MODEL-BASED CODE GENERATION FOR THE HIGH LEVEL ARCHITECTURE FEDERATES

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OF

MIDDLE EAST TECHNICAL UNIVERSITY

BY

BÜLENT MEHMET ADAK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

DECEMBER 2007

Approval of the thesis:

MODEL-BASED CODE GENERATION
FOR THE HIGH LEVEL ARCHITECTURE FEDERATES

Submitted by **BÜLENT MEHMET ADAK** 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. Hakkı Toroslu
Computer Engineering Dept., METU _____

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

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

Assoc. Prof. Dr. İlyas Çiçekli
Computer Engineering Dept., Bilkent University _____

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

Date: _____

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

Name, Last name: Bülent Mehmet Adak

Signature :

ABSTRACT

MODEL-BASED CODE GENERATION FOR THE HIGH LEVEL ARCHITECTURE FEDERATES

Adak, Bülent Mehmet

Ph.D., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Halit Oğuztüzün

December 2007, 261 pages

We tackle the problem of automated code generation for a High Level Architecture (HLA)-compliant federate application, given a model of the federation architecture including the federate's behavior model. The behavior model is based on Live Sequence Charts (LSCs), adopted as the behavioral specification formalism in the Federation Architecture Metamodel (FAMM). The FAMM is constructed conforming to metaGME, the meta-metamodel offered by Generic Modeling Environment (GME). FAMM serves as a formal language for describing federation architectures. We present a code generator that generates Java/AspectJ code directly from a federation architecture model. An objective is to help verify a federation architecture by testing it early in the development lifecycle. Another objective is to help developers construct complete federate applications. Our approach to achieve these objectives is aspect-oriented in that the code generated from the LSC in conjunction with the Federation Object Model (FOM) serves as the base code on which the computation logic is weaved as an aspect.

Keywords: Code Generation, High Level Architecture, Live Sequence Charts, Aspect Oriented Programming, Model-Driven Engineering

ÖZ

YÜKSEK SEVİYE MİMARİ FEDERELERİ İÇİN MODEL TABANLI KOD ÜRETİMİ

Adak, Bülent Mehmet
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Halit Oğuztüzün

Aralık 2007, 261 sayfa

Biz, federenin davranış modelini de içeren federasyon mimari modeli verilen, yüksek seviye mimari (HLA) uyumlu bir federe uygulaması için kod üretiminin otomasyonu problemi ile uğraşmaktayız. Federe davranış modeli, Federasyon Mimari Meta-modeli (FAMM) içinde davranışsal spesifikasyon biçimselleşmesi için adapte edilmiş Canlı Sıralama Çizgelerini baz almaktadır. FAMM meta-GME meta-metamodeline uyumlu olarak inşa edilmiştir. Meta-GME, Jenerik Modelleme Ortamı (GME) tarafından ortaya atılmış bir meta-metamodeldir. FAMM federasyon mimarilerinin betimlenmesi için biçimsel bir dil sunmaktadır. Biz federasyon mimari modelinden direk olarak Java/AspectJ kodu üreten bir kod üreteci sunmaktayız. Bu çalışmanın bir amacı, bir federasyon mimarisini geliştirme yaşam döngüsünün henüz başında test ederek doğrulamaya yardım etmektir. Bir diğer amaç da komple federe uygulamaları oluşturmada geliştiricilere yardım etmektir. Bu amaçlara ulaşmada bizim yaklaşımımız ilgiye odaklı yaklaşımdır. Bu yaklaşımda, Federe Obje Modeli (FOM) ile bütünleşik LSC'den üretilen kod, hesaplama mantığı üzerine bir ilgi olarak örülen, taban kodu olmaktadır.

Anahtar Kelimeler: Kod Üretimi, Yüksek Seviye Mimari Simülasyon, Canlı Sıralama Çizgeleri, İlgi Odaklı Programlama, Model-Güdümlü Mühendislik

To My Family

ACKNOWLEDGEMENTS

I especially thanks to, my supervisor, Assoc. Prof. Dr. Halit Oğuztüzün for supervising me, providing resources, subjects, also offering direction and insightful critism.

Thanks to Assoc. Prof.Dr. Cem Bozşahin and Assoc. Prof.Dr. İlyas Çiçekli for their valuable supervision during my thesis.

I would like to thank to collaborative work with the fellow PhD student Okan Topçu, who developed the Federation Architecture Metamodel. Also thanks to Osman Efe, Kaan Sarıoğlu, Ayhan Molla, Gürkan Özhan, and Deniz Çetinkaya for many supporting comments. I thank Aselsan Inc. for providing a working environment conducive to research.

Finally, I would like to thank my wife and my parents, for their support against all the difficulties that I met.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ACM	Association for Computing Machinery
ADT	Abstract Data Type
ALT	Alternative
AOP	Aspect Oriented Programming
API	Application Programmer's Interface
ATM	Automated Teller Machine
BMM	Behavioral Metamodel
CENG	Computer Engineering
DDM	Data Distribution Management
DDSOS	Dynamic Distributed Service-Oriented Simulation
DMSO	Defense Modeling and Simulation Office
DSL	Domain Specific Language
DSOCS	Dynamic Service-Oriented Collaboration Simulation
EXC	Exception
FAM	Federation Architecture Model
FAMM	Federation Architecture Metamodel
FDD	FOM Document Data
FED	Federation Execution Data
FedMonFd	Federation Monitor Federate
FIFO	First in First out
FOM	Federation Object Model
FSMM	Federation Structure Metamodel
GME	Generic Modeling Environment
HOMM	HLA Object Metamodel
HFMM	HLA Federation Metamodel
HLA	High Level Architecture
HMLib	HLA Methods Library
HMSC	High Level MSC
HSMM	HLA Services Metamodel
IEEE	Institute of Electrical and Electronic Engineers
ITU-T	International Telecommunication Union

ISCIS	International Symposium on Computer and Information Sciences
JDK	Java Development Kit
JSS	Journal of Systems and Software
JVM	Java Virtual Machine
LOC	Lines of code
LSC	Live Sequence Chart
LscRTILib	LSC RTI Library
MSC	Message Sequence Chart
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MetaGME	GME Metamodel
MIC	Model Integrated Computing
METU	Middle East Technical University
MOF	Meta Object Facility
MOM	Management Object Model
M&S	Modeling and Simulation
NSTMSS	Naval Surface Tactical Maneuvering Simulation System
OMG	Object Modeling Group
OMT	Object Model Template
OPT	Option
OOAD	Object Oriented Analysis and Design
PAR	Parallel
PIM	Platform Independent Model
PSML-S	Process Specification and Modeling Language for Services
PSM	Platform Specific Model
pRTI	Pitch RTI
RTI	Runtime Infrastructure
RTILib	RTI Library
SDK	Software Development Kit
SEQ	Sequential
SIW	Simulation Interoperability Workshop
SOA	Service Oriented Architecture
SOM	Simulation Object Model
SoSym	Software and System Modeling
STMS	Strait Traffic Monitoring Simulation
TSE	Transactions on Software Engineering

UML	Unified Modeling Language
UYMS	Ulusal Yazılım Mühendisliği Sempozyumu
XML	Extendible Markup Language
XMSF	Extendible Modeling and Simulation Framework

CHAPTER I

INTRODUCTION

This chapter introduces the motivation and scope of the study, presents the context of the generator and then outlines the organization of the thesis.

1.1. Motivation and Scope

We investigate the applicability of model-based code generation to HLA-compliant federation development. This approach is promising in regards to rapid prototyping of federation designs and semi-automated construction of federate applications. First and foremost, this requires the availability of suitable models, behavioral models in particular. One of the main objectives of modeling is to provide a representation appropriate to identify, analyze and design the systems. The system representation must be clear-cut adequate 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 specification. In our preceding work [Topçu et al. 2007] we introduced a comprehensive metamodel for the description of federation architectures. A salient feature of the metamodel, FAMM (Federation Architecture Metamodel) adopts (Live Sequence Charts) LSCs to specify the communication behavior of federates.

The model of a particular federation architecture constitutes the input to the code generation process. The output is obtained one member federate at a time: An HLA-compliant federate application code that is capable of generating any sequence of communications conforming to the specification, but that lacks the logic to carry out the required computations. Thus, if it were to run as is, it would exhibit a randomized communication pattern conforming to the specification as long as it did not rely on any correctly computed value. To turn it into an appropriate application one would need to provide the algorithms to compute the correct values. In this dissertation we introduce an automated tool that carries out this automated code generation process.

Note that the generated federate application code is HLA-compliant in the sense that its interaction with the RTI complies with the Federate Interface Specification and the Federation Object Model. Further compliance of the federate, as a federation member, with

the HLA Rules can be guaranteed by the designer, who specifies the federate's behavior using FAMM (in particular, Behavioral Metamodel - BMM). The developer, who is providing the computation logic, cannot break the federate's compliance unless his calculations disrupt the control flow within the federate.

The generated code consists of federate base code and computation aspect code, where the latter is weaved onto the former. The federate base code handles the communication between the federate and the RTI, and the computation aspect code allows the user to code the federate's algorithms for computation. To produce an intended federate application, the developer should edit the computation aspect. By providing a simple computation logic (e.g. line-of-sight calculation for radars) the user can obtain prototype federates, thus a prototype federation. This should serve for the verification of the federation architecture. By providing the sophisticated logic as required by the end product (e.g. finite element method calculations for radars) the user can proceed with actual federate application development.

1.2. Context of the Generator

Adopting the Model Driven Engineering (MDE) approach, the system development process can be viewed as a sequence of model transformations [Bezivin 2005]. From this point of view, HLA-based distributed simulation development essentially involves the conceptual model, the federation architecture model, the detailed design model, and the federation in executable form, as illustrated in Figure 1.1. Each layer of models reflects a particular level of abstraction. The conceptual model layer deals with the problem domain entities (for example, a ship); federation architecture deals with concepts of HLA (for example, a ship federate), and the detailed design model layer deals with software objects within federate applications (for example, a component diagram for the ship's hydrodynamic model computations). Finally, we have the federation in some executable form, possibly in some programming language (for example, implementation of the hydrodynamics as a software unit within some federate).

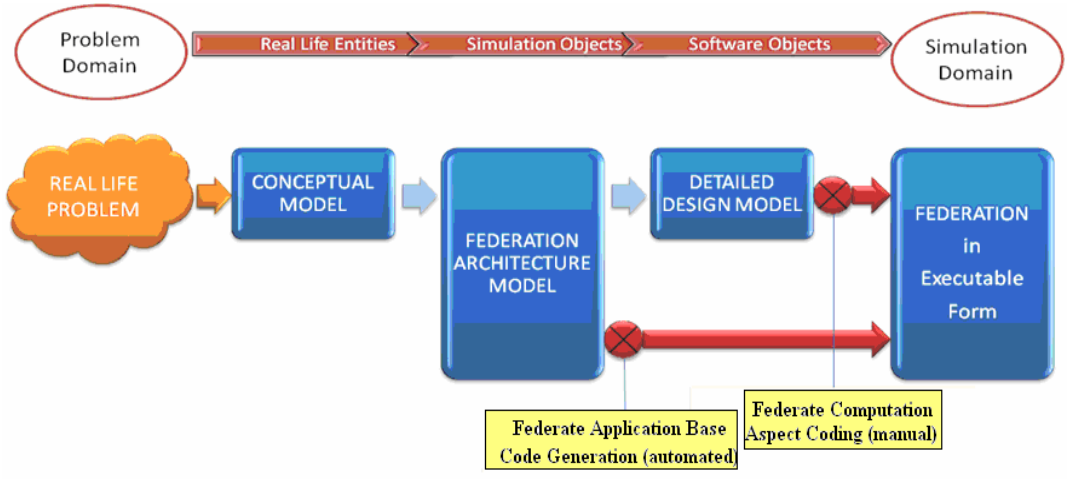


Figure 1.1 Development Methodology for HLA-Based Distributed Simulations, adapted from [Topçu et al. 2007]

"Federate Application Base Code Generation" step is completely automated by the present work. "Federate Computation Aspect Coding" step of the process is to be completed by the developer, based on the Detailed Design Model for each federate. The code generator, without the benefit of a detailed design model for each federate, can only provide the developer with a preliminary computation aspect, which he must edit to introduce code based on the detailed design of the federate.

A federation developer can utilize our work in the following manner:

i. Model the federation architecture, including the behaviors of the new member federates.
ii. Automatically generate code for each new member federate.
iii. Edit the computation aspect in the federate code.
iv. Automatically generate FOM Document Data (FDD).
v. Compile the new federates and run the federation.

1.3 Organization of the Thesis

The preceding sections of this chapter introduce the motivation and scope of the study and present the context of the generator. The remaining chapters are broken down as follows:

- Chapter 2 provides related background information to understand the following chapter.
- Chapter 3 explains how to generate code from a federation architecture model.
- Chapter 4 and Chapter 5 describe code generation from live sequence charts.
- Chapter 6 presents an extensive case-study for a real simulation application.

- Chapter 7 outlines the conclusions reached as a result of this research as well as presenting the way ahead.
- Appendix A explains model patterns and their related codes in the code generation.
- Appendix B gives details of the intermediate form and its generation from input model in the code generation process.
- Appendix C describes the java code generation from the intermediate form in the code generation process.
- Appendix D presents an example to show how the code is generation from model to code in the process.
- Appendix E introduces LSC examples and their code equivalency from the literature for every construct defined in the FAMM.
- Appendix F and Appendix G explain how a domain-specific data model especially HLA methods are integrated with LSC.
- Appendix H is a user guide that describes how the generator is used for code generation.

CHAPTER II

BACKGROUND

In this chapter, background information from the related literature is summarized to help the reader follow the developments in the subsequent chapters more easily. First, generative software development terminology, which is the basis of automated software generation, is discussed. Second, model-driven architecture and model-driven engineering concepts are summarized. In the context of model-driven development, a code generator offers a special kind of transformation from a platform specific model to source code. Third, High Level Architecture which is the domain of our code generator is described. Fourth, Federation Architecture Metamodel (FAMM), which is the metamodel on which our code generator is based, is introduced. Fifth, Generic Modeling Environment tool by which metamodel and input models are constructed is discussed. Sixth, aspect-oriented programming which is the primary approach for the separation of communication behavior and computation concerns of the generated code is outlined. Finally, elementary code generation terminology used in the thesis is defined.

2.1. Generative Software Development

Generative software development [Czarnecki 2005] focuses on automating the creation of software. A required piece of software can be automatically generated from a specification written in some textual or graphical domain-specific language (DSL). A key concept in generative software development is that of a mapping between problem space and solution space, which is also referred to as a generative domain model. Problem space is a set of domain-specific abstractions that can be used to specify the desired system. The solution space, on the other hand, consists of implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstractions from the problem space. The mapping (see Figure 2.1) between these two spaces takes a specification and yields the corresponding implementation. In our case, the mapping is done from the federation architecture model to the federate application source code.

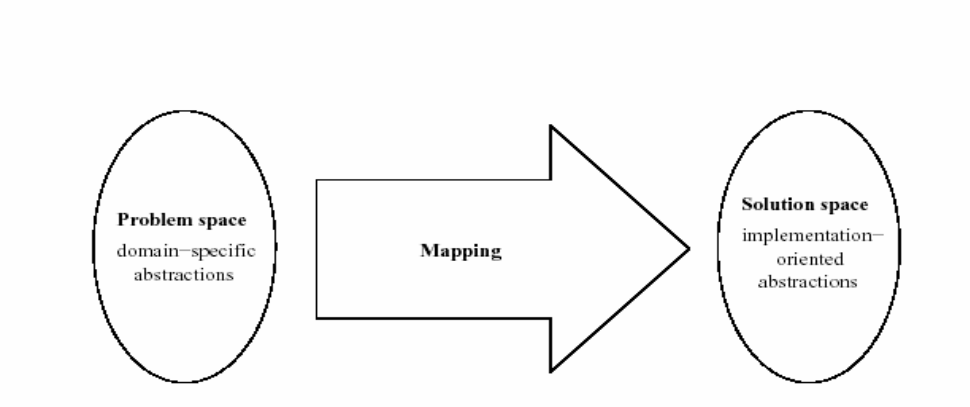


Figure 2.1. Mapping between domain model and implementation-oriented abstractions [Czarnecki 2005]

2.2. Code Generation

Code generation is the technique of writing and using programs that build application code. Typically, it reads in the design, and then builds output code that implements the design. In Figure 2.2, code generation is represented graphically.

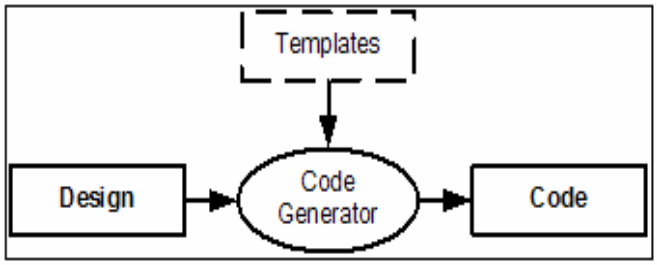


Figure 2.2. Code generation

Code generation, when it is accomplished properly, offers four benefits:

- Quality: output code is as good as having written by hand. It can be standards-compliant.
- Consistency: output code uses consistent class, method, and argument names.
- Productivity: it is faster to generate the code than to write it by hand.
- Abstraction: the design is specified in an abstract form, free of many implementation details.

2.3. Model Driven Architecture (MDA) and Model Driven Engineering (MDE)

"The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform" [Kleppe et al. 2003]. MDA provides an approach for, and enables tools to be provided for:

- specifying a system independently of the platform that supports it,
- specifying platforms,
- choosing a particular platform for the system, and transforming the system specification into one for a particular platform

The primary goals of MDA are portability, interoperability and reusability in the course of architectural separation of concerns. The Model Driven Architecture (MDA) [OMG 2003] is a framework for software development put forth by the Object Management Group (OMG). The MDA development life cycle, which is shown in Figure 2.3 does not look very different from the traditional life cycle in that the same phases are identified. A remarkable difference is the artifacts that are created during the development process. The artifacts are formal models that can be processed by the computers. The following three models are at the core of the MDA.

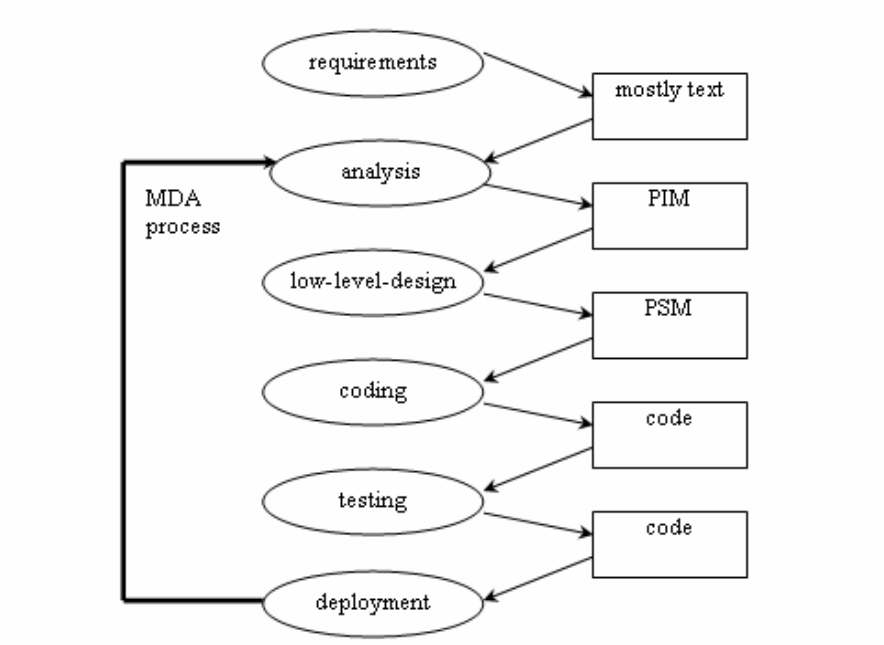


Figure 2.3. MDA software development life cycle [Kleppe et al. 2003]

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

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

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

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

[Kent 2002] remarks that MDA focuses on architecture, on artifacts, on models. Although MDA declares there might be a richer modeling space, it chooses to focus on just one dimension, the transformation between platform independent and platform specific models.

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

[Schmidt 2006] states that MDE technology is a promising approach to address platform complexity. Domain-specific modeling languages formalize the application structure, behavior, and requirements within particular domains. DSMLs are described using metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively.

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

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

An earlier manifestation of MDE is Model Integrated Computing (MIC), which 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 [Ledezci et al 2001].

2.4 High Level Architecture (HLA)

HLA related background material in the section has been extracted from [IEEE 2000a-c, IEEE 2003]. The HLA is common architecture to combine simulations (federates) into a larger simulation (federation). It is based on the publish/subscribe paradigm. A federation execution is a session of a federation executing together. A federation has a name, and involves:

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

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

Federates and the RTI are software. The FOM is data created by the federation developer typically by using a tool. The FOM states agreement on data among the participant federates.

The relationship between the software components is presented in Figure 2.4. Federates are shown in the figure as either simulations, surrogates for live players, or tools for distributed simulation such as data collector, passive viewer. A federate might consist of several processes, perhaps running on several computers. A federate might model a single entity, like a vehicle, or many entities, like all the vehicles in a city.

A federate might have other purposes other than modeling entities: It might be a data collector or viewer, passively receiving data from other federates and generating none for the others, or it might act as a surrogate for human participants in a simulation.

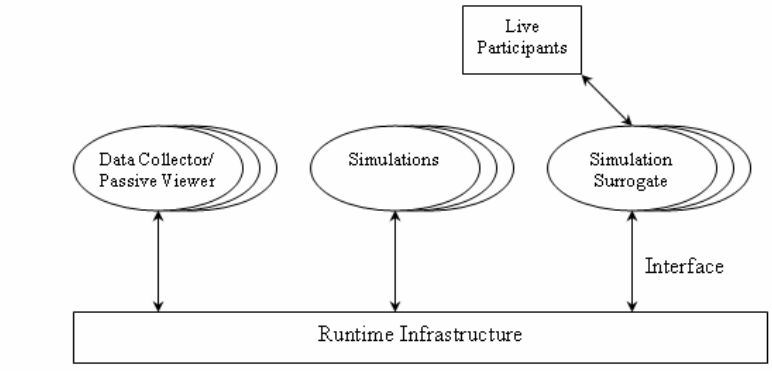


Figure 2.4. Software Components in the HLA [IEEE 2000a]

The HLA is foremost a software architecture, rather than a particular implementation of an infrastructure or tools designed to work with it. The HLA standard supports a variety of implementations. Therefore, it is defined not by software, but by a set of documents. The HLA standard has three parts:

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

For the moment, there are two parallel efforts in progress to follow the adoption of the HLA by standards bodies. One standards adoption effort is through the Object Management Group (OMG), which has adopted version 1.3 of the HLA interface specification as "Facility for Distributed Simulation Systems (FDSS)". The other standards adoption effort is through the IEEE, of whose standards are HLA Framework and Rules [IEEE 2000a], Federate Interface Specification [IEEE 2000b], and OMT [IEEE 2000c].

The Object Model Template (OMT)

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

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

Objects in the RTI refer to simulated entities that are of interest to more than one federate. They persist or endure for some interval of simulated time.

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

HLA Rules

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

The Management Object Model (MOM)

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

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

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

The HLA Services

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

(i) Federation Management

Federation services manage a federation in to ways:

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

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

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

(ii) Declaration Management

The declaration management services are the way for federates to declare their intent to produce (publish) or consume (subscribe to) data. The RTI uses these declarations for routing data, transforming data, and interest management. On the subject of routing, the RTI

uses subscriptions to decide what federates should be informed of the creation or update of entities. Received data go through reduction and re-labeling in accordance with the federate's subscriptions before being delivered. Finally, the RTI uses declarations to indicate interest to publishing federates. The RTI can tell a federate whether any other federate is subscribed to data it intends to produce, so that it can stop producing when no other federate needs the information.

(iii) Object Management

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

(iv) Ownership Management

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

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

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

(v) Time Management

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

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

(vi) Data Distribution Management

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

(vii) Support Services

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

2.5. Federation Architecture Metamodel (FAMM)

FAMM is a proposed metamodel for specifying the architecture of an HLA-compliant federation [Topçu et al 2007]. FAMM formalizes the standard Object Model and Federate Interface Specification. Beyond formalizing the existing HLA standard, FAMM allows the behavioral description of federates based on LSCs. Having the behavioral models of the participating federates gives us the ability to test the federation architecture by executing the federation.

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

- Forming HLA Object Model (federation and simulation object models):
- Specifying the behaviors of participating federates so that they can fulfill their responsibilities within the federation

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

FAMM (Figure 2.5) involves two main sub-metamodels: One for specifying the observable behaviors, and the other for defining the HLA FOM and the HLA service interface.

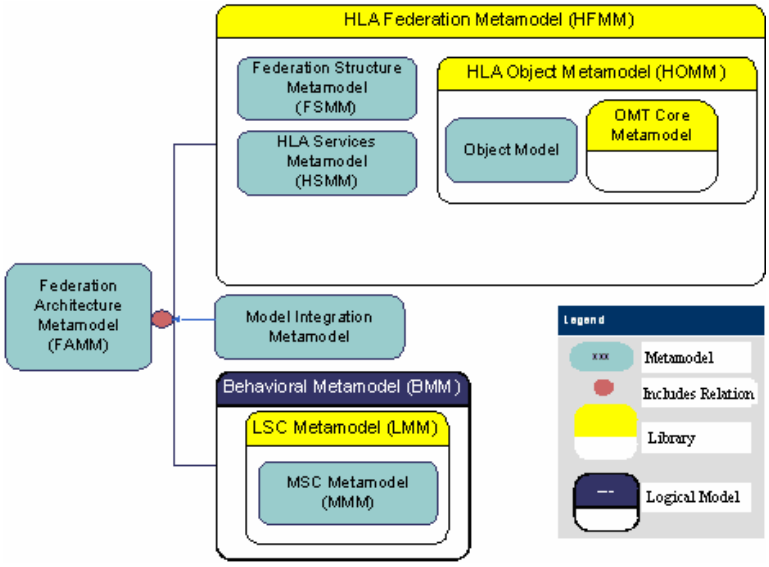


Figure 2.5. Federation Architecture Metamodel Structure ([Topcu et al. 2007])

Figure 2.6 depicts the relationship between FAMM and Federation Architecture. Each participating federate's behavior is modeled using the behavioral metamodel while the FOM is described by using the HLA Object Metamodel.

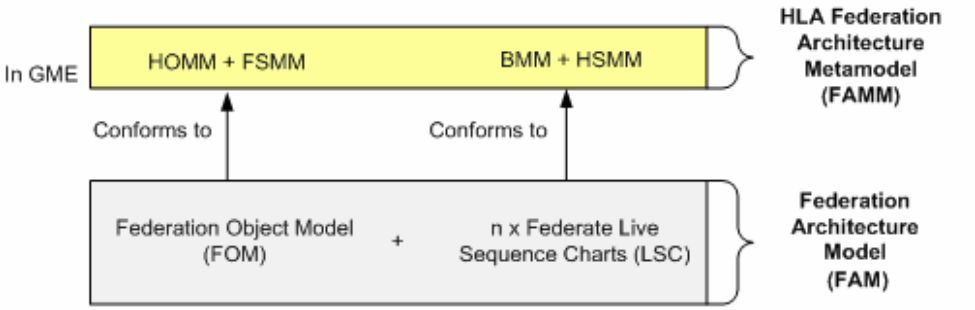


Figure 2.6. Relationship between a Federation Architecture Model and the Metamodel [Topcu et al. 2007]

HLA Object Metamodel (HOMM) is a formalization of HLA Object Model Template (OMT) [IEEE 2000c]. OMT Core folder includes the table contents specified in HLA OMT.

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

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

The HLA Services Metamodel (HSMM) defines the interface of the standard services of Runtime Infrastructure (RTI). These management services provide a functional interface between federates and the RTI. These interfaces arranged into seven basic groups are as follows: Federation management, declaration management, object management, ownership management, time management, data distribution management, and support services [IEEE 2000b].

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

LSC is a graphical language introduced by David Harel and his colleagues [Harel 2001, Damm and Harel 2001, Brill et al. 2004], as an extension of MSC, for specifying the patterns of interactions between components in a concurrent system. MSCs are widely used in the specification of telecommunication systems. The MSC language is standardized by ITU [ITU-T 1998], the most recent standard being Recommendation Z.120 [ITU-T 2004]. Many features of MSCs are adopted in the UML sequence diagrams. LSC extends MSC by providing notations for distinguishing mandatory and optional behavior and by promoting conditions to first class elements.

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

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

2.6. Generic Modeling Environment (GME)

GME serves as a metamodel development environment as well as a customized model building environment once the developed and registered metamodel is invoked. In other

words, GME is a configurable toolkit for creating domain-specific modeling and program synthesis (code generation) environments. It puts the MIC [Ledezci et al 2001] vision into practice. The configuration is achieved through metamodels specifying the modeling language of the application domain. The modeling language contains the syntactic, semantic, and presentation information of the domain. The modeling language defines the family of models that can be created using the resultant modeling environment. The metamodels specifying the modeling language are used to automatically generate the target modeling environment. The generated environment is then used to build domain models. These models can be input to all kinds of model-driven processing, including model transformation and code generation. This kind of process is called model interpretation in GME parlance. There is a metamodeling language, called MetaGME, which configures GME for creating metamodels, called paradigms in GME jargon. These models are then automatically translated into GME configuration information through model interpretation. [GME 2006]

MetaGME meta-metamodel plays the similar role as Meta Object Facility (MOF). MOF is a sister-standard of UML and is maintained by the same standards-publishing body, the Object Management Group (OMG). A metamodel which is an instance of MOF formally specifies the abstract syntax of the set of modeling constructs which constitute a modeling language. In the MOF support context, carries out some research on the direct transformation from MetaGME to MOF [Emerson 2005].

In the code generation, GME BON2 application interface (API) is used. This API enables the developer to walk on the input model. API supports both C++ and Java programming languages. We use Eclipse development environment [Eclipse 2007] for the programming.

2.7. Aspect-oriented Programming (AOP)

AOP [Kiczales et al. 1997] supports modularity of cross-cutting concerns in existing languages, particularly object-oriented programming languages. Some concerns do not align well with existing module boundaries; these are called “cross-cutting concerns”. Well known examples of cross cutting concerns are: error checking/handling, synchronization, performance optimization, monitoring/logging, and debugging support [Elrad et al. 2001].

AspectJ [AspectJ 2007] is an extension of Java that allows modular separation of concerns. Here are some key concepts of AOP couched in AspectJ terms: Aspects are special Java classes that serve as modules to encapsulate concerns in source code. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations.

A joinpoint is a well-defined point in the program (base code) such as a method declaration, a method call, and an assignment statement.

A pointcut is a set of joinpoints. There is a syntactical mechanism for specifying pointcuts. For example, a pointcut can be specified with a regular expression, and the method declarations with matching method names will be picked as the joinpoints.
Advice is AspectJ's mechanism for affecting the behavior at joinpoints. An advice definition comprises a block of code, a pointcut and a specification of whether the block should run before, after or in place of occurrences of the joinpoints in the pointcut when they occur. These options are indicated by the keywords before, after, and around, respectively.

A key idea in our work is to handle computation as an aspect to be weaved onto the base code that handles communication and data model access. A problem with the AOP is that when base code is changed, aspects may become useless. This does not concern our work because the code generator generates both pointcuts and joinpoints. Thus their matching is guaranteed by the code generator.

CHAPTER III

CODE GENERATION FROM A FEDERATION ARCHITECTURE MODEL

This chapter presents the proposed code generation process from the federate or federation developer's point of view. First, we introduce a simple federation to be used as a running example throughout the chapter. Then overview of the code generator and generated code structure is explained. Incorporating HLA related information into code is discussed. How the generated code runs is expressed. After that code generation process from model to code is explained in the running example. Finally, discussion of the example and related works are mentioned.

3.1. Running Example: Strait Traffic Monitoring Simulation

In this section we introduce Strait Traffic Monitoring Simulation (STMS), which will serve as our running example. Later sections will introduce code generation in detail, accompanied by this example.

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: 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. When a ship application joins the federation, this corresponds to entering the strait. When it resigns from the federation, this 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. 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. The conceptual view of the STMS is illustrated in Figure 3.1.

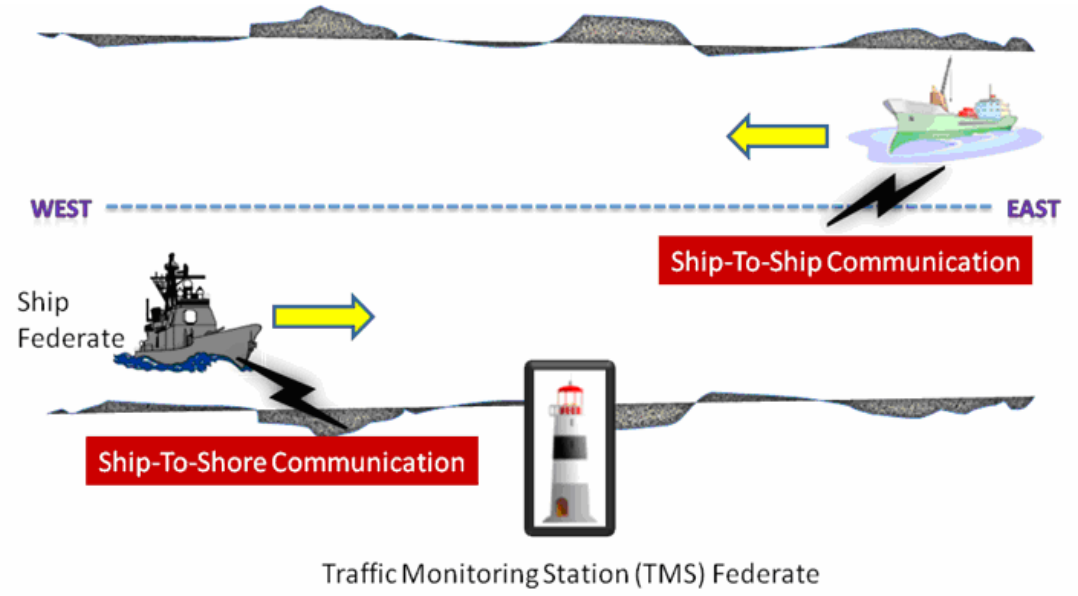


Figure 3.1. Strait Traffic Monitoring Simulation Conceptual View

[Topcu et al. 2007]

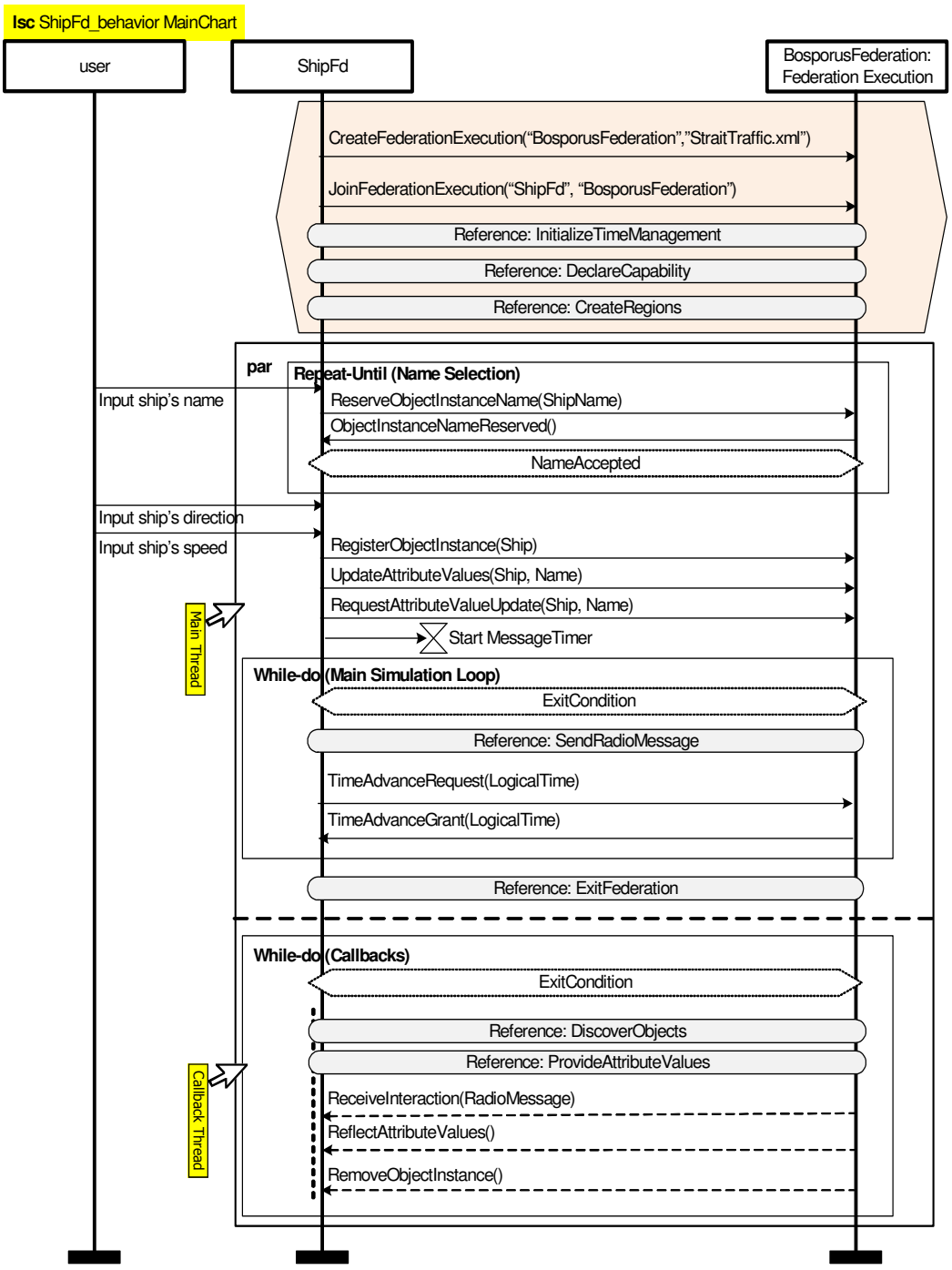


Figure 3.2. Behavior Model for the Ship Federate in LSC Graphical Notation [Topcu et al. 2007]

Focusing on the ship federate in STMS, we identify three LSC instances, representing the behaviors of the federate, the user, the ship federate called *ShipFd*, and the federation

execution called *BosporusFederation*. The behavior model of the ship federate is presented in Figure 3.2 in LSC graphical notation.

