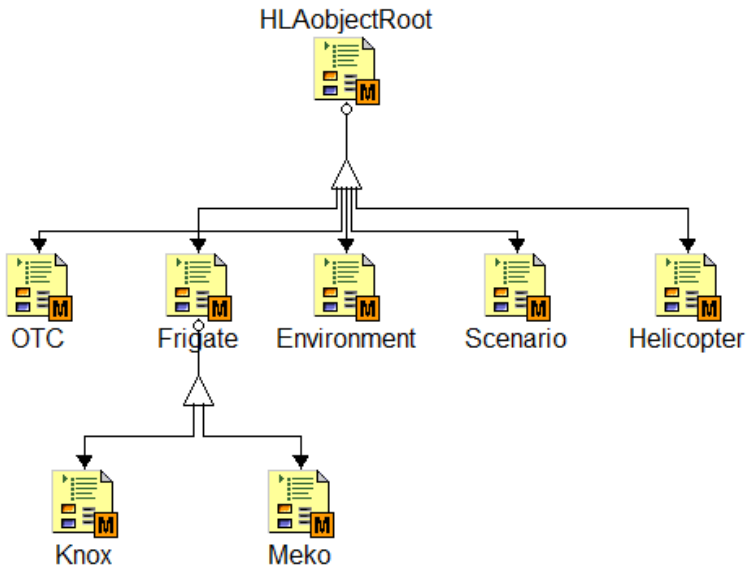


Figure 124. Object Class Hierarchy of NSTMSS

`BeginToPlay` interaction is used to signal the federations to indicate the start of the federation scenario. `Communication` class is used to represent the tactical messages (e.g., task order and formation message) in the virtual environment.

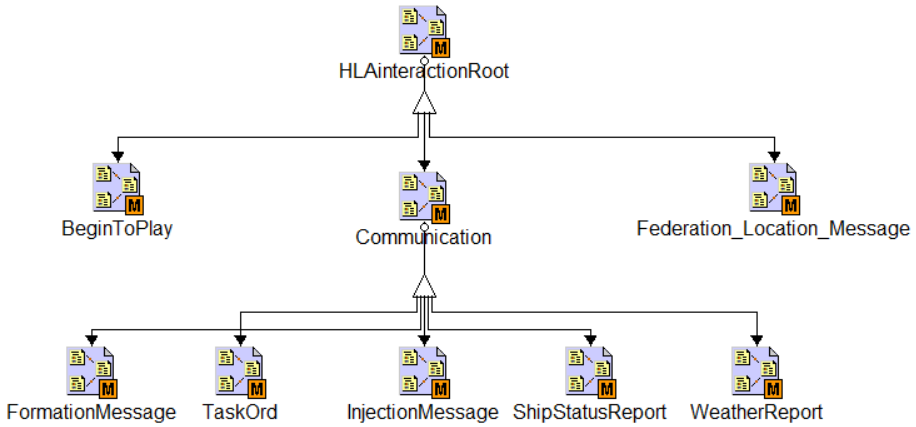


Figure 125. Interaction Class Hierarchy of NSTMSS

Each class is a model where it means that there is a structure in it (i.e., object class models have attributes and interaction class models have parameters). The parameters of a `WeatherReport` interaction are seen in the following figure. It is used to report and change the environment data in the virtual environment.

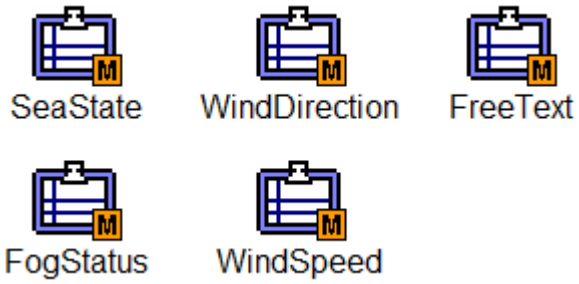


Figure 126. Parameters of a WeatherReport Interaction Class

In addition to the object models, the NSTMSS data types and dimensions are also defined in the federation model. A frequency spectrum dimension is defined in NSTMSS. The UHF and HF regions based on this dimension are created for radio communications in the virtual environment.

As enumerated data types, `FogTypeEnum`, `SeaStateEnum`, and `ShipType` are defined as well as simple data types such as `float` and `int`.

Again, HMOMLib provides its own pre-defined data types and dimensions.

B.2.2 Behavioral Models

The behavioral model of NSTMSS federates are obtained mostly by reviewing the C++ source code and sometimes by observing the run-time action. The behavioral models are first formulated in the LSC graphical representation and then manually transferred to FAMM.

The LSC specification is a simplification of the actual federate' behavior. Because, the main focus was on the interaction between the federate and the RTI. The federate's computation logic is intended to be weaved into the base code using the Aspect Oriented Programming (AOP) [51] techniques after the federate base code generation step [48].

Here, only the behavior model of MekoFd is presented and discussed. For the presentation and discussion of behavior models of ExPFd and FedMonFd, see [49] and [50] respectively.

B.2.2.1 Behavior Model for MekoFd

MekoFd is a typical platform-level and interactive simulation, where the player drives a platform.

Program flow of MekoFd is divided into three phases: initialization, main simulation loop, and system termination. In initialization phase, MekoFd tries to create the federation, joins it, and then declares its interests (i.e., publish and subscribes its object and interactions). In main simulation loop, MekoFd creates its object (i.e., the type of Meko object class), sends interaction (i.e., ship status reports), and discovers objects (i.e., other frigates in the environment, scenario, or environment). In final phase, MekoFd resigns federation, deletes its created objects, and tries to destroy the federation.

Although the program flow dictates three phases, the behavior model of the MekoFd is dispersed into two main charts, the pre-chart and the main chart, as seen in Figure 127. The pre-chart mainly involves the initialization phase activities. Pre-chart behaves as a prerequisite. Unless it is satisfied, the main chart is never activated. In other words, unless MekoFd successfully completes its initialization phase, it will never execute its main simulation loop. The main chart includes two parts; the main and the callback threads. The main thread covers the main simulation loop as the loopback covers the federate ambassador methods (i.e., the HLA callback methods initiated by the RTI).