Figure 3.2 depicts a pre-chart, which consists of two parts: the pre-part (diamond shaped), attached to the body-part (rectangle shaped). The events in the body-part depend on the completion of the event occurrences in the pre-part. Thus, unless federation creation, time management initialization, declaration capability and regions creation complete successfully, the federate must not progress. Time management initialization, declaration capability and regions creation methods are modeled in the separate charts for reasons of modularity, thus, references to them are included in the pre-chart.

There is a parallel inline expression in the body-part, marked by "Par" indicator, and drawn as a rectangular shape that is divided by horizontal dashed lines to identify its operands. The operands run in parallel. (This is handled by a separate thread implementation for each operand.) In the example, there are two operands, handling of federate simulation loop called "main thread" and handling of call-back events which may arrive any time, called "callback thread". There are repeat-until (do-while) and while-do constructs in the first operand, and a while-do in the second operand. Ship name is selected (reserve object instance name) in the repeat-until construct and then radio messages (interactions) are sent to the *BosporusFederation* execution in the while-do. When the federate execution ends, *ExitFederation* condition is set to true in the first operand.

A repeat-until construct is marked by a "Repeat-Until" indicator and drawn as a rectangular shape. At the bottom of the rectangle is a dashed-diamond shape that denotes the loop condition. The loop is repeated until the condition is satisfied, in our case, until the unique ship name is selected by the interactive user.

A while-do construct is marked by a "While-Do" indicator and drawn as a rectangular shape. The loop condition is located at the top portion of the rectangle indicating that the loop is to be repeated as long as the condition is satisfied. In this example, the loop is repeated while the *ExitCondition* is not true. (*ExitCondition* is set when the federate resigns).

In the second operand of the parallel construct, call-back events are received in arbitrary order from the federation execution in the while-do. This loop is repeated until the federate is resigned from the federation. Unordered receiving of callbacks is specified by a coregion, which is indicated by a vertical dashed line parallel to the location. There are two references in the while-do construct (*DiscoverObjects* reference and *ProvideAttributeValues*). LSC cold messages are indicated by horizontal dashed arrows, and hot ones by solid arrows. Cold messages are not guaranteed to arrive. For this reason, all the events coming from the federation execution have cold designation.

In the context of the example, some LSC concerns are discussed. Except for them, complete LSC structures and their semantic meanings are reached from [Harel 2001, Damm and Harel 2001, Brill et al. 2004].

3.2. Federation Application Code generation from FAM

In this section we introduce our code generator from the viewpoint of a federate application developer. We address the more technical behavioral model oriented issues (LSC) in Chapter 4 and Chapter 5.

3.2.1. Overview of the Code Generator

The input FAM includes the behavioral models of the participating federates as well as the FOM they have in common. The behavioral model of a federate is presented as a single LSC document consisting of one or more LSC diagrams.

Here are some highlights of the features of the code generator:

- All RTI interface specification methods in the standard
- All MSC/LSC features with few exceptions such as synchronous messages

Following the classical schema, there are two sequentially connected modules forming our generator, namely, Intermediate Form Generation Module (the front end) and Java Code Generation Module (the back end). Figure 3.3 shows the overall data flow diagram of the generator. The Intermediate Form Generation module walks on the source FAM model, using the model interpreter API of GME, and constructs the intermediate form that holds the model in a convenient internal form. Then the Java Code Generation module walks on the intermediate form and produces the diagram class, the federate class, the computation aspect, and the federation execution aspect. The produced codes are fed into the AspectJ compiler. Further details about front end and back end modules are presented in Appendix B and Appendix C respectively. An example which follows this code generation process is also presented in Appendix D. In this appendix, the example's concrete model, GME-model, intermediate form, and code are illustrated.

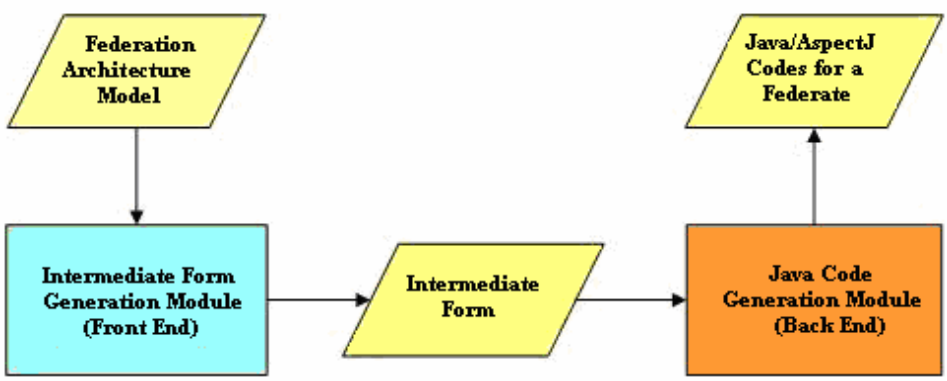


Figure 3.3. Code Generator Data Flow Diagram

The current version of our code generator generates Java and AspectJ codes. However, only the Java Code Generation Module needs to be re-implemented if another target language is desired.

3.2.2. Structure of the Generated Code

One LSC Document is assumed per federate application. For each LSC Diagram in the document a diagram class is generated. An instance class is generated for each LSC instance in the diagram. In case an LSC instance occurs in multiple diagrams an LSC instance class is generated for each occurrence. Code for each LSC instance in a diagram is started by the diagram class, and runs in its own thread.

In an LSC instance, in regards to federate – federation execution communication, an RTI Ambassador Method call is generated for every LSC message-out event, and similarly, a Federate Ambassador call-back for every LSC message-in event. An LSC Instance Aspect class (i.e. a computation aspect) code is generated for every LSC instance to handle the standard RTI Ambassador methods and Federate Ambassador call-back methods, and the LSC-specific auxiliary methods for the preliminary computation. The latter essentially help resolve, in a randomized fashion, the nondeterminism inherent in an LSC specification. For example, auxiliary methods randomly determine loop counts (within bounds), choice of alternatives, order of sending events in a coregion, etc.

A dictionary structure guides nondeterministic choices regarding conditions and temperatures. Specifically, it holds

a) conditions (whose names are keys) and whether they are satisfied or not (which are associated values),

b) LSC model elements (whose names are keys) and their temperatures (which are associated values).

The dictionary is defined in the computation aspect. The developer can edit it so that the choices (i.e. values associated with keys) are determined according to the simulation logic (rather than randomly).

Declarations of both RTI methods and auxiliary methods in LSC instance code (federate base code) constitute the join points targeted by the LSC instance aspect (computation aspect). For every join point in the federate's base code an advice code is generated in the computation aspect code. The developer can change the preliminary computation by editing advices associated with the pointcuts, and then weave the edited aspect onto the base code, which requires the use of AspectJ [AspectJ 2007] compiler. In AOP terminology, this is called production aspect usage. This is the only place where the developer's intervention is needed to produce a properly functioning federate. In other words, only the LSC instance aspect can be edited by the developer; all other generated codes are read-only. Well-known examples of cross-cutting concerns are non-functional, e.g. logging, authentication, etc. In our use of AOP, however, we take a functional concern (addressed by the computation aspect) as a cross-cutting concern.

All argument information of FAM events (Ambassador Methods) that flow between instances (federates, federation execution) are carried by *LSCObjects*. For example, object classes, interaction classes and their attribute and parameter information are all held by this data structure.

LscRTILib library essentially serves as an RTI interface layer. This library takes an *LSCObject*, unpacks it into the actual RTI method parameters, and calls the actual RTI method. In other words, generated federate codes and LscRTILib communicate with each other over *LSCObject*. Thus, the generated federate application code is independent of the vendor specific implementation of the RTI API. This library is in fact an adapter, in the sense of a design pattern, between federate code and the specific RTI, such as Pitch pRTI 1516 [Pitch RTI 2007]. Note that LscRTILib does not attempt to redefine the programming model offered by the RTI or simplify its programming interface.

A Federation Execution Aspect class is generated for each federation execution in which this federate can participate. This aspect code catches the Federate Ambassador call-back methods of the RTI by using LscRTILib library and forwards them to the generated federate base code. Handling of call-backs is discussed in section 3.2.4.2.

In Figure 3.4, the static structure of the generated federate code is presented as a class diagram.

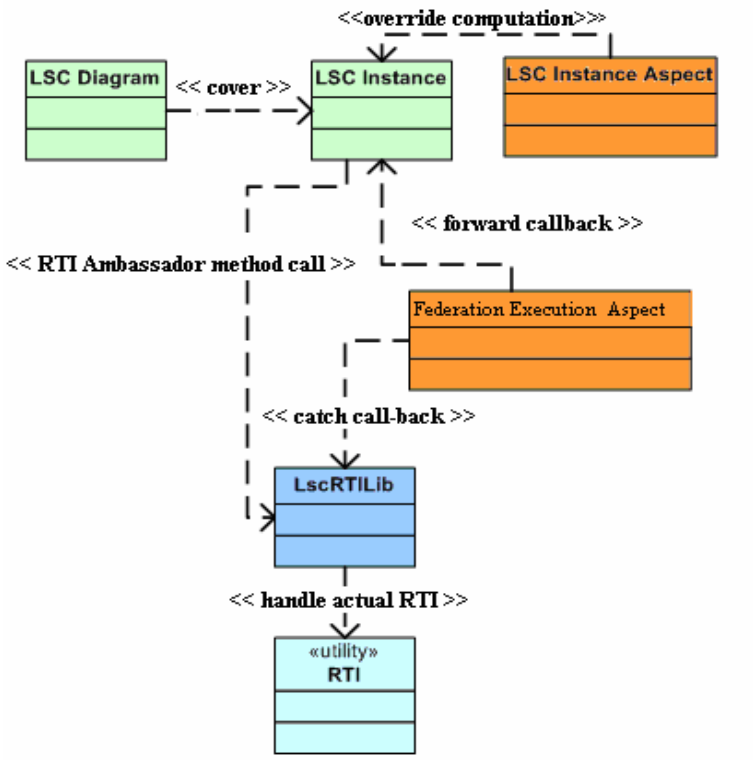


Figure 3.4. Structure of the Generated Federate Application

Let us now revisit the generated code in AOP terms referring to Figure 3.4. LSC Diagram and LSC Instance codes constitute the federate base code, which brings about the federate's communication behavior as specified by the BMM of FAM (see Figure 2.4). Each LSC Instance Aspect, which implements the preliminary computation, is weaved onto the related LSC Instance. Separately, Federation Execution Aspect, generated per federation execution, is weaved onto LscRTILib library. Aspect weaving is carried out by the AspectJ compiler [AspectJ 2007], which produces the "weaved intermediate Java byte code", which is then run on the Java Virtual Machine [JVM 2007].

3.2.3. Incorporating HLA Object Model and Services into Code

In this subsection we discuss how HLA Object Model (FOM or SOM) and Federate Interface Specification (RTI methods along with their parameters, called in the federate base code) are reflected to the generated code. At the metamodel level these issues are represented in HFMM (see Figure 2.4).

Information to weave is obtained from the input FAM. For example, in order to create a federation, the federation name and the path for the FDD file are required. These are obtained from the FAM (specifically, from HSMM) and then they are included in the computation aspect. (See *CreateFederationExecution* method's advice in Figure 3.5a) Then,

it is weaved onto the federate's base code, specifically, it replaces the body of *CreateFederationExecution* method, a joinpoint shown in Figure 3.5b. Thus, *CreateFederationExecution* method will be called with appropriate parameters.

```
pointcut pcSendCreateFederationExecutionCFEBosporusFederation(…)//pointcut definition
{//advice block begin
      FederationName="BosporusFederation";// from FAM
      FedFile="c:\\eclipse-SDK-3.0.1-
       win32\\eclipse\\workspace\\FedCodeGen1516\\StraitTraffic.xml";// from FAM
      (…) \\proceed and return
}//advice block end
```

Figure 3.5 a. CreateFederationExecution Method (advice) in Computation Aspect Code

```
Public static boolean SendCreateFederationExecutionCFEBosporusFederation(String
FederationName,String FedFile)
//corresponding joinpoint
{
        /* dummy code */  // It is overridden by the (Figure 3.5a) computation aspect
}
```

Figure 3.5 b. *CreateFederationExecution* Method (join point) in Federate Base Code.

In Table 3.1, the retrieved information and its source in the FAM is described with respect to the RTI Interface Specification service areas. The information retrieved from FAM by the code generator front end is then placed in the preliminary computation aspect by the back end (see Figure 3.3). In general, RTI methods along with their parameters information are retrieved from HSMM in the following service areas.

Table 3-1. Information Retrieved from FAM and Placed in the Generated Code

Service Area	**Information**	**Source in FAMM**
Federation Management	federation name, federate name and related federation execution instance name, Synchronization points	FSMM and HSMM
Declaration Management	interaction and object classes	HOMM and HSMM
Object Management	interaction and object classes, dynamic object instances	HOMM and HSMM
Ownership Management	Object classes	HOMM and HSMM
Data Distribution Management	Dimensions, dynamic regions,	HOMM and HSMM
Time Management	Timestamps, lookaheads and receive/timestamp orders	HOMM and HSMM
Support Service	Evoke time periods	HSMM

3.2.4. How a Generated Federate Runs

In Figure 3.6a and 3.6b, collaboration diagrams of the generated federate is illustrated. Figure 3.6a shows how an RTI Ambassador method is called. First, the RTI Ambassador Method declared in LscRTILib is called by the federate (i.e. the LSC Instance class). At this time, LSC Instance Aspect interferes and catches the RTI method (in LSC Instance class) calls. Then it overrides the arguments of the method. After that, RTI Ambassador Method call proceeds to LscRTILib. Finally, LscRTILib calls the actual RTI Ambassador Method with the actual arguments. This RTI method calling process is exemplified in section 3.2.4.1

In Figure 3.6b, how a call-back is received from RTI is sketched. First, LscRTILib receives the call-back from RTI and calls its own method (with the same name) which has a single *LSCObject* type argument. This method call is caught by federation execution aspect, which then forwards this call to federate's LSC instance. Thus, the call-back reaches the

federate. Finally, LSC Instance aspect also catches the forwarded call-back and presents it for overriding. This handling process is exemplified in section 3.2.4.2

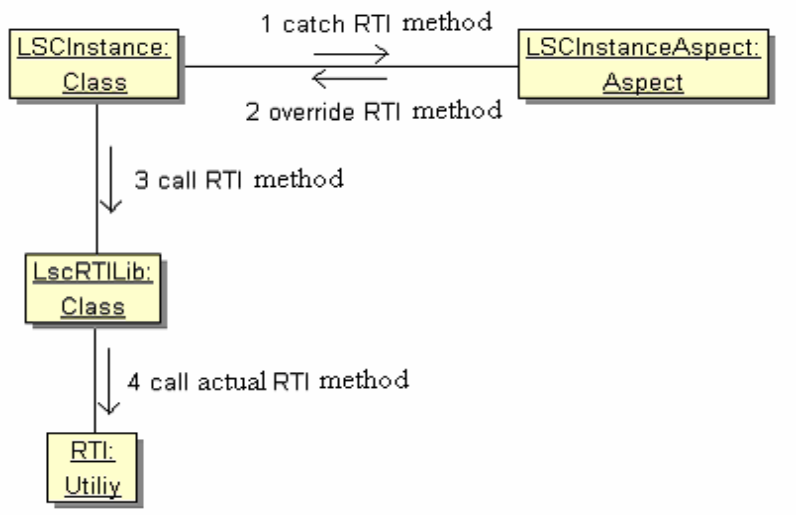


Figure 3.6 a. Collaboration Diagram of Calling RTI Ambassador Method in the Generated Federate

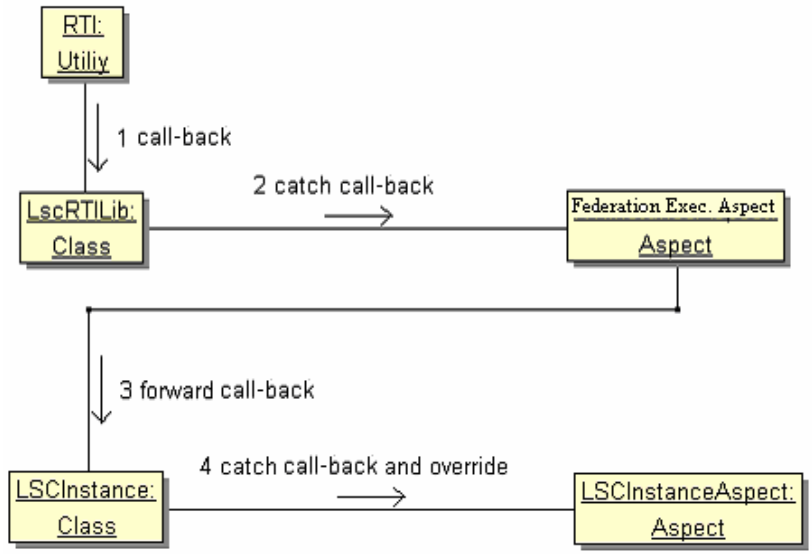


Figure 3.6 b. Collaboration Diagram of Handling Federate Ambassador Call-back in the Generated Federate

3.2.4.1. Handling the RTI Methods

In the federate's RTI Ambassador Method call, related LscRTILib (*BosporusFederationRTILib*) method is called (cf. Figure 3.6a, action 3) with the *LSCObject* argument. Example code is presented in Figure 3.13a.

This RTI method (in *LSCInstance* class) is caught (cf. Figure 3.6a, action 1) by LSC Instance Aspect and the arguments of the call can be overridden (cf. Figure 3.6a, action 2). In our example in Figure 3.14a, the values assigned to the message and *CallSign* interaction class parameters in the base code are to be overridden. The overriding code is shown in italic form in the figure. In the corresponding LscRTILib method, actual RTI Ambassador Method is called (cf. Figure 3.6a, action 4) with the particular RTI (Pitch RTI in our case) specific arguments (Figure 3.7).

```
public  void sendInteractionWithRegion (LSCLib.LSCObject procUpdate)
{
    (…)//procUpdate is adapted to the vendor specific RTI method parameters
    mRtiAmb.sendInteractionWithRegions(proc.iHandle,phvpset,rSet,null,lt);
    //actual RTI method
    (…)//method exception and other code details
}
```

Figure 3.7. RTI Ambassador Method in the *LscRtiLib*

3.2.4.2. Handling the Call-back Methods

When a Federate Ambassador call-back event occurs, related LscRTILib method (same named) is called (cf. Figure 3.6b, action 1). Example code is presented in Figure 3.8a.

In the method of LscRTILib, arguments of method are packed into our common data structure, namely *LSCObject*. Then, the same named method in the library that has an *LSCObject* argument is called. Example code is presented in Figure 3.8b.

```
public final void receiveInteraction (…)//parameter definitions
{
ambLib.receiveInteraction(iClassHandle,phvpset, sendOrder,tType,lt, recvOrder, mrh);
//LscRTILib method is called when the events come
}
```

Figure 3.8. a Federate Ambassador Call-back Method in the *RTIFederateAmbassador*

```
public  void receiveInteraction(…)//parameter definitions
{
    (…)//proc is declared and callback method parameters are packed into it.
    receiveInteraction(proc);//same named method is called with the LSCObject as argument
    (…)//return code
}
public  void receiveInteraction(LSCLib.LSCObject proc)//same named callback method
{}//this method is caught by federate execution aspect since it is a joinpoint for it.
```

Figure 3.8. b. Federate Ambassador Call-back Method in the *LscRtiLib*

When this calling is done, federation execution aspect (Example advice code is shown in Figure 3.9a.) catches this method call (cf. Figure 3.6b, action 2) and forwards (cf. Figure 3.6b, action 3) it to the corresponding method in the federate base code. Example federate Ambassador call-back code is presented in Figure 3.9b.

```
pointcut ReceiveInteraction(…) //pointcut definition
{
    ShipFd.ReceiveInteraction(proc);//calling the same named method in  base code
}
```

Figure 3.9. a. Federate Ambassador Call-back Method (advice) in the Federation Execution Aspect

```
public static void ReceiveInteraction(LSCLib.LSCObject proc)
{
    if (proc.name.compareTo("RadioMessage")==0)
    //received interaction is compared with the interaction class names of the federate
    {
        //receive the interaction:
       RecvReceiveInteractionRadioMessageBosporusFederation(proc);
    }
}
```

Figure 3.9. b. Federate Ambassador Call-back Method in Federate Base Code

And finally code to receive interaction is introduced (cf. Figure 3.6b, action 4) to the developer (Figure 3.14b).

3.3. More On The Code Generator

3.3.1. Participating in Multiple Federations

A federate may be a member of more than one federation at the same time. For each joined federation a federation execution aspect is generated. An LscRTILib is declared and instantiated for accessing the actual RTI in a federation execution aspect code.

For example, if we have two federation executions, we have two different aspects, in which an LscRTILib library is declared (*ShipFd.BosporusFederationRTILib*). Example library declaration is presented in Figure 3.15 and its usage is presented in Figure 3.13a.

3.3.2. Retargeting another RTI

Developers use different RTIs offered by various vendors. Although the current RTIs must conform to the IEEE 1516 standard, their APIs exhibit minor differences (e.g. variations to the HLA standard data types) among the vendors. The code generator can target vendor specific RTIs by customizing the LscRTILib library, which serves an RTI adapter layer.

In cases of methods and data types outside IEEE 1516, modifications to the HSMM will be necessary. In fact a vendor-specific version of HSMM can be constructed. Consider a vendor-specific method, say *Mnew*, not mentioned in the standard. Definition of *Mnew* must be introduced to HSMM, and *LscRtiLib* must be extended with the mapping from *Mnew* method definition in HSMM to constituents of a *Mnew* method call.

3.3.3. Code Clarity

The readability of the generated code is crucial as the application developers deal with it directly. Therefore, care is taken to generate understandable codes that closely reflect the model structure and to separate and hide those parts that are not subject to aspect weaving. Code generation follows the coding standard *CamelCase*. Moreover, user-supplied comments on the input model are carried to the code to ease the task of navigating the code.

3.3.4. Support for Model-Code Traceability

Developer can attach comments to model elements; the comments are carried over to the generated code in the code generation. Therefore traceability is provided according to the comments. Comment support also gives us more readable and qualified codes.

Model element names are used as the keys for the dictionaries (implemented as a *hashtable*). For example, in the implementation of an alternative inline expression, alternative choice value is hold in the dictionary where alternative inline expression model element name is a key. In addition, some model element names are used directly as variable names in the generated code. These types of usages give us a capability to establish a traceability between model and code.

3.3.5. Availability of the Generator

The presented code generator is under GNU Public License, and detailed documentation can be obtained from “http://www.ceng.metu.edu.tr/~e73883“.

Our generator source code is almost 15.000 lines of code (LOC). It is developed in Eclipse 3.3 environment with Java programming language [Eclipse 2007]. It is packaged as a GME model interpreter.

Generated codes are Java and AspectJ, and can be compiled and run in Eclipse. Eclipse AspectJ plug-in [AspectJ 2007] is used for compiling AspectJ codes. The generator currently supports Pitch RTI (certified for IEEE-1516).

3.4. Code Generation Example

We now return to our example STMS federation to walk through the code generation process.

3.4.1. Steps in Using the Code Generator:

Step i: Constructing the FAM

The STMS FAM is built as conforming to FAMM.

Step ii: Configuring the Generator

A configuration document based on XML, called *GeneratorConf.xml*, is provided with initial values for configuration parameters.

STMS is configured by setting values of the following parameters:

- Seed for the random number generator,
- The path for the generated code,
- Maximum poll count for receiving an optional (cold) message, and
- Waiting (sleep) time between two successive polls.

So final configuration XML file is presented (Figure 3.10) as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Confs>
<Random seed="123456"> <!— for random number generator -->
</Random>
<Sleep time="100" passes="50">
<!—sleep time and number of passes for cold message receiving-->
</Sleep>
<PATH>
<Generated path="c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\FedCodeGen\">
</Generated> <!—destination path for the generated code-->
<Generator path="c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\NewCodeGenProject\">
</Generator> <!— path of the generator code-->
</PATH>
<External-InstanceLibs>
 <InstanceLib name="RTILib" prefix="RTI"> <!— external library used in the generator-->
</InstanceLib>
</External-InstanceLibs>
</Confs>
```

Figure 3.10. XML Configuration File for the Code Generator for *ShipFd* Application

Step iii: Running the Generator

The code generator is run in GME as a model interpreter [GME 2006]. The generated code files are placed in the folder specified in the configuration file. In our case: *Ship_MSC*

(Diagram class), *ShipFd* (Ship federate class), User (Live entity class), *ShipFdAspect* (computation aspect of ship federate), *UserAspect* (computation aspect of user) and *BosporusFederationLibAspect* (federation execution aspect) are generated. Generated three classes and three aspects are shown with a class diagram in Figure 3.11. This class diagram reflects the static structure of a generated federate application, cf.Figure 3.4.

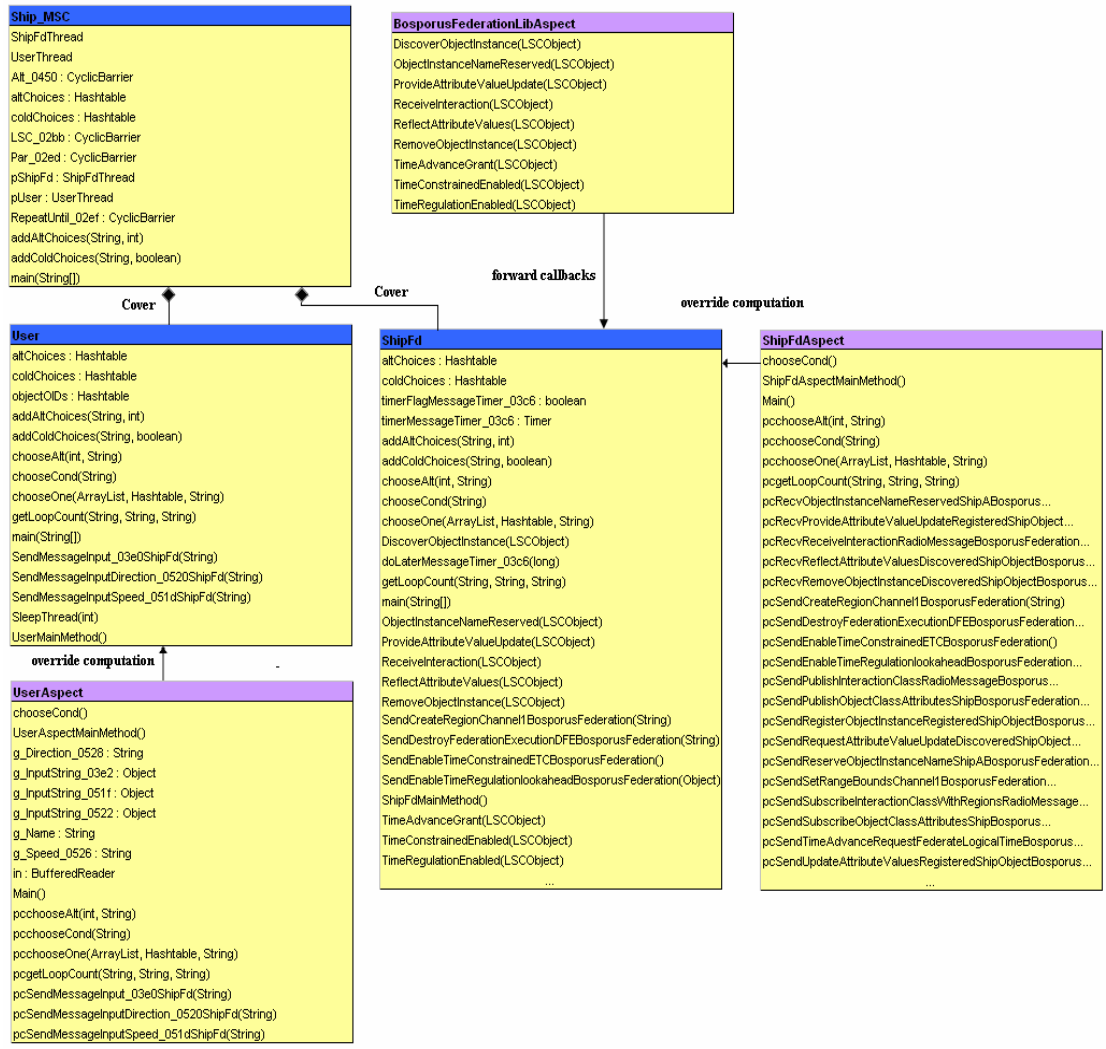


Figure 3.11. Class Diagram of the Ship Federate

a) Base Codes

The *Ship_MSC*, *ShipFd* and *User* classes constitute the base code of the ship federate application as shown in Figure 3.11. *Ship_MSC* is a diagram code in which the *ShipFd* and User threads are defined and run. *ShipFd* is an instance code where federate RTI methods and LSC-specific auxiliary methods are generated. User is also an instance code in which user sends ship name, direction and speed to the ship federate.

```
public static void ShipFdMainMethod (){
 (…) // prechart 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(); // ship's name comes from the user
   // Reserve Object Instance Name is sent to RTI:
   SendReserveObjectInstanceNameROINBosporusFederation("s0");
   // "s0" is to be overridden by the computation aspect which will take ship name from user
   // Object Instance Name Reserved (OINR) is received from RTI
   condRecvObjectInstanceNameReservedOINRBosporusFederation();
   (…) //If OINR succeeds leave the loop
   } while (!((Boolean)Ship_MSC.coldChoices.get("until_0300")).booleanValue());
   (…)// Other Inputs: direction and speed come from the user.
   SendRegisterObjectInstanceRegisteredShipObjectBosporusFederation(...);
   // Register Object Instance is sent to RTI for the Ship object
   SendUpdateAttributeValuesRegisteredShipObjectBosporusFederation(…);
   // Update Attribute Values is sent to RTI for the Ship object
   SendRequestAttributeValueUpdateDiscoveredShipObjectBosporusFederation(…);
   // Request Attribute values Update is sent to RTI for the Ship object
   doLaterMessageTimer_03c6(100); //timer is started for periodically send interactions
   // While-Do Main Simulation Loop begins
   while (((Boolean)Ship_MSC.coldChoices.get("ExitCondition_040f")).booleanValue()) {
    //loop is repeated until the federate is resigned.
    (…) // The code generated for SendRadioMessage chart is inserted here.
    // when a timeout occurs radio message interactions are sent and timer is restarted
    // Time Management  methods begin
     SendTimeAdvanceRequestTARBosporusFederation(new Double(55.0));
    // Timestamp type Double comes from FAM.  Timestamp value (55) should be overridden.
   condRecvTimeAdvanceGrantTAGBosporusFederation();
   // Time Advance Grant is received from RTI.
 }//end of main simulation loop.
 (…) // The code generated for Exit Federation chart goes in here.
 //federate is resigned and federation is destroyed.
}//end of the main thread
```

Figure 3.12. Excerpts from the Generated Java Code of Ship Federate Application

To give a sense of the generated code, a part of the ship federate's (see Figure 3.12) and a sample RTI Ambassador Method (*sendinteraction* in Figure 3.13a) and a federate

Ambassador method (*receiveinteraction* in Figure 3.13b) are shown in the figures. The first operand (main thread) of the parallel inline expression (see Figure 3.2) of the generated *shipFd* code is exemplified in Figure 3.12. For every operand in a parallel inline expression occurring in the LSC, a thread (e.g. MainThread_02ee and CallbackThread_032c) is generated. For loop idioms, "while-do" or "repeat-until" code statements are generated. Values of loop conditions are retrieved from the dictionary (implemented as *hashtable* named *coldChoices*) defined in the computation aspect. In place of the chart references in the LSC model, the referenced charts' codes are generated. For example, for *CreateRegions* reference, *CreateRegion* and *SetRangeBounds* methods are generated.

In Figure 3.13a, interaction parameters are packed into an object of *LSCObject*. Then the corresponding LscRTILib method (in this case, *sendInteraction*) is called. In Figure 3.13b, a federate Ambassador method (in this case, *receiveinteraction*) example in the federate base code is shown.

```
public static boolean
SendSendInteractionWithRegions_0536RadioMessageBosporusFederation(…)//parameters
    {
        LSCLib.LSCObject proc= new LSCLib.LSCObject();
        //interaction class information comes from HOMM.
        proc.name="RadioMessage"; //interaction class name
        proc.pars=new ArrayList(); //parameter list of the interaction class
        LSCLib.LSCAttribute parNew0 =new LSCLib.LSCAttribute();
        //parameter1 is declared
        parNew0.name="CallSign"; //parameter1's name
        parNew0.type="Object"; //parameter1's type in Java
        parNew0.objClass="HLAASCIIstring"; //parameter1's type in HLA datatype
        parNew0.objVal=CallSign; //parameter1's value
        proc.pars.add(parNew0); //parameter1 is added to the parameter list
        (…)//parameter2 is added.
            //dimension and region data is added to the parameter list
            //time stamp data is added to the parameter list
        BosporusFederationRTILib.sendInteractionWithRegion(proc);
        //same named LscRTILib method is called
    }
```

Figure 3.13. a. A Sample *SendInteraction* RTI Ambassador Method in Federate Base Code (*ShipFd*)

```
public static void RecvReceiveInteractionRadioMessageBosporusFederation
(LSCLib.LSCObject iClass,String TimeStamp,int SentOrderType,int ReceiveOrderType,String
TransportationType)
{}//received interaction parameter values are held in iClass.
```

Figure 3.13. b. A Sample *ReceiveInteraction* Federate Ambassador Callback Method in Federate Base Code (*ShipFd*)

b) Codes for Aspects

Two computation aspects and a federation execution aspect are generated, namely *ShipFdAspect*, *UserAspect*, and *BosporusFederationLibAspect*. *ShipFdAspect* overrides all RTI methods in the *ShipFd* federate base code. In *ShipFdAspect*, dictionaries and LSC-specific auxiliary methods' (i.e. *chooseOne*, *getLoopount*) advices are also generated.

Two sample advices, namely, RTI Ambassador Method's (send interaction) advice and a federate Ambassador method's (receive interaction) advice, are shown in Figure 3.14a and Figure 3.14b, respectively. In Figure 3.14a, federate send interaction method (cf. Figure

3.13a) is caught in the *ShipFd* base code and preliminary logic (in italic) is filled in. The developer can edit this advice as described in the subsequent "Editing the Computation Aspect" section.

In Figure 3.14b, federate receive interaction method (cf. Figure 3.13b) is found on the *ShipFd* base code and received data is placed in its advice in the *ShipFdAspect*. This received data is the values of all parameters of the interaction class. In this example, the interaction class is *RadioMessage* with parameters *callsign* and message.

```
pointcut pcSendSendInteractionWithRegions_0536RadioMessageBosporusFederation()
{//pointcut definition
    CallSign=new Boolean(true);
    //call sign is given  as preliminary computation in the computation aspect
    Message="Radio Message Sample"; //message is given as preliminary computation
    (…) //declaration detail of dimension  is get outed
    parChannelDimension2_0.strVal="ChannelDimension"; //dimension comes from FAM
    (…) //declaration details of region
    parChannel13_0.strVal="Channel1"; // region comes from FAM
    (…) //other  details of dimension and region
    TimeStamp=new Double(2.0);//must be overridden
    proceed(CallSign,Message,RadioMessagewithRgnsDims,TimeStamp);
    return true;
}
```

Figure 3.14 a. A sample RTI Ambassador Method (advice) in Computation Aspect (*ShipFdAspect*)

```
pointcut pcRecvReceiveInteractionRadioMessageBosporusFederation (…)//pointcut definition
{
      Object CallSign= (Object)((LSCLib.LSCAttribute)iClass.pars.get(0)).objVal;
      System.out.println("Received CallSign Parameter:"+CallSign);
     //callsign interaction class parameter is printed.
      Object Message= (Object)((LSCLib.LSCAttribute)iClass.pars.get(1)).objVal;
      System.out.println("Received Message Parameter:"+Message);
     //Message interaction class parameter is printed.
      System.out.println("Received TimeStamp:"+TimeStamp);
      System.out.println("Received SentOrderType:"+SentOrderType);
      System.out.println("Received ReceiveOrderType:"+ReceiveOrderType);
      System.out.println("Received TransportationType:"+TransportationType);
      proceed(iClass,TimeStamp,SentOrderType,ReceiveOrderType,TransportationType);
}
```

Figure 3.14 b. A sample Federate Ambassador Method (advice) in Computation Aspect (*ShipFdAspect*)

BosporusFederationLibAspect (federation execution aspect) is mainly used to catch call-back methods from the Bosporus federation execution. A *BosporusFederationRTILib* object is instantiated from *LscRtiLib* in this aspect and it is used to reach actual RTI. A sample *LscLibRTI* definition (*BosporusFederationRTILib*) and a sample (*ReceiveInteraction*) advice are presented in Figure 3.15.

In Figure 3.15, *ReceiveInteraction* call-back method is caught by the federation execution aspect (*BosporusFederationLibAspect*) and forwarded to the federate (*ShipFd.ReceiveInteraction*).

```
public static RTILib ShipFd.BosporusFederationRTILib= new RTILib(); // LscRtiLib declaration for
the federate
(…)//unrelated code
pointcut ReceiveInteraction(…)//pointcut definition
{
      RTILib rtiLib = (RTILib)thisJoinPoint.getThis();
      // compare received callback with federation name as there might be other federations
      if (rtiLib.federatename.compareTo("BosporusFederation")==0)
         ShipFd.ReceiveInteraction(proc); //federate method in the base code is called
}
```

Figure 3.15. A LscRTILib Definition and a Sample Advice in Federation Execution Aspect (*BosporusFederationLibAspect*)

Step iv: Editing the Computation Aspect

After running the generator, *ShipFdAspect* and *UserAspect* (generated preliminary computation) can be edited by the developer in order to effect the desired computation. Consider, for example, how ship name is retrieved from the user to send a *reserveobjectinstance* event to the federation. In the automatically generated preliminary computation, a "sample string" is sent to the *ShipFd* as a ship name by *UserAspect*. Naturally we would like the name to be entered by the user. User types in a name in the advice. The corresponding edited code is illustrated as italic form in Figure 3.16.

```
pointcut pcSendMessageInput_03e0ShipFd(…)//pointcut definition
{
   System.out.print("Name:> ");
  try {
       g_Name = in.readLine(); //name is read from console
  } catch (Exception ignored)   { }
  Name=g_Name;
  proceed(Name);
  return true;
}
```

Figure 3.16. Adding a Computation to User Ship Name Selection Method in User's Computation Aspect
(Modifications to the generated preliminary advice are in italic)

Step v: Running the Generated Code

With the editing of *ShipFdAspect* and *UserAspect* completed, the ship federate is ready to be compiled by AspectJ. Then the federate runs and joins the Bosporus Federation with the station federate joined as well. Preparation of the station federate follows the same steps. A view from the running ship federate is presented in Figure 3.17.

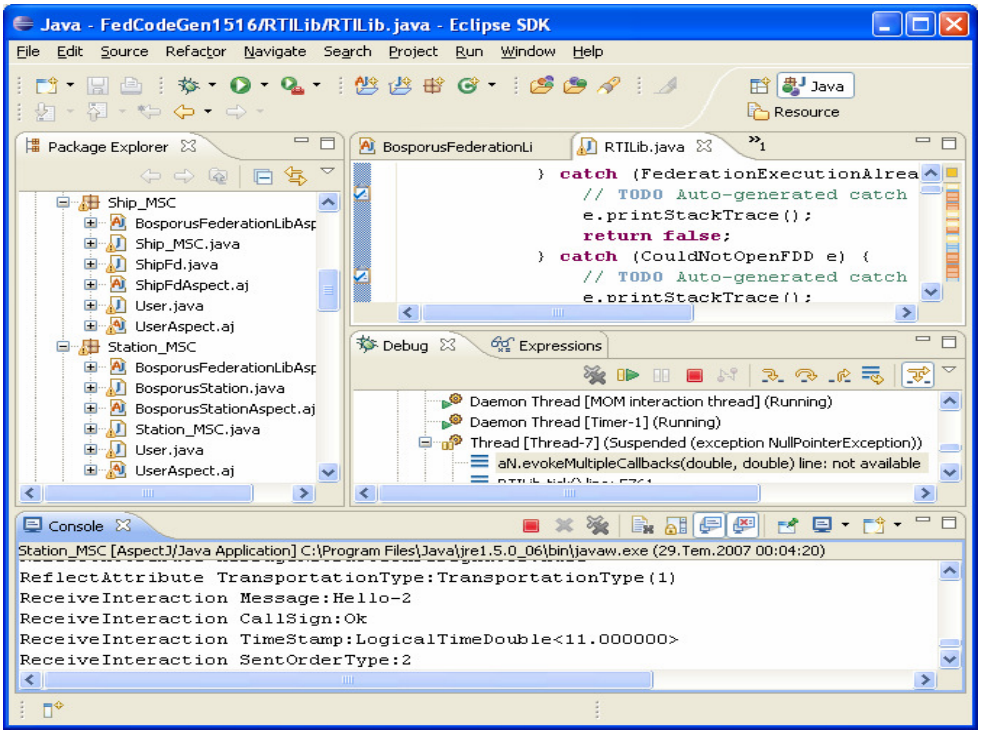


Figure 3.17. A View of the Ship Federate Running (pRTI snapshot)

3.4.2. Discussion of the Case Study

Final STMS code has over 4100 LOC (lines of code). Federate application code metrics (in terms of LOC) are given in Table 3.2 according to the generated classes and aspects, where those for *ShipFd* are shown in Figure 3.11. Manually introduced computation code (shown in column 3) is a small fraction, less than 2%, of the whole federate code (shown in column 2). This figure could be taken as an indication for the developer's manual contribution to a first-cut prototype. Of course, with a sophisticated computational logic (e.g. a high fidelity hydrodynamics model for the ship) this figure could boost up dramatically. Presumably, specialized computational codes of this nature are utilized as a library, and should not be included in the LOC count.

The automatically generated federate code size is estimated one fourth larger than that of the corresponding hand-crafted code, which does not carry any aspect-orientation

(essentially, pointcuts) overhead. Regardless of the code size, as aspect weaving takes place at compile time there is no execution time overhead.

In comparison with developing a single federate from scratch, most effort is now spent on constructing the model. The real benefit of aspect-orientation is expected to be manifest in the course of maintenance. Changes to an algorithm, for instance, can be made in one place (i.e. the relevant advice in the computation aspect) without being bothered with the rest of the code.

The metamodel enforces required references from the behavioral model to the object model (SOM or FOM). The intellectual effort to be spent by the programmer while coding the computation to keep the code consistent with the object model is saved. Changes to SOM/FOM, e.g. adding a new attribute to some object class, is reflected to the whole federate automatically. The programmer needs only to update the affected computations.

Table 3-2. STMS Code Metrics (in LOC)

Class/Aspect	**Federate Application**	**Edited Portion**
Ship_MSC (LSC Diagram Class)	50	-
ShipFd (LSC Instance Class)	1448	-
ShipFdAspect (LSC Instance Aspect)	520	27
User (LSC Instance Class)	116	-
UserAspect (LSC Instance Aspect)	134	30
BosporusFederationLibAspect (Federation Execution Aspect)	65	-
Total (Ship Federate)	**2333**	**57**
Station_MSC (LSC Diagram Class)	51	-
BosporusStation (LSC Instance Class)	1130	-
BosporusStationAspect (LSC Instance Aspect)	372	16
User (LSC Instance Class)	112	-
UserAspect (LSC Instance Aspect)	88	2
BosporusFederationLibAspect (Federation Execution Aspect)	63	-
Total (Station Federate)	**1816**	**18**
STMS Total	**4149**	**75**

3.5. Related Works

Although there are many useful HLA federation development tools in the marketplace, they fall short of generating code for federate behavior. Existing tools can generate class declarations corresponding, essentially, to the structural part of FAMM, which is the HLA Object Model. For example, most closely related tool, Calytrix SIMplicity [Simplicity 2007] does not support code generation for dynamic behavior of federates. Because, from a model-based point of view, no workable metamodel that accounts for federate dynamics and supports executable code generation was available until FAMM.

A promising approach to federate and federation development is component-based development, as proposed by [Radeski and Parr 2002]. They present a component-based development framework and a set of tools that can be used to simplify the development effort. The simulation framework provides the services that enable the bi-directional communications between applications and the RTI. During the generation and implementation phase of the framework, the component descriptor is processed by a code generator to create the appropriate source file stubs. Once the stubs are generated, the developer must insert the "simulation logic", using appropriate callback methods, into the generated code. In our AOP approach, the user provides the same logic in terms of advices in the computation aspect. Further, as a benefit of our behavior modeling facility, simulation control flow is determined by the user specifically for each federate, rather than being the "integration logic" part of some framework.

In [Yuan et al. 2003], a framework for designing and executing parallel simulation using the RTI is introduced. With the code library from the framework, the modeler is able to complete the design of a parallel simulation that runs on RTI by specifying the simulation configuration and the handling detail of each event. The modeler can specify the "logical processes" in the simulation and the events that are sent or received by the LPs. The federate model hides the HLA implementation from the parallel simulation modeler. The framework incorporates automatic code generation. Code generator will generate the Federation Execution Data (FED) file and the executable federate code based on the modeler's specifications.

[Tsai et al. 2006] presents the DDSOS (Dynamic Distributed Service-Oriented Simulation) framework, which supports the simulation, development, and evaluation of large scale distributed systems such as network-centric and system-of-systems applications. The framework features automated simulation code generation from a specification. The DDSOS framework provides two layers of modeling support. The upper layer, the target system's components and the relationship among the components are specified. At the lower layer,

PSML-S (Process Specification and Modeling Language for Services) is utilized to specify the more detailed system specifications. The upper layer system architecture specified can be automatically converted into PSML-S model. Once the simulation tasks are specified in the PSML-S language, the automated code generation service can be applied to translate the processes into executable. They [Tsai et al. 2007] also present the Dynamic Service-Oriented Collaboration Simulation (DSOCS) framework, which supports the dynamic collaboration, development, simulation, and evaluation of large scale SOA systems. It also features automated simulation code generation from the specification based on the PSML-S. The relation of this work to HLA is through XMSF.

A common point in the above cited works is that each proposes a setting for simulation construction that is at a higher level of abstraction than what is offered by the HLA standard. Our present work, clearly, has no such ambitions. Additional expressive power due to behavioral description of federates with LSCs comes about at the level of abstraction provided by the standard. We contend that the abstraction issue could be addressed in a model-driven way, by means of transformations from conceptual models to architectural models.

In the recent modeling and simulation literature there have been numerous calls to apply model-driven engineering to distributed simulation systems. In particular, Tolk [Tolk 2002] publicizes the potential advantages of adopting MDA for development of HLA-compliant federations. Clearly, to realize the touted benefits one needs model-based tools, and to start building them the metamodels they rely on must be available.

[Parr and Russell 2003] also argue that applying the MDA to HLA is the next step for simulation development. UML notations, UML profiles, a component model and tools must all be developed if HLA is to align itself with the goals of MDA. HLA is technically well positioned to leverage the advantages of MDA. Our work adopts the same line, although we do not necessarily commit to UML.

At the heart of the federate code generator is our MSC/LSC code generator, which handles all the essential features required for federate communication behavior specification. Executable code generation from behavioral specifications in LSC is an ongoing quest, see [Homme and Ramsland 2003, Maoz and Harel 2006]. There is also a body of literature dealing with transforming LSCs to some executable form, in particular, statecharts [Bontemps et al 2005, Kruger et al 1999]. We favor executable code generation directly from LSC as this approach tends to yield more readable code.

To sum up, the edge the present generator has over existing efforts can be traced back to the behavioral modeling facility afforded by FAMM. Furthermore, existing tools do not take advantage of AOP, most prominently, modularity of cross-cutting concerns. In our approach to code generation, computation and communication are separated, yet under complete control of the developer.

CHAPTER IV

CODE GENERATION FROM LIVE SEQUENCE CHARTS

Generation of code for a federate's communication behavior is based on code generation from an LSC as LSC is the language we adopt for the behavioral specification of federates. Code generation from LSCs, however, is a topic of independent interest. This chapter presents code generation from LSCs in its own right from the application developer's viewpoint. Structure of the generated code, running of the generated code, integration with the domain-specific data model is mentioned mainly in this chapter. Also a running example, ATM money withdrawal is used to give the code generation process.

Automatic code generation using the system specifications is one of the important goals of the software engineering since the emergence of the third generation programming languages and the visual modeling languages. The more software engineering practices emphasize the analysis and design, the more modeling of software gained importance. Initially, visual modeling languages are used in software engineering only as a representation of the system in analysis and design. The code and its model were two different artifacts that must be treated equally. For example, maintenance of software is generally carried out in the code level, rarely in the model level. This caused a gap between a model and its code, where the latter transforms into a complete another system during the years. But soon it is understood that the gap between a model and its code is a problem area that must be addressed. The modern approaches anticipate only the maintenance of the model, where the code is generated automatically from the model. As a result, coding is leaving its importance to modeling day by day.

Visual modeling languages such as LSC are more often used in representation of the observed behavior of systems. In this respect, automatic code generation from the behavioral specification, LSC in our case, is the major goal of this study. LSCs are used to specify the communication behaviors of a distributed system. It has a powerful specification for this manner. But, in literature there is no complete code generation solution for the LSC.

LSCs are used to specify or describe the communication behaviors of a distributed system. It has a powerful specification for this manner. But, in literature there is no complete metamodel and code generation solution for the LSC. By using our solution, developer can

model the behaviors of his application in conjunction with an object model, generate the base code, and develop his application in an aspect oriented way. Basically, code generator generates code directly from LSC models. The input of the generator is an LSC model with an abstract syntax. This abstract syntax is declared by a metamodel for LSC models.

4.1. Motivation and Scope

We investigate the applicability of model-based code generation for application development—not restricted to HLA. This approach is promising in regards to rapid prototyping of an application design and semi-automated construction of applications. First and foremost, this requires the availability of suitable models, behavioral models in particular. LSCs can be used to specify the communication behavior of components of an application, which might be distributed.

A behavioral specification in the language of LSC along with an associated object model constitutes the input to the code generation process. Input model specifies the behavior and provides the data model associated with the interactions (events). The output is an application code that is capable of generating any sequence of communications conforming to the specification, but that lacks the logic to carry out the required computations for the targeted application. Thus, if it were to run as is, it would exhibit a randomized communication pattern conforming to the specification as long as it did not rely on any correctly computed value. To turn it into an appropriate application, one would need to supply the algorithms to compute the correct values. In this chapter, we elaborate our approach to code generation from LSCs and introduce a semi-automated tool that carries out this code generation process.

The generated code consists of application base code and computation aspect code, where the latter is weaved onto the former. The application base code handles the communication between the application and other external applications, which have interfaces to the application, and the computation aspect is the place where the user puts the codes for the application's algorithms for computation. To produce the intended application, the developer should edit the preliminary computation aspect. By providing a simple computation logic (e.g. line-of-sight calculation for radars), the user can obtain a prototype application. By advancing to a sophisticated logic (e.g. solving the radar equation) as required by the end product, the user can proceed with actual application development.

4.2. Context of the Generator

Adopting the Model Driven Engineering (MDE) approach, the system development process can be viewed as a sequence of model transformations [Bezivin 2005]. From this point of view, an application development essentially involves the platform independent model

(PIM), platform specific model (PSM), and the executable code. Each layer of models reflects a particular level of abstraction. A PSM is customized to specify an application in terms of implementation constructs that are in one specific implementation technology. It is clear that a PSM will only seem sensible to a developer who has knowledge about the specific platform. In our case, the PSM corresponds to an LSC model, which conforms to the LSC metamodel. Finally, code presents the application in some executable form.

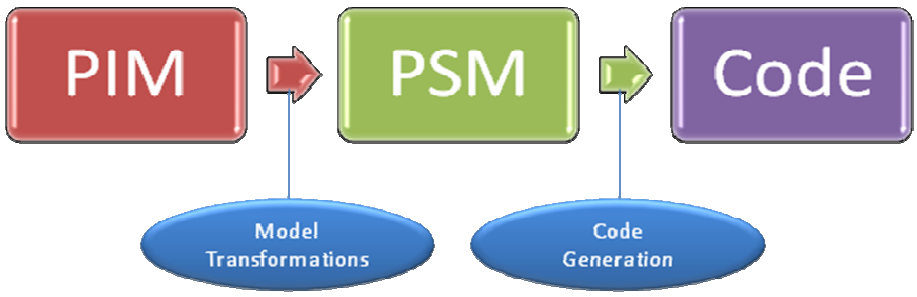


Figure 4.1Development Methodology for an Application

"Code Generation" step is semi-automated by the present work. The code generator can only provide the developer with a preliminary computation aspect, which he must edit to introduce code based on the detailed design of the intended application. Detailed design describes the computation logic of the application in addition to the behavioral specification and its object model. Object model presents the type information for the parameters of the events in the behavioral specification. Correct values of these parameters are to be computed by the computation logic. The behavioral specification pertains to the communication patterns where the message parameters must conform to the object model.

A developer can utilize our work in the following manner:

i. Model the system's observable behavior and the object model it refers to.
ii. Automatically generate code.
iii. Edit the computation aspect.

Lastly, compile and run the application.

4.3. Code Generator

The current version of the code generator handles LSC/MSC features found in [ITU-T 2004, Brill et al. 2004, Damm and Harel 2001, Harel 20001]. Implementation detail for the handling of LSC/MSC features is described in Chapter 5.

Here are some highlights of the features of the code generator:

- Asynchronous messages
- Conditions
- Inline expressions and idioms (i.e., frequently used constructs that are packaged as idioms in the LSC metamodel such as while-do)
- Timer events
- Temperature for LSC elements. Messages, conditions, inline expressions and locations, can be hot (mandatory) or cold (optional); charts can be universal or existential.
- Coregions
- Local and multi-instance general ordering
- Composition of diagrams by using the MSC references
- Local invariants
- Simultaneous regions
- High Level MSCs (HMSCs)
- Gates

In Figure 3.3 The Intermediate Form Generation module walks on the source LSC model, using the model interpreter BON2 API of GME, and constructs the intermediate form that holds the model in a convenient internal form. Then the Java Code Generation module walks on the intermediate form and produces the diagram class, the instance class, and the computation aspect. The generated codes are fed into the AspectJ compiler. The current version of our code generator generates Java and AspectJ codes. However, only the back-end needs to be re-implemented if another target language is desired.

The LSC metamodel is based on the instance-oriented textual representation in which an MSC/LSC can be completely defined by giving the behavior of each instance separately. Thus, the model traversing is instance-oriented. The code generator walks on each instance of the diagram separately and generates the instance base code and the instance aspect code (preliminary computation aspect). While generating the codes for all instances in the diagram, the generator also generates a base code for the whole diagram. Diagram base code contains instance threads that are declared and started, and the shared (multi-instance) variables, which are declared.

4.3.1 Running Example: ATM Money Withdrawal Application

In this section, how code generation is applied is showed on a simple example. In this example, the behavioral description of drawing money from an Automatic Teller Machine (ATM) is specified. First, the client inserts his bank card into the card slot of the machine.

Then, a password entry window appears and the client enters his password. If the password is valid, an operations menu is displayed. If it is not valid, the password entry window appears again. This operation can be repeated at most three times. If the third attempt also fails, the card gets blocked.

In the operations menu, money withdrawal option is chosen. A box to enter money amount is presented, and the client enters the amount. If the entered amount exceeds the balance of the client's account, the box is presented again. If the amount is less than or equal to the balance, the client draws the money from the machine. The operations menu appears again to take the next request. Finally, the client selects the quit option from the menu, and removes his card (Figure 4-2).

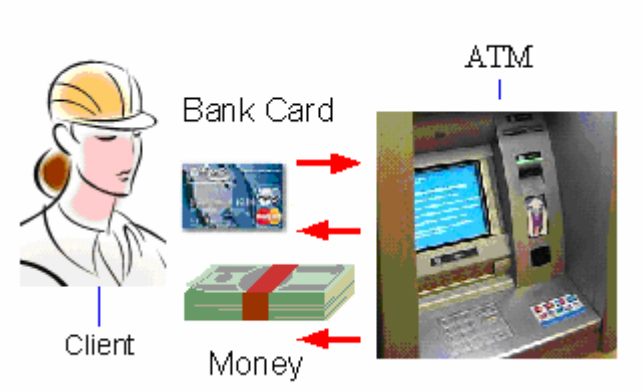


Figure 4.2. ATM Money Withdrawal Scenario Conceptual View

4.3.2. Structure of the Generated Code

The automatically generated base code consists of a diagram code, an instance code, and an aspect code (computation aspect). In the instance base code:

- Sending/receiving method definitions of events (every event in the model corresponds to a method in the code such as *SendMessageInput_0026ATM* and *RecvMessageOutput_003cATM*,
- Empty method bodies of randomization (auxiliary) methods such as *chooseOne* and *getLoopount*,
- Message queue class definitions for receiving messages,
- Dictionary definitions for temperature property of events, conditions and locations,
- Some variable definitions used for the implementation of barrier synchronization of inline expressions, general ordering of events, coregion,

- The main function of the instance (i.e., *ATMMainMethod* and *ClientMainMethod* in Figure 4.13), where method calls correspond to the LSC event sequences, is generated.

In the diagram base code, thread definitions in which instances are run; dictionary and condition definitions shared by two or more instances (multi-instance) are generated.

An LSC Instance Aspect code is generated for every LSC instance to handle the sending/receiving methods and the LSC-specific auxiliary methods of the base code for the preliminary computation. The latter (auxiliary methods) essentially help resolve, in a randomized fashion, the non-determinism inherent in an LSC specification. For example, auxiliary methods randomly determine loop counts (within bounds), choice of alternatives, order of sending events in a coregion, etc

In Figure 4.3, the structure of the generated base code is represented as a class diagram.

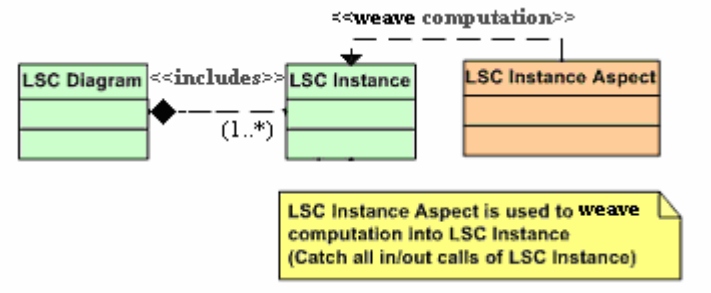


Figure 4.3. Structure of the Generated Code

4.3.3. Running the Generated LSC Instance Code Alone

This section describes how sending and receiving methods behave at run-time.

a. Handling the Sending Methods in the LSC Instance Code

In the body of sending method, an object (*LSCObject* type) is created from the method arguments. The receive method of the target LSC instance is called with the created object parameter (i.e. *ATM.RecvMessageInput_0026Client(proc)* in Figure 4.4). In the called method, sending object is received and put in the queue of receiving event of that application. (For asynchronous event receiving, a message queue is defined and used for every receiving event. See chapter 5 section 5.7 for more detail.). But in case of sending a cold event, sending method draws lots whether to send the message or not before calling. Note that in the generated code, some numeric values, such as *_0026*, are used in the function and variable declarations. Function and variable names comes from the model. In the model, name of the model element is given by the modeler and there may be model

elements that share the same names. But, in the code, it is not acceptable. So model element name and numeric model id which is created by the GME becomes unique in the model. This combination is used in the code.

```
public static boolean SendMessageInput_0026ATM(String Password_0029) {
      LSCLib.LSCObject proc= new LSCLib.LSCObject();
      (...)//proc (LSC Object) object is set.
      ATM.RecvMessageInput_0026Client(proc);
      //target (ATM) receive method is called
      return true;
}
```

Figure 4.4 A Sample Sending Method in the Client LSC Instance Code

b. Handling the Receiving Methods in the LSC Instance Code

At execution time, the receiving method of instance class first looks at the queue of the received event. If queue is not empty, the front message is taken from the queue and it is introduced to the user in a method. If queue is empty and received event is hot, receiving method waits the message until the message is put in the queue. However, if the received event is cold, receiving method checks the queue in pre-defined intervals specified as a configuration parameter. If the waiting event comes in this period, the receiving method received the event; otherwise, the receiving method stops waiting and proceeds with the next event. A sample receiving method in the client LSC instance code is depicted in the following Figure 4.5.

```
public static void  RecvMessageOutput_003cATM(Object obj)
{
 //received data is introduced to the user in the computation aspect.
  //Because of this, empty –body function is generated.
}
```

Figure 4.5 A Sample Receiving Method in the Client LSC Instance Code

4.3.4 Running the Generated Aspect Code with the Base Code

In this section, how sending and receiving methods of the base code are captured by the computation aspect and how the application logic is overridden in the aspect are explained:

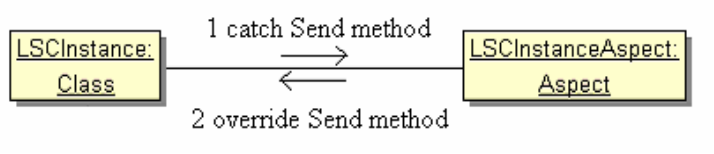


Figure 4.6.a. Collaboration Diagram of Sending Method

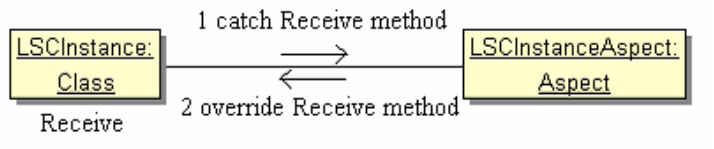


Figure 4.6 b. Collaboration Diagram of Receiving method

a. Handling the sending Methods in the Aspect

Sending method in the LSC instance code is caught (cf. Figure 4.6a, action 1) by the Aspect (computation aspect) and arguments of the method can be overridden (cf. Figure 4.6a, action 2). In AOP terminology, an aspect pointcut definition catches the joinpoints in the base code and then weaves its advice on the caught joinpoints. In our case, matching a pointcut definition is exactly one joinpoint, which is a method definition whose body is replaced by the advice in the pointcut. In our example (see Figure 4.7), *password* is edited. Then, edited aspect is weaved on the LSC instance code in the execution time and so modified password is sent.

```
pointcut pcSendMessageInput_0026ATM(String Password_0029):
execution(static boolean Client.SendMessageInput_0026ATM
(String))&& args(Password_0029); boolean around(String Password_0029):
pcSendMessageInput_0026ATM(Password_0029)
{
        //password is edited as the following
       String _password = in.readLine();
        password=_password; //preliminary computation
          (…)//proceed and return
}
```

Figure 4.7 A Sample Sending Method's Pointcut in the Client Aspect

b. Handling the Receiving Methods in the Aspect

Receive methods in the LSC instance base code are also captured (cf. Figure 4.6b, action 1) by the computation aspect in a similar way and introduced to (cf. Figure 4.6b, action 1) the developer. Developer can edit the advices of the methods to apply his application logic. For

example, receiving messages are printed on the console. In this example, edited aspect is weaved on the LSC instance base code at the run-time and received messages are printed on the console (see Figure 4.8).

```
pointcut pcRecvMessageOutput_002aATM(Object obj):
execution(static void Client.RecvMessageOutput_002aATM(Object))&& args(obj);
void around(Object obj):pcRecvMessageOutput_002aATM(obj)
{
    System.out.println("Received message:"+obj);    //received message is printed on the  console
    proceed(obj);
}
```

Figure 4.8 A Sample Receiving Method's Pointcut in the Client Aspect

4.3.5. Editing the Computation Aspect

In the aspect code editing process, generated base code (LSC instance code and diagram code) is not to be touched; all modifications and additions must be made on the advices of base code's methods in the computation aspect. Note that this constraint is not forced as the developer has the generated code at his disposal so that the developer should be carefully on the modification process. Also generator marks the mandatory editing points in the computation aspect by giving comments such as "must be edited". For instance, sending method of the LSC base code can be changed in its advice and the changed advice body is run instead of the method body in the base code. However, the pointcut definitions must not be touched in editing because if they change, aspect may not catch the intended joinpoint in the base code.

Furthermore, in the computation aspect, advices of the randomization methods whose empty bodies are generated in the base code can be edited and application specific logic can be replaced with the preliminary randomization logic. For example, generated preliminary random logic for alternative inline expression that selects an alternative randomly from the all possible alternatives can be edited in the advice of the randomization method (*ChooseAlt*). Instead of the random selection logic, user can select the alternative from the console (see Figure 4.9).

```
pointcut pcchooseOne(ArrayList selectedList, Hashtable orderList,String tag):
execution(static int ATM.chooseOne(..)) && args(selectedList,orderList,tag);
int around(ArrayList selectedList, Hashtable orderList,String tag):
pcchooseOne(selectedList,orderList,tag)
{
 int ch=0;
 int choice=0;
 /*Random r = new Random(123456);
 ch=r.nextInt(selectedList.size());
 choice=((Integer)selectedList.get(ch)).intValue();*/ //preliminary  code is commented
 System.out.println("enter the choice please:> "); //new logic is added
 try {
   choice = in.readLine(); // choice is read
 } catch (Exception ignored) {}
 (…)
 return choice;
}
```

Figure 4.9 Editing Auxiliary *ChooseAlt* Method in the Aspect

4.3.6. Weaving the Computation Aspect

AspectJ parser weaves the AspectJ codes on the Java base codes at compile time either as they are generated or after editing the computation aspect, and produces a native Java byte code. After compilation, the generated Java byte code can be run on the Java virtual machine.

4.3.7. Metamodel Support for Code Generation

Since the input model conforming to LSC metamodel is instance based, code generator generates instance codes (i.e. for every instance a Java class is generated) separately (see Figure 4.3). Reference usage is a key facility in traversing the model. Generator can reach desired model elements such as atoms, models, and connections in the model by using only references of them.

Using global lists such as instance list and condition list, allows and makes easy for the generator to reach instances and condition in the model by using references. For example, in instance based code generation, instance codes are generated according to the global instance list one by one. While traversing the input model, especially the nested model elements, only references of the instance are located. Generator readily reaches the global instance by using these references. In contrast, without a global instance list, an LSC instance could be met

anywhere in any nested model hierarchy level. For example, an LSC instance of the LSC diagram may be in topmost LSC diagram, or, in case the LSC instance has one or more inline expressions, it may be in one of the inline expression's operand. This situation would have caused to implement a messy case analysis for the generator to generate instance based code.

If a domain specific data model is not integrated into the model, then the code generator generates standard code for the MSC/LSC messages. However, if the domain specific messages are available as specializations of MSC/LSC message then the generator reaches them through references and generates the specific method bodies. Method parameter information and other related data is retrieved from the data model of the domain. For example, in HLA based code generation [Adak et al. 2007], the HLA methods and related data are obtained from the federation architecture model.

4.3.8. Integration with Domain-Specific Data Models

LSC metamodel, (in its pure form) only describes the receive and send events but it does not deal with the internal parameters of the events. These internal parameters of the events can also be modeled as a domain specific data model. We provide an integration interface to the domain specific data models. However, if the domain specific data model has a metamodel, integration of LSC model and data model is possible. In this case, generator can retrieve parameter information of the events from the data model.

Furthermore, in case parameter values are entered to the model, these values are put in the computation aspect as preliminary in the code generation. For example, we assume *SendPassword* event has a string typed parameter. The value of the parameter comes from the data model and its value is "hello world" (see Figure 4.10). Code generator, in the LSC instance base code, generates a method which has a string typed parameter and in the computation aspect, an advice that overrides the parameter value with "hello world".

```
pointcut pcSendMessageInput_0026ATM(String Password_0029):execution(
static boolean Client.SendMessageInput_0026ATM(String))&& args(Password_0029);
boolean around(String Password_0029):pcSendMessageInput_0026ATM(Password_0029)
{
        password="hello world"; //edit  the method of instance base code in the advice.
          (…)
}
```

Figure 4.10 Integration with Domain-Specific Data Model Example

This approach is applied in the FAMM where LSC metamodel (BMM) and HLA metamodels (HOMM, HFMM) are integrated [Topcu et al. 2007]. There are two extensions on the front end and back end modules (Figure 3.3) of the generator. In the first extension, domain specific information is reflected to the intermediate form, and in the second extension, domain specific code segments (data model related) are produced form the domain-specific part of the intermediate form. Further details about front end and back end extension modules' are presented in Appendix F and Appendix G respectively.

4.4. ATM Money Withdrawal Application

In this example, two instances namely a Client and an ATM is modeled. Their LSC models are presented in concrete syntax in Figure 4.11. After the proper installation, the generator can be used. In the following, code generation is described on the running example step by step.

4.4.1 Steps

Step i. Modeling

The ATM Money Withdrawal model is built as a model of the LSC metamodel.

Step ii. Configuring the Generator

In our generator, a configuration document based on XML, called *GeneratorConf.xml*, is provided with initial values for configuration parameters. Generator generates code according to the parameters which are:

a. Seed for the random number generator function for randomization process,
b. The path for the generator,
c. The path for the generated code,
d. Maximum poll count for receiving an optional (cold) message, and
e. Waiting (sleep) time between to successive polls.
f. External library name of the domain specific model.

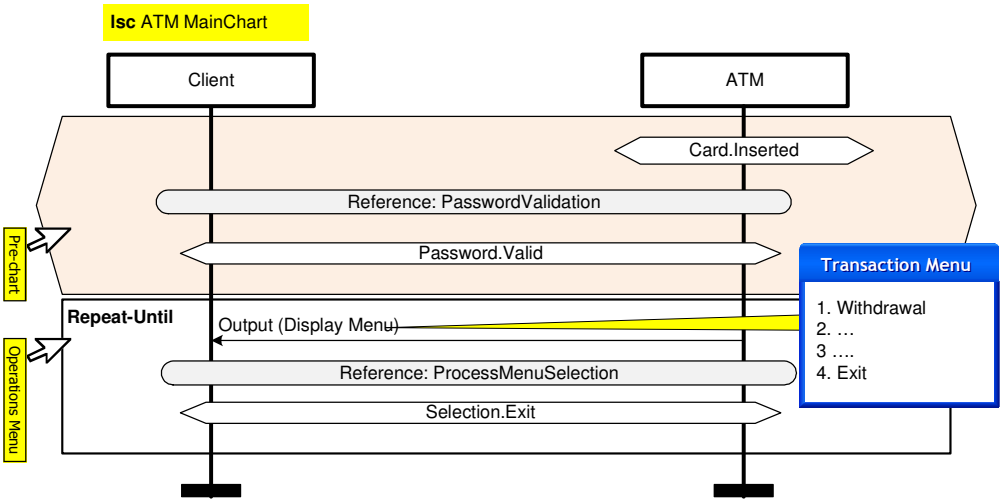


Figure 4.11. a. LSC for ATM Money Withdrawal at Topmost View

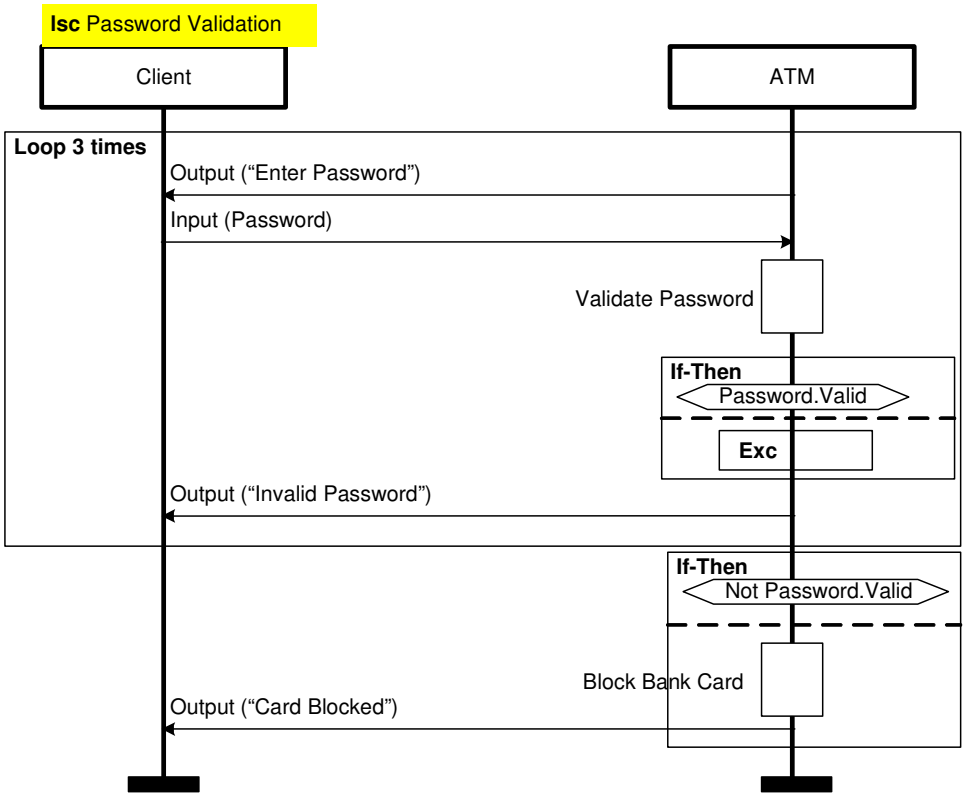


Figure 4.11. b. LSC for Password Validation Reference

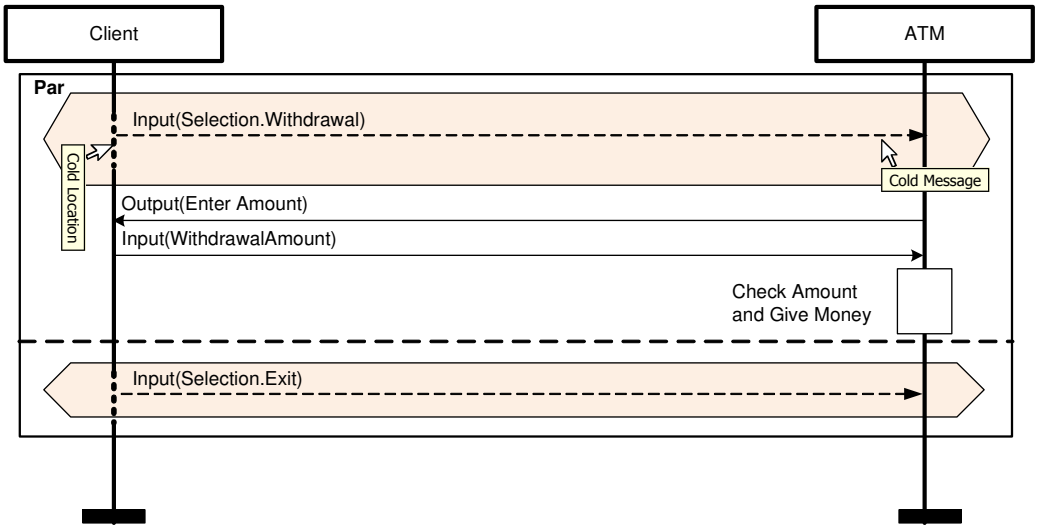


Figure 4.11. c. LSC for Process Menu Selection Reference

Our example's configuration XML file is presented in Figure 4.11 as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Confs>
<Random seed="123456">
</Random>
<Sleep time="5" passes="5">
</Sleep>
<PATH>
<Generated path="C:\eclipse-SDK-3.0.1-win32\eclipse\workspace\AtmCodeGen\">
</Generated>
<Generator path=
"c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\NewCodeGenProject\">
</Generator>
</PATH>
</Confs>
```

Figure 4.12. XML Configuration File for the Code Generator for ATM Money Withdrawal Application

Step iii. Runnig the Generator

After completing the configuration, the generator is run in GME as a model interpreter as defined in GME documentation [GME 2006]. The generated code files are placed in the folder specified in the configuration file.

In our case: *MainChart, ATM, Client, ATMAspect* and *ClientAspect* are generated in the folder. Three classes and two aspects are generated. Generated classes and aspects are represented in a class diagram in Figure 4.13. The class diagram reflects the static structure of the application represented in Figure 4.3.

a) Base Code

Class *MainChart is* the diagram code where *ATM* and *Client* (LSC instances) threads are defined and run. Class *ATM* and *Client* are instance codes where sending methods, receiving methods and main-loop method (main function) of application are generated.