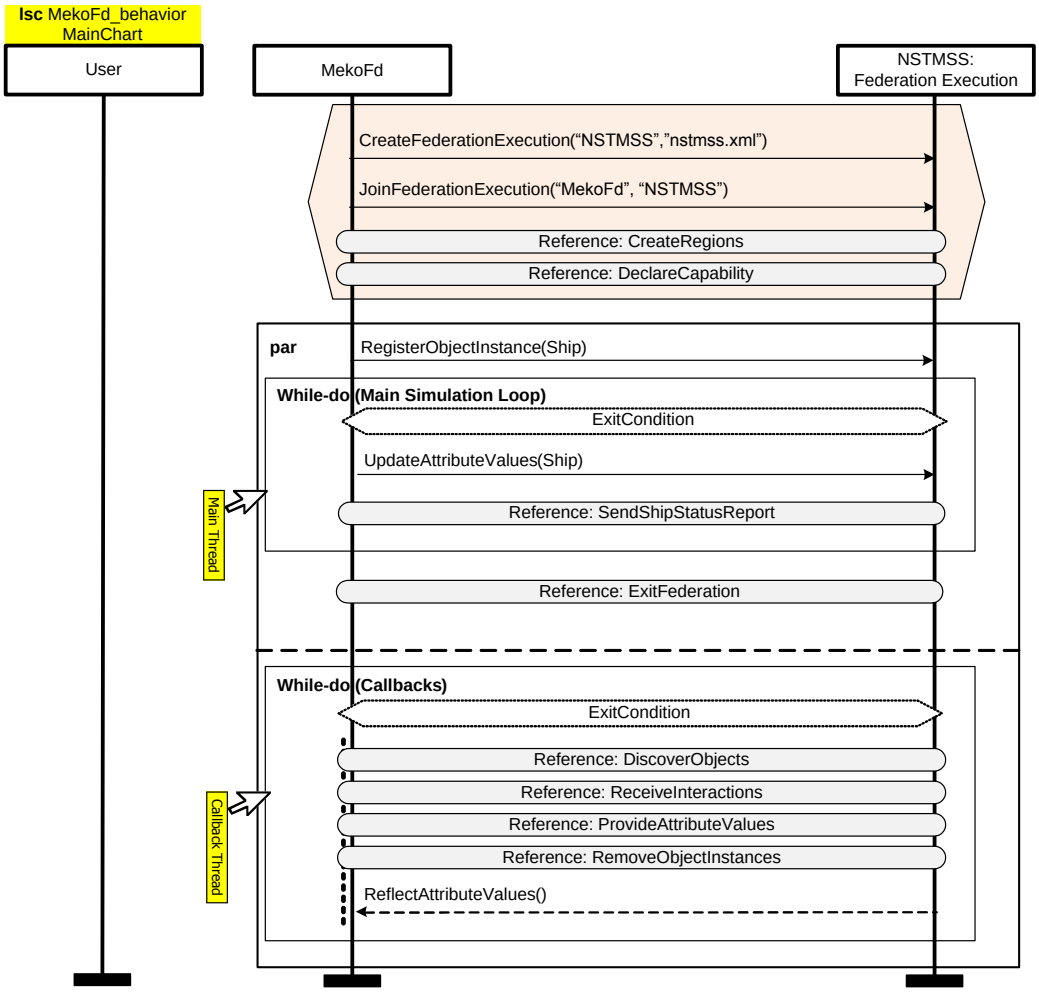


Figure 127. MekoFd Main LSC

As a modeling principle, the details of interactions (events) between the MekoFd and the RTI are encapsulated in sub-charts (e.g., `CreateRegions` and `ExitFederation`), which are referenced from the main chart. Each sub-chart is a functional part of the behavior. Dividing the main LSC into sub-charts increases the readability, understandability, and modularity of the architecture.

The main chart in FAMM notation is depicted in Figure 128. The parts of the main chart are modeled using operands in a parallel structure. Thus, the main thread and the callback thread runs in parallel.

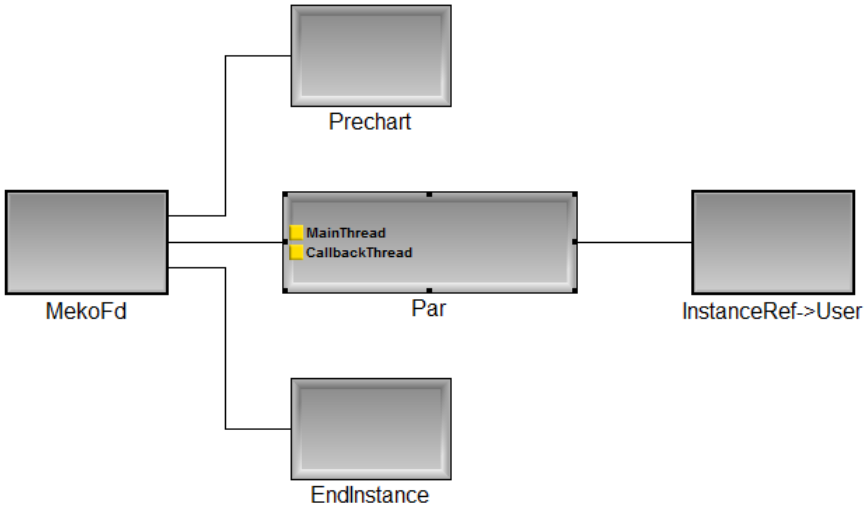


Figure 128. MekoFd Main LSC in FAMM

An interesting part of the behavior is sending the ship status report. After the scenario has begun, the ship periodically broadcasts a status message to the environment (i.e., send an interaction). The transmitting period is defined in the attribute (i.e., `ReportInterval`) of the scenario object. To model the report interval, a timer is set when scenario-begin interaction (i.e., `BeginToPlay`) has received. Whenever the timer ticks, a ship-status-report interaction is sent.

Figure 129 presents the send-ship-status sub-chart. Figure 130 presents the corresponding model in FAMM notation.

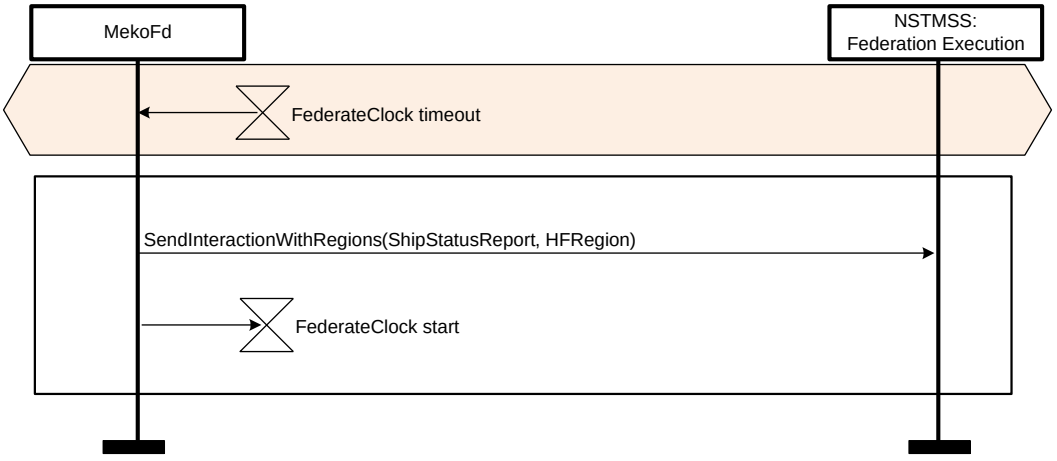


Figure 129. Send Ship Status Report Sub-chart

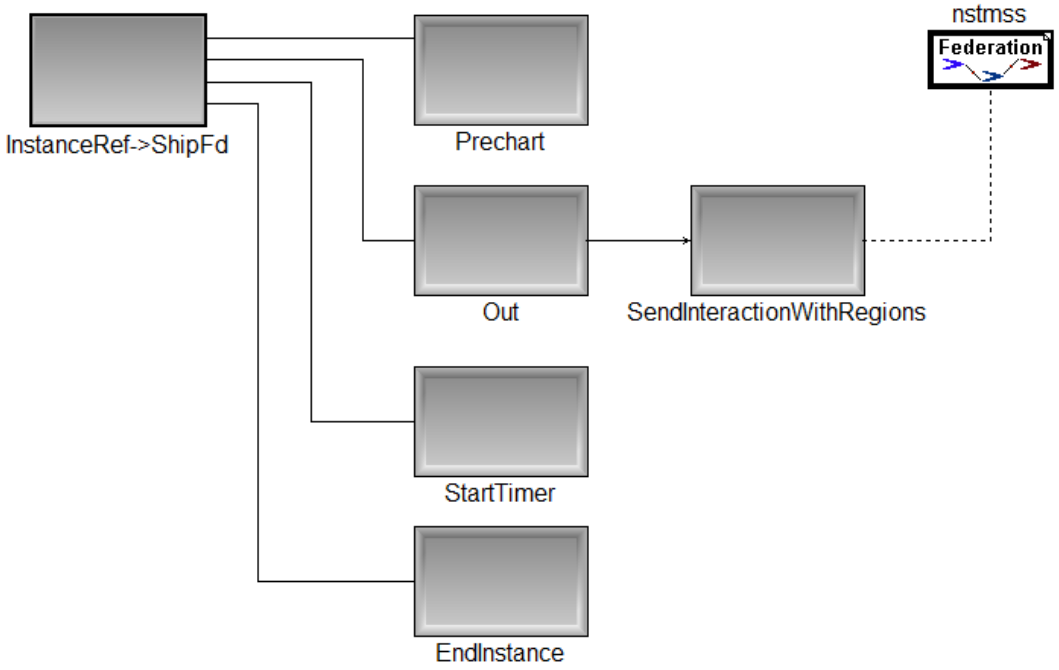


Figure 130. Send Ship Status Report Sub-chart in FAMM

Figure 131 presents the receive interactions sub-chart. As seen in the last operand of the figure, when the federate receives a `BeginToPlay` interaction, it sets its federate clock (i.e., starts a timer). The sub-chart also depicts the events between the federate and the user. Whenever an interaction is received, the federate reports the event to the user using a basic console interface.

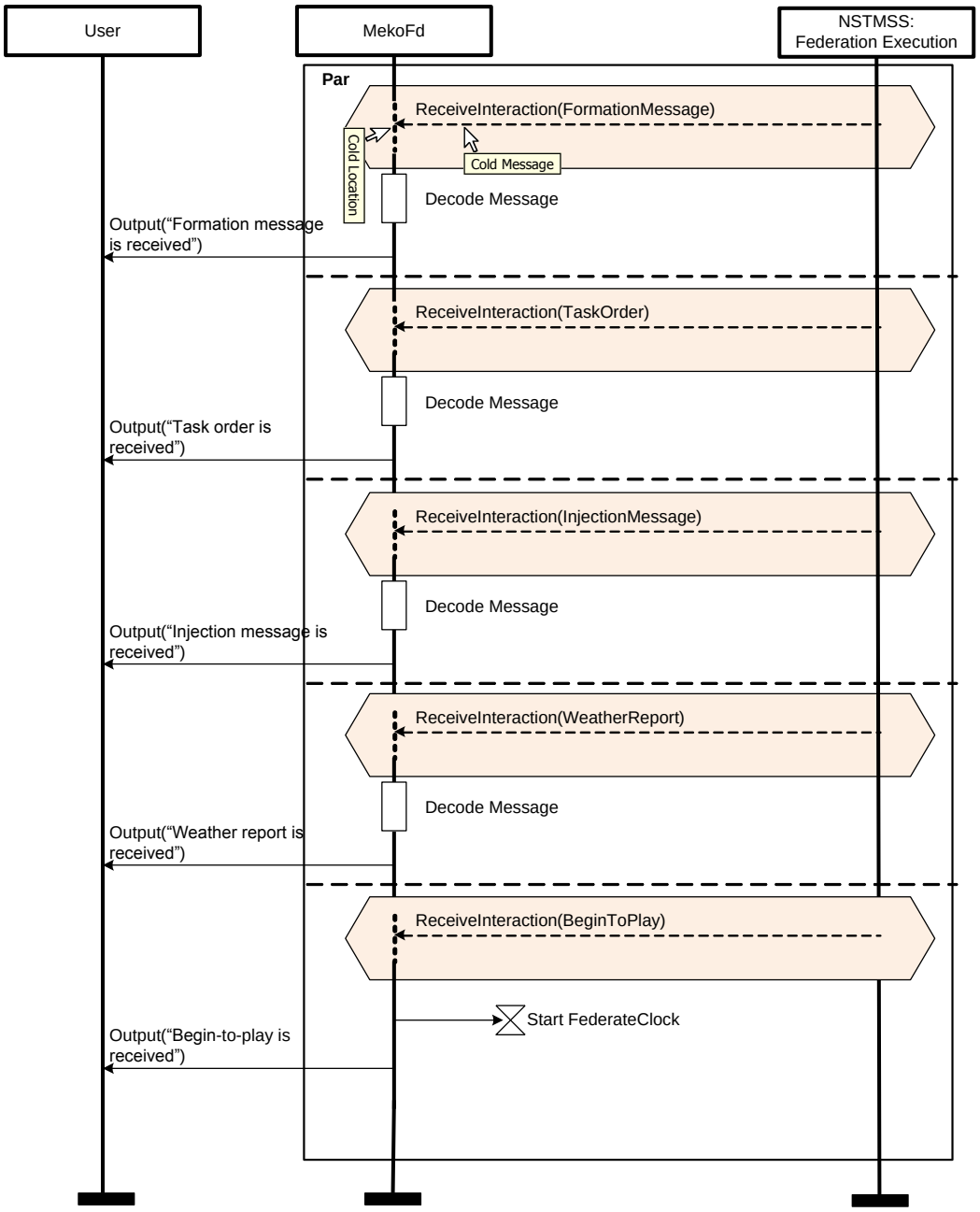


Figure 131. Receive Interactions Sub-chart

The passage of time between sending and receiving of messages is ignored. Therefore, `instantaneous messages` are used to represent the calls.

All the sub-charts are placed into the utility part of the MSC document as seen in Figure 132. `Ship_MSC` is the main LSC and is placed in the defining part.

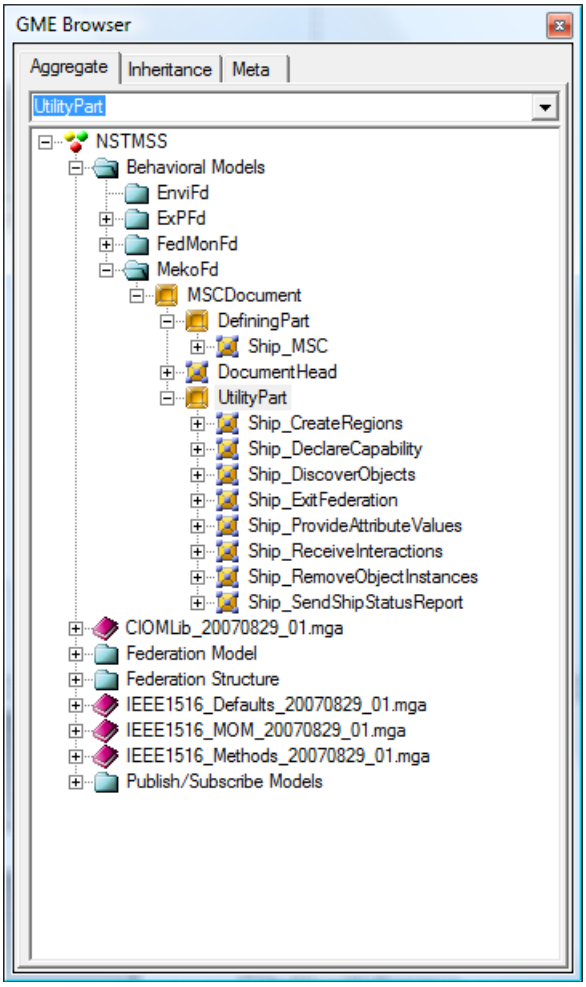


Figure 132. NSTMSS Federation Architecture in GME

B.3 Publish/Subscribe Models

The P/S models are generated by extracting the P/S interests of federates using the P/S Model Generator. The generated models help the modeler to analyze the P/S interests.

In Figure 133, the MekoFd-based P/S model is depicted. A careful analysis of the model has revealed that MekoFd never subscribes to the OTC object class where in fact it should. This was noted as a bug in the current MekoFd implementation.