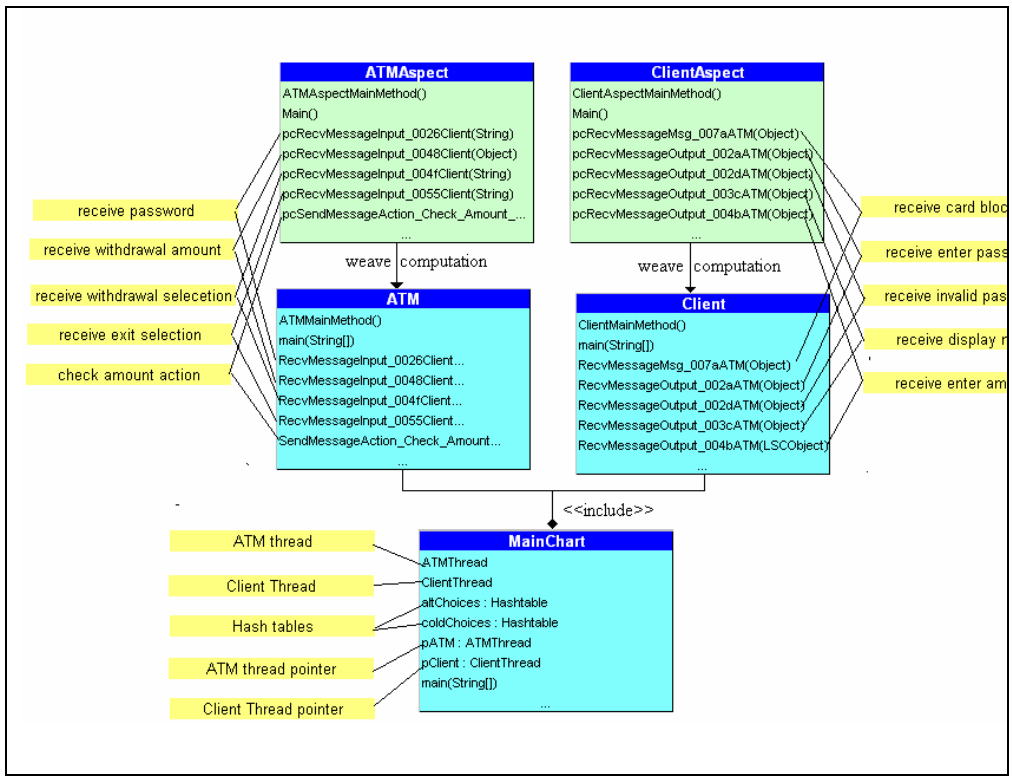


Figure 4.13. Class Diagram of the Money Draw

To give a sense of the generated code, the first do-while (password query) inline expression (Figure 4.11) of the generated *Client* code are exemplified in Figure 4.14. And also a sample sending method (*sendpassword)* (Figure 4.4) and a receiving method (*receiveloginWindow*) (Figure 4.5) are shown in the following figures.

In the operand of the do-while inline expression, password menu method is received from the ATM and user password method is sent to the ATM. Value of the do-while condition is retrieved from the dictionary named *coldChoices* by using condition model element name that is defined in the computation aspect.

```
public static void ClientMainLoop(){
 (…)//code from the beginning
            do
            { //loop, user sends password
                        condReceiveLoginWindow();
                        SendPasswordATM ("123");
            } while(((Boolean)coldChoices.get("PasswordOK")).booleanValue())
(…)//code from while-do to the end.
}
```

Figure 4.14. A Part of the Client Do-While Loop

b) Computation Aspects

Two computation aspects namely *ATMAspect* and *ClientAspect* are generated. In these aspects, all methods of the LSC instances (natural joinpoints) are accessed and their method bodies are overridden in their corresponding advices. In this computation aspect, also dictionaries and other related support methods bodies such as *chooseOne, getLoopount* are overridden in their advices of the methods.

A sample sending method's (*sendpassword*) and a receiving method's (*receiveloginWindow*) pointcuts (accessing methods) and corresponding advices are shown in Figure 4.7 and Figure 4.8, respectively. In Figure 4.7, *sendpassword* method is caught and then preliminary logic fills in the method advice. The developer can edit this method advice as described in the next section.

In Figure 4.8, the receiving method is caught from the LSC instance code and received data is displayed in the preliminary computation aspect. This received data is the values of all parameters of the method.

Step iv. Editing the Computation Aspect

After the running of the generator, automatically generated preliminary computation can be edited by the developer in order to effect the desired computation. Consider, for example, how *sendpassword* method is handled. In the preliminary computation, a sample password string ("123") is sent to the ATM, as shown in Figure 4.7. Naturally we would like the password to be entered by the user. User types in a password and hits the enter key, thus password is sent. When user types right password the do-while condition (*MSCGuard2*) becomes true and *Client* is continued. Otherwise, password entering process repeated three

times more. Corresponding code is illustrated in the following Figure 4.15. Italic codes in the figure are edited.

```
pointcut pcSendMessageInput_0026ATM(String Password_0029):execution
(static boolean Client.SendMessageInput_0026ATM(String))&& args(Password_0029);
boolean around(String Password_0029):
pcSendMessageInput_0026ATM(Password_0029) {
        String _password="";
        System.out.print("enter the password please:> ");
        try {
                _password= in.readLine(); //password is read
        } catch (Exception ignored) {}
         Password_0029=_password; //password is edited
        System.out.println(_password+" send to ATM");
        proceed(Password_0029);
        return true;
}
```

Figure 4.15.Adding a Sample Computation to the Sending Method's Advice

Step v. Running the Generated Code

With the computation aspect edited, the application code is ready to run. The code is copied into the Eclipse workspace. Eclipse is run and added codes are open as an AspectJ project. After that, code is compiled (Aspects are weaved on the LSC instance base codes by the aspect compiler) and run. A view from the running application is represented in Figure 4.19.

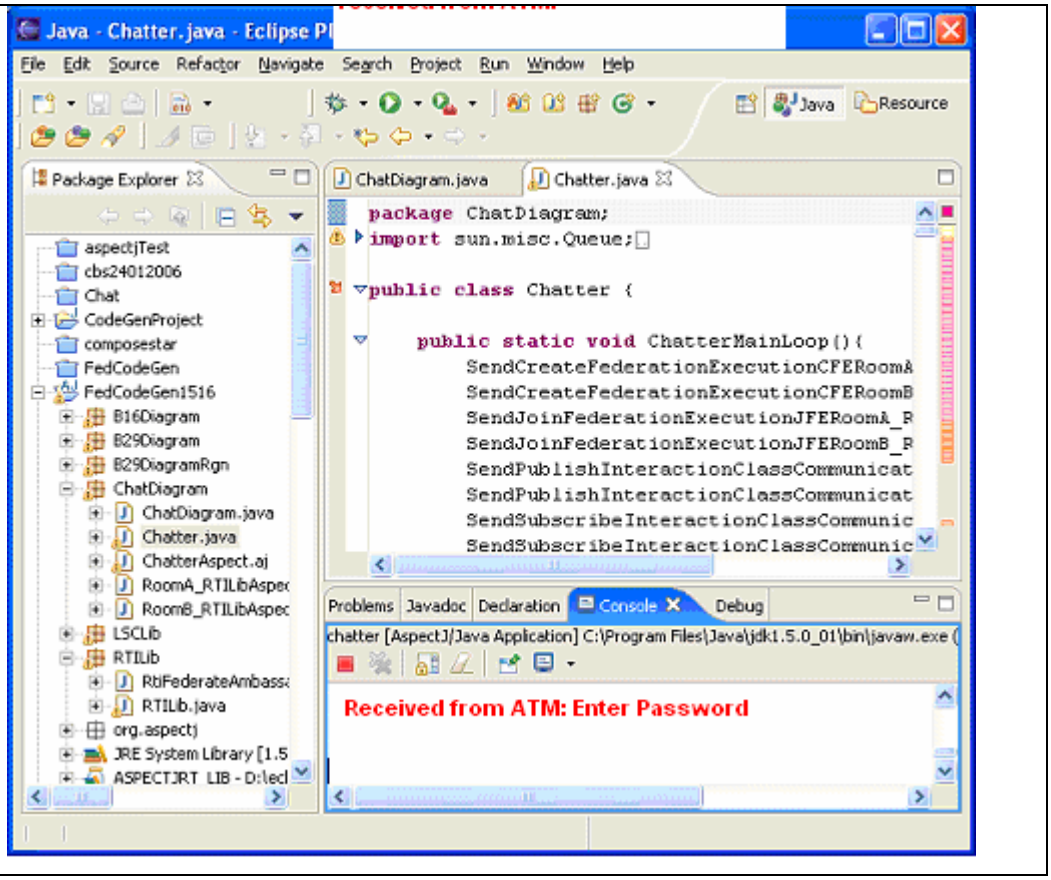


Figure 4.16. A View of the Money Withdrawal Application Running (Eclipse Screenshot)

4.4.2. Related Work

Executable code generation from behavioral specifications in LSC/MSC is an ongoing and an open challenge quest 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 [Harel 2001] as an implementation mechanism for LSC, it only provides a simulation of the execution of the LSC diagrams by playing out different 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 application code generator or interpreter in accordance with his data model. In literature, few studies address the issue of code generation from LSCs: one master thesis's [Homme and Ramsland 2003] and a new published conference papers [Maoz and Harel 2006].

The Homme-Ramsland thesis studies how code generation can be applied in different stages of system development. It explores the possibility of synthesizing system behavior in the form of state-charts from LSCs, and from there using available tools to generate the actual code. It also proposes a way of generating code directly from an LSC specification with certain constraints.

In Maoz paper, the LSC model becomes an aspect and it is weaved on the computation application, on the other hand, in ours work, computation is as an aspect and it is weaved on the LSC base code that represents the LSC behavioral model. Our approach allows complete control over the process of matching the pointcut definitions (in aspect code) with the joinpoints (in base code). This is the major difference of our work. They said that they did not support multi-threading application development in their work and it is the future work for them. However, we support multi-threading and it is the major base point for us in the code generation. While no study we perused claims to cover LSC completely, the present one covers LSC and MSC almost completely, with respect to the current MSC standard [ITU-T 2004], and our interpretation of the LSC references [Damm et al. 2003 and Brill et al. 2004. Also our domain is different from the Maoz's. Although they concentrate on scenario development and model the inter-objects behavior, we focus on outer behaviors of system rather than inner objects. In general, we look at the system from outside, they examine the system internally.

There is also a body of literature dealing with transforming LSCs to some executable form, in particular, statecharts [Bontemps et al 2005, Kruger et al 1999]. We favor executable code generation directly from LSC as this approach tends to yield more readable and traceable code. Structure of the input models is reflected in the structure of the generated code.

Furthermore, we could not see any explicit metamodel component in these related works. We hold that in any model-based approach, an explicit metamodel must be presented to the users.

CHAPTER V

IMPLEMENTATION VIEW OF CODE GENERATION

In this chapter, implementation details of code generation are presented. First, the intermediate form is described, and then, LSC constructs and their corresponding implementation approach is introduced. How the constructs are handled, how their code's are generated, how their operational semantics are considered are explained. Further details about LSC/MSC code generation of constructs are presented in Appendix A by giving module patterns and their corresponding codes. In the implementation perspective, for the federate code generation, only the LSC message model element is inherited and HLA method model is obtained from it. In this HLA method model element, method parameters are retrieved from the FAMM as explained in Chapter 3. Further details are delegated to Appendix F and Appendix G.

5.1. Intermediate Form

Basically, code generator generates code directly from LSC models. The input of the generator is an LSC model with an abstract syntax. This abstract syntax is defined by a metamodel for LSC models [Topcu et al. 2007].

Intermediate form is implemented as a dynamic list (list of *Block*) structure called *blockList*. The *Block* class has simple-typed class members such as name, type and also a dynamic list structure (list of *Operand*) called *Operands*. *Operand* class has class members such as name and two dynamic lists: Messages (list of *LSCObject*) and *blockList* (list of *Block*) for nested inline expressions. *LSCObject* class has simple typed class members such as name, type, cold condition, composed typed class members such as order and a dynamic list (list of *LSCAttribute*). *LSCAttribute* class has only simple typed class members such as name, type and value. Order class also has only simple typed class members such as owner instance name, name of the ordered event. Class diagram of the intermediate form is depicted in Figure 5.1

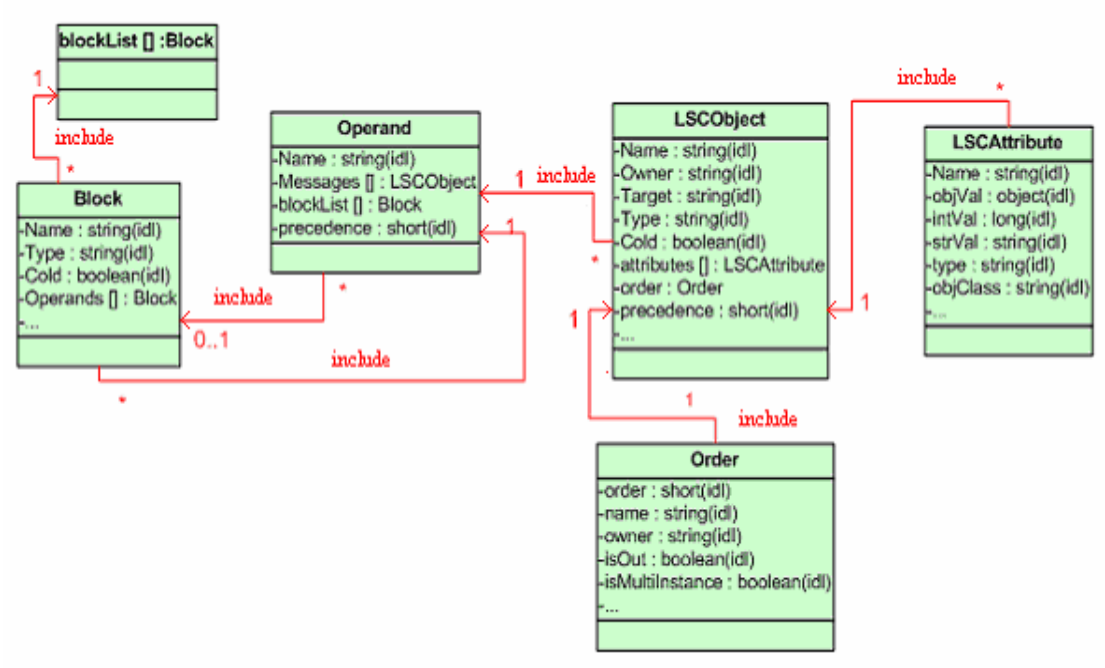


Figure 5.1 Class Diagram of the Intermediate Form

BlockList structure that holds the blocks is implemented as a dictionary data structure (i.e., a kind of dynamic list data type). A block represents a referencing environment, which constitutes a scope for a local declaration (as defined in between curly braces ("{" and "}" in Java, C and its descendents). Corresponding to an LSC chart is a block. Corresponding to each operand of a nested inline expression in the LSC chart is also a block. We consider the LSC chart as a sequential inline expression that has one operand. Thus an operand addresses a block. Operand (or block) structure holds the message list where event structures are inserted and nested-block list where nested block structures of inline expressions are added.

Events are defined in *LSCObject* class, which is the primary class for holding event information in the model. An object of this class holds mainly event name, event type such as sending or receiving, cold condition, order information and attributes/parameters of it. Attributes/parameters are defined in *LSCAttibute* class that holds the event attribute information such as name, type and value.

Order class is used for general ordering. Order class holds desired new order of the event, the owner instance of the event, whether the event is multi-instance or single-instance, name of the ordering model element and finally whether the event is sending or receiving. It is instantiated for each event that is to be ordered.

An example LSC diagram to describe the object model which corresponds to the class diagram of the intermediate form is represented Figure 5.2, in which an ALT (alternative)

inline expression has two operands. Both operands include only an event *m* and *n* respectively. LSC chart of the example (*msc A*) is considered as a SEQ (sequential) inline expression at the top of the ALT inline expression. Resulting object model of the example is depicted in Figure 5.3. Note that further details about intermediate form generation are presented in Appendix B.

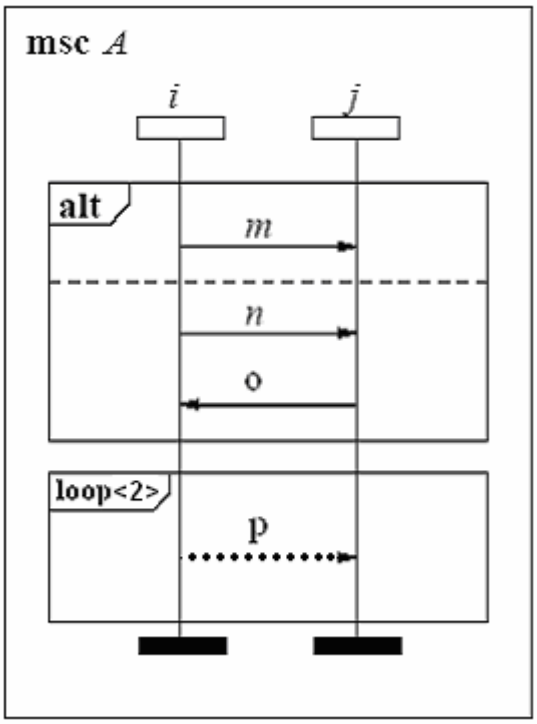


Figure 5.2 LSC Diagram of the Example for Intermediate Form Representation Aim

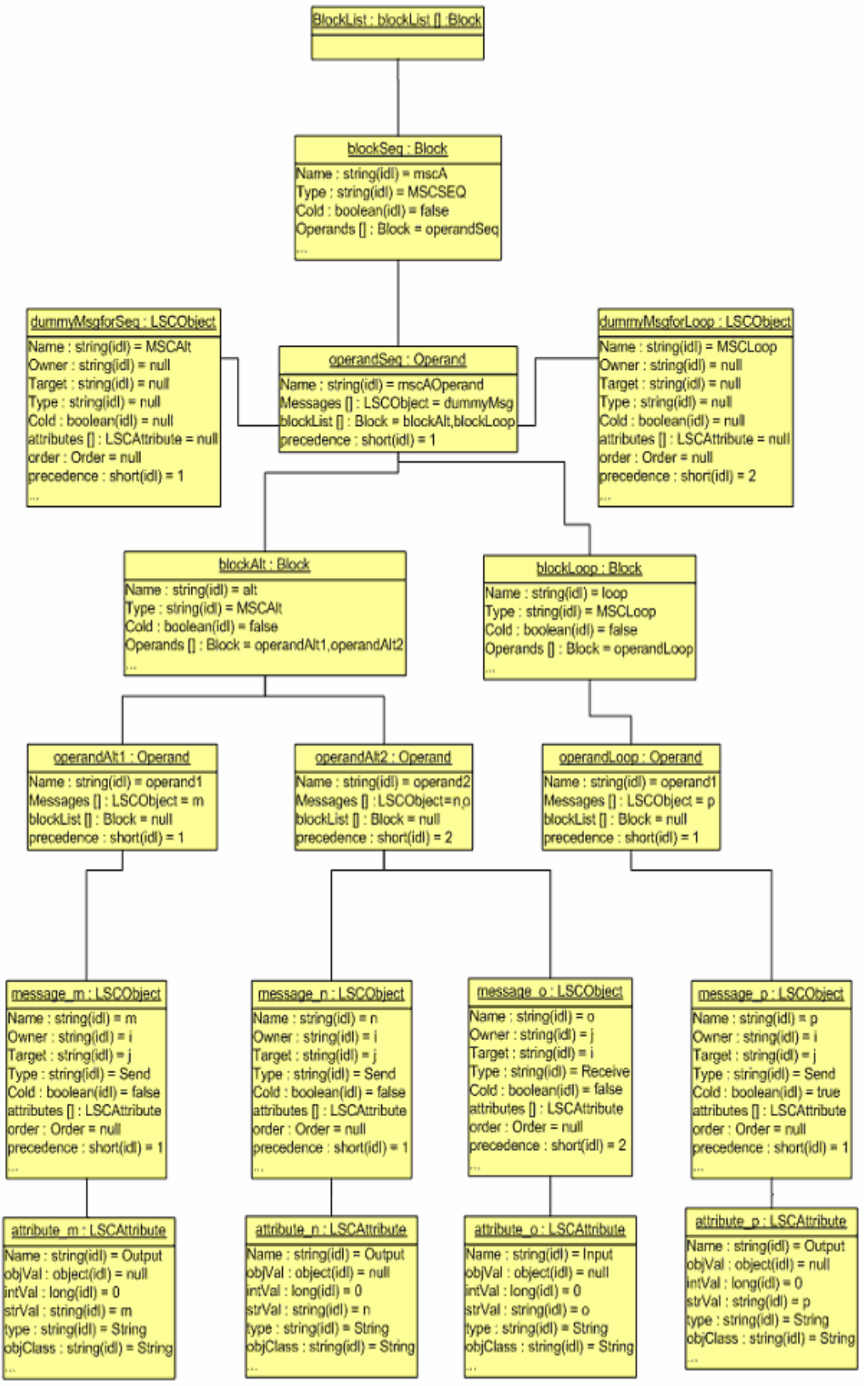


Figure 5.3 Object Diagram of the Example in Figure 5.2

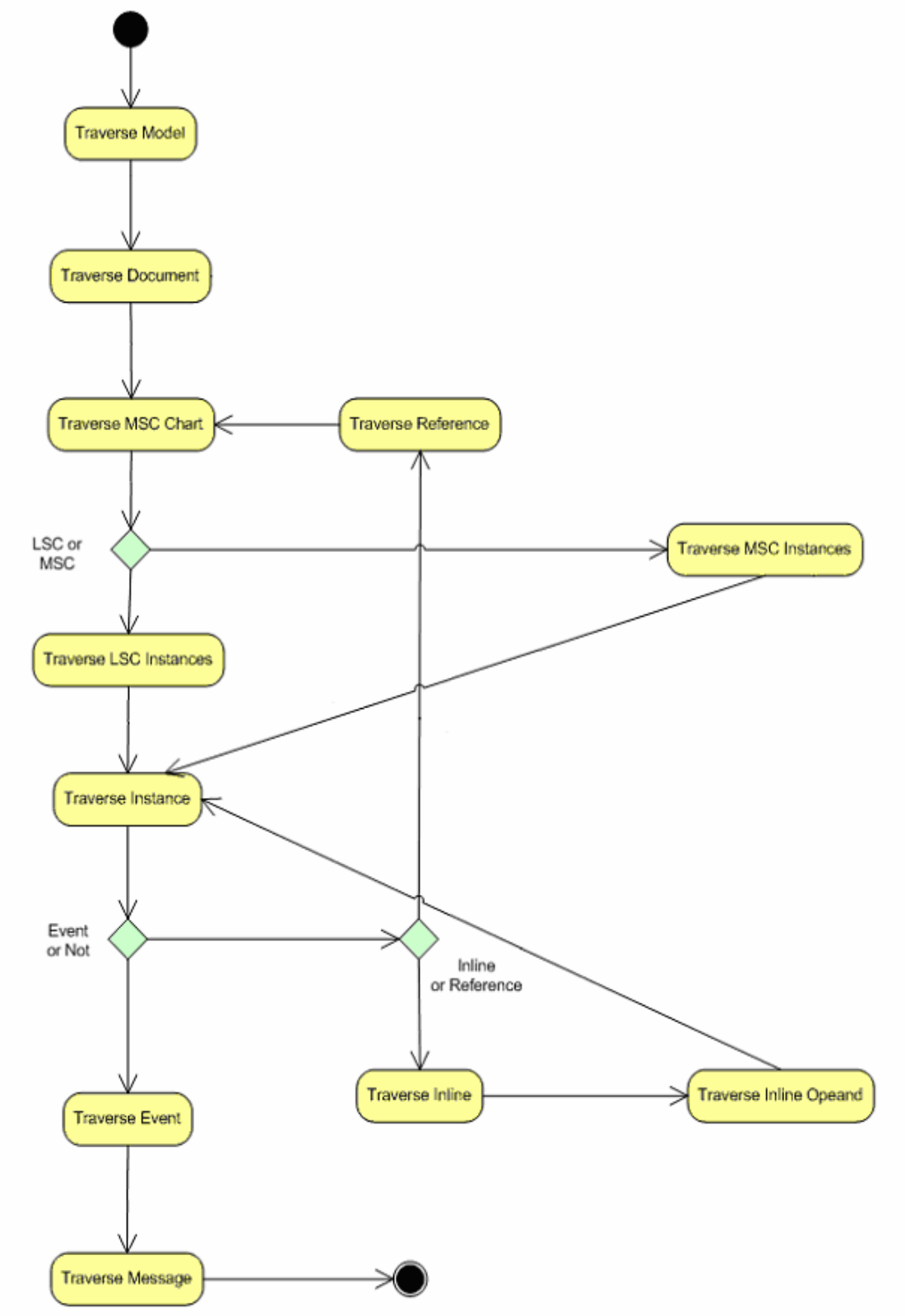


Figure 5.4. Activity Diagram of the Front-End Module

5.2. Intermediate Form Generation - Front End

In Figure 5.4, an activity diagram of the model walking module (front end in Figure 3.3) is represented. In this figure, generator starts to traverse from LSC model (top) to LSC message (bottom). For every instance in the chart (diagram), an intermediate form is constructed. There are two selections in the walking. First selection is carried out in the chart for LSC and MSC cases of the chart. Especially for the LSC case, universal/existential chart attribute of the Block object is filled. Second selection is made in the instance for event, inline expression and reference cases. When generator meets an event, *LSCObject* object is instantiated.

When generator meets an inline expression, it creates a Block object and adds the block object whose key is the precedence value of the inline expression (precedence on the connection between the inline expression and the instance in the model) into the *blocklist* for the inline expression. It also adds a dummy event into the message list (Messages). Therefore, a binary relation, such as <*dummyMsgForSeg,blockAlt*>, between the dummy event in the message list and the inline expression block in *blocklist* can be established by using the precedence value. When the Back End module meets an empty event object, it interprets that an inline expressing is handled. Note that precedence value, which is entered to the LSC model by the modeler, expresses the execution order of the event in the time slot.

5.3. Target Code Generation – Back End

Back End module (Figure 3.3) generates the Java codes from the constructed intermediate form. Activity diagram of the back end module is depicted in Figure 5.5 back end module first generates function declarations such as sending, receiving, auxiliary and timer declaration and aspect advices of the methods. After the method declaration, main function of the instance code (made up of inline expressions and method calls) is generated. For every inline expression block object in the intermediate form, different code is generated. For example, “while clause” code segment is generated from the operand (block object) of the loop inline expression. These selections are represented decision markers in the activity diagram.

When code generation meets an empty event which matches an inline expression, it retrieves the inline expression block from *blocklist* by using the empty event’s precedence value. And generator recursively continues to generate nested inline expression code by using the block.

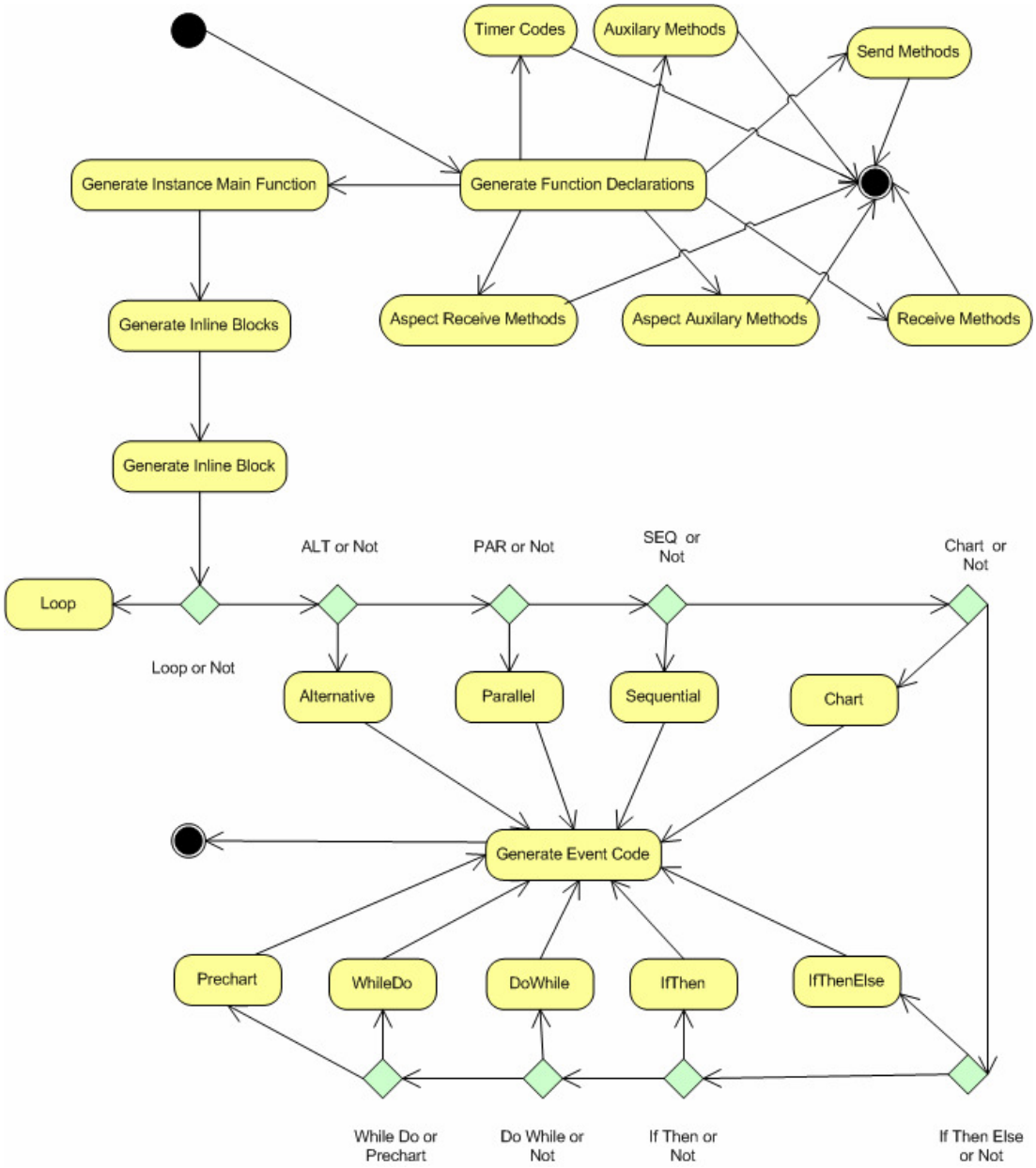


Figure 5.5 Activity Diagram of the Back End Module

5.4. Dictionary Usage in the Generated Code

A dictionary is a collection of pairs such that in each pair there is a key and its corresponding information. Java *hashtable* data structure is used for the dictionary implementation. Dictionary is used for choice operations in the generated code. Name of the model elements such as conditions, alternative inline expressions becomes a key for the dictionary.

In the generated preliminary computation aspect, this value is determined randomly. However, this value can be edited in the computation aspect by using its key. Sample usage is given in the figures of section 5.7.

5.5. Multi-threaded Realization of Instances

We employ threads for realizing instances and diagrams. For each instance and each diagram, a thread is allocated. There are two types of instance in the LSC: static and dynamic. Although static events are created at the beginning, dynamic instances are created by a *create-instance* event Threads of both static and dynamic instances are declared in the diagram base code. The threads of an static instance is started at the point of declaration in the diagram code while a dynamic instance thread is started when an LSC *create-instance* event is received, and stopped when an *stop-instance* event is received.

Moreover, threads are also used in parallel inline expression (PAR) and LSC simultaneous region implementations. More detailed explanations are given for inline expressions in section 5.10, and for simultaneous region in section 5.15.

5.6. Events

Typical events are either a sending or a receiving. Sending and receiving events are specialized such as *in, out, call, receive, replyin* and *replyout*. There are also conditions, *create-instance/stop-instance* events, and timer events

A condition can be local or multi-instance. Local condition is confined to a single instance. A multi-instance condition is shared between two or more instances.

In the code generation time, when a local condition event is met, an “if clause” (see Figure 5.6) code is generated. Condition value of “if-clause” comes from the dictionary of the instance whose key is the name (i.e. *Condition (Card.Inserted)_000f in* Figure 5.6) of the condition model element. Also an “if-clause” is generated for multi-instance condition. But, in this case, value of the condition comes from the diagram’s dictionary because the diagram is common for all participant instances.

If condition event comes from an instance, condition is valid until to the end of the chart where instance is located. If the condition event comes from nested inline expressions, it is valid until to the end of the inline expression’s operand block.

```
if (((Boolean)coldChoices.get("Condition (Card.Inserted)_000f")).booleanValue())
//Condition is model element name and comes from the model
  (….)//codes in the condition block
else //for only the hot conditions
   return; //if condition is not satisfied, the nearest method is exited
```

Figure 5.6 Hot Condition Example

Instance creation may be desirable to divide the "application instance" into many sub-instances in case of instance decomposition property in LSC. Sub-instances are also implemented by using threads similar to native instances. Every sub-instance will be located in a separate thread also.

In the generated code execution time, when a *create-instance* method is called, related thread (target instance of event in the model) is started (i.e. *ClientDiagram.pClient.start()* in Figure 5.7). On the other hand, when a *stop-instance* method is called related thread is stopped.

```
(…) //other unrelated codes
ClientDiagram.pClient.start();//create-instance event
(…)//unrelated code
ClientDiagram.pClient.stop();//stop-instance event
(…)//unrelated code
```

Figure 5.7 Instance Creation/Stop Example

There are three timer events, namely *starttimer*, *stoptimer* and *timeout* in the model. A Java *Timer* class is used for timer event implementation. Java timer class definition and timer functions' codes are generated in the LSC instance code. For every Java timer, there is a timer function that runs whenever a timer event such as *timeout*, *starttimer* occurs.

In the code generation time, when a *starttimer* event's model element is met, the timer set value is read from the model and then the timer's setter code is generated according to this value. When a *re-starttimer* event's model element is met, timer's stopper and then re-starter (reset) codes are generated. Timer's reset value is treated in the same manner timer's set value.

At run time, when the timer's setting time is passed, timer's function is called. When the timer's timeout time is elapsed, timer's function is called and also a timer flag (i.e. *timerFlagTimer1* in Figure 5.8) is set to true in it. (Timer flag is a boolean variable that indicates whether the timeout has occurred or not) When reaching the timeout method execution in the LSC instance code, the timeout received warning may be introduced to the user if flag is true in the timeout method (i.e. *RecvTimer1Timeout()*).

```
doLaterTimer1(1000); // Start Timer
(...)//codes between start and reset timer events
stopTimer1();//Stop Timer
doLaterTimer1(1000); //first stop then re-start timer.
(...)//codes between reset timer and timeout events
if(timerFlagTimer1)//timer flag to be set
{
   RecvTimer1Timeout();// Timeout function
}
```

Figure 5.8 Example for Timer Events

When a lost or a found attribute of a message or a method call event is met in the code generation time; only a method definition which has an empty body is generated for the message in the LSC instance code. Action is also a kind of local event similar to lost and found. When an action event's model element is met, only a method definition is generated.

5.7. Buffering of Received Messages

Buffering is used for receiving events of instance. For every event, a FIFO (first in first out) message queue is declared and used. In the implementation, a standard queue (i.e., *sun.misc.Queue in Java)* class is used for the FIFO data structure.

When an event is received, it is put into its queue (i.e. *queMessageInput_0026Client.enqueue(proc)* in Figure 5.9). When executing the LSC model and meeting a received event, an event is de-queued. But if the queue is empty (i.e. *while (!boolMessageInput_0026Client())*), event receiving is waited in a "while" loop statement for next receiving events that are mandatory (i.e.: hot event). If received event is declared a hot event, loop is only broken when an event comes. However, if received event is cold, declared number of polls in the configuration file is applied. If this waited event is received in these polls, it is accepted. If not, loop is broken and LSC execution is continued.

In this approach, all received messages are put in a queue dedicated for the reception of this message and they are taken from the queue one by one. This capability enables the receiving asynchronous messages because asynchronous messages are sent or received at any time and only buffering is used to handle this uncertainty. Code generations for synchronous messages are left as a future work.

```
public static void RecvMessageInput_0026Client(LSCLib.LSCObject proc)
{
      queMessageInput_0026Client.enqueue(proc);
}// Received event is put into the queue

public static boolean condRecvMessageInput_0026Client()
 {
      while (!boolMessageInput_0026Client())
         SleepThread(100);
      LSCLib.LSCObject proc=null;
      try {
         proc = (LSCLib.LSCObject) queMessageInput_0026Client.dequeue();
      } catch (InterruptedException e) {
         e.printStackTrace();
         return  false;
      }
      ProcessRecvMessageInput_0026Client(proc);
      return  true;
   }//Dequeue example
```

Figure 5.9 A Buffering Example

5.8. Temperature Property

Conditions, events, charts (universal/existential), locations and timers can be hot or cold. All these are hot except for conditions in the generated preliminary computation aspect. Condition defines a block which surrounds the events on which condition is applied. The events are in such as chart and operand.

In the generated code execution time, if a cold condition is satisfied, all other events from the condition to the end of the block are executed. If it is not, the events are not executed. However, if the condition is hot and not satisfied, events in the condition block are not executed and instance code is exited from the closest method where the condition is in. This is done by generating a “return statement” (see Figure 5.6) at the end of the block in the code generation time.

Behavior of a hot/cold event is different whether it is sending or receiving in the generated code execution time. If the sending event is hot, it must be sent; but if it is cold, a lot is drawn and the event is sent or not according to the result. If a receiving event is hot, the event must be waited until it comes. But, if the event is cold, the event is waited for

according to the parameters (i.e. *50* and *100* in Figure 5.10) of the configuration file. If it does not arrive within this waiting period, instance code stops waiting and continues the execution for next event in the LSC.

```
for (int i=0;i<50;i++) //loop count retrieved from configuration file
{
 if (boolMessageInput_004fClient())//if the event in the queue, if condition is satisfied.
  {
        condRecvMessageInput_004fClient();//event is retrieved from the queue.
        break;
    }
  SleepThread(100); //waiting time comes from the configuration file.
 }
```

Figure 5.10 Receiving a Cold Event

If a chart is hot or universal, the inner events of the chart must always be executed, but if it is cold or existential, also a lot is drawn and according to the result, the events of the chart are executed or not. If a location is hot, all events on the location behave as described earlier in this section. If the location is cold, sending hot/cold events and receiving hot events behave also the same. In the cold event receiving, the events are also waited according to the parameters of the configuration file as the hot location case. But differently if they are not come in this waiting period, instance code stops waiting and do not continue the execution for next statement in the LSC. In the code generation time, a “return statement” is added to the end of the location so instance code is exited from the closest method where the events are in.

In timer events, *starttimer* and *stoptimer e*vents act the same behavior as the standard sending events. Only the timeout event acts different behavior in the generated code execution time. If a timer is cold and timeout time is elapsed, the timeout event is received and only warning message which says “a timeout is occur” is presented to the user. However, a hot timeout is violated, similar to the hot condition, it is exited from the method in which timer is active.

5.9. Resolving Non-determinism by Randomization

Non-determinism is inherent in the LSC/MSC operational semantics. For example, receiving/sending cold events, loop count in a range for loop inline expression, and alterative inline operand selection all involve nondeterministic choices.

We apply randomization for cold event sending, for alternative inline operand selection, and for loop inline expressions, fixing iteration count within the range between the prescribed min and max. The random number seed can be set in the configuration file to support repeatability. If this seed number is not set, seed number becomes the “current time” in initial.