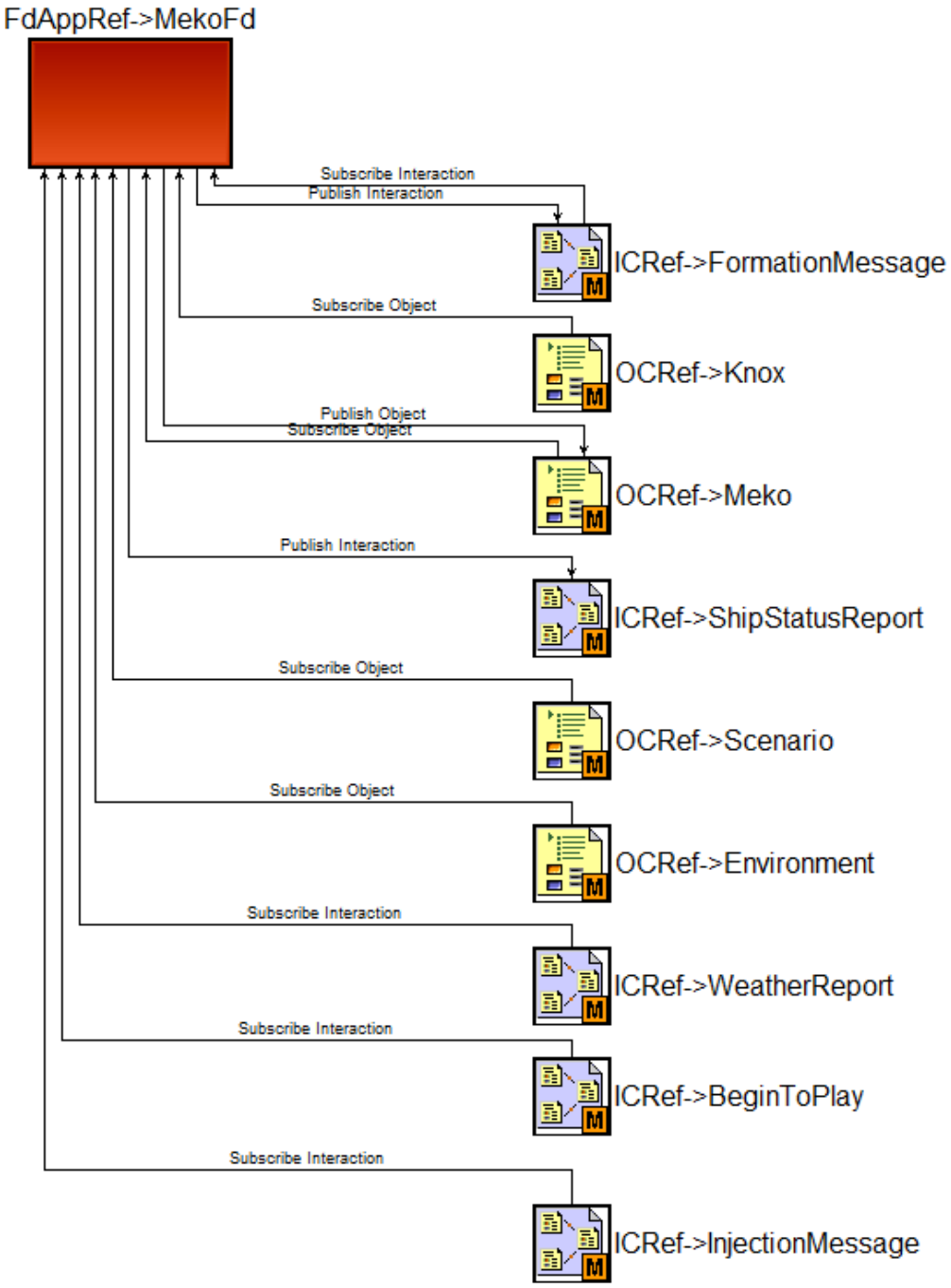


Figure 133. MekoFd-Based P/S Model

Moreover, using the class-based P/S models, in a quick view, the modeler can see, for example, which federates publish or subscribes to the ship status report. The P/S model for ship status report is depicted in Figure 134. Here, we can see that only MekoFd has the capability to send a ship status report while the ExPFd has an interest on it.

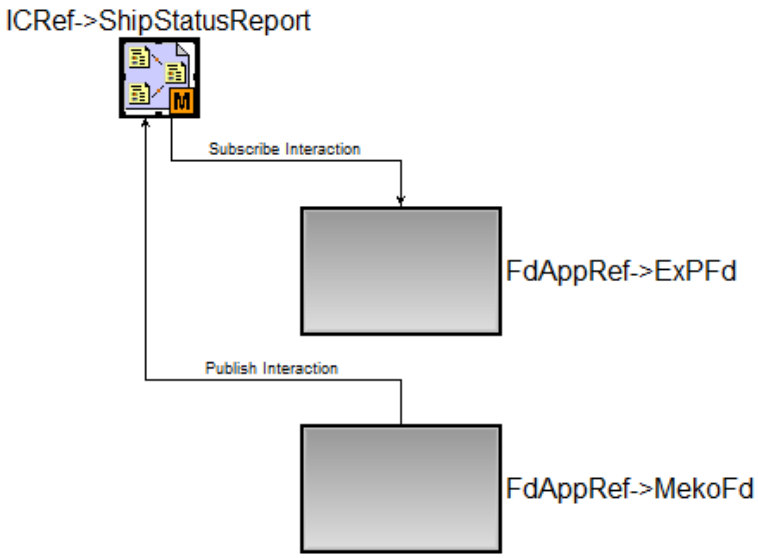


Figure 134. Ship Status Report P/S Model

B.4 Discussions

FAMM serves well as a representation language for the HLA federation architectures. Its success comes from its representation power for the federation model (object model and structure) and the behavior of the federates, providing a complete cover for the federation.

To provide an objective assessment of FAMM [54] by the modelers of NSTMSS federates; ExPFd and FedMonFd, is presented first, then the author provides a discussion.

> Using FAMM, modeling federate behavior plays an important role in terms of employing model driven approach to federation and federate development. FAMM provides a facility to achieve that. However, there are some challenges that we have encountered during this work. It should be noted that these are not result of a thorough evaluation of FAMM, but rather feedbacks derived from our experiences in using FAMM.
>
> Using IMLib provided with FAMM, instead of building a more specialized one relieved the modeling effort; however, there are some differences between the library and the standard interface specification such as the differences in the number of arguments to HLA methods. This caused frequent lookups to the IEEE

documentations and additional adaptation effort to map an HLA service to the method provided by the IMLib.

Specifying behavior in terms of LSCs was not a very hard task, since only the federates' interactions with the outside (RTI and interactive users) were concerned. However, we spent considerable time in transferring LSCs into FAMM in GME environment. Since GME environment only provides boxes and connectors, transferring an LSC into FAMM involves inserting instances, LSC keywords, HLA functions and then providing connections and enumerating them in time order. Even though modeling tasks can be comfortably handled for simple behaviors, it can quickly get complex and unmanageable for complicated behaviors. Lack of some navigational aids in GME (e.g. it only provides a rudimentary view of structural elements and does not show the depths of the charts) caused loss of time.

A full person-month of effort was spent on the modeling of each of MekoFd, ExPFd, and FedMonFd in FAMM, including the overall NSTMSS federation structure.

By its very nature model is strictly tied to the underlying metamodel. For this reason, apparently slight changes in FAMM may completely invalidate the existing modeling efforts. We believe that coping with (meta) model evolution is a serious issue for any nontrivial model driven development effort.

It would be worthwhile developing a front-end interface for utilizing FAMM's expressive power in a more user friendly way. With the help of such a tool user can synthesize live sequence charts in their usual graphical notation instead of inserting boxes and connecting them. Such a front end would then automatically generate FAMM conforming behavioral models.

Actually, there are three important issues emphasized in above text. The first issue concerns with the usability of FAMM. As discussed in Chapter 7, FAMM is not created targeting handy modeling, instead; it is created with the automatic generation idea in mind. As specified in future work, a user-friendly front-end tool

will ease the modeler work and decrease the modeling time and effort. This tool will also overcome the GME usage problems.

On the other hand, it is useful to emphasize that once the architecture is specified conforming to FAMM, the modeler harvests the goods; code generation, FDD generation, and P/S model generation. Each adds a remarkable value. Code generation plays an important role to dynamically verify the federation architecture. It shows the errors made by the modeler (e.g., modeler forgets to publish an interaction while federate tries to send it) as well as the program logic errors (e.g., federate never discovers OTC objects because of no subscription to it). Automatically generated P/S models provide the modeler a useful view of the architecture. For example, after re-engineering the MekoFd code, the P/S models generated by the P/S Model Generator showed us that MekoFd never subscribes to the OTC object.