The randomization logic is coded within the preliminary computation aspect. Methods in which randomization is applied can be edited in the corresponding advices of the methods of the computation aspect by the developer according to the appropriate application logic (Figure 4.9).

5.10. Inline Expressions

Code generation for MSC/LSC inline expressions in the LSC instance base code is explained in this section. For the alternative (ALT) inline expression, a “switch case clause” code (see Figure 5.11) is generated. In generated code execution time, according to the making choice (i.e. *Alt1* in Figure 5.11), chosen operand (block) code is executed.

```
int Alt1=chooseAlt(2,"Alt1_0065");
switch(Alt1){
 case 0:
   SendEventM(); //alternative 1
   break;
 case 1:
   SendEventN(); //alternative  2
   condRecvEventO();//
   break;
}
```

Figure 5.11 Alternative Inline Expression Example in Figure 5.2

For the parallel inline expression, a thread code is generated for each operand (i.e. *op1_0041* in Figure 5.12) of the parallel inline expression. In the generated code execution time, these threads are run in parallel.

```
class op1_0041 extends Thread { //inline thread definition for the operand of PAR.
  op1_0041 () {}
  public void run() {// thread running function
     (…)//codes in the operand of the parallel
     stop();
   }
}
op1_0041 p0 = new op1_0041 ();//thread is declared
p0.start();//thread is started
```

Figure 5.12 Parallel Inline Expression Example

For the loop (LOOP) inline expression, a "while clause" which uses the min and max parameter values, code is generated. If these parameters are "*inf*, *inf*", an infinite loop ("while(true)"); if they are "*x*, *inf*", a loop that iterates a number of times between x and infinite; if they are "*inf*, x", a loop that iterates a number of times between 0 and x; if they are "x, y" (i.e. in Figure 5.13 *x=0* and *y=2*), a loop that iterates a number of times between x and y and finally if they are "x, x", a loop iterates exactly x times is generated. All this selection is made randomly in the generated preliminary computation aspect.

Besides the Loop inline expression, for "while-do" and "do-while" structures called idiom, code is generated. For the while-do idiom, a "while clause", for the do-while idiom, a "do while clause" code is produced in the code generation time.

```
boolean loopCond=false;
int countLoop1=0;
int loopCount = getLoopCount("0","2");//loop iterates at most two times.
if(loopCount==-1)
loopCond=true;
while(countLoop1 <loopCount || loopCond)
{
        (…)//cold message sending related code
      SendEventP();
        (…)//cold message sending related code
      countLoop1++;
}//end of loop
```

Figure 5.13 Loop Inline Expression Example in Figure 5.2

In the same way, code is generated for "if-then" and "if-then-else" idioms. For "if-then" idiom, an "if clause", and for "if-then-else" idiom, an "if-then-else clause" code is produced.

For the exception (EXC) inline expression, a "try-catch" clause code is generated. According to the try body code execution, exception part is run at the generated code execution time.

For the option (OPT) inline expression, operand (block) of the option inline expression is surrounded with an "if-then" clause and according to the condition value; operand code is executed or not executed at the generated code execution time. At last, for sequential (SEQ) inline expression, all generated operand codes of the inline expression are appended sequentially.

5.11. Barrier Synchronization

Barrier synchronization is used for executing multi-instance inline expressions synchronously at generated code execution time. *CyclicBarriers* (a Java class for the barriers synchronization) are used for this aim. For every multi-instance inline expression, a *CyclicBarrier* (i.e. *RepeatUntil_02ef* in Figure 5.14) is declared in the diagram code. When an instance reaches to the end of the multi-instance inline expression, it calls *await* method so the *CyclicBarrier* blocks the thread until the all other instance's threads call the *await* method. If all instance threads reach the end of the inline expression, blocking is broken and the multi-instance inline expression is synchronized properly.

```
do {
        (…)//other unrelated code in the while block
         try {
          Ship_MSC.RepeatUntil_02ef.await();
          //do-while  (or repeat-until) idiom is blocked.
          } catch (…)
} while (!((Boolean)Ship_MSC.coldChoices.get("until_0300")).booleanValue());
```

Figure 5.14 Barrier Synchronization Example

5.12. Prechart

Pre-chart is consists of two separate blocks (pre and body blocks). In pre-chart semantic, if any events of the pre-block were executed, then the events of the body block are executed. In the generated LSC instance base code, all methods return a boolean argument whether to indicate they are executed successfully or not. To determine the occurrences of the any events of the pre block, a flag (i.e. *condPar_0040* in Figure 5.15) is used. Initially the flag is set to false. An "if-then" clause code is generated for the body block.

In the generated code execution time, returned argument of every event in the pre block and the flag is compared by OR (||) operator and new flag value is set. This is repeated to last event of the pre-part. Finally, with respect to the obtained flag value, body block is executed or not executed. .

Pre-chart is handled as an inline expression that have two operands (blocks), first operand points out the pre block, second operand does the body block.

```
boolean condPar_0040=false; //pre-chart condition flag
 for (int i=0;i<50;i++)
 {
  if (boolMessageInput_004fClient())
    {
      condPar_0040=condRecvMessageInput_004fClient() || condPar_0040;
      //cold message
      break;
    }
  SleepThread(100);
  }//cold message received in the pre-part
  if (condPar_0040) //if flag is true, body of the prechart is executed
  {//start of body
     SendMessageOutput_004bClient(new Object());
     condRecvMessageInput_0048Client();
     SendMessageAction_Check_Amount_and_Give_MoneyACTION(new Object());
  }//end of body
```

Figure 5.15 Prechart Example

5.13. Coregion

The events in the coregion happen in arbitrary ordering. Messages to be sent in the coregion are sent in a randomly generated order. In the sending algorithm, there is a while loop. In this loop, randomly a number is selected in the range of event count and corresponding event is executed. This loop iterates until the all events are executed. If sending event is cold, before sending, a lot is drawn and event is send or not in the generated execution time.

Round-robin algorithm is used to receive events in the coregion when they are come. In this algorithm, there is a while loop and in this loop, all message-in events are checked one by one whether they come in the queue or not. If an event came, this message is received from the queue. This algorithm also guarantees to give an equal chance to all events (i.e. *chooseOne* method achieves this in Figure 5.16). Loop iteration count depends on the number of hot and cold events (i.e. *nHot=3* and *nCold=0* in Figure 5.16) in the coregion. When the coregion has at least one hot event, loop iteration continues until the all hot events are received. On the other hand, when the coregion has only cold events, loop iterates finite number of times which is determined by the polling count and period time in the configuration file.

Regarding to the MSC standards, local general ordering is only applied in the coregion. In local general ordering, orders of the two events of the same instance can be changed. Local general ordering process differs whether events are sending or receiving.

For the sending events, rules that indicate the event order are defined. A rule dictates a binary relation. This binary relation says that first event occurs before the second event. It is represented as a vector (a, b). In the randomly sending events process of the coregion, first, rules are looked at, if the rules are satisfied, the event is sent. But, if any rule is violated, random selection is repeated until the rules are satisfied.

For the receiving events, before the former coming event (e1) is received, the latter event (e2) is not controlled by the round-robin algorithm whether it comes or not in the queue. (e1 > e2: means e1 occurs before e2) Therefore, after the former (e1) is received, algorithm starts to control latter event (e2). This queue control restriction does not cause any data loss because of the event queue mechanism. If the latter event (e2) comes first, it is put and waited in the queue. When the former event (e1) is arrives first, it is executed and then the latter event (e2) is retrieved from the queue and executed. Consequently, general ordering is ensured for both sending and receiving events.

```
ArrayList selectedList = new ArrayList();
(…)//other declarations such as nHot, iHot, nCold, iCold etc
nHot=3; iHot=0; //we  assume that there are  three hot sending events in the coregion
nCold=0;iCold=0; //we  assume that there is no cold  sending events in the coregion
while(iHot+iCold<nHot+nCold) //round-robin algorithm
{
        int choice=chooseOne(selectedList,null);
          // select random message from the selected list.
          // This function guarantees to select a different choice for every iteration
        switch(choice)
        {//switch
        case 1:
                SendEvent1();
                break;
        case 2:
                SendEvent2();
                break;
        case 3:
                SendEvent3();
                break;
        }//switch
        iHot++;
}
selectedList.clear();
```

Figure 5.16 Coregion Example

5.14. General Ordering

In this section multi-instance general ordering is described. Local general ordering is explained in the coregion (5.13) section. In the multi-instance general ordering (e1>e2), if the latter receiving event (e2) comes first, it is waited until the former event (e1) comes. To establish waiting, before the latter event (e2), a while loop is added to the code. Loop condition is false at the beginning. When the former event (e1) comes first, it changes the condition of the loop to true and this breaks the loop of latter event (e2). This loop condition is declared in the diagram shared by the two instances. As a result, multi-instance general ordering property is provided (see Figure 5.17).

```
SendEvent1();//former event: e1
Diagram1.setgeneralorder1=true;
```

Figure 5.17 a General Order Example (instance i)

```
while(!Diagram1. setgeneralorder1);
CondRecvEvent1();//latter event:e2
```

Figure 5.17 b General Order Example (instance j)

5.15. Simultaneous Region

Events in a simultaneous region are perceived as simultaneous, i.e. happening at the same instant of time. In other words, these events are executed at the same time and before all of them are finished, any other event does not happen. Threads are used in the simultaneous region implementation. In this implementation, each event of the region is executed in a separate thread (i.e. *SimultaneousRegion1* thread in Figure 5.18). While these events execute, other event executions are waited. After all the threads stop, waiting is ended, and LSC execution continues.

```
class SimultaneousRegion1 extends Thread
{ //simultaneous region thread.
  SimultaneousRegion1() {}
  public void run()
  {
       SendEvent1();
        stop();
   }
}
//we assumes that there are two events in the simultaneous region. So two threads is defined and started.
SimultaneousRegion1 pSimultaneousRegion1 = new SimultaneousRegion1();
pSimultaneousRegion1.start(); //thread of the first event is started
(…)//thread code of the second event.
SimultaneousRegion2 pSimultaneousRegion2 = new SimultaneousRegion2();
pSimultaneousRegion2.start();//thread of the  second event is started
while(!pSimultaneousRegion1.alive()&&!pSimultaneousRegion2.alive());
//wait for threads and other events are not executed.
```

Figure 5.18 Simultaneous Region Example

5.16. Gate

In the MSC chart structure, gates are used to send an event to outside of the chart and receive an event from the outside of the chart. Events are sent to gate and received from the gate. We also use queue solution for the gate implementation similar to event. In this solution, a message queue is defined for each gate. Sending messages are put in the gate's queue (i.e. *queMessageTextGate.enqueue(proc*) in Figure 5.19); receiving events are retrieved from the queue. These queues are declared in the LSC diagram code, different from the event's queues. Hence, all instances of the diagram can use the same gate's queue.

```
//In LSC instance base  code
public static boolean SendMessageTextGate(Object obj)
{
  LSCLib.LSCObject proc= new LSCLib.LSCObject();
  (…)//proc (LSC Object) object is set.
  MSC.RecvMessageTextGate(proc);//gate queue  is declared in the diagram base code.
  return true;
  }
//in LSC diagram base code
public static void RecvMessageTextGate(LSCLib.LSCObject proc)
{
   queMessageTextGate.enqueue(proc);//sending event is put into the queue.
}
```

Figure 5.19 Gate Example

5.17. Local Invariant

Invariant is a property that must be satisfied at each point in the interval over which it is defined. Hence, it must be checked before and after each event in the interval. In the code generation time, when a start invariant event is met, in the front of the all subsequent events, an "if-then-else" clause code is appended. In the "else" part of the clause, a "return statement" is added (i.e. *Invariant1* in Figure 5.20). When an end of invariant event is seen, appending is stopped. This invariant condition is handled similarly as the standard (horizontal) condition. Because, invariant is a kind of vertical condition.

When the generated code runs, events are executed according to the invariant condition value. If the condition is violated, the closest method of the instance base code that encloses the condition is exited.

```
if(((Boolean)coldChoices.get("Invariant1")).booleanValue()){ //invariant condition
  SendMessageOutj(new Integer(0));
  SendMessageActionACTION(new Integer(0));
}
else //if condition is not satisfied, it is aborted.
 return;
//end of invariant
```

Figure 5.20 Local Invariant Example

5.18. Namespacing

In the generated base code, every diagram code is generated in a separate namespace. In Java programming language, every namespace indicated a different folder (directory). As a result, there can be LSC instance classes named the same in different diagrams in the application.

5.19. LSC/MSC Composition

LSC charts are composed in parallel, sequentially or alternatively by using their references. In the code generation time, whenever a reference to a chart is encountered in the input model, the referenced chart model is traversed and finally code for it is generated at the point of reference in the base code.

5.20. High Level MSC (HMSC)

HMSC is a kind of representation of the composition of the diagrams by using references of them. In the MSC composition operation, LSC diagrams of the MSC/LSC document are composed in parallel, alternatively and sequentially. HMSC shows these compositions more clear, understandable and simple form. Thus, code generation for MSC composition is used for the code generation for the HMSC similarly. In the following figure, an HMSC and corresponding MSC composition model is represented.

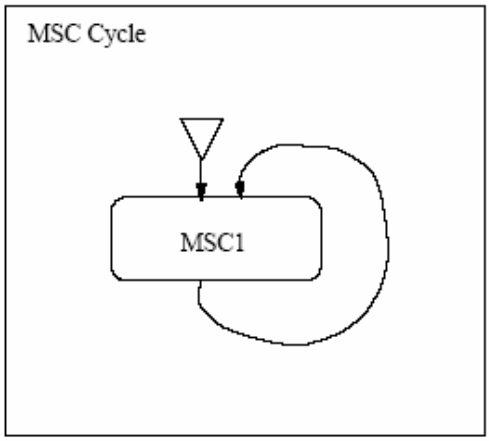


Figure 5.21 a High-Level MSC (B35 in [ITU-T 1998])

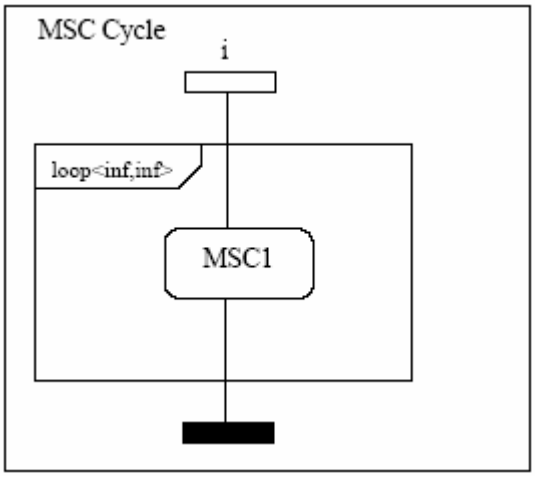


Figure 5.21 b HMSC's corresponding MSC Composition (B35 in [ITU-T 1998]))

CHAPTER VI

CASE STUDY: CONSTRUCTION OF A FEDERATION MONITOR FEDERATE

In this chapter, more comprehensive example is introduced. Firstly, the simulation system is described, and then code generation process is applied on its architecture model throughout the chapter.

6.1 Introduction to Case-study

Naval Surface Tactical Maneuvering Simulation System (NSTMSS, see http://www.ceng.metu.edu.tr/~otopcu/nstmss/) is a 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 (i.e., scenario management and simulation monitoring).

NSTMSS has been developed by using the concepts of HLA, which provides a structural basis for interoperability and reusability. NSTMSS uses Runtime Infrastructure (RTI) for data communication and object exchange, SGI OpenGL Performer for 3D graphical interfaces and virtual environment. UML has been used for Object Oriented Analysis and Design (OOAD).

Federation Monitor Federate (FedMonFd) is NSTMSS' stealth observer federate. FedMonFd enables generic data collection and reporting on HLA federates about their usage of underlying RTI services by using HLA Management Object Model (MOM) interface.

FedMonFd 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.

FedMonFd provides displays to monitor the status of the federation and the federates. Federation Monitor Federate displays the federation name, Federation Definition Data (FDD) file name used in the federation, RTI version used in the federation, federates in the federation, federation save names (i.e., Last Save Name and Next Save Name), and federation save times (i.e., Last Save Time and Next Save Time).

Monitor federate displays the info about federates in the federation. The information displayed consists of federate id, federate name, the host computer name on which the

federate is running, the status of the federate, and the federate handle, which is used by RTI. The information on the displays is automatically updated according to the update period, which is set by the user.

6.2. Federation Architecture Model Featuring FedMonFd

Federation Monitor Federate has a simple structure. Federation is connected to federate application to denote the members of the federation. Federation is connected *FOMReference*. The *FOMReference* references to MOM. Moreover FedMonFed's structure has a minor difference from other NSTMSS federates. Other federates has a connection between *FederationApplication* and *SOMReference*, but FedMonFd has not *SOMReference* so there is no such a connection.

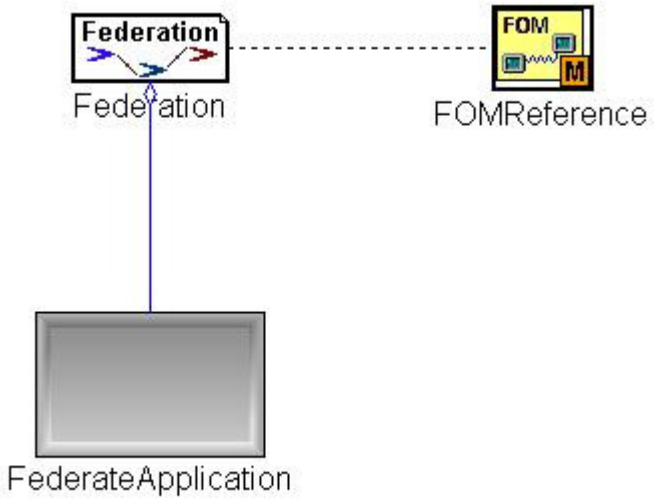


Figure 6.1. FedMonFd Federation Structure (FSMM)

In FedMonFd, IEEE 1516.1 Management Object Model (MOM) Library is used in place of Federation Object Model. This library provides the required object models for HLA MOM.

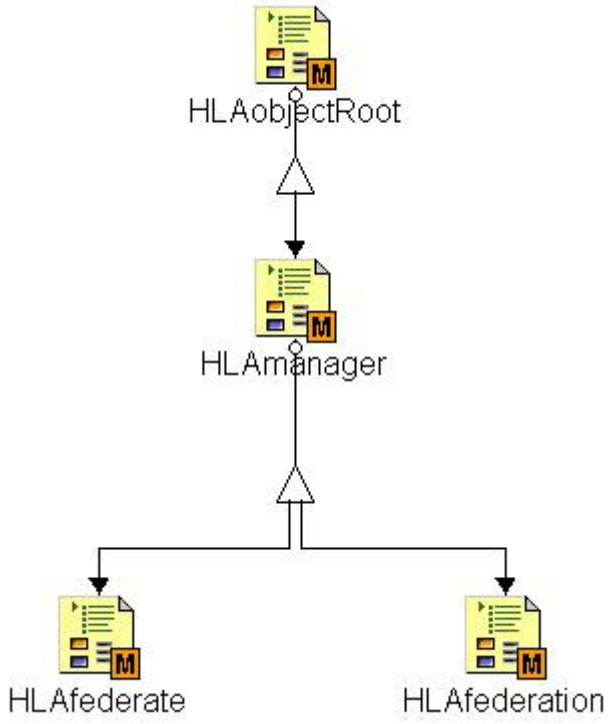


Figure 6.2. FedMonFd Object Classes (HOMM)

The Management Object Model (MOM) was designed to provide management information and control of the RTI, federation, and individual federates through objects and interactions. In addition to obtaining management information about the federation, the MOM provides federates with the ability to control the federation through interactions. Under this mechanism, the federates initiate control interactions that are sensed and reacted to by the RTI.

Using the MOM a federate can,

- Obtain management data directly from the RTI.
- Control the federation through interactions.
- Extend the MOM to provide federation-specific management functions.

MOM Objects

The MOM consists of two object classes that are used to provide persistent data about the federation, the RTI, and individual federates: a *Manager.Federation*, and a *Manager.Federate*

The attributes of *Manager.Federation* provide federation information, such as:

- Federation name
- List of federates

- FED file ID
- RTI Version
- Save Status

The attributes of *Manager.Federate* provide federate information such as:

- Federate type and ID
- Host name of computer
- Time management information
- State of the federate
- Object and interaction information
    - Number of objects and interactions sent
    - Number of interactions sent and received
    - Number of objects updated and reflected
    - Number of objects owned

MOM Interactions

There are four classes of MOM interactions:

- Adjust interactions control aspects of the federation, federate, and the RTI
- Request interactions obtain RTI information from another federate
- Report interactions report RTI data about a federate; the RTI issues them in reply to Request interactions
- Service interactions are used to invoke RTI services on behalf of another federate

The type of control available through the Adjust interactions include

- Timing of attribute updates
- Ownership of attributes
- Setting service and reporting logging

The type of information available through the Report and Request interactions include

- Subscription and publication information
- Ownership information
- Update and Reflection information
- Alert status

The types of control available through the Service interactions include

- Resignation of federates
- Saving and restoring of a federation
- Publication and subscriptions of federates
- Setting ownership and transportation of attributes
- Setting federates time management parameters

To understand the behavioral model of the FedMonFd, the legacy FedMonFd application's code is analyzed. This application was written as a one of the federate of NSTMSS application. To start out, the FedMonFd application was analyzed to understand the behavior of the Federation Monitor Federate.

In the second phase of the work, LSCs were drawn with MS Visio. In this phase, modeling was refined and got ready to transfer to the GME environment. In written FedMonFd application, DMSO RTI NG 1.3 was used. But we planned to model Federation Monitor Federate according to IEEE 1516 Standard. So a mapping was done between DMSO RTI and IEEE 1516 Specification.

In the third phase of the work, behavior modeling was realized in GME according to these drawings.

FedMonFd is constructed following parts:

- Federate Initialization
- Refreshing All Monitors
- Timer interactions
- RTI callbacks interactions
- Federate resign and federation destruction

FedMonFd behavioral model (BMM) can be seen in the following live sequence charts (in Figure 6.3).

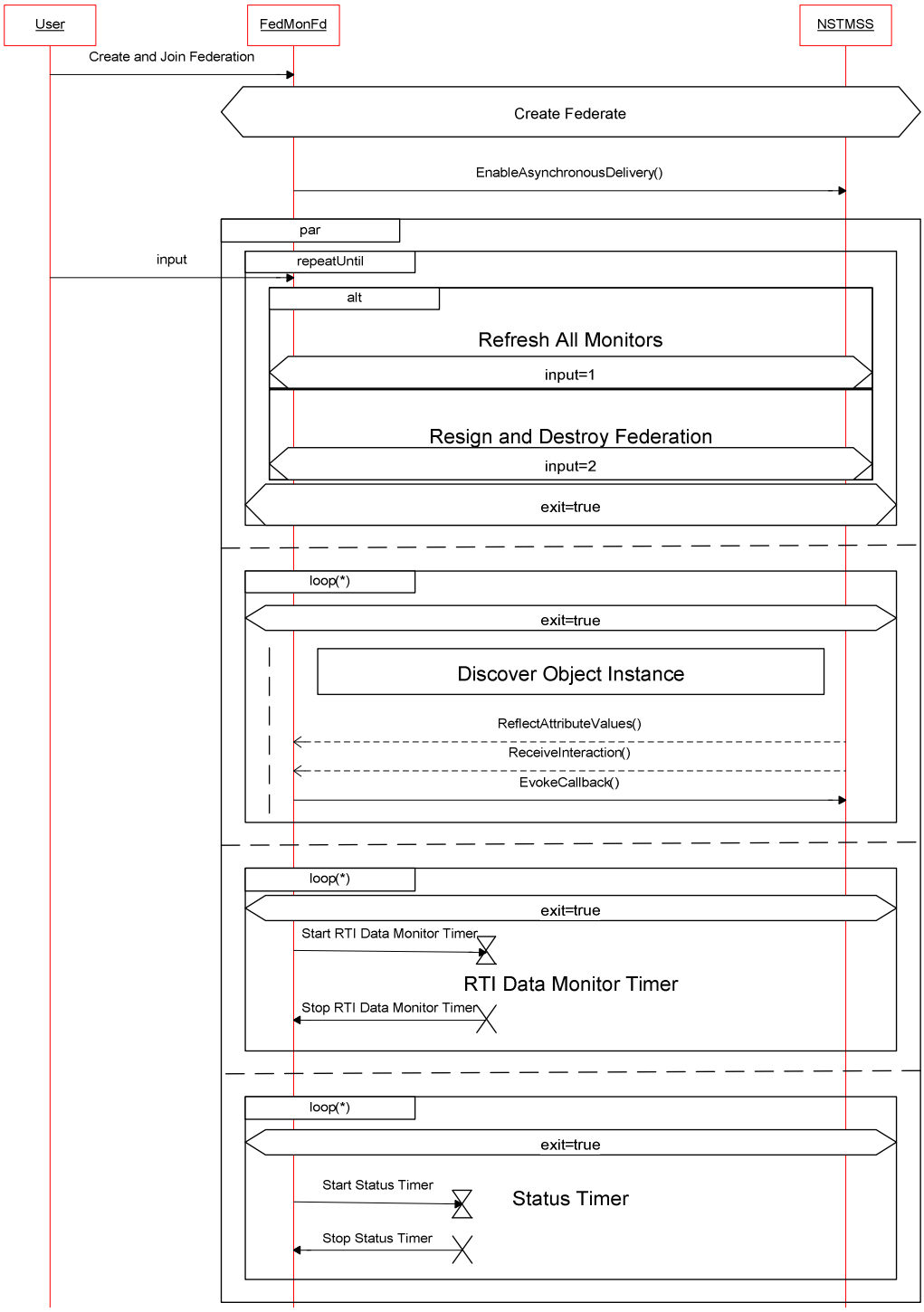


Figure 6.3. FedMonFd Behavioral Model

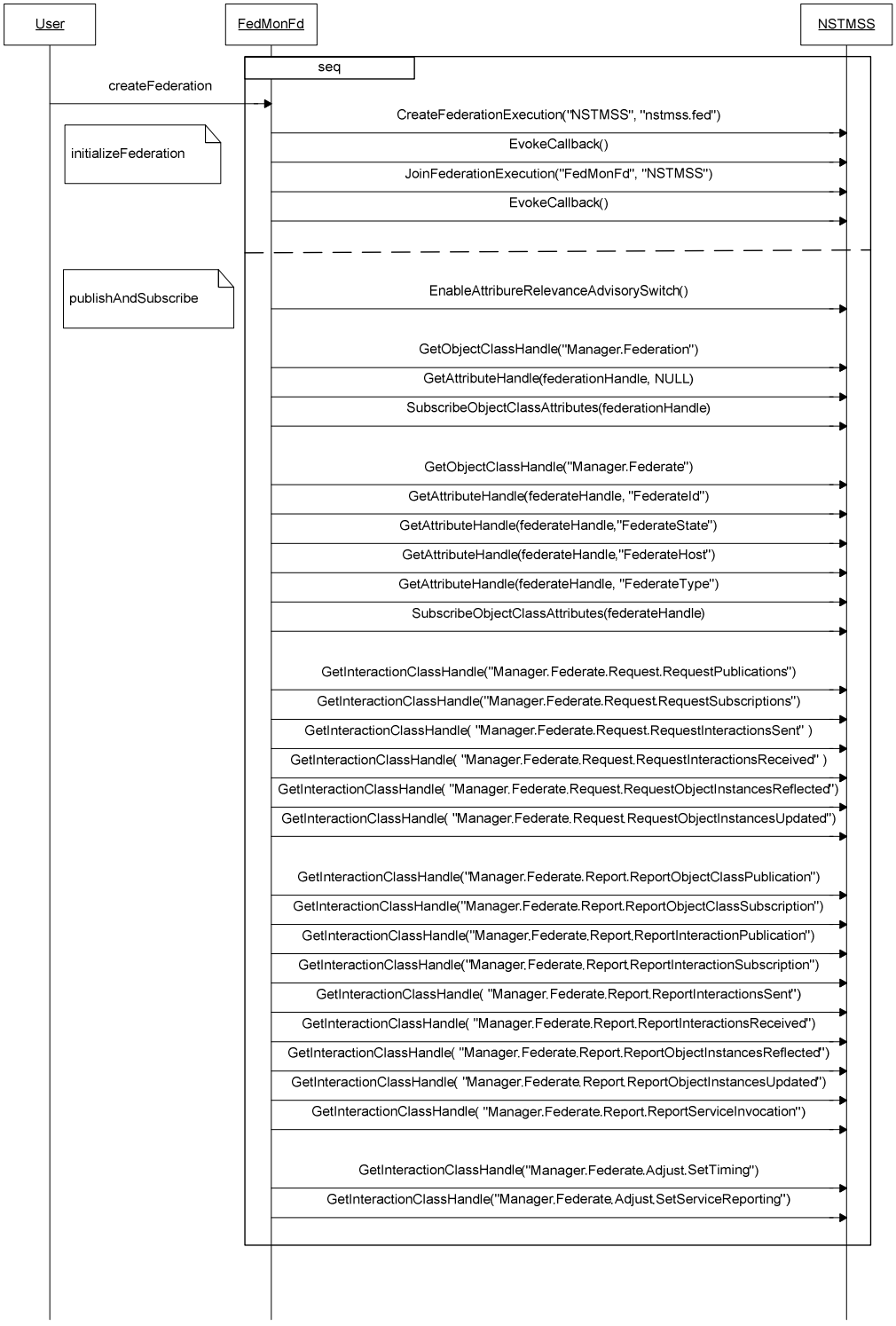


Figure 6.3 FedMonFd Behavioral Model (Continued)

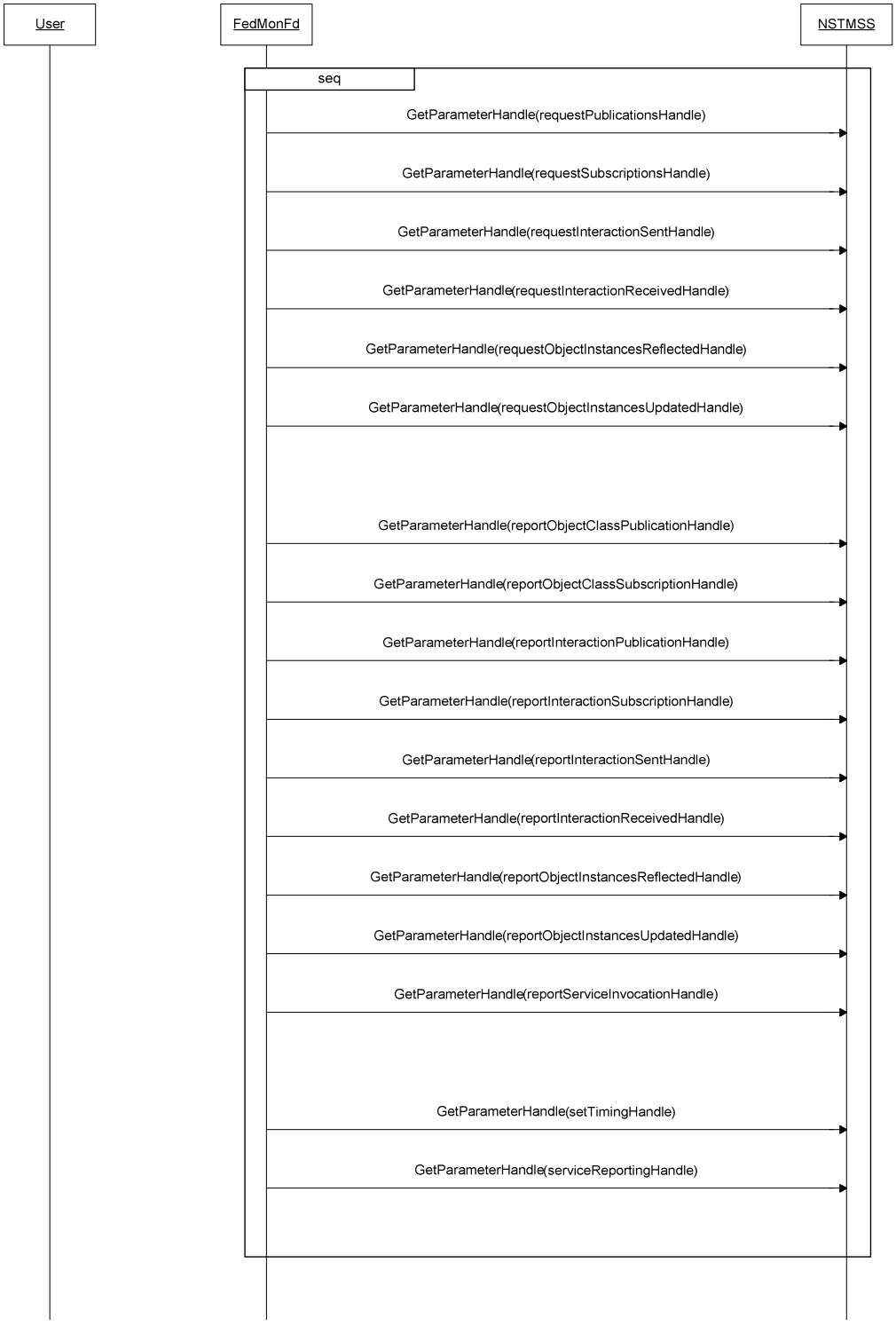


Figure 6.3 FedMonFd Behavioral Model (Continued)

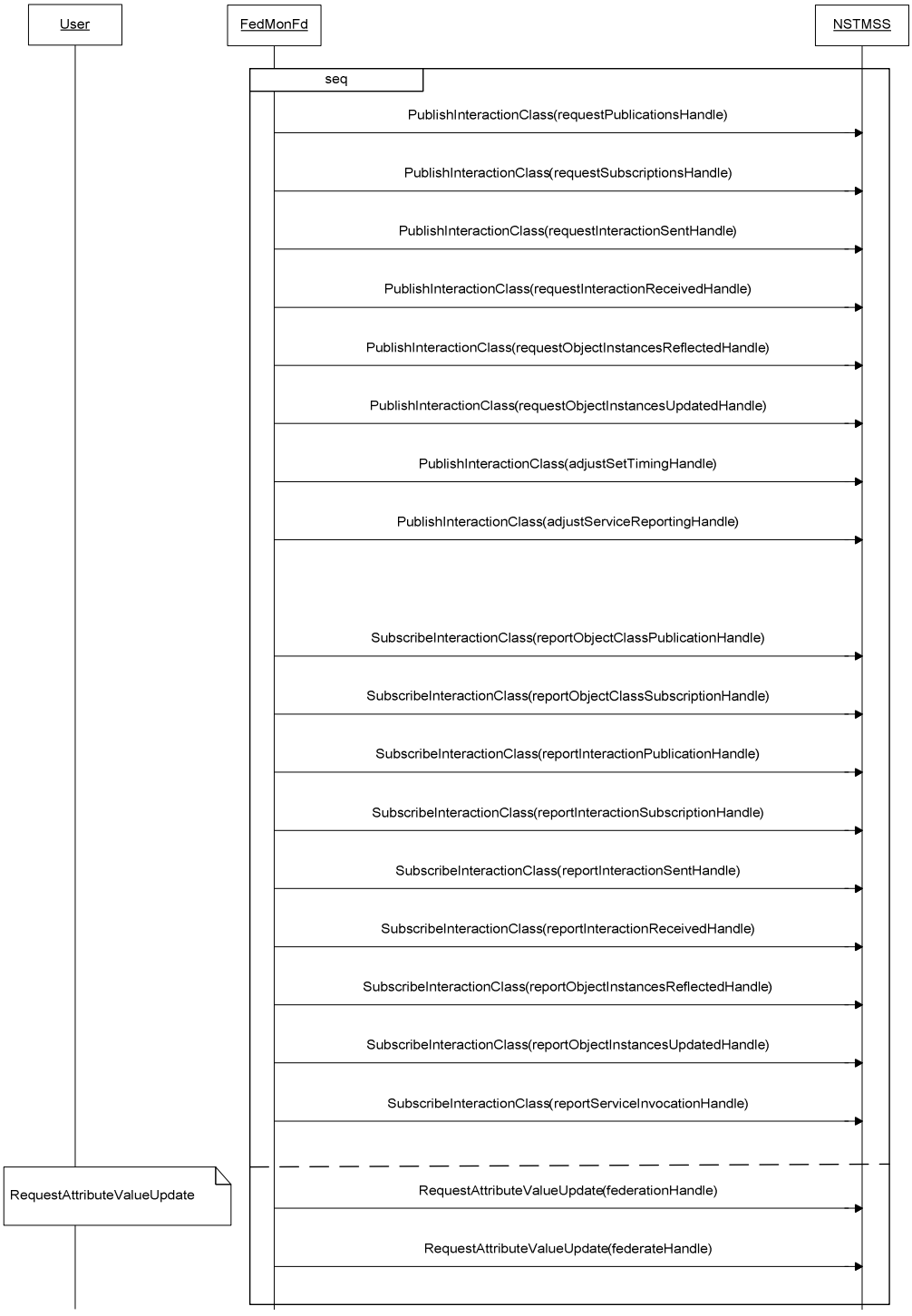


Figure 6.3. FedMonFd Behavioral Model (Continued)