The second issue is not about FAMM directly, but about the HLA Methods Library. The modelers point out that:

- In some methods, the “number” of parameters in IMLib and in IEEE Interface specification is different. For example, `DiscoverObjectInstance` service specification requires three parameters, which are `Object Instance Handle`, `Object Class Designator`, and `Object Instance Name`. FAMM HLA Methods library on the other hand requires only `Object Class Reference` in supplied parameters.
- In some methods, the “type” of parameters is different. For example, `GetObjectClassName` returns the name of the object. However, the counterpart in IMLib returns an object class reference.

Actually, the library provides the services based on a using-minimum-element approach where this approach can be defined as “if an FAMM element is capable of expressing more than one argument, then it is sufficient to use only that element”. If we return to the `DiscoverObjectInstance` method given as an example above, all its arguments; `Object Instance Handle`, `Object Class Designator`, and `Object Instance Name`, can be expressed by giving only one `Object Class Reference`, which refers to an object instantiated from the object class. The interpreters can obtain all the three arguments by traversing this reference. If the library had provided three references for each of them, the

modeler must have pointed all to the same object causing a more error-prone modeling.

Names of objects or interactions are generally obtained from the model element itself. Therefore, instead of providing a string value for names, a reference to the model element is provided.

The mappings that show which arguments in HLA interface specification correspond to which elements in IMLib are provided in Appendix C. Of course, the modeler spends a time to be familiar with the mappings. This can be seen as decreasing the usability. Nevertheless, note that IMLib is provided with FAMM as a convenience to the modeler. The modeler can always create his/her own library.

The last issue is about the changes in the metamodel, which can be frequently seen in non-mature metamodels. In fact, this is a serious issue as the models conforming to FAMM are increasing. Each change or update in FAMM may invalidate the current models. Accordingly, as FAMM evolves, this issue mandates that each update or change in FAMM must be backward compatible.

APPENDIX C

TRANSITION BETWEEN HLA METHODS LIBRARY AND IEEE 1516.1 FEDERATE INTERFACE SPECIFICATION

This appendix provides the mapping between the elements of the HLA methods library and the arguments specified in IEEE 1516.1 Federate Interface Specification [2]. It also gives an analysis and a quick overview of HLA Services. Tables are presented according to the HLA management groups and they are summarized from [2]. The first column presents the service number. The second column presents the service name. The third column presents the arguments of the service as specified in [2] as the last column presents the metamodel element found in the HLA methods library corresponding to the argument specified in [2]. The bold cells in tables represent the returned arguments while the white ones represent the supplied arguments.

C.1 Federation Management

Federation management includes the necessary services for the creation, dynamic control, modification, and deletion of a federation execution.

Federation management services, 24 in total, are presented in Table 8.

Table 8. Federation Management Services

NO	SERVICE	ARGUMENT	METAMODEL ELEMENT
1	Create Federation Execution	Federation Execution Name	Federation Execution Reference (FedRef) The name of the Federation element to which “FedRef” refers

Table 8 (cont'd)

		FOM Document Designator	The value of the attribute "FDD Designator" of "Federation" element to which "FedRef" refers.
		None	None
2	Destroy Federation Execution	Federation Execution Name	FedRef The name of the instance from "Federation" element
		None	None
3	Join Federation Execution	Federate Type	StringTypeReference
		Federation Execution Name	FedRef The name of the Federation element to which "FedRef" refers.
		Joined Federate Designator	Federate Application Reference (FdAppRef)
4	Resign Federation Execution	Action Argument (see the standard [2])	ResignFederationActionArgument [1] Unconditionally divest ownership of all owned instance attributes [2] Delete all object instances for which the joined federate has the delete privilege [3] Cancel all pending instance attribute ownership acquisitions [4] Perform action [2] and then action [1] [5] Perform action [3], action [2], and then action [1] [6] Perform no actions.
		None	None
5	Register Federation Synchronization Point	Sync Point Label	SynchronizationNAReference
		User-supplied Tag	TagsReference
		Optional Set of Joined Federate Designators	Set of FdAppRef
		None	None

Table 8 (cont'd)

6	Confirm Synchronization Point Registration[1] †	Sync Point Label	SynchronizationNAReference
		Registration-success indicator	IndicatorReference
		Optional Failure Reason	StringTypeReference
		None	None
7	Announce Synchronization Point †	Sync Point Label	SynchronizationNAReference
		User-supplied Tag	TagsReference
		None	None
8	Synchronization Point Achieved	Sync Point Label	SynchronizationNAReference
		None	None
9	Federation Synchronized †	Sync Point Label	SynchronizationNAReference
		None	None
10	Request Federation Save	Federation Save Label	StringTypeReference
		Optional Time Stamp	TimeStampReference
		None	None
11	Initiate Federate Save †	Federation Save String	StringTypeReference
		Optional Time Stamp	TimeStampReference
		None	None
12	Federate Save Begun	None	None
		None	None
13	Federate Save Complete	Federate Save-success indicator	IndicatorReference
		None	None
14	Federation Saved †	Federation Save-Success Indicator	IndicatorReference
		Optional Failure Reason	StringTypeReference
		None	None

[1] All RTI initiated services are denoted with a † (printer's dagger) after the service name

Table 8 (cont'd)

15	Query Federation Save Status	None	None
		None	None
16	Federation Save Status Response	List of Joined Federates and Save Status for Each	List of Federate_SaveStatus
		None	None
17	Request Federation Restore	Federation Save String	StringTypeReference
		None	None
18	Confirm Federation Restoration Request †	Federation Save String	StringTypeReference
		Request Success Indicator	IndicatorReference
		None	None
19	Federation Restore Begun †	None	None
		None	None
20	Initiate Federate Restore †	Federation Save String	StringTypeReference
		Joined Federate Designator	FdAppRef
		None	None
21	Federate Restore Complete	Federate Restore-Success Indicator	IndicatorReference
		None	None
22	Federation Restored †	Federation Restore-Success Indicator	IndicatorReference
		Optional Failure Reason	StringTypeReference
		None	None
23	Query Federation Restore Status	None	None
		None	None
24	Federation Restore Status Response †	List of Joined Federates and Restore Status for Each	List of Federate_RestoreStatus
		None	None

C.2 Declaration Management

Declaration management services, 12 in total, are presented in Table 9.

Table 9. Declaration Management Services

NO	SERVICE	ARGUMENT	METAMODEL ELEMENT
1	Publish Object Class Attributes	Object Class Designator	OCReference
		Set of Attribute Designators	Set of AttributeReference
		None	None
2	Unpublish Object Class Attributes	Object Class Designator	OCReference
		Optional Set of Attribute Designators	Set of AttributeReference
		None	None
3	Publish Interaction Class	Interaction Class Designator	ICReference
		None	None
4	Unpublish Interaction Class	Interaction Class Designator	ICReference
		None	None
5	Subscribe Object Class Attributes	Object Class Designator	OCReference
		Set of Attribute Designators	Set of AttributeReference
		Optional Passive Subscription Indicator	IndicatorReference True means active subscription, false means indicator is not present
		None	None
6	Unsubscribe Object Class Attributes	Object Class Designator	OCReference
		Optional Set of Attribute Designators	Set of AttributeReference
		None	None
7	Subscribe Interaction Class	Interaction Class Designator	ICReference
		Optional Passive Subscription Indicator	IndicatorReference
		None	None

Table 9 (cont'd)

8	Unsubscribe Interaction Class	Interaction Class Designator	ICReference
		None	None
9	Start Registration For Object Class †	Object Class Designator	OCReference
		None	None
10	Stop Registration For Object Class †	Interaction Class Designator	ICReference
		None	None
11	Turn Interactions On †	Interaction Class Designator	ICReference
		None	None
12	Turn Interactions Off †	Interaction Class Designator	ICReference
		None	None

C.3 Object Management

Object management services, 19 in total, are presented in Table 10.

Table 10. Object Management Services

NO	SERVICE	ARGUMENT	METAMODEL ELEMENT
1	Reserve Object Instance Name	Name	StringTypeReference
		None	None
2	Object Instance Name Reserved †	Name	StringTypeReference
		Reservation Success Indicator	IndicatorReference
		None	None
3	Register Object Instance	Object Class Designator	OCReference
		Optional Object Instance Name	OCReference (name of Instance of OC that this reference refers)
		Object Instance Handle	OCReference (refers to the Instance of OC)

Table 10 (cont'd)

4	Discover Object Instance †	Object Instance Handle	OCReference (refers to the Instance of OC)
		Object Class Designator	(Type of) OCReference
		Object Instance Name	OCReference (name of Instance of OC that this reference refers)
		None	None
5	Update Attribute Values	Object Instance Designator	OCReference (refers to the Instance of OC)
		Constrained Set of Attribute Designator and Value Pairs	Set of Attribute_Value
		User-supplied Tag	TagsReference
		Optional Time Stamp	TimeStampReference
		Optional Message Retraction Designator	MessageRetractionDesignatorReference
6	Reflect Attribute Values †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Constrained Set of Attribute Designator and Value Pairs	Set of Attribute_Value
		User-supplied Tag	TagsReference
		Sent Message Order Type	OrderType
		Transportation Type	TransportationRef
		Optional Time Stamp	TimeStampReference
		Optional Receive Message Order Type	OrderType
		Optional Message Retraction Designator	MessageRetractionDesignatorReference
		Optional Set of Sent Region Designators	Set of RegionReference
		None	None

Table 10 (cont'd)

7	Send Interaction	Interaction Class Designator	ICReference
		Constrained Set of Interaction Parameter Designator and Value Pairs	Set of Parameter_Value
		User-supplied Tag	TagsReference
		Optional Time Stamp	TimeStampReference
		Optional Message Retraction Designator	MessageRetractionDesignatorReference
8	Receive Interaction †	Interaction Class Designator	ICReference
		Constrained Set of Interaction Parameter Designator and Value Pairs	Set of Parameter_Value
		User-supplied Tag	TagsReference
		Sent Message Order Type	OrderType
		Transportation Type	TransportationRef
		Optional Time Stamp	TimeStampReference
		Optional Receive Message Order Type	OrderType
		Optional Message Retraction Designator	MessageRetractionDesignatorReference
		Optional Set of Sent Region Designators	Set of RegionReference
		None	None
9	Delete Object Instance	Object Instance Designator	OCReference (refers to the Instance of OC)
		User-supplied Tag	TagsReference
		Optional Time Stamp	TimeStampReference
		Optional Message Retraction Designator	MessageRetractionDesignatorReference

Table 10 (cont'd)

10	Remove Object Instance †	Object Instance Designator	OCReference (refers to the Instance of OC)
		User-supplied Tag	TagsReference
		Sent Message Order Type	OrderType
		Optional Time Stamp	TimeStampReference
		Optional Receive Message Order Type	OrderType
		Optional Message Retraction Designator	MessageRetractionDesignatorReference
		None	None
11	Local Delete Object Instance	Object Instance Designator	OCReference (refers to the Instance of OC)
		None	None
12	Change Attribute Transportation Type	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		Transportation Type	TransportationRef
		None	None
13	Change Interaction Transportation Type	Interaction Class Designator	ICReference
		Transportation Type	TransportationRef
		None	None
14	Attributes In Scope †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None
15	Attributes Out Of Scope †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None

Table 10 (cont'd)

16	Request Attribute Value Update	Object Instance Designator or Object Class Designator	OCReference
		Set of Attribute Designators	Set of AttributeReference
		User-supplied Tag	TagsReference
		None	None
17	Provide Attribute Value Update †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		User-supplied Tag	TagsReference
		None	None
18	Turn Updates On For Object Instance †	Object Instance Designator	OCReference
		Set of Attribute Designators	Set of AttributeReference
		None	None
19	Turn Updates Off For Object Instance †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None

C.4 Ownership Management

Ownership management services, 17 in total, are presented in Table 11.

Table 11. Ownership Management Services

NO	SERVICE	ARGUMENT	METAMODEL ELEMENT
1	Unconditional Attribute Ownership Divestiture	Object Instance Designator	OCReference
		Set of Attribute Designators	Set of AttributeReference
		None	None
2	Negotiated Attribute Ownership Divestiture	Object Instance Designator	OCReference
		Set of Attribute Designators	Set of AttributeReference
		User-supplied Tag	TagsReference
		None	None

Table 11 (cont'd)

3	Request Attribute Ownership Assumption †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		User-supplied Tag	TagsReference
		None	None
4	Request Divestiture Confirmation †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None
5	Confirm Divestiture	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		User-supplied Tag	TagsReference
		None	None
6	Attribute Ownership Acquisition Notification †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		User-supplied Tag	TagsReference
		None	None
7	Attribute Ownership Acquisition	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		User-supplied Tag	TagsReference
		None	None
8	Attribute Ownership Acquisition If Available	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None
9	Attribute Ownership Unavailable †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None

Table 11 (cont'd)

10	Request Attribute Ownership Release	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		User-supplied Tag	TagsReference
		None	None
11	Attribute Ownership Divestiture If Wanted	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators (for which the joined federate is willing to divest ownership)	Set of AttributeReference
		Set of Attribute Designators (for which ownership has actually been divested)	Set of AttributeReference
12	Cancel Negotiated Attribute Ownership Divestiture	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None
13	Cancel Attribute Ownership Acquisition	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None
14	Confirm Attribute Ownership Acquisition Cancellation †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		None	None
15	Query Attribute Ownership	Object Instance Designator	OCReference (refers to the Instance of OC)
		Attribute Designator	AttributeReference
		None	None
16	Inform Attribute Ownership †	Object Instance Designator	OCReference (refers to the Instance of OC)
		Attribute Designator	AttributeReference

		Ownership Designator (could be a joined federate, RTI, or unowned)	Either FdAppRef (i.e., Joined Federate) or FedRef (i.e., RTI) If both are null then it means an indication that the instance attribute is available for acquisition (i.e., un-owned)
		None	None
17	Is Attribute Owned By Federate	Object Instance Designator	OCReference (refers to the Instance of OC)
		Attribute Designator	AttributeReference
		Instance Attribute Ownership Indicator	IndicatorReference

C.5 Time Management

Time management services, 23 in total, are presented in Table 12.

Table 12. Time Management Services

NO	SERVICE	ARGUMENT	METAMODEL ELEMENT
1	Enable Time Regulation	Lookahead	LookaheadReference
		None	None
2	Time Regulation Enabled †	Current Logical Time of the Joined Federate	TimeStampReference
		None	None
3	Disable Time Regulation	None	None
		None	None
4	Enable Time Constrained	None	None
		None	None
5	Time Constrained Enabled †	Current Logical Time of the Joined Federate	TimeStampReference
		None	None
6	Disable Time Constrained	None	None
		None	None

Table 12 (cont'd)

7	Time Advance Request	Logical Time	TimeStampReference
		None	None
8	Time Advance Request Available	Logical Time	TimeStampReference
		None	None
9	Next Message Request	Logical Time	TimeStampReference
		None	None
10	Next Message Request Available	Logical Time	TimeStampReference
		None	None
11	Flush Queue Request	Logical Time	TimeStampReference
		None	None
12	Time Advance Grant †	Logical Time	TimeStampReference
		None	None
13	Enable Asynchronous Delivery	None	None
		None	None
14	Disable Asynchronous Delivery	None	None
		None	None
15	Query GALT	None	None
		GALT Definition Indicator	IndicatorReference
		Optional Current Value of Invoking Joined Federate's GALT	TimeStampReference
16	Query Logical Time	None	None
		The Invoking Joined Federate's Current Logical Time	TimeStampReference
17	Query LITS	None	None
		LITS Definition Indicator	IndicatorReference
		Optional Current Value of Invoking Joined Federate's LITS	TimeStampReference
18	Modify Lookahead	Requested Lookahead	LookaheadReference
		None	None

Table 12 (cont'd)

19	Query Lookahead	None	None
		The Invoking Joined Federate's Current Actual Lookahead	LookaheadReference
20	Retract	Message Retraction Designator	MessageRetractionDesignatorReference
		None	None
21	Request Retraction †	Message Retraction Designator	MessageRetractionDesignatorReference
		None	None
22	Change Attribute Order Type	Object Instance Designator	OCReference (refers to the Instance of OC)
		Set of Attribute Designators	Set of AttributeReference
		Order Type	OrderType
		None	None
23	Change Interaction Order Type	Interaction Class Designator	ICReference
		Order Type	OrderType
		None	None

C.6 Data Distribution Management

Data distribution management services, 12 in total, are presented in Table 13.