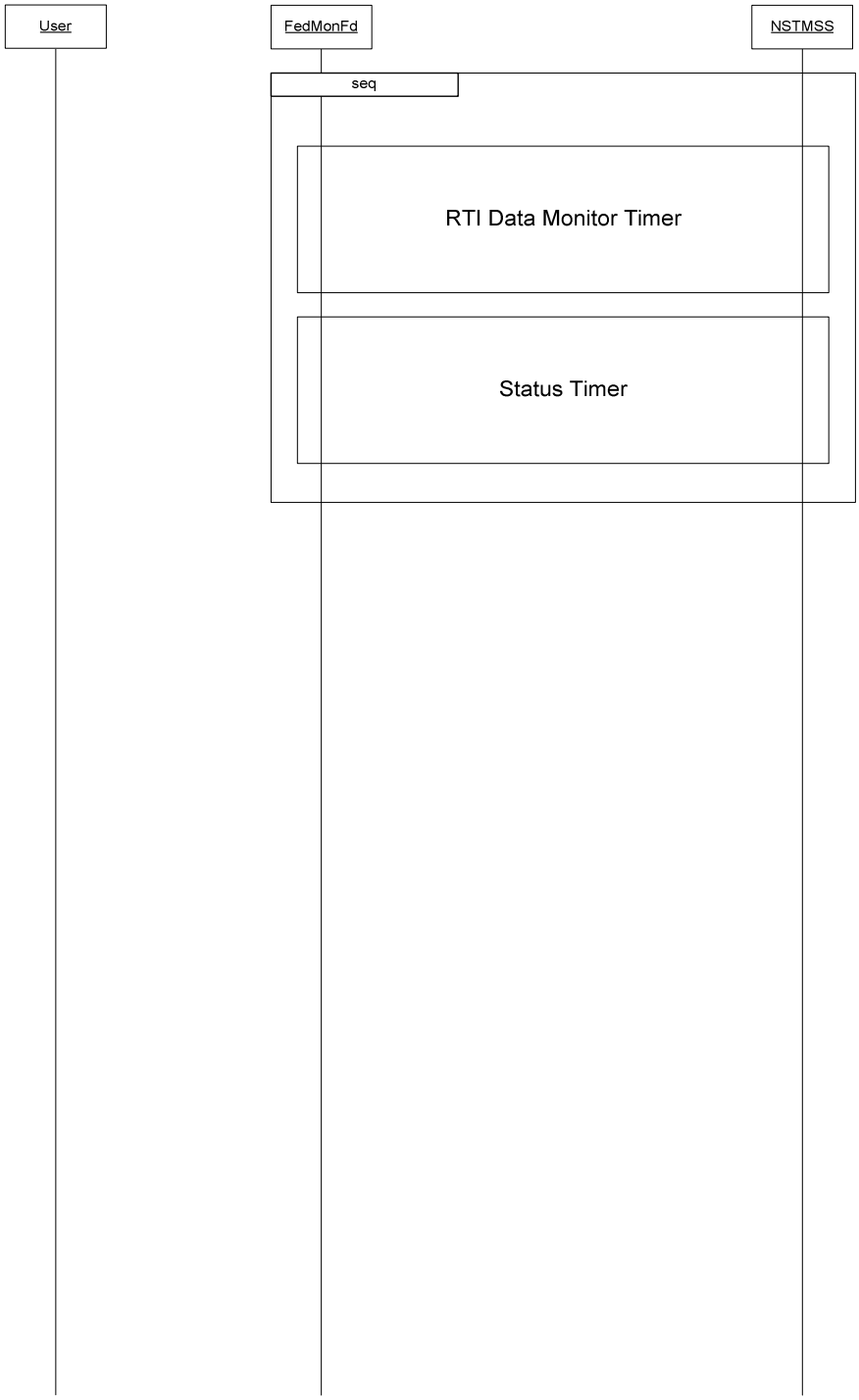


Figure 6.3. FedMonFd Behavioral Model (Continued)

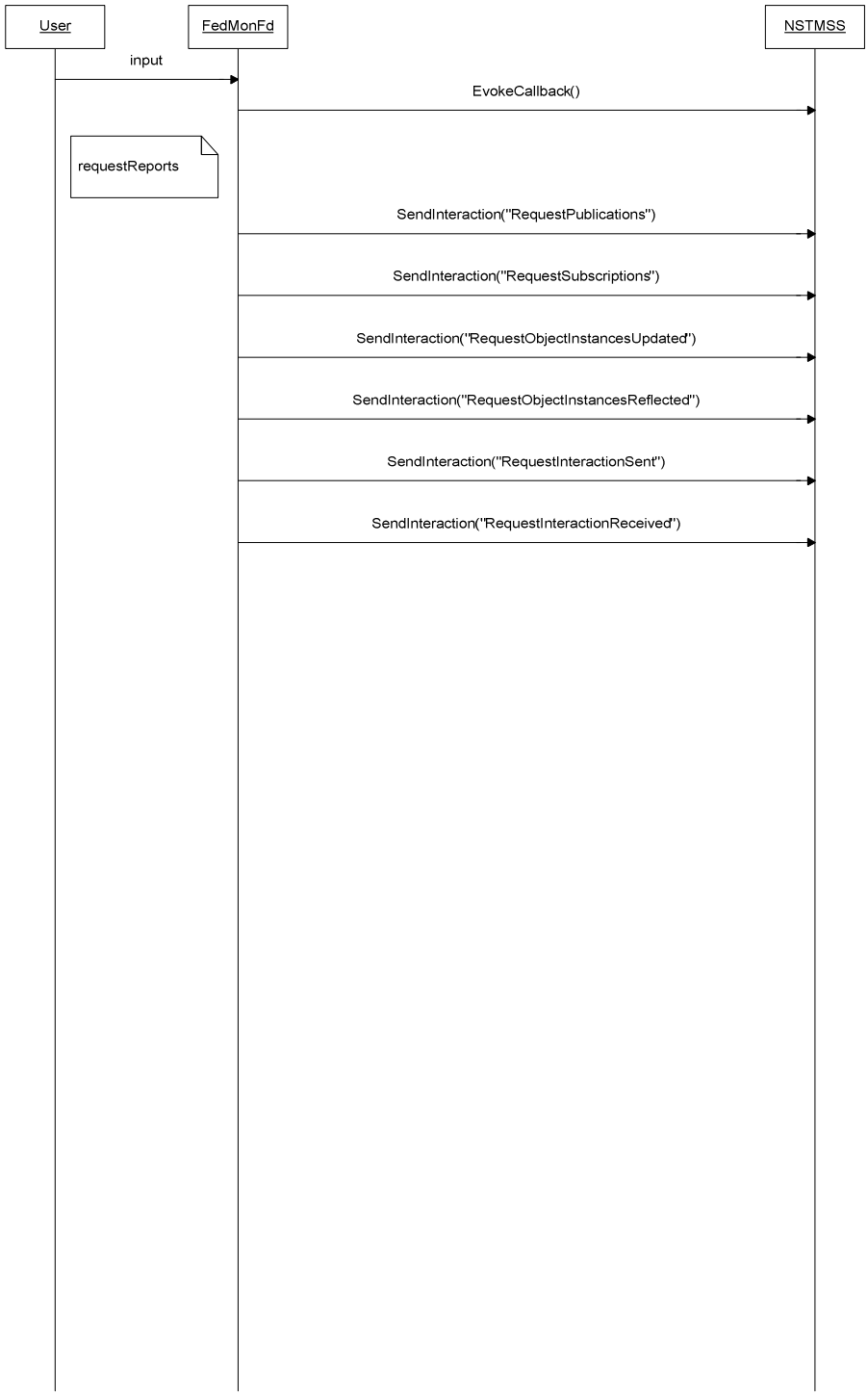


Figure 6.3. FedMonFd Behavioral Model (Continued)

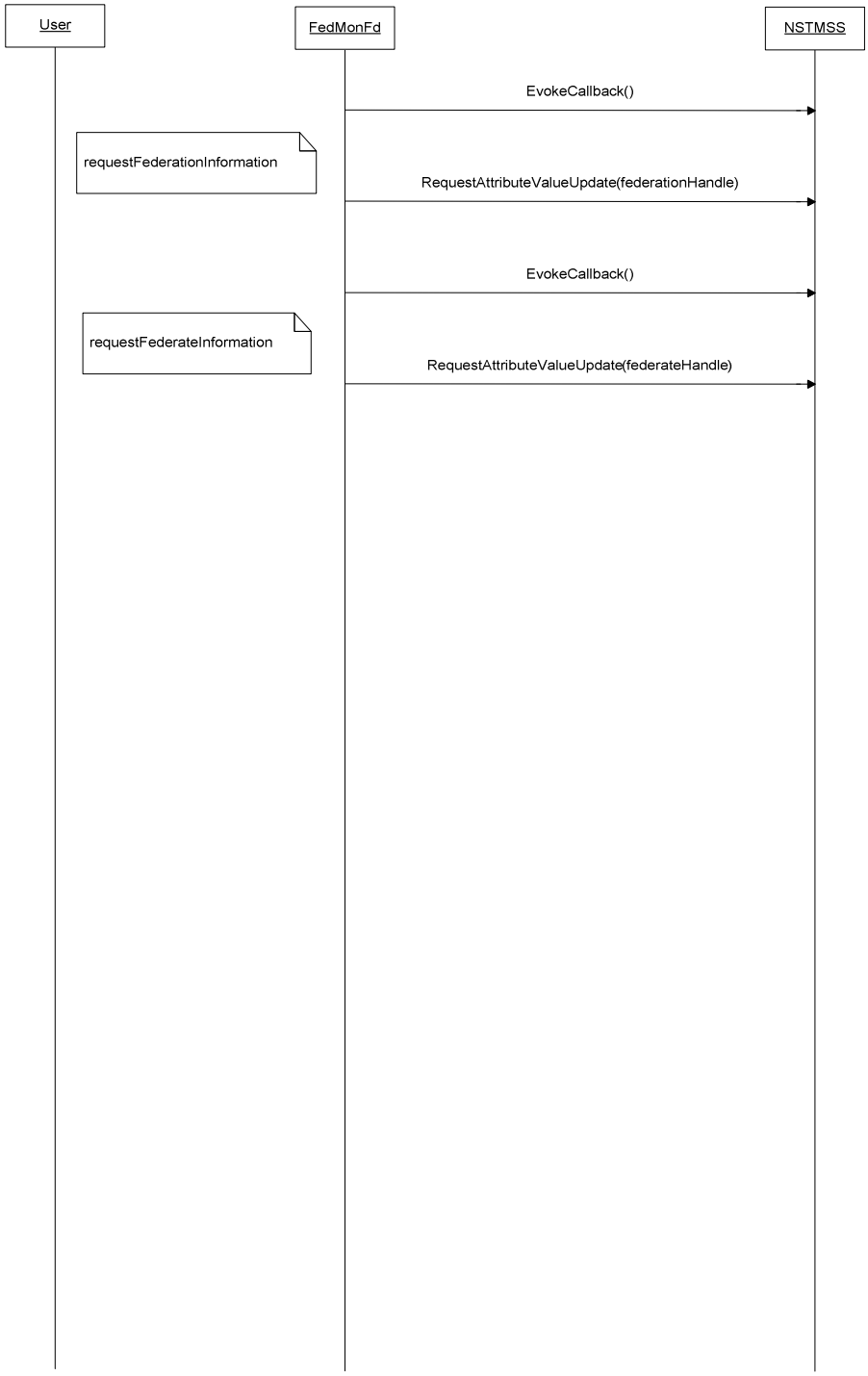


Figure 6.3 FedMonFd Behavioral Model (Continued)

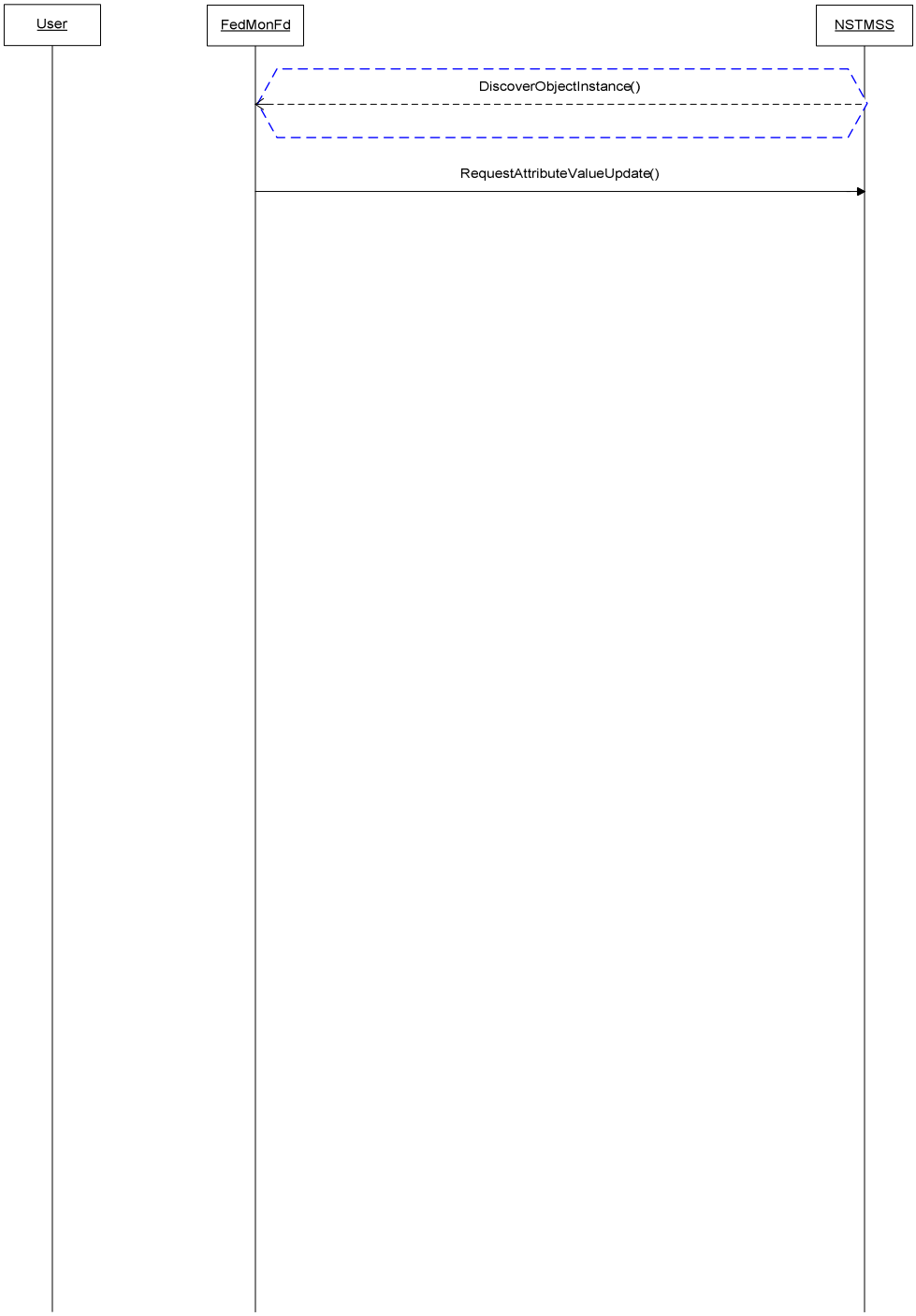


Figure 6.3 FedMonFd Behavioral Model (Continued)

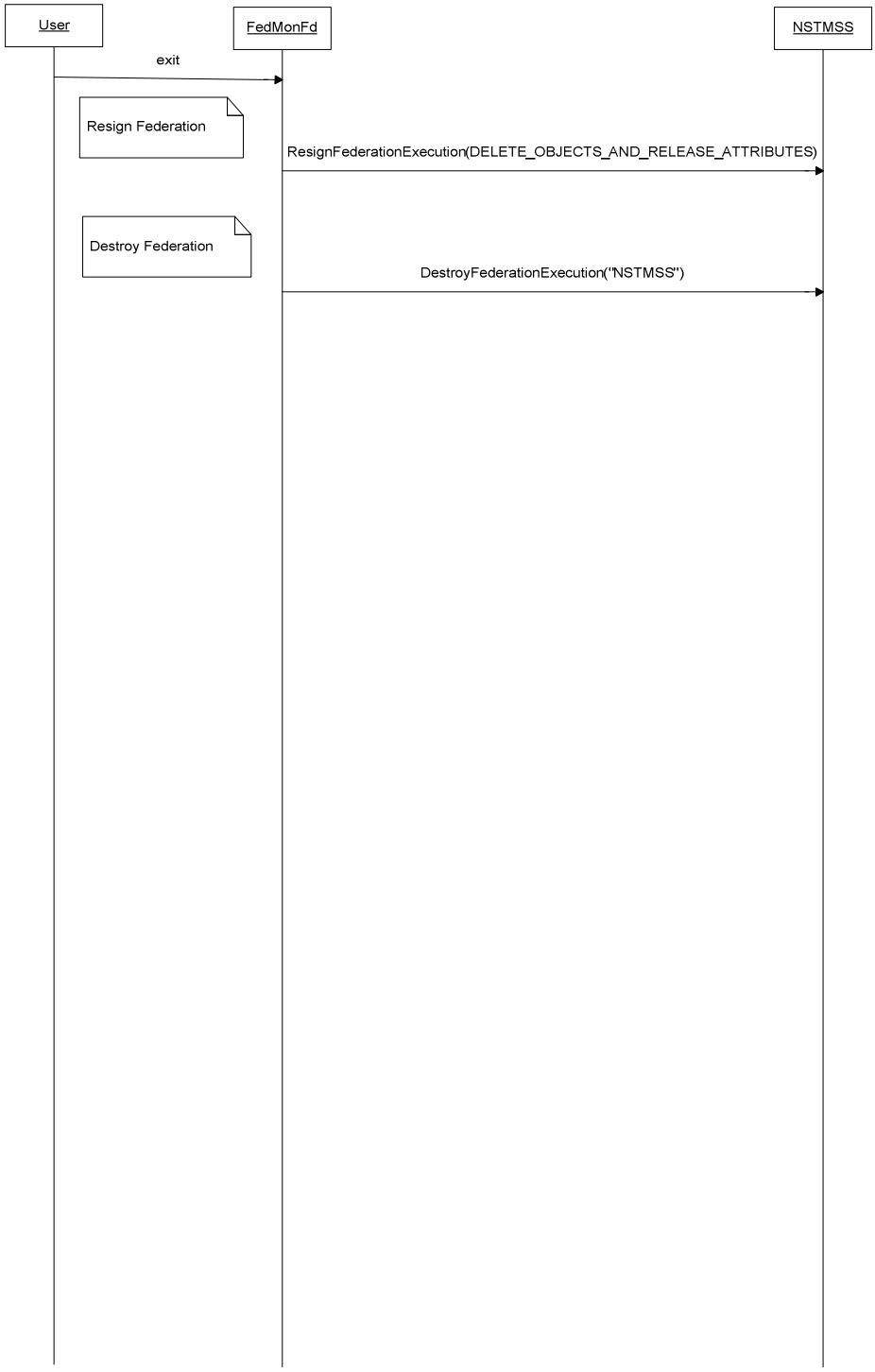


Figure 6.3 FedMonFd Behavioral Model (Continued)

Note that there is no modeling element related to user interface. After completing the last phase of the study, the generated code will have not user interface functionality. But to

complete the Federation Monitor Federate, user interface functionality must be included. For readability issues, abstraction is used. One can focus on the next phase by drilling down on the model element.

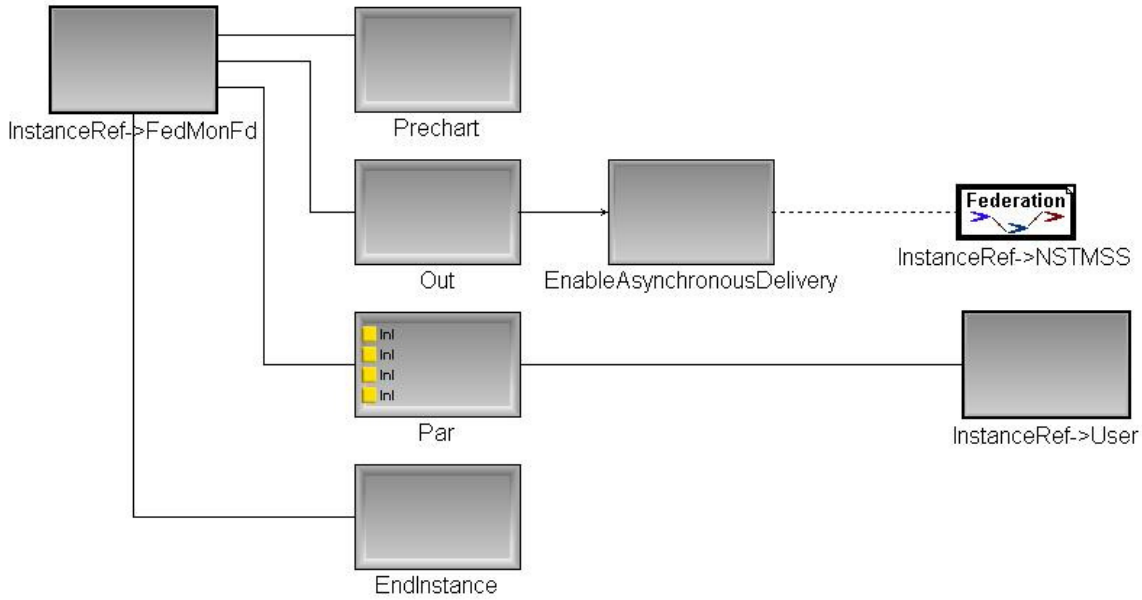


Figure 6.4. FedMonFd Main Chart in FAMM

Figure 6.5. Sequential Operator in Pre-chart in Figure 6.4

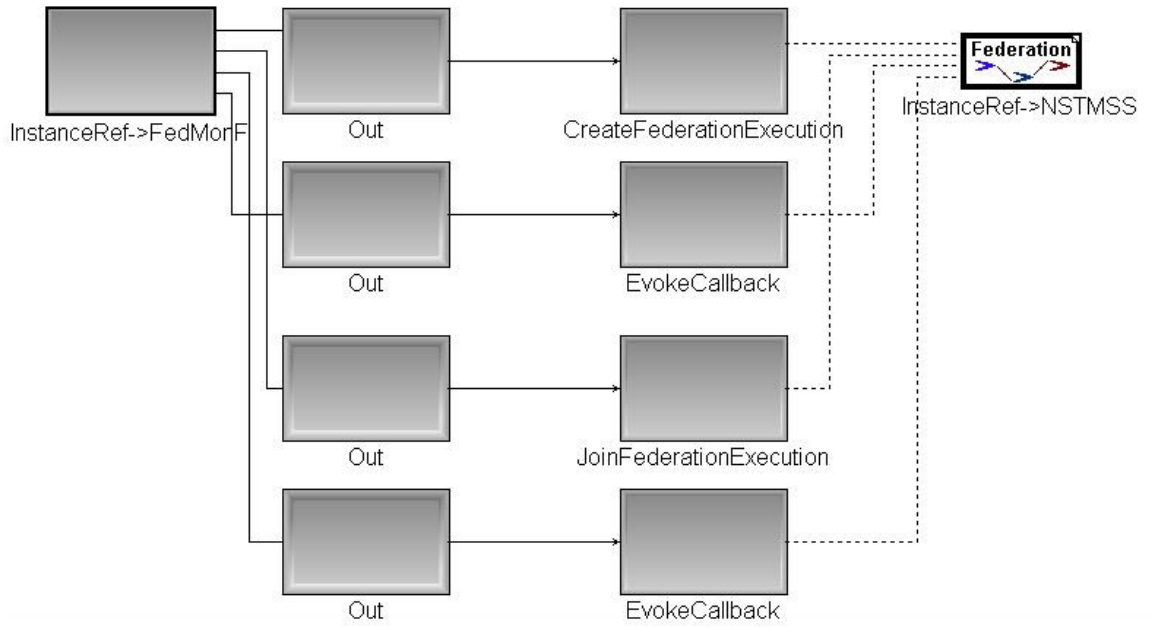


Figure 6.6. Initialize Federation Operand in Figure 6.5

6.3. Code Generation for the FedMonFd

We now represent a walkthrough of the code generation process.

6.3.1. Steps in Using the Code Generator:

Step i: Construct the FAM

The FedMonFd FAM is built conforming to the metamodel FAMM as described above in section 6.2.

Step ii: Configure the Generator

Final configuration XML file is presented (Figure 6.7) as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Confs>
<Random seed="123456">
</Random>
<Sleep time="100" passes="50">
</Sleep>
<PATH>
<Generated path="c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\FedCodeGen\">
</Generated>
<Generator path="c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\NewCodeGenProject\">
</Generator>
</PATH>
<External-InstanceLibs>
 <InstanceLib name="RTILib" prefix="RTI">
</InstanceLib>
</External-InstanceLibs>
</Confs>
```

Figure 6.7. XML Configuration File for the Code Generator for FedMonFd Application

Step iii: Run the Generator

After completing the configuration, the generator is run [GME 2006]. The generated code files are placed in the folder specified in the configuration file. In our case: *FedMonFdChart* (Diagram class), *FedMonFd* (monitor federate class), *User* (Live entity class), *FedMonFdAspect* (computation aspect of monitor federate), *UserAspect* (computation aspect of user) and *NTSMSSLibAspect* (federation execution aspect) are generated in the output folder. Generated three classes and three aspects are shown with a class diagram in Figure 6.8.

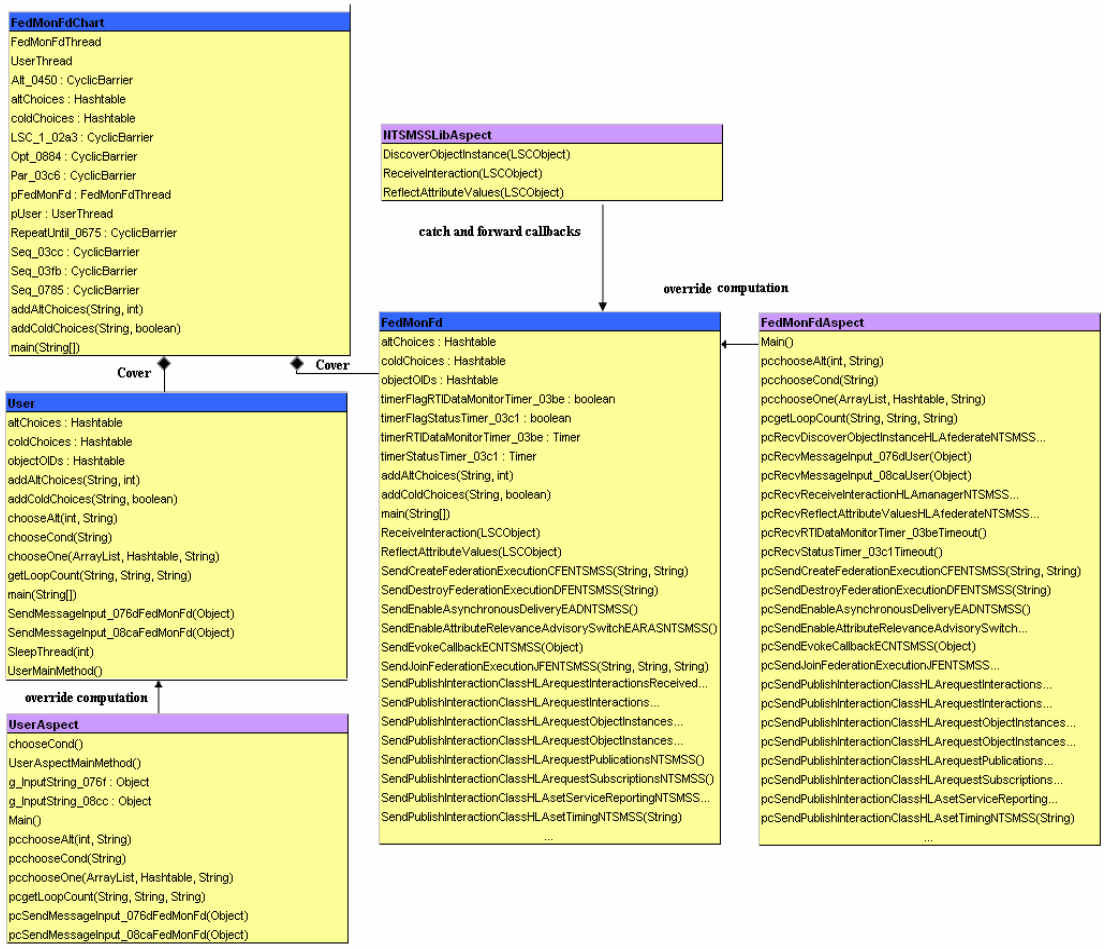


Figure 6.8. Class Diagram of the FedMonFd Federate

a) Base Code

FedMonFdChart, *FedMonFd and User* are the base code of the monitor federate application in Figure 6.8. *FedMonFdChart* is a diagram code where *FedMonFd* and *User* thread is defined and run. *FedMonFd* is an instance code where federate RTI methods, and LSC-specific auxiliary methods are generated.

```
public static void FedMonFdMainMethod() {
    if (((Boolean)FedMonFdChart.coldChoices.get("LSC_1_02a3")).booleanValue()) {
      boolean condLSC_1_02a3=false;
      condLSC_1_02a3=SendCreateFederationExecutionCFENTSMSS("s0","s1") || condLSC_1_02a3;
      condLSC_1_02a3=SendEvokeCallbackECNTSMSS(new Object()) || condLSC_1_02a3;
      condLSC_1_02a3=SendJoinFederationExecutionJFENTSMSS("s0","s1","s2") || condLSC_1_02a3;
      condLSC_1_02a3=SendEvokeCallbackECNTSMSS(new Object()) || condLSC_1_02a3;
      condLSC_1_02a3=SendEnableAttributeRelevanceAdvisorySwitchEARASNTSMSS()
      || condLSC_1_02a3;
      condLSC_1_02a3=SendSubscribeObjectClassAttributes_04b8HLAfederationNTSMSS
      ("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9") || condLSC_1_02a3;
      condLSC_1_02a3=SendSubscribeObjectClassAttributes_04bdHLAfederateNTSMSS
       ("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9","s10","s11","s12","s13","s14","s15",
        "s16","s17","s18","s19","s20","s21","s22","s23","s24","s25","s26","s27","s28") || condLSC_1_02a3;
      condLSC_1_02a3=SendPublishInteractionClassHLArequestPublicationsNTSMSS()
      || condLSC_1_02a3;
      condLSC_1_02a3=SendPublishInteractionClassHLArequestSubscriptionsNTSMSS()
      || condLSC_1_02a3;
      condLSC_1_02a3=SendPublishInteractionClassHLArequestInteractionsSentNTSMSS()
      || condLSC_1_02a3;
      condLSC_1_02a3=SendPublishInteractionClassHLArequestInteractionsReceivedNTSMSS()
      || condLSC_1_02a3;
      condLSC_1_02a3=SendPublishInteractionClassHLArequestObjectInstancesReflectedNTSMSS()
      || condLSC_1_02a3;
      condLSC_1_02a3=SendPublishInteractionClassHLArequestObjectInstancesUpdatedNTSMSS()
      || condLSC_1_02a3;
      condLSC_1_02a3=SendPublishInteractionClassHLAsetTimingNTSMSS("s0") || condLSC_1_02a3;
      condLSC_1_02a3=SendPublishInteractionClassHLAsetServiceReportingNTSMSS("s0")
      || condLSC_1_02a3;
      condLSC_1_02a3=SendSubscribeInteractionClassHLAreportInteractionPublicationNTSMSS("s0")
      || condLSC_1_02a3;
      condLSC_1_02a3=SendSubscribeInteractionClassHLAreportObjectClassPublicationNTSMSS
       ("s0","s1","s2")
       || condLSC_1_02a3;
      condLSC_1_02a3=SendSubscribeInteractionClassHLAreportInteractionSubscriptionNTSMSS("s0")
      || condLSC_1_02a3;
       condLSC_1_02a3=SendSubscribeInteractionClassHLAreportObjectClassSubscriptionNTSMSS
       ("s0","s1","s2","s3")
       || condLSC_1_02a3;
      condLSC_1_02a3=SendSubscribeInteractionClassHLAreportInteractionsSentNTSMSS("s0","s1")
       || condLSC_1_02a3;
```

Figure 6-9. Excerpts from the Generated Java Code of Monitor Federate

```
condLSC_1_02a3=SendSubscribeInteractionClassHLAreportInteractionsReceivedNTSMSS("s0","s1")
 || condLSC_1_02a3;
condLSC_1_02a3=SendSubscribeInteractionClassHLAreportObjectInstancesUpdatedNTSMSS("s0")
 || condLSC_1_02a3;
condLSC_1_02a3=SendSubscribeInteractionClassHLAreportServiceInvocationNTSMSS
  ("s0","s1","s2","s3","s4","s5") || condLSC_1_02a3;
condLSC_1_02a3=SendRequestAttributeValueUpdate_066bHLAfederationNTSMSS
 ("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9") || condLSC_1_02a3;
condLSC_1_02a3=SendRequestAttributeValueUpdate_0670HLAfederateNTSMSS
("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9","s10","s11","s12","s13","s14",
 "s15","s16","s17","s18","s19","s20","s21","s22","s23","s24","s25","s26","s27","s28")
|| condLSC_1_02a3;
if (condLSC_1_02a3) {//if clause start
   SendEnableAsynchronousDeliveryEADNTSMSS();
   class op1_RefreshAllMonitors_03d0 extends Thread {
      op1_RefreshAllMonitors_03d0() {}
      public void run() {
         do {
            if (((Boolean)coldChoices.get("Opt_0884")).booleanValue()) {
               {
                  condRecvMessageInput_08caNTSMSS();
                  SendEvokeCallbackECNTSMSS(new Object());
                  SendSendInteraction_08bdHLArequestPublicationsNTSMSS(new Object());
                  SendSendInteraction_08b7HLArequestSubscriptionsNTSMSS(new Object());
                  SendSendInteraction_08b1HLArequestObjectInstancesUpdatedNTSMSS(new Object());
                  SendSendInteraction_08a4HLArequestObjectInstancesReflectedNTSMSS
                  (new Object());
                  SendSendInteraction_089eHLArequestInteractionsSentNTSMSS(new Object());
                  SendSendInteraction_0898HLArequestInteractionsReceivedNTSMSS(new Object());
                  SendEvokeCallbackECNTSMSS(new Object());
                  SendRequestAttributeValueUpdate_0886HLAfederationNTSMSS
                  ("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9");
                  SendEvokeCallbackECNTSMSS(new Object());
                  SendRequestAttributeValueUpdate_088bHLAfederateNTSMSS
                   ("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9","s10","s11","s12","s13","s14",
                   "s15","s16","s17","s18","s19","s20","s21","s22","s23","s24","s25","s26","s27","s28");
               }
            }
            try {
               FedMonFdChart.RepeatUntil_0675.await();
            } catch (InterruptedException e) {
               e.printStackTrace();
```

Figure 6-9. Excerpts from the Generated Java Code of Monitor Federate (Continue)

```
            } catch (BrokenBarrierException e) {
               e.printStackTrace();
            }
         } while (!((Boolean)FedMonFdChart.coldChoices.get("Condition_0883")).booleanValue());
         stop();
      }
   }
   op1_RefreshAllMonitors_03d0 p0 = new op1_RefreshAllMonitors_03d0();
   p0.start();
   class op2_Callbacks_03d1 extends Thread {
      op2_Callbacks_03d1() {}
      public void run() {
         while (((Boolean)FedMonFdChart.coldChoices.get("Condition_0883")).booleanValue()) {
            int nWhileDo_06f80;
            int iHotWhileDo_06f80;
            int iColdWhileDo_06f80;
            int passWhileDo_06f80;
            int recvChoiceWhileDo_06f80;
            ArrayList selectedListWhileDo_06f80 = new ArrayList();
            selectedListWhileDo_06f80.add(new Integer(2));
            selectedListWhileDo_06f80.add(new Integer(5));
            iHotWhileDo_06f80=0;
            iColdWhileDo_06f80=0;
            nWhileDo_06f80=4;
            passWhileDo_06f80=0;
            recvChoiceWhileDo_06f80=0;
            while (iHotWhileDo_06f80+iColdWhileDo_06f80<nWhileDo_06f80) {
               int choiceWhileDo_06f80=-1;
               if (selectedListWhileDo_06f80.size()>0)
                  choiceWhileDo_06f80=chooseOne(selectedListWhileDo_06f80,null,"WhileDo_06f80");
               switch (choiceWhileDo_06f80) {//switch
               case 2:
                  if (((Boolean)coldChoices.get("LSC_2_02a4")).booleanValue()) {
                     boolean condLSC_2_02a4=false;
                     condLSC_2_02a4=condRecvDiscoverObjectInstanceHLAfederateNTSMSS()
                     || condLSC_2_02a4;
                     if (condLSC_2_02a4) {//if clause start
                        SendRequestAttributeValueUpdate_0774HLAfederateNTSMSS
                         ("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9","s10","s11","s12",
                         "s13","s14","s15","s16","s17","s18","s19","s20","s21",
                          "s22","s23","s24","s25","s26","s27","s28");
```

Figure 6-9. Excerpts from the Generated Java Code of Monitor Federate (Continue)

```
                    }////if closed end
                  }//end of cold condition
                  iColdWhileDo_06f80++;
                  break;
               case 5:
                  SendEvokeCallbackECNTSMSS(new Object());
                  iHotWhileDo_06f80++;
                  break;
               }//switch
               switch (recvChoiceWhileDo_06f80) {//switch
               case 0:
                  if (boolReflectAttributeValuesHLAfederateNTSMSS()) {
                     condRecvReflectAttributeValuesHLAfederateNTSMSS();
                     iHotWhileDo_06f80++;
                  }
                  break;
               case 1:
                  if (boolReceiveInteractionHLAmanagerNTSMSS()) {
                     condRecvReceiveInteractionHLAmanagerNTSMSS();
                     iHotWhileDo_06f80++;
                  }
                  break;
               }//switch
               recvChoiceWhileDo_06f80=(recvChoiceWhileDo_06f80+1)%2;
               if (iHotWhileDo_06f80==nWhileDo_06f80-1&&iColdWhileDo_06f80<1)
               //n-number of cold
               {
                  SleepThread(100);
                  passWhileDo_06f80++;
                  if (passWhileDo_06f80==50)
                     break;
               }
            }
            selectedListWhileDo_06f80.clear();
         }
         stop();
      }
   }
   op2_Callbacks_03d1 p1 = new op2_Callbacks_03d1();
   p1.start();
   class op3_RTIDataMonitorTimer_03d2 extends Thread {
```

Figure 6-9. Excerpts from the Generated Java Code of Monitor Federate (Continue)

```
    op3_RTIDataMonitorTimer_03d2() {}
    public void run() {
      while (((Boolean)FedMonFdChart.coldChoices.get("Condition_0883")).booleanValue()) {
        doLaterRTIDataMonitorTimer_03be(666);
        SendEvokeCallbackECNTSMSS(new Object());
        SendSendInteraction_0722HLArequestPublicationsNTSMSS(new Object());
        SendSendInteraction_0728HLArequestSubscriptionsNTSMSS(new Object());
        SendSendInteraction_072eHLArequestObjectInstancesUpdatedNTSMSS(new Object());
        SendSendInteraction_0734HLArequestObjectInstancesReflectedNTSMSS(new Object());
        SendSendInteraction_073aHLArequestInteractionsSentNTSMSS(new Object());
        SendSendInteraction_0740HLArequestInteractionsReceivedNTSMSS(new Object());
        calcelRTIDataMonitorTimer_03be();
      }
      stop();
    }
  }
  op3_RTIDataMonitorTimer_03d2 p2 = new op3_RTIDataMonitorTimer_03d2();
  p2.start();
  class op4_StatusTimer_03d3 extends Thread {
    op4_StatusTimer_03d3() {}
    public void run() {
      if (((Boolean)FedMonFdChart.coldChoices.get("Condition_0883")).booleanValue()) {
       //cond start
        doLaterStatusTimer_03c1(66);
        SendEvokeCallbackECNTSMSS(new Object());
        SendRequestAttributeValueUpdate_075aHLAfederationNTSMSS
        ("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9");
        SendEvokeCallbackECNTSMSS(new Object());
        SendRequestAttributeValueUpdate_075fHLAfederateNTSMSS
        ("s0","s1","s2","s3","s4","s5","s6","s7","s8","s9","s10","s11","s12",
        "s13","s14","s15","s16","s17","s18","s19","s20","s21",
         "s22","s23","s24","s25","s26","s27","s28");
        calcelStatusTimer_03c1();
        stop();
      }//cond end
      else//Hot cond
        return;//Hot cond
    }
  }
  op4_StatusTimer_03d3 p3 = new op4_StatusTimer_03d3();
  p3.start();
```

Figure 6-9. Excerpts from the Generated Java Code of Monitor Federate (Continue)

```
            while (p3.isAlive()||p2.isAlive()||p1.isAlive()||p0.isAlive())
               SleepThread(100);
            condRecvMessageInput_076dUser();
            SendResignFederationExecutionRFENTSMSS(0);
            SendDestroyFederationExecutionDFENTSMSS("s0");
         }////if closed end
      }//end of cold condition
   }
```

Figure 6.9. Excerpts from the Generated Java Code of Monitor Federate (Continue)

To give a sense of the generated code, a part of the monitor federate's base code (see Figure 6.9) and a sample RTI Ambassador method (*sendinteraction* in Figure 6.10) and a federate Ambassador method (*receiveinteraction* in Figure 6.11) are shown in the respective figures.

The main method of the monitor federate (see Figure 6.3) of the generated FedMonFd code is exemplified in Figure 6.9. For every operand in a parallel inline expression occurring in the LSC, a thread (e.g. *op1_RefreshAllMonitors_03d0*, *op2_Callbacks_03d1*, *op3_RTIDataMonitorTimer_03d2* and *op4_StatusTimer_03d3*) is generated. For loop idioms, "while-do" or "repeat-until" code statements are generated. Values of loop conditions are retrieved from the dictionary (implemented as *hashtable* named *coldChoices*) defined in the computation aspect. In place of the references in the LSC model, corresponding referenced charts code are generated and added. For example, for *FedMonFdChart2*, corresponding methods are generated.

In Figure 6.10, interaction information is put together in an object of the common data type *LSCObject*. Then the corresponding *LscRTILib* method (in this case, *sendInteraction*) is called.