Table 13. Data Distribution Management Services

NO	SERVICE	ARGUMENT	METAMODEL ELEMENT
1	Create Region	Set of dimension designators	Set of DimensionRef
		Region Designator	RegionReference
2	Commit Region Modifications	Set of region designators	Set of RegionReference
		None	None
3	Delete Region	Region designator	RegionReference
		None	None

Table 13 (cont'd)

4	Register Object Instance With Regions	Object Class Designator	OCReference
		Collection of attribute designator set and region designator set pairs	Collection of "Set of Attributes" and "Set of RegionReference"
		Optional Object instance Name	(Name of) OCReference
		Object Instance Designator	OCReference (refers to the Instance of OC)
5	Associate Regions For Updates	Object Instance Designator	OCReference (refers to the Instance of OC)
		Collection of attribute designator set and region designator set pairs	Collection of "Set of Attributes" and "Set of RegionReference"
		None	None
6	Unassociate Regions For Updates	Object Instance Designator	OCReference (refers to the Instance of OC)
		Collection of attribute designator set and region designator set pairs	Collection of "Set of Attributes" and "Set of RegionReference"
		None	None
7	Subscribe Object Class Attributes With Regions	Object Class Designator	OCReference
		Collection of attribute designator set and region designator set pairs	Collection of "Set of Attributes" and "Set of RegionReference"
		optional passive subscription indicator	Indicator or IndicatorReference
		None	None
8	Unsubscribe Object Class Attributes With Regions	Object Class Designator	OCReference
		Collection of attribute designator set and region designator set pairs	Collection of "Set of Attributes" and "Set of RegionReference"
		None	None
9	Subscribe Interaction Class With Regions	Interaction class designator	ICReference
		set of region designators	Set of RegionReference
		optional passive subscription indicator	IndicatorReference
		None	None

Table 13 (cont'd)

10	Unsubscribe Interaction Class With Regions	Interaction class designator	ICReference
		set of region designators	Set of RegionReference
		None	None
11	Send Interaction With Regions	Interaction class designator	ICReference
		Constrained set of parameter designator and value pairs	Set of Parameter_Value
		Set of region designators	Set of RegionReference
		user-supplied tag	TagsReference
		Optional time stamp	TimeStampReference
		Optional Message Retraction Designator	MessageRetractionDesignatorReference
12	Request Attribute Value Update With Regions	Object Class Designator	OCReference
		Collection of attribute designator set and region designator set pairs	Collection of "Set of Attributes" and "Set of RegionReference"
		user-supplied tag	TagsReference
		None	None

C.7 Support Services

Support services, 39 in total, are presented in Table 14.

Table 14. Support Services

NO	SERVICE	ARGUMENT	METAMODEL ELEMENT
1	Get Object Class Handle	Object Class Name	OCReference
		Object Class Handle	OCReference
2	Get Object Class Name	Object Class Handle	OCReference
		Object Class Name	OCReference
3	Get Attribute Handle	Object Class Handle	OCReference
		Class Attribute Name	AttributeReference
		Class Attribute Handle	AttributeReference

Table 14 (cont'd)

4	Get Attribute Name	Object Class Handle	OCReference
		Class Attribute Handle	AttributeReference
		Class Attribute Name	AttributeReference
5	Get Interaction Class Handle	Interaction Class Name	ICReference
		Interaction Class Handle	ICReference
6	Get Interaction Class Name	Interaction Class Handle	ICReference
		Interaction Class Name	ICReference
7	Get Parameter Handle	Interaction Class Handle	ICReference
		Parameter Name	ParameterReference
		Parameter Handle	ParameterReference
8	Get Parameter Name	Interaction Class Handle	ICReference
		Parameter Handle	ParameterReference
		Parameter Name	ParameterReference
9	Get Object Instance Handle	Object Instance Name	OCReference (refers to the Instance of OC)
		Object Instance Handle	OCReference (refers to the Instance of OC)
10	Get Object Instance Name	Object Instance Handle	OCReference
		Object Instance Name	OCReference
11	Get Dimension Handle	Dimension Name	DimensionRef
		Dimension Handle	DimensionRef
12	Get Dimension Name	Dimension Handle	DimensionRef
		Dimension Name	DimensionRef
13	Get Dimension Upper Bound	Dimension Handle	DimensionRef
		Dimension Upper Bound	DimensionRef (upperBound attribute of Dimension)
14	Get Available Dimensions For Class Attribute	Object Class Handle	OCReference
		Class Attribute Name	AttributeReference
		A Set of Dimension Handles	Set of DimensionRef

Table 14 (cont'd)

15	Get Known Object Class Handle	Object Instance Handle	OCReference
		Object Class Handle	OCReference
16	Get Available Dimensions For Interaction Class	Interaction Class Handle	ICReference
		A Set of Dimension Handles	Set of DimensionRef
17	Get Transportation Type	Transportation Name	TransportationRef
		Transportation Type	TransportationRef
18	Get Transportation Name	Transportation Name	TransportationRef
		Transportation Type	TransportationRef
19	Get Order Type	Order Name	OrderType
		Order Type	OrderType
20	Get Order Name	Order Name	OrderType
		Order Type	OrderType
21	Enable Object Class Relevance Advisory Switch	None	None
		None	None
22	Disable Object Class Relevance Advisory Switch	None	None
		None	None
23	Enable Attribute Relevance Advisory Switch	None	None
		None	None
24	Disable Attribute Relevance Advisory Switch	None	None
		None	None
25	Enable Attribute Scope Advisory Switch	None	None
		None	None
26	Disable Attribute Scope Advisory Switch	None	None
		None	None
27	Enable Interaction Relevance Advisory Switch	None	None
		None	None
28	Disable Interaction Relevance Advisory Switch	None	None
		None	None

Table 14 (cont'd)

29	Get Dimension Handle Set	Region Handle	RegionReference
		A Set of Dimensions	Set of DimensionRef
30	Get Range Bounds	Region Handle	RegionReference
		Dimension Handle	DimensionRef
		Range Lower Bound	NumericType
		Range Upper Bound	NumericType
31	Set Range Bounds	Region Handle	RegionReference
		Dimension Handle	DimensionRef
		Range Lower Bound	NumericType
		Range Upper Bound	NumericType
32	Normalize Federate Handle	Federate Handle	DimensionRef
		Normalized Value	NumericType
33	Normalize Service Group	Service Group Indicator	DimensionRef
		Normalized Value	NumericType
34	Initialize RTI	Set of Strings	Set of StringTypeReference
		Set of Strings	Set of StringTypeReference
35	Finalize RTI	None	None
		None	None
36	Evoke Callback	Minimum Amount of Wall-clock time	TimeStampReference
		Pending Callback Indicator	IndicatorReference
37	Evoke Multiple Callbacks	Min.Amount of Wall-clock time	TimeStampReference
		Maximum Amount of clock time	TimeStampReference
		Pending Callback Indicator	IndicatorReference
38	Enable Callbacks	None	None
		None	None
39	Disable Callbacks	None	None
		None	None

APPENDIX D

HLA ARGUMENTS

This appendix provides the details of the arguments specified in the HLA Services Metamodel for the library developers.

Table 15 presents the arguments specified in [2], while Table 16 presents the additional arguments created for DMSO RTI NG 1.3v6.

Table 15. Arguments for IEEE Interface Specification

NO	ARGUMENT	EXPLANATION	EXAMPLE FOR USAGE
1	Federation	It represents a federation or federation execution (instance of federation).	Not used.
2	FedRef	It is a reference to a Federation.	It can be used to extract the name of the federation execution and the FDD path and name.
3	FederateApplication	It represents a federate application or a joined federate.	Not used.
4	FdAppRef	It is a reference to a FederateApplication.	It can be used to represent the joined federate designators.
5	Region	It represents a region	Not used.
6	RegionReference	It is a reference to a Region.	It can be used to represent the region designators.
7	ObjectClass	It represents an HLA object class or an object.	Not used.
8	OCReference	It is a reference to an ObjectClass.	It can be used to represent both an object instance designator and an object class designator.

Table 15 (cont'd)

9	AttributeReference	It is a reference to an object class attribute.	It can be used to represent the attribute designators.
10	ICReference	It is a reference to an interaction class.	It can be used to represent an interaction class designator.
11	DimensionRef	It is a reference to a dimension in object model.	It can be used to represent a dimension handle and name.
12	TimeStampReference	It is a reference to a time stamp in object model.	It can be used to represent a pointer to a time stamp.
13	SynchronizationNAReference	It is a reference to a synchronizationNA in object model.	It can be used to represent a pointer to a synchronization point label.
14	MessageRetractionDesignator	It represents a message retraction designator.	Not used.
15	MessageRetractionDesignator Reference	It is a reference to a message retraction designator.	It can be used to represent a message retraction designator.
16	TransportationRef	It is a reference to transportation in object model.	It can be used to represent a transportation type.
17	TagsReference	It is a reference to a tag in object model.	It can be used to represent a user-type tag.
18	LookaheadReference	It is a reference to a lookahead in object model.	It can be used to represent a lookahead.
19	ParameterReference	It is a reference to an HLA interaction parameter specified in object model.	It can be used to represent a parameter.
20	Collection	It represents a collection of two sets.	For example, it is used for collection of attribute designator set and region designator set pairs.
21	List	It represents a list of federate-save status or federate-restore status.	For example, it is used for list of Federate_SaveStatus.
22	Set	It represents a set of region references, or parameter and value pairs, or federate application references, or attribute and value pairs, or attribute references, or dimension references, or string types, or string type references.	For example, it is used for set of attribute designators.

Table 15 (cont'd)

23	OrderType	It represents an order type. It is either receive or time stamp order.	For example, it is used to represent a sent message order type.
24	ResignFederationActionArgument	It represents the action arguments while resigning a federation execution.	It is used in `ResignFederationExecution`.
25	NumericType	It represents numeric type arguments.	It is used for bounds and normalized values.
26	NumericTypeReference	It is a reference to a numeric type.	Not used.
27	StringType	It represents a string type argument.	Not used.
28	StringTypeReference	It is a reference to a string type.	It is used for failure reasons, labels, strings, names, and a federate type.
29	Indicator	It represents a Boolean type.	Not used.
30	IndicatorReference	It is a reference to an indicator.	It is used for indicators such as passive subscription indicator.

Table 16. Additional Arguments for DMSO RTI NG 1.3v6

NO	ARGUMENT	EXPLANATION	EXAMPLE FOR USAGE
1	DMSOActionArgument	It represents the action arguments while resigning a federation execution.	It is used in DMSO `ResignFederationExecution`
2	DMSOFederateAmbassador	It is used for the representation of Federate Ambassador.	It is used in DMSO `JoinFederationExecution`
3	DMSORoutingSpace	It represents a routing space.	It is used in DMSO `CreateRegion`
4	DMSORoutingSpaceReference	It represents a reference to `DMSORoutingSpace`	In DMSO `GetDimensionName`

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Topçu Okan

Nationality: Turkish (TC)

Date and Place of Birth: 19 May 1971, Edirne

Marital Status: Married

Phone: +90 312 260 86 18

Email: okantopcu@gmail.com

EDUCATION

Degree	Institution	Year of Graduation
MS	METU Computer Engineering	1999
BS	Naval Academy	1993
High School	Naval High School	1989

WORK EXPERIENCE

Year	Place	Enrollment
2003-Present	Turkish General Staff	Chief of System Administration Branch
2002-2003	Canada Defense Research and Development Center	NATO Exchange Officer
2000-2002	Turkish General Staff	Network and System Administrator
1996-1999	Naval Forces Headquarter / METU	MS Student
1993-1996	Navy	Branch Officer in Various Naval Ships

FOREIGN LANGUAGES

Advanced English

PUBLICATIONS

Journals

1 Topçu Okan, Adak Mehmet, and Oğuztüzün Halit, “A Metamodel for Federation Architectures”, ACM Transactions on Modeling and Computer Simulation (to appear).

2 Topçu Okan, Adak Mehmet, and Oğuztüzün Halit, “Metamodeling Live Sequence Charts for Code Generation”, submitted, 2007.

3 Adak Mehmet, Topçu Okan, and Oğuztüzün Halit, “Model-based Code Generation for HLA Federates”, submitted, 2007.

4 Adak Mehmet, Topçu Okan, and Oğuztüzün Halit, “Code Generation for Live Sequence Charts and Message Sequence Charts”, submitted, 2007.

5 Topçu Okan and Oğuztüzün Halit, “Developing an HLA Based Naval Maneuvering Simulation”, in Naval Engineers Journal by American Society of Naval Engineers (ASNE), vol.117 no.1, pp. 23-40, Winter 2005.

International Conferences

1 Molla A., Sarıoğlu K., Topçu Okan, Adak M., and Oğuztüzün Halit, “Federation Architecture Modeling: A Case Study with NSTMSS”, 07F-SIW-052, In the Proceedings of 2007 Fall Simulation Interoperability Workshop (SIW), Orlando, Florida, USA, September 16-21, 2007.

2 Topçu Okan, Oğuztüzün Halit, and Hazen G. M., “Towards a UML Profile for HLA Federation Design, Part II”, in the Proceedings of Summer Computer Simulation Conference (SCSC-2003), pp. 874-879, Montreal, Canada, July 19-24, 2003.

3 Topçu Okan and Oğuztüzün Halit, “Towards a UML Extension for HLA Federation Design”, in the Proceedings of 2nd Conference on Simulation Methods and Applications (CSMA-2000), pp 204-213, Orlando, FL, USA, Oct. 29-31, 2000.

4 Topçu Okan and Oğuztüzün Halit, “Design of a Naval Surface Tactical Maneuvering Simulation System”, in the Proceedings of 31st Summer Computer and Simulation Conference (SCSC-1999), pp 319-324, Chicago, Illinois, USA, July 11-15, 1999.

Technical Reports / Memoranda

1 Topçu Okan and Oğuztüzün Halit, “A Metamodel for Live Sequence Charts and Message Sequence Charts”. Technical Report (METU-CENG-TR-2007-3), Middle East Technical University, May 2007.

2 Topçu Okan, “Naval Surface Tactical Maneuvering Simulation System Technical Report (Draft)”, manuscript, 2007.

3 Topçu Okan, “Development, Representation, and Validation of Conceptual Models in Distributed Simulation”, Defence R&D Canada – Atlantic (DRDC Atlantic) Technical Memorandum (TM 2003-142), Halifax, NS, Canada, February 2004.

4 Topçu Okan, “Review of Verification and Validation Methods in Simulation: Literature Survey, Concepts, and Definitions”, Defence R&D Canada – Atlantic (DRDC Atlantic) Technical Memorandum, TM 2003-055, Halifax, NS, Canada, April 2003.

Thesis

1 Topçu Okan, “Naval Surface Tactical Maneuvering Simulation System”, MSc Thesis, The Department of Computer Engineering, The Graduate School of Natural and Applied Sciences, Middle East Technical University (METU), Ankara, Turkey, December 1999.

HOBBIES

Sailing, Scuba Diving, Swimming