```
public static boolean SendSendInteraction_08b7HLArequestSubscriptionsNTSMSS(Object Time)
{
      LSCLib.LSCObject proc= new LSCLib.LSCObject();
      proc.name="HLArequestSubscriptions";
      proc.pars=new ArrayList();
      LSCLib.LSCAttribute parNew0 =new LSCLib.LSCAttribute();
      parNew0.name="Time";
      parNew0.type="Object";
      parNew0.objClass="Double";
      parNew0.objVal=Time;
      proc.pars.add(parNew0);
      NTSMSSRTILib.sendInteraction(proc);
      return true;
 }
```

Figure 6.10. Sample *SendInteraction* RTI Ambassador Method

In Figure 6.11, a federate Ambassador method (in this case, *receiveinteraction*) example in the federate base code is shown.

```
public static void RecvReceiveInteractionHLAmanagerNTSMSS(LSCLib.LSCObject iClass, String
TimeStamp, int SentOrderType, int ReceiveOrderType, String MessageRetractionDesignator, String
TransportationType)
{}
```

Figure 6.11. A Sample *ReceiveInteraction* Federate Ambassador Call-back Method

b) Default Aspect Code

Two computation aspect and a federation execution aspects are generated, namely *FedMonFdAspect*, *UserAspect*, and *NTSMSSLibAspect*. In *FedMonFdAspect*, all methods of the federate are accessed and method bodies of them are overridden in their corresponding advices. In FedMonFdAspect, dictionaries and LSC-specific auxiliary methods' (i.e. *chooseOne*, *getLoopount*) advices are also generated.

```
pointcut pcSendSendInteraction_0728HLArequestSubscriptionsNTSMSS
(Object Time):execution(static boolean
 FedMonFd.SendSendInteraction_0728HLArequestSubscriptionsNTSMSS(Object))&&  args(Time);
boolean around(Object Time):
pcSendSendInteraction_0728HLArequestSubscriptionsNTSMSS(Time)
{
      Time=new Object();
      proceed(Time);
      return true;
}
```

Figure 6.12. A sample RTI Ambassador Method (advice)

A sample RTI Ambassador method's advice (send interaction) and a federate Ambassador method's (receive interaction) advices (accessing methods) are shown in Figure 6.12 and Figure 6.13, respectively. In Figure 6.12, federate send interaction method (cf. Figure 6.10) is caught in the FedMonFd base code and default logic filled in its advice. The developer can edit this advice as described in the next "Editing the Default Computation Aspect" section.

In Figure 6.13, federate receive interaction method (cf. Figure 6.11) is found on the FedMonFd base code and received data is displayed in the its advice in the *FedMonFdAspect*. This received data is interaction class and its parameters' values. In our case, interaction class is *HLArequestSubscriptions*.

```
pointcut pcRecvReceiveInteractionHLAmanagerNTSMSS(LSCLib.LSCObject iClass,String
TimeStamp,int SentOrderType,int ReceiveOrderType,  String MessageRetractionDesignator,String
TransportationType):execution(static void FedMonFd.RecvReceiveInteractionHLAmanagerNTSMSS
(LSCLib.LSCObject,String,int,int,String,String)) &&
args(iClass,TimeStamp,SentOrderType,ReceiveOrderType,
MessageRetractionDesignator,TransportationType);
 void around(LSCLib.LSCObject iClass,String TimeStamp,int SentOrderType,int
ReceiveOrderType,String MessageRetractionDesignator,String TransportationType):
pcRecvReceiveInteractionHLAmanagerNTSMSS(iClass,TimeStamp,SentOrderType,ReceiveOrderT
ype, MessageRetractionDesignator,TransportationType)
{
      System.out.println("Received message:"+TimeStamp);
      System.out.println("Received message:"+SentOrderType);
      System.out.println("Received message:"+ReceiveOrderType);
      System.out.println("Received message:"+MessageRetractionDesignator);
      System.out.println("Received message:"+TransportationType);
      proceed(iClass,TimeStamp,SentOrderType,ReceiveOrderType,
      MessageRetractionDesignator,TransportationType);
}
```

Figure 6.13. A sample Federate Ambassador Method (advice)

NTSMSSLibAspect (federation execution aspects) is mainly used to catch call-back methods from the respective federation executions. In our case, only an aspect is generated since a monitor federate can join in a federation. *NTSMSSRTILib* (*LscRTILib*) is declared in this aspect and it is used to reach actual RTI. A sample *LscRTILib* definition (*NTSMSSRTILib*) and a sample (*ReceiveInteraction*) advice are presented in Figure 6.14

In Figure 6.14, *ReceiveInteraction* call-back method is caught by the federation execution aspect (*NTSMSSLibAspect*) and forwarded to the federate (*FedMonFd.ReceiveInteraction*).

```
public static RTILib FedMonFd.NTSMSSRTILib= new RTILib();; // LscRTILib Definition
(…)
pointcut ReceiveInteraction(LSCLib.LSCObject proc):
execution(public void RTILib.receiveInteraction(..))&& args(proc); //LscRTILib method is caught
   after(LSCLib.LSCObject proc):ReceiveInteraction(proc)
{
      RTILib rtiLib = (RTILib)thisJoinPoint.getThis();
      if (rtiLib.federatename.compareTo("NTSMSS")==0)
         FedMonFd.ReceiveInteraction(proc); //federate method is called
}
```

Figure 6.14. A LscRTILib Definition and A Sample Advice

Step iv: Edit the Default Computation Aspect (Optional)

After running the generator, *FedMonFdAspect* (generated default computation) can be edited by the developer in order to effect the desired computation. Consider, for example, how timestamp is retrieved to send a send-interaction event to the federation. In the automatically generated default computation, a "*new Object ()"* is sent to the federation. The corresponding edited code is illustrated in the figure as italic form in Figure 6.15.

```
pointcut pcSendSendInteraction_0728HLArequestSubscriptionsNTSMSS(Object Time):
execution(static boolean FedMonFd.
SendSendInteraction_0728HLArequestSubscriptionsNTSMSS(Object))&& args(Time);
boolean around(Object Time):
pcSendSendInteraction_0728HLArequestSubscriptionsNTSMSS(Time)
{
      //Time=new Object();
      Time=g_TimeStamp;
      proceed(Time);
      return true;
}
```

Figure 6.15. Adding a Computation to the RTI Ambassador Method
(Modifications to the advice are in italic).

In this example, the advice that catches the method of *SendSendInteraction_0728HLArequestSubscriptionsNTSMSS* in the federate base code is edited. Generated code statement (*Time=new Object ()*) is commented and new statement that provides sending of the timestamp is added (*Time=g_TimeStamp*).

Step v: Run the Generated Code

With the *FedMonFdAspect* (computation aspect) edited, the *FedMonFd* (federate application) code is ready to run. After that, code is compiled and the AspectJ compiler weaves the aspects on the base code. After the compiling, code is run. A view from the running FedMonFd is represented in Figure 6.16.

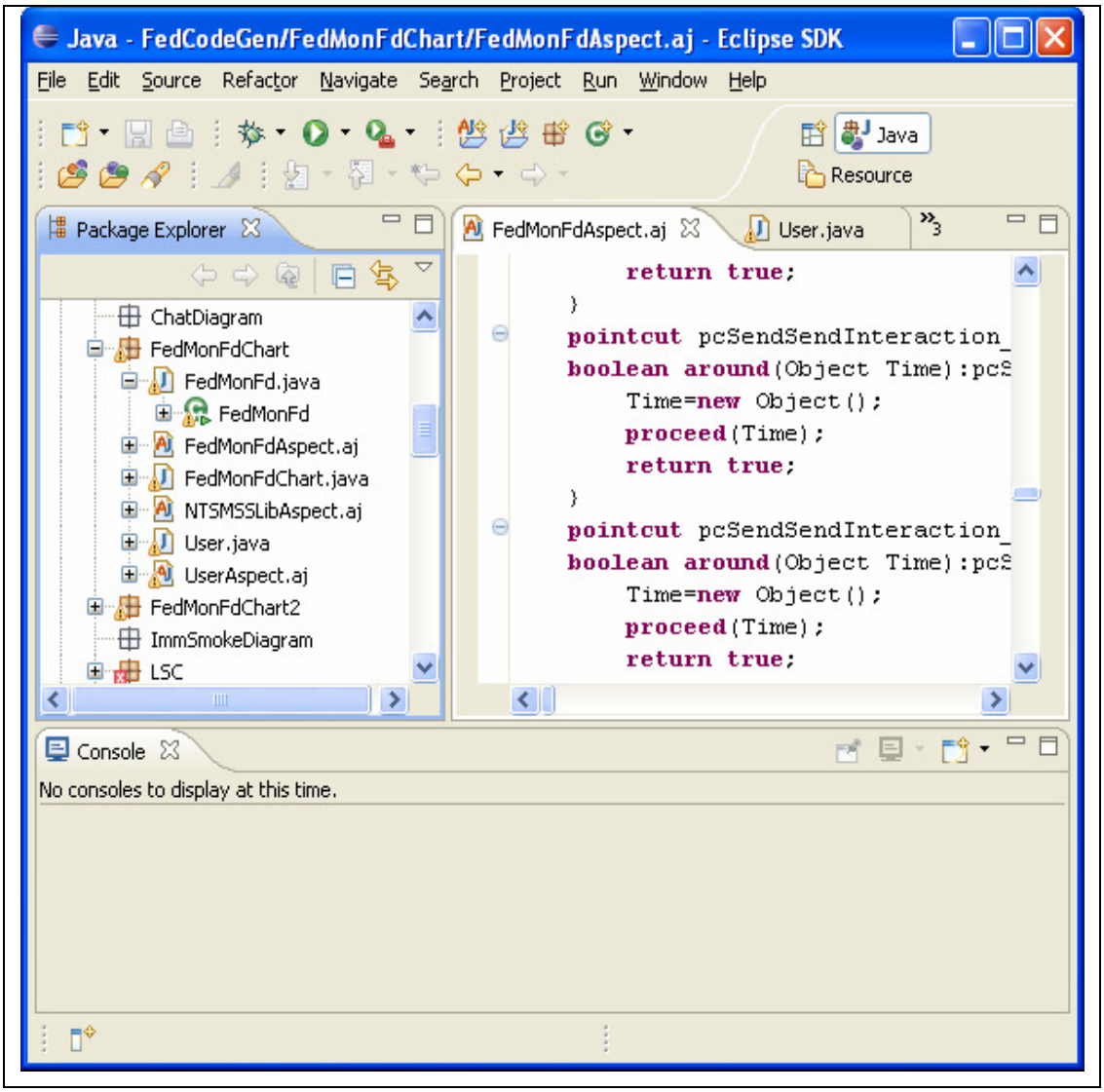


Figure 6.16. A View of the FedMonFd Application Running (pRTI snapshot)

6.3.2. Discussion of the Case Study

There are some points that must be discussed after the study. One of major drawback of this study is remodeling from the beginning when a major update is realized in the metamodel or library. Until FAMM become mature, these updates will potentially be problematic.

This study is important for Federation Design Verification. This can be done by two ways: Static checking and dynamic verification. Both Federation scenario(s) and Federation Design Model is represented using LSCs. So, the static model checking is performed using the model interpretation over both LSCs where Federate LSCs must include the Federation Scenario LSC. But Verification can be interpreted in the dynamic (federation execution) sense. Dynamic model checking is based on the automatic code generation.

Generated code has almost 3000 LOC. Detailed generated code statistics are given the following table. This case study gives us a hint in the code generation of larger application. FedMonFd models and corresponding codes are obtained from our Web site (www.ceng.metu.edu.tr/~e73883). Also in Appendix E, an example for each LSC/MSC construct such as ALT, SEQ and its corresponding generated code is presented.

Table 6-1. FedMonFd Code Metrics (in LOC)

Class/Aspect	**Line of Codes**
FedMonFdChart (Diagram base code)	53
FedMonFd (Federate Base Code)	1978
FedMonFdAspect (Federate Computation Aspect)	664
User	108
UserAspect	95
NTSMSSLibAspect	25
Total	**2923**

CHAPTER VII

CONCLUSION

The primary contribution of this thesis is automatic generation of a federate application code from a model of the federation architecture and a model of the federate behavior. The federate code generator offers the ability for early prototyping of a federation with the ability to proceed with full-fledged implementation. As a core part of this work, but generic in its nature, we present a code generator that generates Java base code directly from Live Sequence Charts (LSCs). This is the secondary contribution of this thesis.

The generated federate code carries out the communication behaviors of a federate in an HLA 1516 compliant federation. The code generator is built upon the foundation of a metamodel, namely FAMM, for describing both the static and dynamic views of the architecture of a federation.

Details of code generation from the federate developer's perspective have been illustrated with the help of a running example, STMS. On a larger scale, a Federation Monitor Federate (FedMonFd) is modeled and its code is generated [Sarioglu et al. 2007]. Supplementary material and produced code can be obtained from our Web site.

Adopting the aspect-oriented approach, communication-related base code and computation-related aspect codes are separated. The developer can edit the latter so that the particular computation (in general, non-communication) logic can be weaved onto the generated base code. An obvious advantage is applying variations on the algorithms by editing only the computation aspect without touching the federate base code. Generally speaking, the whole spectrum of AOP techniques is at the disposal of the federate developer. If a new pattern of communication is desired the behavior model of the federate must be modified accordingly and then the federate base code must be generated again. There is no need to modify the computation aspect provided that the method arguments that are used in the base code remain the same. In other words, pointcut definitions (method declarations in the base code) must not be touched during aspect editing.

The developer works on the federate's communication behavior at the model level rather than at the code level. Modeling provides an abstract view of the federation and the participating federates to the developer. Developer, for example, can add a new event

between two instances in the LSC model. During code generation, the related pointcut definition and the corresponding method are automatically generated. Developer is not forced to dealing with implementation details of code. Only the advices of the computation aspect code needs to be edited. Editing points are marked by comments in the generated code so these advices should be easy to locate in the code. Behavioral codes and all related method definitions are generated in the base code automatically. Corresponding pointcut definitions and advices that access the data model is also generated as an aspect.

Generation of federate code from the model takes negligible time. Developer's time is consumed by constructing the architecture model plus coding the computational logic by aspect editing. The compliance of the computational code to the SOM is dictated by the automatically generated portions of the aspect code thanks to the referencing mechanism employed in the metamodel FAMM. Using the techniques reported here, full generation of utility federates, e.g. for testing, monitoring, logging, etc. looks feasible.

In a more general setting, this work has achieved code generation for communication behaviors of applications described in LSC. The LSC code generator is built upon the foundation of an LSC metamodel, for describing both the static and dynamic views of the application. The MSC/LSC metamodel provides a flexible and extendible input specification for code generator. The offered code generator covers both MSC and LSC specifications. For example, for LSC chart, it generates LSC based code, for MSC chart it generates MSC based code. In the LSC base code, existential chart properties and universal chart properties are handled.

Code generation from LSCs and MSCs allows the execution of the behavioral model supporting an early validation of a behavioral specification expressed in MSC or LSC. Modeling also enables developer to understand behaviors of the application.

Regarding the semantics of MSCs and LSCs, we had to do many clarifications as the relevant literature is obscure on many points. For instance, timer semantics have been defined rather tersely in Z.120.

The clarity of the generated code is crucial as the application developer may have to deal with it directly. This is also important for editing. Moreover, comments are added to the code to help the developer navigate the code. The developer can trace the comments from model to code as well.

Only Java and AspectJ codes are generated presently by our generator. However, the back end (Java code generator module) can be re-implemented to target another programming language. Most important obstacle to achieve retargeting is the AOP language maturity. Only a few of the programming languages are mature enough such as AspectJ and AspectC [AspectC 2007] till now.

Details of LSC code generation from the developer's perspective have been illustrated with the help of a running example, ATM Machine Money Withdrawal. Another case study has been carried out to animate the behavioral specifications [Efe 2007]. The message exchanges among the components of the specified system are modeled as LSCs. The subject of modeling in that study is the radio communications among the members of a field artillery team, which are the fire control center, the firing unit, and the forward observer. Using the code generator, the Java and AspectJ codes are automatically generated from the communications model. The animation code is weaved on the generated base code. Execution of the generated code animates the radio messages as a sequence of events respecting to the partial order specified in the LSC. Animation can help validate conceptual models, e.g. in face validation, and clarify system specifications. The generated code can also be utilized as a first-cut prototype for the intended simulation.

REFERENCES

ADAK M. AND OGUZTUZUN H. 2007. A Code Generator for Live Sequence Charts and Message Sequence Charts. Technical Report (METU-CENG-TR-2007-4), Middle East Technical University, May 2007.

ADAK M., TOPÇU O., AND OĞUZTÜZÜN H. 2007, Model-Based Code Generation for HLA Federates, submitted to ACM Transactions on Modeling and Simulation, 2007

ASPECTC. 2007. AspectC. http://www.cs.ubc.ca/labs/spl/aspects/aspectc.htmlj. Last accessed at August 8, 2007.

ASPECTJ. 2007. AspectJ Project. http://www.eclipse.org/aspectj. Last accessed at August 8, 2007.

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

BONTEMPS Y., HEYMANS P. AND SCHOBBENS P.Y. 2005. From live sequence charts to state machines and back: a guided tour, *IEEE Transactions on Software Engineering,* doi.ieeecomputersociety.org

BRILL M., DAMM W., KLOSE J., WESTPHAL B. AND WITTKE H. 2004. Live Sequence Charts: An Introduction to Lines, Arrows, and Strange Boxes in the Context of Formal Verification. *Springer-Verlag LNCS 3147*, 374-399.

CZARNECKI K. 2005. Overview of Generative Software Development. In J.-P. Banâtre et al. (Eds.): Unconventional Programming Paradigms (UPP) 2004, Mont Saint-Michel, France, LNCS 3566, pp. 313–328

DAMM W. AND HAREL D. 2001. LSCs: Breathing Life Into Message Sequence Charts. In *Formal Methods in System Design*, 19, 45-80.

EFE O. 2007. Animation of Behavioral Specification through Code Generation. Project Report, Department of Computer Engineering, METU, Ankara

ELRAD T., AKSIT M., KICZALES G., LIEBERHERR K. AND OSSHER H. 2001. Discussing Aspects of AOP. *Communications of the ACM*, vol. 44 no.10, pp. 33-38.

ECLIPSE. 2007. Eclipse Project. http://www.eclipse.org/. Last accessed at August 8,2007

EMERSON, J. M. 2005. GME-MOF: An MDA Metamodeling Environment For GME, *Thesis Submitted to the Faculty of the Graduate School of Vanderbilt University in partial fulfillment of the requirements for the degree of Master of Science in Computer Science*, Nashville, Tennessee.

GME. 2006. A Generic Modeling Environment GME 6 User's Manual v6.0, Institute for Software Integrated Systems (ISIS) Vanderbilt University

HAREL D. 2001. From play-in scenarios to code: an achievable dream, Fac. of Math. & Comput. Sci., Weizmann Inst. of Sci., Rehovot, Computer 53-60

HOMME T. AND RAMSLAND J.E. 2003. From Live Sequence Charts to Implementation, A study of the LSC specification, the execution of behavioral requirements and exploring the possibilities to use an LSC model to generate Java code, *Masters thesis in Information and Communication Technology*, Grimstad

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

IEEE 2000b. *Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification (IEEE 1516.1).*

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

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

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

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

JVM. Java Virtual Machine. 2007. http://java.sun.com/docs/books/jvms. Last accessed at August 8, 2007.

KENT S. 2002. Model Driven Engineering. *Lecture Notes In Computer Science; Vol. 2335 Proceedings of the Third International Conference on Integrated Formal Methods*, 286 - 298

KICZALES G., LAMPING J., MENHDHEKAR A., MAEDA C., LOPES C, LOINGTIER J. M. AND IRWIN J. 1997. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proc. European Conf. on Object-Oriented Programming*, volume 1241 of LNCS. Springer-Verlag, Berlin, Heidelberg, and New York, 220–242.

KLEPPE A., WARMER J. and BAST W. 2003, MDA Explained: Practice and Promise, *Addison*

Weslesy.

KRUGER I., GROSU R., SCHOLZ P. AND BROY M. 1999. From MSCs to Statecharts. Distributed and Parallel Embedded Systems. informatik.tu-muenchen.de

LEDEZCI A., BAKAY A., MAROTI M., VOLGVESI P., NORDSTORM G., SPRINKLE J., KARSAI G. 2001. Composing Domain-Specific Design Environments. In IEEE Computer, vol.34 no.11, pp. 44-51

MAOZ S AND HAREL D. 2006. From multi-model scenarios to code: compiling LSCs into AspectJ. *Procedings of the 14th ACM SIGSOFT International symposium on Foundations of the software engineering*, Portland, Oregon, 219-230.

OMG 2002. Meta Object Facility (MOF), 1.4, OMG Document formal/02-04-03, http://www.omg.org/cgi-bin/doc?formal/2002-04-03, Last accessed at August 8, 2007

OMG 2003. MDA Guide Version 1.0.1. Object Management Group. http://www.omg.org/mda Last accessed at August 8, 2007

PARR S. AND RUSSELL K.M. 2003. The Next Step - Applying the Model Driven Architecture to HLA, *Proceedings of the 2003 Spring Workshop*[C]. Paper ID: 03S-SIW-123 Calytrix Technologies Pty Ltd.

PITCH RTI. 2007. Pitch Technologies AB. pRTI 1516 v3.1.1 certified for IEEE 1516. http://www.pitch.se. Last accessed at August 8, 2007.

RADESKI A. AND PARR S. 2002. Towards a Simulation Component Model for HLA. *2002 Fall Simulation Interoperability Workshop*

SARIOGLU K., ADAK M. AND OĞUZTÜZÜN H. 2007. Fedaration Monitoring Federate, Technical Report (METU-CENG-TR-2007-7), Middle East Technical University, June 2007

SCHMIDT D.C. 2006. Guest Editor's Introduction: Model-Driven Engineering Computer, 2006 - csdl.computer.org

SIMPLICITY. 2007. http://www.calytrix.com/siteContent/SIMplicity/intro.php. Last accessed at August 8, 2007.

TOLK A. 2002. Avoiding Another Green Elephant – A Proposal for the Next generation HLA based on the Model Driven Architecture. In *Proceedings of 2002 Fall Simulation Interoperability Workshop (SIW).*

TOPÇU O., ADAK M. AND OĞUZTÜZÜN H. 2007. A Metamodel for Federation Architectures, accepted from *ACM Transactions on Modeling and Simulation (TOMACS), September 2007.*

TOPÇU, O. AND OĞUZTÜZÜN H. 2007. A Metamodel for Live Sequence Charts and Message Sequence Charts, Technical Report (METU-CENG-TR-2007-03), Middle East Technical University, May 2007

TSAI W.T., FAN C., CHEN Y. AND PAUL R. 2006. DDSOS: A Dynamic Distributed Service-Oriented Simulation Framework, *In 39th Annual Simulation Symposium*, 160--167. Huntsville, AL, USA

TSAI W.T., HUANG Q., SUN X. AND CHEN Y. 2007. Dynamic Collaboration Simulation in Service-Oriented Computing Paradigm, *In Proceedings of 40th Annual Simulation Symposium* (ANSS), March 2007, Norfolk, VA, USA, pp.41-48

YUAN Z., CAI W. AND LOW M.Y.H. 2003. A Framework for Executing Parallel Simulation using RTI, *Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real-Time Applications* (DS-RT'03)

APPENDIX A

PATTERNS AND RELATED CODES

In this appendix, model patterns and their corresponding generated code segments are presented exhaustively. For example, M1 represents a message pattern, and it may be sending message, start timer event, etc.

1. Messages

Pattern:

Code Generated for Pattern:

(a) Send Message

```
Send[name of message][target instance](new Object());
```

(b) Receive Message

```
Recv[name of message][target instance] ();
```

(c) Dynamic instance Create

```
[name of diagram].p[name of instance].start();
```

(d) Dynamic instance Stop

```
[name of diagram].p[name of instance].stop();
```

(e) Start Timer

```
doLater[name of timer]([timer set value]);
```

(f) Reset Timer

```
cancel[name of timer] ();

doLater[name of timer] ([timer set value]);
```

(g) Timeout

```
if(timerFlag[name of timer]){Revc[name of timer]Timeout();}
```

(h) Lost Message

```
Send[name of message]LOST(new Object ());
```

(i) Found Message

```
contract[name of message]FOUND();
```

(j) Action

```
Send[name of message]ACTION(new Object ());
```

2.4.2. Conditions

Pattern:

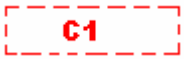
(Hot)

Code Generated for Pattern:

```
if (((Boolean)coldChoices.get("C1")).booleanValue())// model elements name is key.
  (…)
else
  return; //if condition is not satisfied, the most nearest method is aborted
```

Figure A.1 Code Generated for C1-hot

Pattern:



(Cold)

Code Generated for Pattern:

```
if (((Boolean)coldChoices.get("C1")).booleanValue())
  ….
```

Figure A.2 Code Generated for C1-cold

2.4.4. Temperature Property

Cold Message Pattern:

Code Generated for Pattern M1:

```
if(((Boolean)coldChoices.get("M1")).booleanValue()){
        M1
}//cold
```

Figure A.3 Code Generated for cold message

Cold Location Pattern:

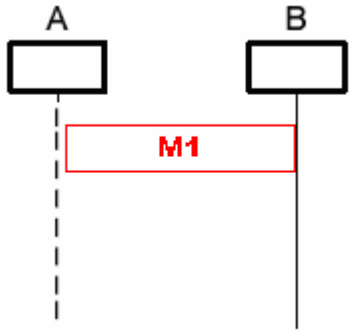


Figure A.4 Cold Location Pattern

Code Generated for Pattern:

```
boolean cold[name of location]=false;
for (int i=0;i<50;i++) {
 if (bool[name of event]) {
        M1
        cold[name of location]=true;
       break;
  }
 }
 if(!cold[name of location])//for cold receive message in cold location
   return;
```

Figure A.5 Code Generated for cold location

Existential-Chart Pattern:

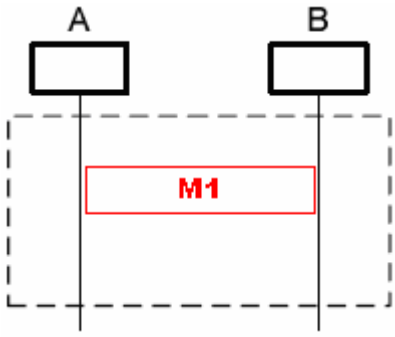


Figure A.6 Existential-Chart Inline Expression Pattern

```
if(((Boolean)coldChoices.get("[name of chart]")).booleanValue())
{
        M1
}
```

Figure A.7 Code Generated for Chart

2.4.8. Barrier Synchronization

Pattern:

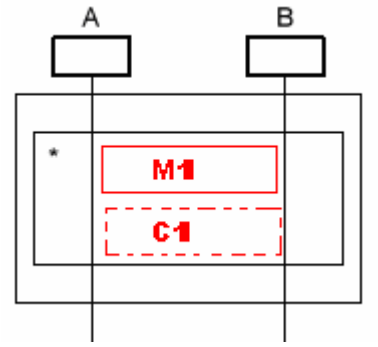


Figure A.8 Barrier Synchronization Pattern

Code Generated for Pattern:

```
do {
        M1
        try {
            Ship_MSC.RepeatUntil_02ef.await();//cyclic barrier is used
            //wait other instances calling "await" method.
        } catch (…)
} while(!C1);
```

Figure A.9 Code Generated for the Pattern

2.4.9. LSC/MSC composition

Pattern:

Composition

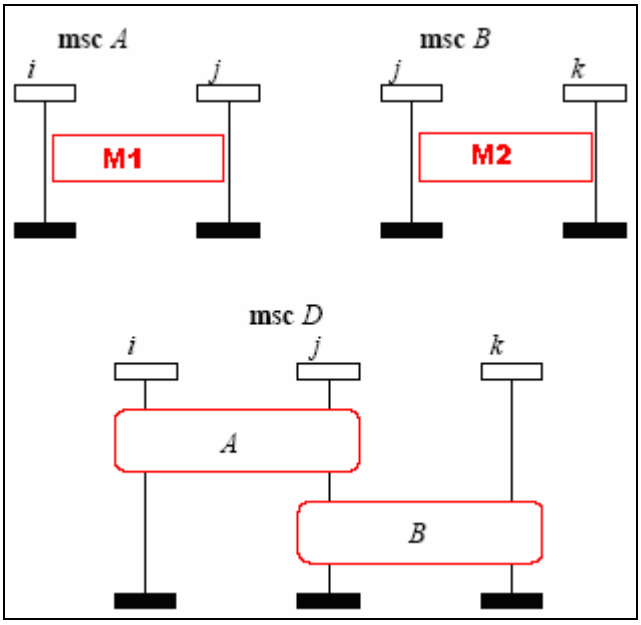


Figure A.10. Composition Pattern

Code Generated for Pattern:

```
M1
```

Figure A.11. a Code Generated for Instance “i”

```
M2
```

Figure A.11. b Code Generated for Instance “k”

```
M1

M2
```

Figure A.11. c Code Generated for Instance j (composition)

2.4.10. Coregion

Pattern:

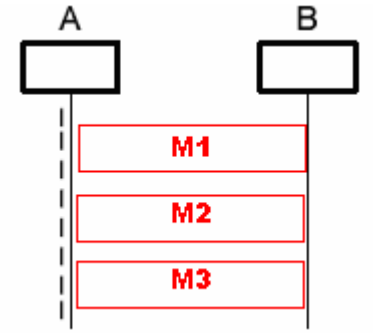


Figure A.12. Coregion Pattern

Code Generated for Pattern:

Assume that all events are send messages. Receiving case is not presented in here.

Order is changed randomly in A

```
ArrayList selectedList = new ArrayList();
int n;
int i;
selectedList.add(new Integer(1));
selectedList.add(new Integer(2));
selectedList.add(new Integer(3));
n=3;
i=1;
while(i<=n) //round-robin algorithm
{
        int choice=chooseOne(selectedList,null);// select random message
        switch(choice)
        {//switch
        case 1:
                M1
                break;
        case 2:
                M2
                break;
        case 3:
                M3
                break;
        }//switch
        i++;
}
selectedList.clear();
```

Figure A.13. a Code Generated for Coregion in instance A

Order is not changed in B

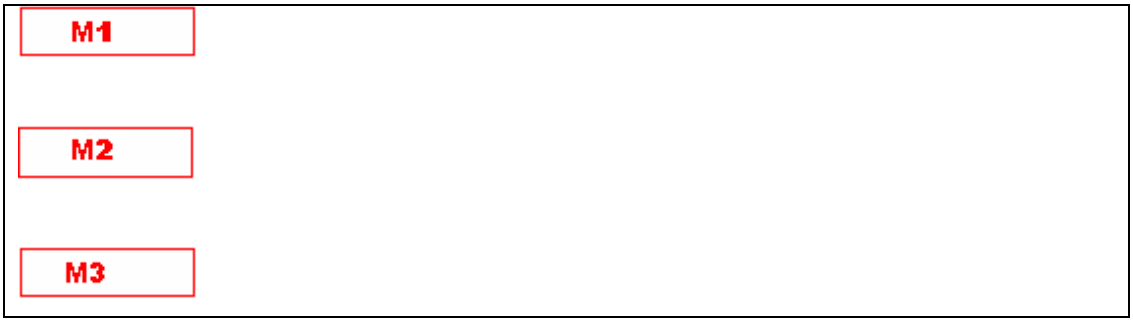


Figure A.13. b Code Generated for Coregion in Instance B

2.4.11. Inline Expressions

Alternative Pattern:

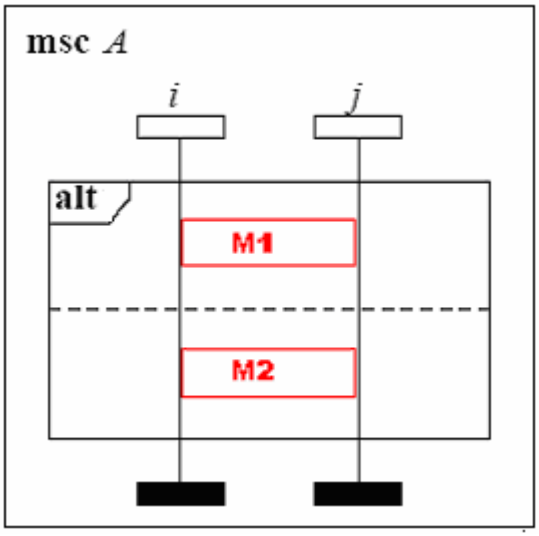


Figure A.14. ALTernative Inline Expression Pattern

Switch blocks are chosen in the computation aspect randomly. The name of ALT is put in the *hashtable* as a key. Value of the key is set randomly at the run-time. According to random choices, an alternative part of the operator code is run.

Code Generated for Pattern:

int [model *name of alternative inline*]=

((Integer)altChoices.get("[model *name of alternative inline*]")).intValue();

switch([model *name of alternative inline*]){

case 0:

M1

break;

case 1:

M2

break;

}

Figure A.15. Code Generated for ALT

Parallel Pattern:

INLINE

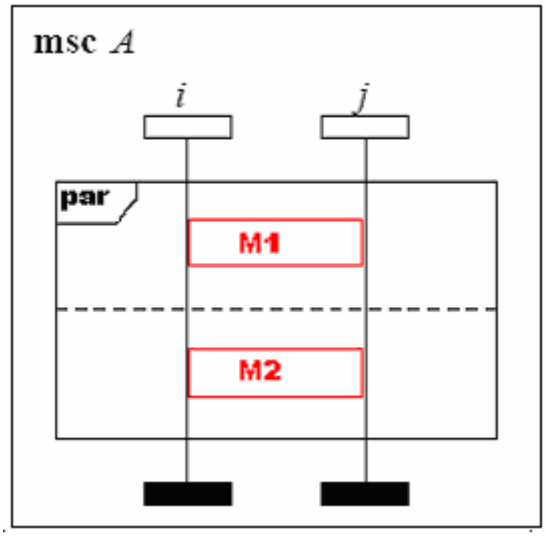


Figure A.16. PARallel Inline Expression Pattern

This expression is represented by threads that contain parallel blocks of the operator in the code. So the parallel blocks are run in parallel threads.

Code Generated for Pattern:

```
class [name of operand1] extends Thread {
        [name of operand1] () {}
        public void run() {
                M1
                stop();
        }
}
[name of operand1] p0 = new [name of operand1] ();
p0.start();
class [name of operand2] extends Thread {
        [name of operand2] () {}
        public void run() {
                M2
                stop();
        }
}
[name of operand2] p1 = new [name of operand2] ();
p1.start();
while(p1.isAlive()||p0.isAlive());
```

Figure A.17. Code Generated for PAR

Loop Pattern:

INLINE

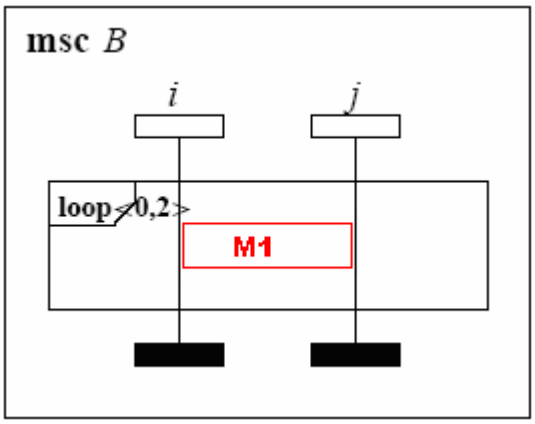


Figure A.18. LOOP Inline Expression Pattern

If loop is infinite (*loop<inf, inf>*), a "*while (true)*" code is generated. However, if the loop is definite, counter variables are randomly selected and loop is iterated according to those variables.

Code Generated for Pattern:

```
boolean loopCond=false;
int count[name of loop]=0;
int loopCount = getLoopCount("0","2");
if(loopCount==-1)
loopCond=true;
while(count[name of loop <loopCount || loopCond)
{
        M1
        count[name of loop]++;
}//end of loop
```

Figure A.19. Code Generated for LOOP

Sequential Pattern:

INLINE

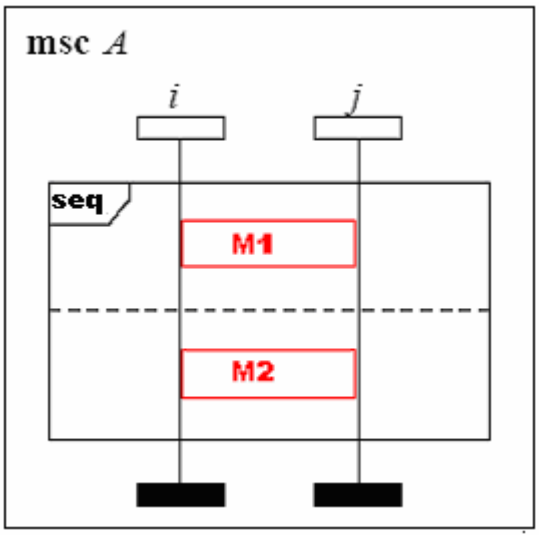


Figure A.20. SEQential Inline Expression Pattern

This expression is represented simply by adding sequential parts successively.

Code Generated for Pattern:

```
M1

M2
```

Figure A.21. Code Generated for SEQ

Exception Pattern:

INLINE

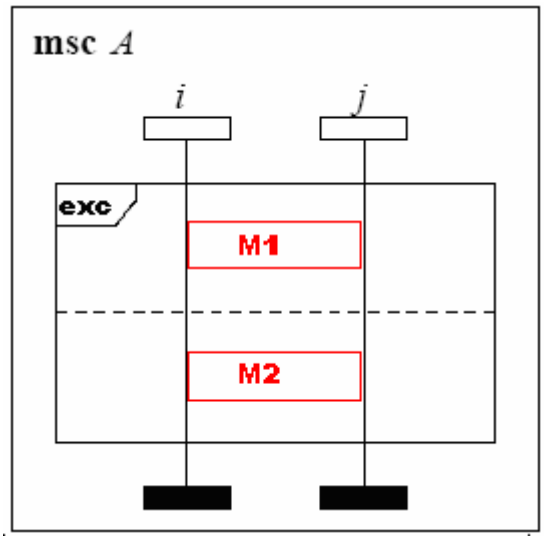


Figure A.22. EXCeption Inline Expression Pattern

This expression is specialized by an EXC operator. There are two operands in this expression. First operand presents the message traffics (try block) and second operand presents the exceptional message traffics (catch block) when an exception is raised in the LSC.

Code Generated for Pattern:

```
try{
        M1
}catch(Exception ex)
{
        M2
}
```

Figure A.23. Code Generated for EXC

Optional Pattern:

INLINE

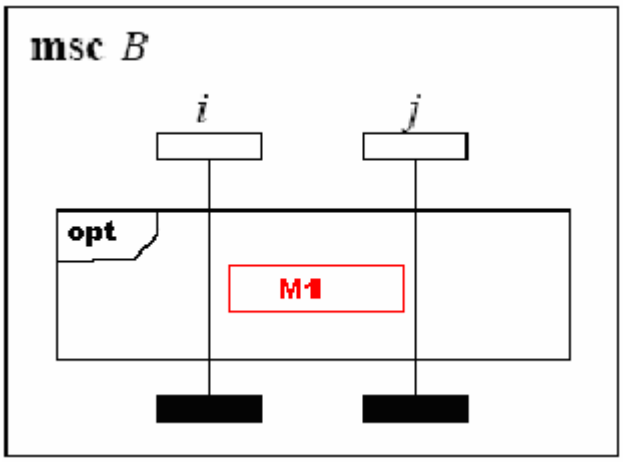


Figure A.24. OPTional Inline Expression Pattern

Opt operand is surrounded with an if-clause and if the condition is satisfied, operand is executed otherwise it is not. Condition is randomly selected in the aspect.

Code Generated for Pattern:

```
if(((Boolean)coldChoices.get("[name of optional inline]")).booleanValue())
{
        M1
}
```

Figure A.25. Code Generated for OPT

Do-While Pattern:

INLINE

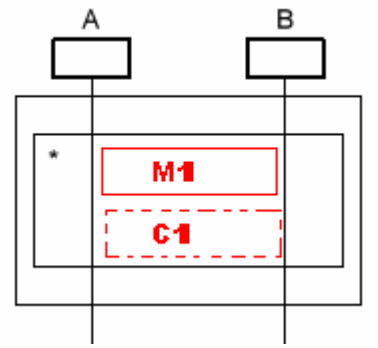


Figure A.26. Do-While Inline Expression Pattern

This expression is represented by a “do-while loop” clause.

Code Generated for Pattern:

```
do {
        M1
        C1
} while(!  );
```

Figure A.27. Code Generated for Do-While

While-Do Pattern:

INLINE

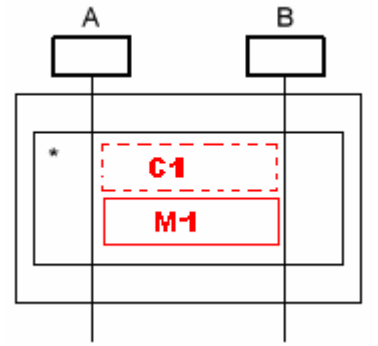


Figure A.28. While-Do Inline Expression Pattern

This expression is represented by a “while loop” clause.

Code Generated for Pattern:

```
while( C1 ){
        M1
}
```

Figure A.29. Code Generated for While-Do

If-Then Pattern:

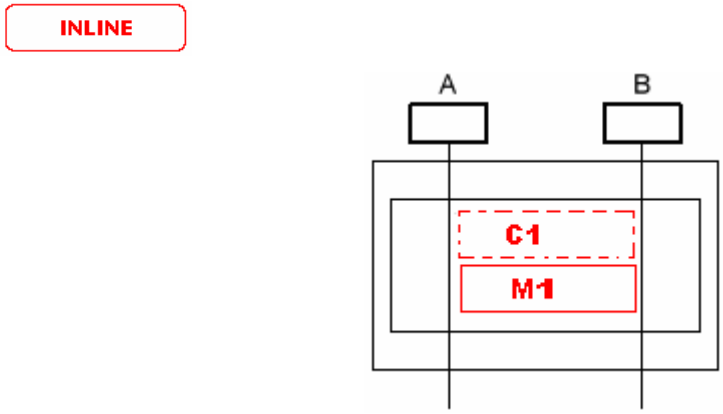


Figure A.30. If-Then Inline Expression Pattern

This expression is represented by an “if-then” clause.

Code Generated for Pattern:

```
if( C1 ){
        M1
}
```

Figure A.31. Code Generated for If-Then

If-Then-Else Pattern:

INLINE

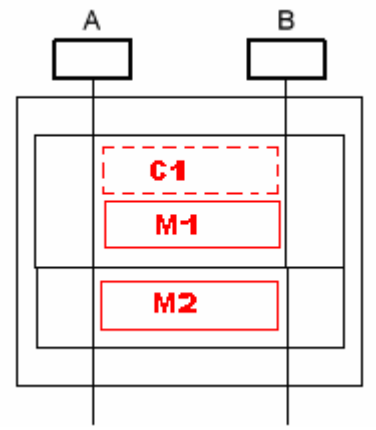


Figure A.32. If-Then-Else Inline Expression Pattern

This expression is represented by an “if-then-else” clause.

Code Generated for Pattern:

```
if( C1 ){
        M1
}
else {
        M2
}
```

Figure A.33. Code Generated for If-Then-Else

2.4.13. General Ordering

Local general order is only defined in a coregion.

General Ordering (Local) Pattern:

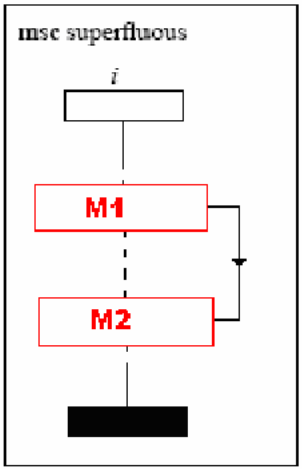


Figure A.34. Local General Ordering Pattern

Code Generated for Pattern:

```
(…)
Hashtable orderList = new Hashtable ();

orderList.put(new Integer(2),new Integer(1)); //1 happens earlier than 2
while(i<=n) //round-robin algorithm
{
        int choice=chooseOne(selectedList,orderList);// orderList indicates the order
        switch(choice)
        {//switch
        case 1:
                M1
                break;
        case 2:
                M2
                break;
        }//switch
        i++;
}
selectedList.clear();
```

Figure A.35. Code Generated for Local General Ordering

General Ordering (Multi-Instance) Pattern:

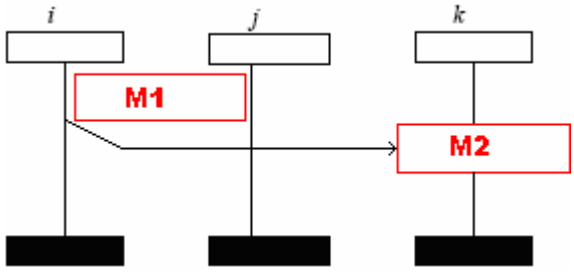


Figure A.36. Multi-Instance (Shared) Local General Ordering Pattern

Code Generated for Pattern:

M1

[*name of diagram*].set[*name of general order*]=true;

Figure A.37. a Code Generated for Multi-instance General Ordering in Instance i

while(![name of diagram].set[name of general order]);

M2

Figure A.37. b Code Generated for Multi-instance General Ordering in Instance k

2.4.14 Pre-chart

Pattern:

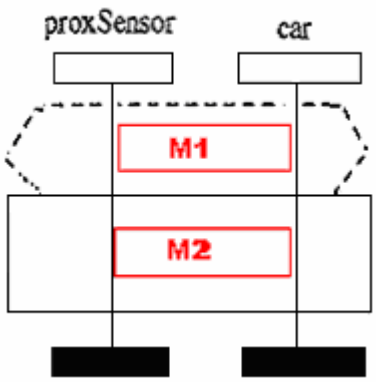


Figure A.38. Pre-Chart Inline Expression Pattern

Code Generated for Pattern:

boolean cond[*name of Prechard*]=true;

condo[*name of Prechard*]= M1 || condo[*name of Prechard*]);

if(cond[*name of Prechard*]){

M2

}////if closed end

Figure A.39. Code Generated for Pre-Chart

2.4.15. Local Invariant

Local Invariant Pattern:

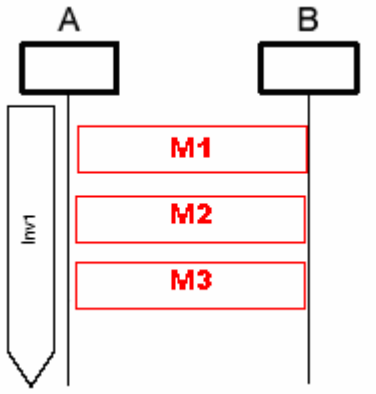


Figure A.40. Local Invariant Pattern

Code Generated for Pattern:

if(((Boolean)coldChoices.get("[*name of Inv1*]")).booleanValue()){

M1

M2

M3

}//end of invariant

Figure A.41. Code Generated for Local Invariant

2.4.16. Simultaneous Region

Simultaneous Region Pattern:

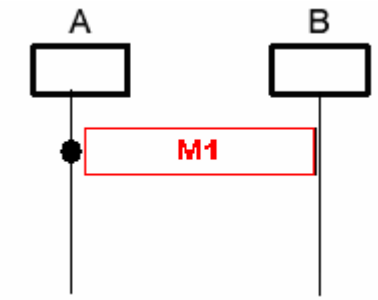


Figure A.42. Simultaneous Region Pattern

Code Generated for Pattern:

```
class [name of simultaneous region] extends Thread {
 [name of simultaneous region]() {}
 public void run()
  {
          M1
     stop();
  }
}
[name of simultaneous region] p[name of simultaneous region]=
new [name of simultaneous region]();
p[name of simultaneous region].start();
```

Figure A.43. Code Generated for Simultaneous Region

APPENDIX B

INTERMEDIATE FORM GENERATION

Main class and traversing method definitions are presented in the following. This presentation is kind of abstract data type (ADT) of the intermediate form. ADT defines an encapsulation of a data structure (Figure B.1) by giving the main classes/methods and their explanation in a formal way.

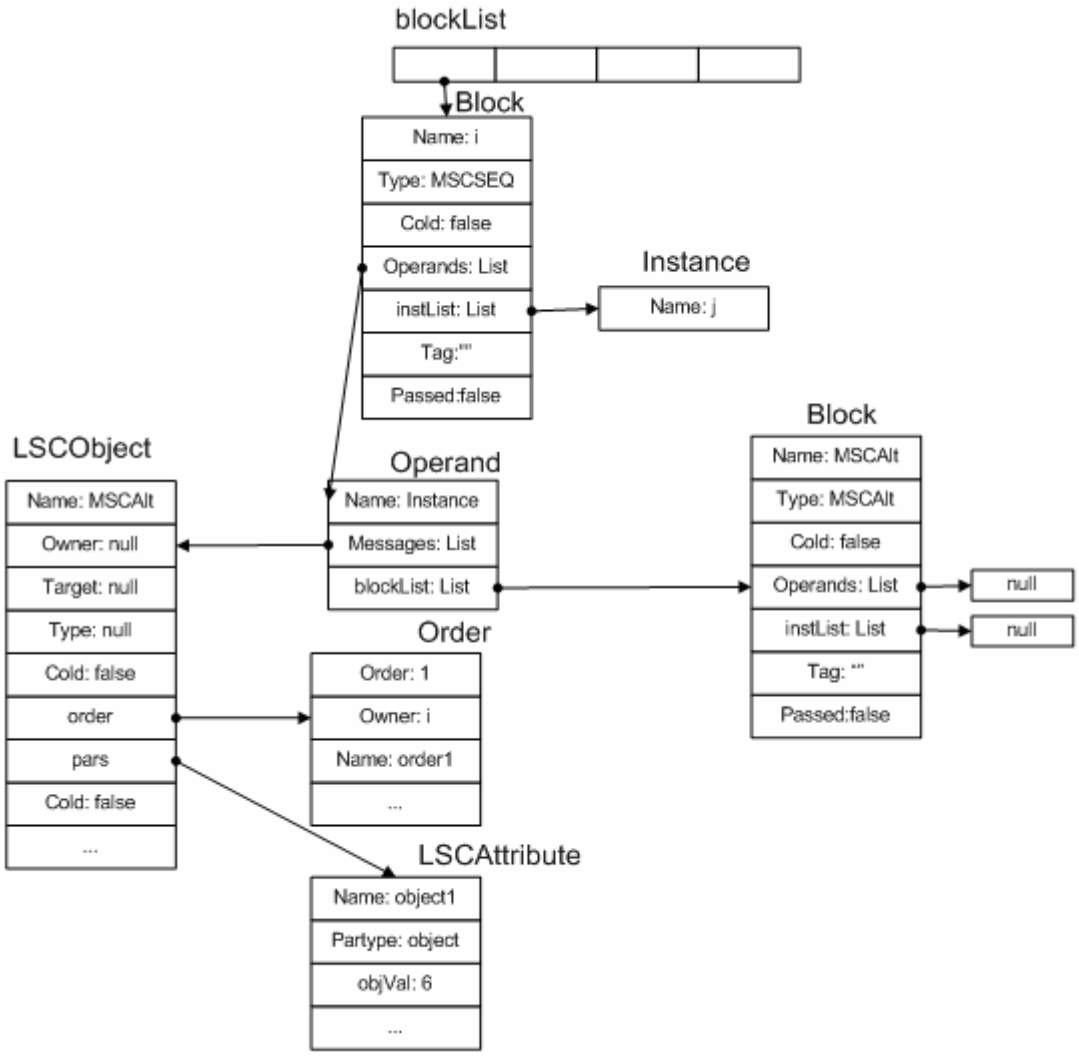


Figure B.1. Intermediate Form Data Structure

Block Structure

A block holds the main block and the nested sub-blocks thereof in the time axis. Nested sub-blocks are inline expressions of the instance. Main block is the main body of instance.

Classes:

1. Block

Object of this class holds the main block (as a kind of sequential inline expression) and nested inline expressions such as parallel, sequential.

Attributes	
String name	Block name
String type	Block type such as alternative, parallel, optional
boolean cold	Whether block is hot or cold
boolean passed	Indicate whether block is visited or not
Hashtable operands	List holds the operands of block. (Operand objects list)
ArrayList insList	List holds LSC instances where block is located.
String minCount	For the loop inline expression minimum count number
String maxCount	For the loop inline expression maximum count number
boolean isMultiIns	Block is multi-instance or not
String activationCondName	For the activation chart case, condition name

Operand

Object of this class holds the messages of operand.

Attributes	
String name	Operand name (copied from the input model)
Hashtable messages	List holds the messages in the operand (*LSCObject* objects list)
Hashtable blockList	List holds the nested blocks located in the operand. For example, alternative block in parallel block.

1.1.1 LSCObject

This class is the primary class for holding event (message) information in the model. An object of this class holds mainly event name, and event type such as sending, receiving, condition.

Attributes

String name	Event name generated by the code generator in the application (it does not occur in the model.)
boolean inCoregion	Whether event is in a coregion or not.
ArrayList pars	Holds the event parameters. Parameter information is declared as *LSCAttribute* object and it is stored in the "pars". (*LSCAttribute* objects list)
String blockName	Name of the innermost block name of the event. Block may be a main loop or an inline expression.
String targetInstance	Name of the destination instance of the event.
String ownerInstance	Name of the source instance of the event.
boolean coldCond	Whether event is cold or hot.
String insname	Event name (object name) is copied from the model
ArrayList timerList	List holds all the outer timers.
ArrayList newOrder	If event is subject to general ordering, order information is held here. (Order objects list)
boolean isMultiIns	Whether event is multi-instance or not
ArrayList instList	If event is multi-instance event, other instances are stored in this list.
boolean boolRecv	Whether event is receiving event or not
boolean boolSend	Whether event is sending event or not
boolean boolCrep	Whether event is process creation event or not
boolean boolStop	Whether event is process stop (termination) event or not
boolean boolActn	Whether event is local action event or not
boolean boolSetT	Whether event is timer set event or not
boolean boolRstT	Whether event is timer reset event or not
boolean boolTout	Whether event is timer timeout event or not
boolean boolCond	Whether event is condition event or not
boolean boolGuard	Whether event is guarding condition event or not
boolean boolCrgn	Whether event is coregion event or not or not
boolean boolLost	Whether event is a lost event or not
boolean boolFound	Whether event is a found event or not
boolean boolCall	Whether event is reference calling event or not
boolean boolReplyin	Whether event is a reply-in event or not
boolean boolReceive	Whether event is a receive event or not
boolean boolCallMtd	Whether event is a method call event or not

boolean boolReplyout	Whether event is a reply-out event or not
boolean boolMethod	Whether event is a method event or not
boolean boolSetting	Whether event is a setting condition event or not
boolean isLocStart	Whether event is cold location starting event or not
boolean isLocStop	Whether event is cold location ending event or not
boolean isCrgnStart	Coregion starting event
boolean isCrgnStop	Coregion stopping event
boolean isSuspensionStart	Suspension starting event
boolean isSuspensionStop	Suspension stopping event
boolean isInvariantStart	Invariant starting event
boolean isInvariantStop	Invariant stopping event
int timerSemanticMethod	If event is timer event, timer semantic is assigned to it.
String msg	It is used to pass information from the first pass to the second.
String sid	Symbolic id of the LSCObject
boolean isGateRelated	Whether event is gate related or not
String eventName	Event name comes from model
String msgName	Message name if event is a message
String mtdKind	If event is a method, kind of the method
String simultaneousName	If event is in a simultaneous region, indicate region name

1.1.1.1 LSCAttribute

This class is the primary class for holding the event parameter information in the model. An object of this class holds mainly parameter name and parameter type.

Attributes

String name	Attribute name.
String type	Attribute type such as int, string, object (similar to union)
String strVal	If attribute type is string, string value of attribute
int intVal	If attribute type is integer, integer value of attribute
Object objVal	If attribute type is object, object value of attribute
int value	Holds the timer set value if event is a timer event.
String objClass	If attribute is object, class name of the object
boolean isGlobal	Whether attribute is global for instance or not
Object [] arrVal	If attribute type is array, array value of attribute

LSCObject LSCObjectVal	If attribute type is *LSCObject*, *LSCObject* value of attribute

1.1.1.2 Order

This class is used to order events in general ordering of LSC. Object of this class holds the event precedence such as order, instance of events such as owner, whether events is multi-instance or single-instance, name of the ordering model element and finally whether event is sending or receiving. It is instantiated for each event that is to be ordered.

Attributes

int order	Precedence of event (copied from the model).
String owner	Instance name of event
String name	Name of general ordering model element that copied from the model
boolean isMultiInstance	Indicates whether event is multi-instance or not
boolean isSingleInstance	Indicates whether event is single-instance or not
boolean isOut	Indicates whether event is sending or receiving

Global Lists:

These global lists are constructed by using above classes for every instance.

ArrayList sendListObjectHandlers	List holds all the sending events (same events may be repeated) on the instance. Events are added to the list as *LSCObject* instances.
ArrayList recvListObjectHandlers	List holds the all receiving events (same events may be repeated) on the instance. Events are added to the list as *LSCObject* instances.
ArrayList sendListClassHandlers	List holds the all different sending events on the instance. Events are unique and repetitions are removed. Events are added to the list as *LSCObject* instances.
ArrayList recvListClassHandlers	List holds the all receiving events on the instance. Events are unique and repetitions are removed. Events are added to the list as *LSCObject* instances.
ArrayList allTimerList	List holds the all timers on the instance.
ArrayList orderedEventList	List holds the all events that are ordered according to the general ordering principles

ArrayList blocksForInstance	Lists all blocks located on the executing instance.
ArrayList variablesForInstance	Lists all model variables located on the executing instance.

Traversing Methods

These methods traverses on the LSC input model and creates the intermediate form (Intermediate Form Generation Module in Figure 3.3). These methods are described in the calling order. Other word, call graph is represented in Figure B-2 and orders of the methods are presented by using the numbered bullets. For example, *traverseOnModel* method (numbered bullet 1) calls the *traverseDocument* (numbered bullet 1.1) method.

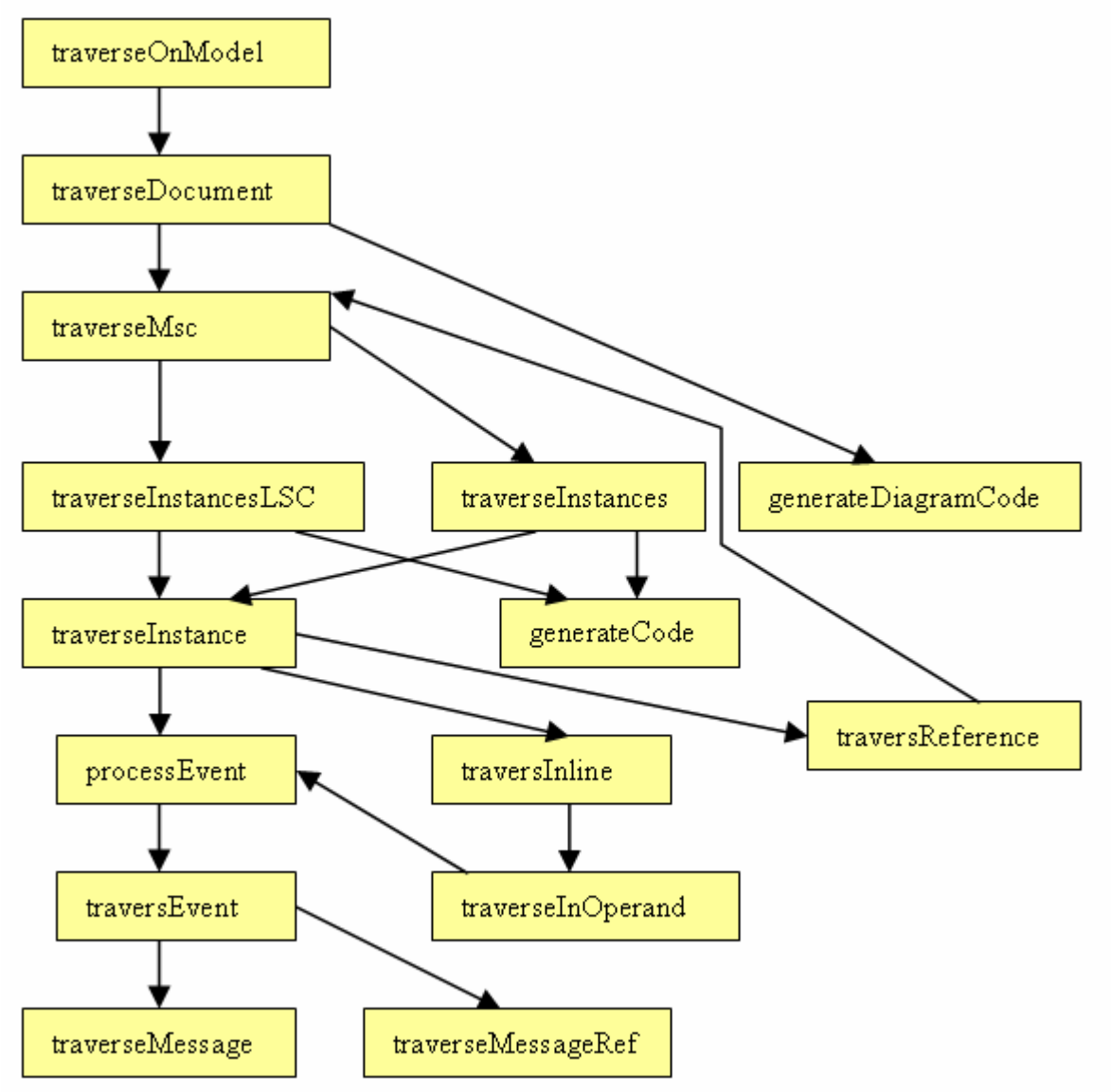


Figure B.2. Call-graph of the Intermediate Form Generation Module (Front End)

1. traverseOnModel

This method traverses the input LSC model and gets all the diagrams. After that it calls the traverse document method (namely *traverseDocument*) one by one. It also creates the block list that holds the blocks of the diagrams in the model.

Parameters

Model builder	Main model element

1.1. traverseDocument

This method walks on the document model and retrieves the all diagram found in it. Then it calls the traverse diagram method for each of them.

Parameters

JBuilderModel mscDocument	Document model element
Hashtable blockList	List holds blocks
int index	Precedence of reference model element in case of reference model composition

1.1.1. traverseMsc

This method traverses the diagram model and it calls the traverse instances method (namely *traverseInstances* and *traverseInstancesLSC*) that walks on all instances in the diagram. After that it calls the *generateCode* method to generate corresponding source code segment for the diagram from the intermediate form.

Parameters

JBuilderModel MSC	Diagram model element
Hashtable blockList	List holds blocks
int index	Precedence of reference model element in case of reference model composition

1.1.1.1. traverseInstancesLSC

This method traverses the LSC diagram model and gets the all instances located in it. Then it calls the traverse instance method (namely *traverseInstance*) for each of them.

Parameters

JBuilderModel Lsc	Diagram model element
Hashtable blockList	List holds blocks
int index	Precedence of reference model element in case of reference model composition
boolean coldCond	Indicate whether diagram is existential or universal chart

1.1.1.2. traverseInstances

This method traverses the MSC diagram model and gets the all instances located in it. Then it calls the traverse instance method (namely *traverseInstance*) for each of them

Parameters

JBuilderModel MscBody	Diagram model element
Hashtable blockList	List holds blocks
int index	Precedence of reference model element in case of reference model composition

1.1.2. generateDiagramCode

Recall that a separate thread is provided in a separate source file for every instance in the model, and that a diagram consists of instances. The threads of instances are declared and started if they are not dynamic instances. (An instance is dynamic if it is started when process creation event occurs.) In the diagram code, diagrams have also own threads. These diagram codes are generated by this method. Moreover, the preliminary computation aspect codes for the diagram are generated in it.

Parameters

ArrayList InstanceList	List holds instances in the diagram
String diagramName	Diagram name copied from the model
ArrayList dynInstanceList	List holds dynamic instances in the diagram
ArrayList blocksForInstance	List holds blocks in the current instance.

1.1.1.1.1 traverseInstance

This method traverses the instance and gets the events, inline expressions, and composition reference calls; and then forwards them to the related methods, namely, *processEvent*, *traversInline*, and *traversReference*.

Parameters

Model lscInstance	Instance model element
JBuilderReference InstanceRef	Instance model reference
Hashtable blockList	List holds blocks.
ArrayList dynInstanceList	List holds dynamic instances.
int index	Precedence of diagram reference in composition. In case of composition, diagram references are connected to the instance time-line.
ArrayList targetList	List holds target instances
JBuilderModel RefInline	If this method called from a inline expression, it is inline's model element

1.1.1.1.2 generateCode

Recall that an instance in the model is handled by a thread in a separate source file as a separate class definition. The instance codes (Class definitions, main blocks, messages, and message declarations) are generated in this method. Also, aspect codes for the instance are generated by generator in a separate aspect file as an aspect. Developer can catch the join points (obvious point cuts in the base code) in the application and can weave the advice code into these points to impose the computation logic in the aspect codes. In the code generation, a preliminary is generated randomly.

Parameters

String FedName	Instance name
Hashtable blockList	List holding blocks.
String activeDiagramName	Diagram name of instance (copied from the input model).

1.1.1.1.1.1 LSCObject processEvent

This method gets event details and constructs the event objects. (namely *LSCObject*) It is called by *traverseEvent* method.

Parameters

Model Event	Event model element

String instanceName	Instance name of the event
ArrayList dynFedList	List holding the dynamic instances
ArrayList targetList	List holds target instances

1.1.1.1.1.1.1 LSCObject traversEvent

This method traverses through to the event model element (namely *traverseMessage*) and gets the event information. Process creation events are also caught in this method. If events are multi-instance, instance names are obtained (from the input model).

Parameters

Model Event	Event model element
LSCObject obj	Event object
ArrayList dynFedList	List of dynamic instance
ArrayList targetList	List holds target instances

1.1.1.1.1.1.1.1 LSCObject traverseMessage

This method traverses through to the message model elements. But, message parameters of object details are not handled in the present work. Only target instance name is returned by this method. Because, messages are connected to the target instances in the model.

Parameters

Model Message	Message model element
LSCObject obj	Source event object. (Events are connected to messages in the model)
ArrayList targetList	List holds target instances

1.1.1.1.1.1.1.2 LSCObject traverseMessageRef

This method traverses through to the message model by using reference's of it. But, message parameters of object details are not handled in the present work. Only target instance name is returned by this method. Because, messages are connected to the target instances in the model

Parameters

JBuilderReference MessageRef	Message model reference
Model Message	Message model element

LSCObject obj	Source event object. (Events are connected to messages in the model)
ArrayList targetList	List holds target instances

1.1.1.1.1.2 traversInline

It traverses recursively the inline expressions, which could be nested, (*traversInline* method).

Parameters

Model Inline	Inline expression model element
Hashtable blockList	Block list where constructed blocks are inserted
String instName	Instance name of the block
ArrayList dynInstanceList	Dynamic instance list
int index	Precedence of the block (inline expression)
ArrayList targetList	List holds target instances
JBuilderModel ownerInstance	Owner instance of the inline
boolean isLSC	Inline's temperature value

1.1.1.1.1.2.1 traverseInOperand

This method traverses the operand of the inline. And constructs the block data structure (*processEvent* method) for each operand.

Parameters

JBuilderModel MSCOperand	Operand model element
String instName	Current instance name
Block instBlock	Block structure for the operand model
Operand instOperand	Operand structure for the operand model
ArrayList dynInstanceList	Dynamic instance list
ArrayList targetList	List holds target instances
JBuilderModel RefInline	If inline is nested, parent inline model element
boolean isLSC	Inline's temperature value

1.1.1.1.1.3 traversReference

This method traverses the reference composition model elements and gets the referenced diagram. Then it calls the related diagrams. Reference has also own instance and diagrams. Also separate thread and file are generated for it. In the instance code, methods of referenced diagram are called. Although same instance may occur in other diagrams, different threads are created for it.

Parameters

Model Reference	Reference model element
Hashtable blockList	List holds blocks.
String instName	Instance name of diagram reference.
Operand operand	Parent operand. References are put in the parent *hashtable*.
ArrayList dynFedList	List holds the dynamic instances
int index	Precedence of reference model element.
ArrayList targetList	List holds target instances
JBuilderModel lscInstance	Instance model element
JBuilderModel RefInline	If this method called from a inline expression, it is inline's model element

APPENDIX C

JAVA CODE GENERATOR

Source code of the input LSC is generated from the intermediate form of the LSC (Java Code Generation Module in Figure 3.3). In this section, main generating methods are described. These methods are expressed according to the call order. Other word, call graph is represented n Figure C-1. Orders of methods are illustrated by using the numbered bullets. For example, *createHeadSourceCodes* method (numbered bullet 1) calls the *writeInstanceLoopMethod* (numbered bullet 1.1) method.

Attributes	
boolean locationIsCold	Indicate whether location is cold or hot.
boolean coregionIsStarted	Indicate whether coregion is started.

Global Lists:

These global lists are constructed by using above classes for every instance.

currTimerList	List the current timers during the emitting process
currConList	List the current conditions during the emitting process

1.1 LSCTimer

Object of this class holds the timer information.

Attributes	
String name	Timer name (copied from the input model)
Int time	Time interval value
int timerSemanticMethod	Timer semantics for different implementation

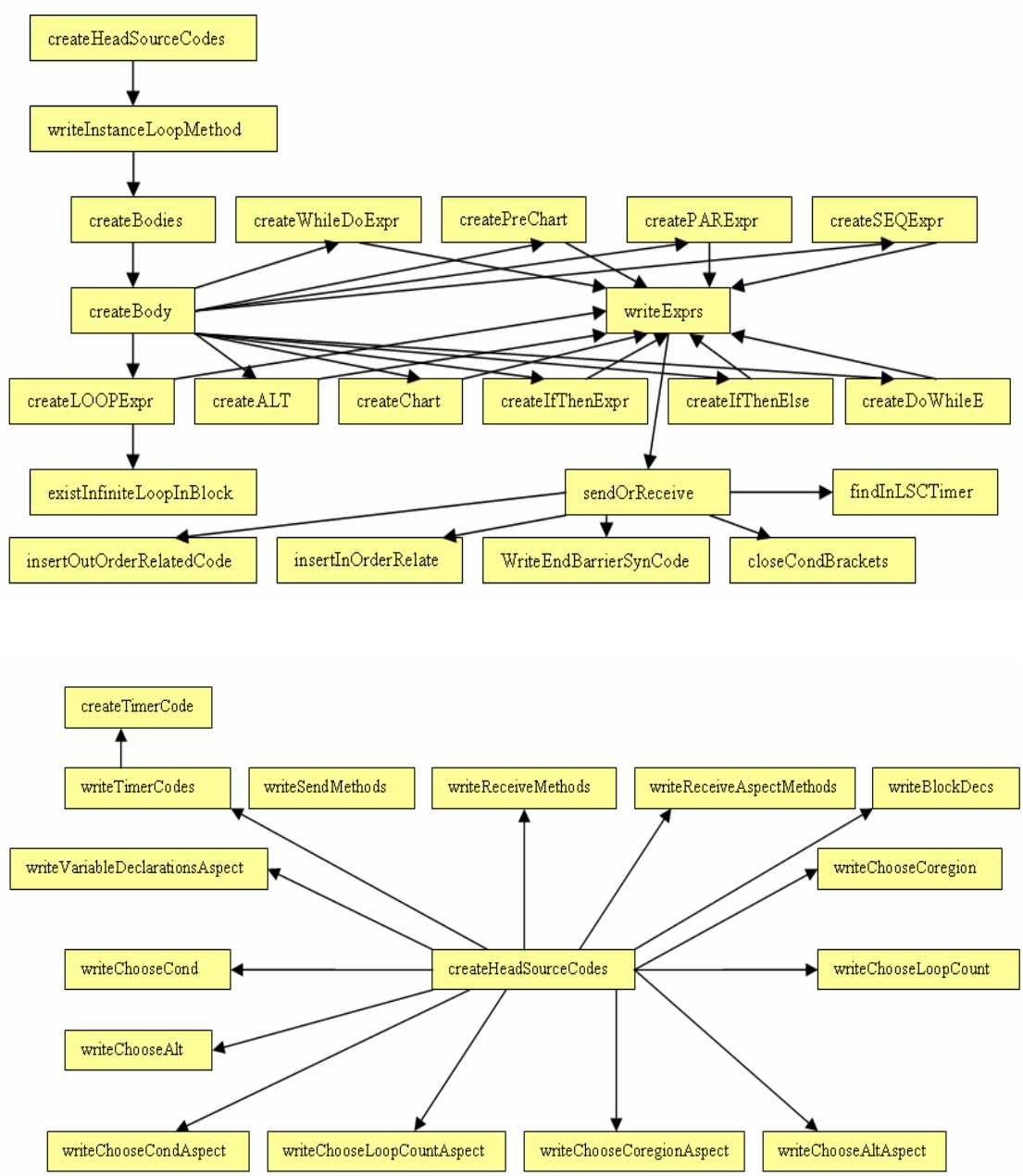


Figure C.1. Call-graph of the Java Code Generation Module (Back End)

Generating Methods

1. createHeadSourceCodes

This method writes the global declarations and main body codes of the instance into the Java source file and related aspect source file. It is also the main compositional method that calls the all emitting methods. An other word, this method generates the Java and AspectJ code of the instance.

Parameters

ArrayList sendListClassHandlers	List holds sending events of the instance.
ArrayList recvListClassHandlers	List holds receiving events of the instance.
boolean ExternalEnabled	Indicate external library usage for the domain-specific model integration
String LSClib	Namespace of the *LSCObject* class
String className	Name of the instance as a class name
String ExternalLib	External library name
ArrayList allTimerList	List holds the timers of the instance
Hashtable blockList	List holds the blocks of the instance
String activeDiagramName	Name of the diagram is copied from input LSC model.
ArrayList variablesForInstance	List holds the instances of block.

1.1. writeInstanceLoopMethod

This method writes the main loop method declaration and related aspect code of the instance.

Parameters

Hashtable blockList	List holds the blocks.
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram is copied from input LSC model.

1.1.1. createBodies

This method scans the block list and it calls the inline expression writer method for each.

Parameters

Hashtable blockList	List holds blocks of the instance
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram is copied from input LSC model.

1.1.1.1 createBody

This method writes an inline expression (block) content that consists of events. It calls the inline expression writer method according to the block type such as loop, parallel.

Parameters	
Block block	Block of the inline expression.
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram is copied from input LSC model.
LSCObject prc	Input event
Block pcBlock	Parent block of the block for nested case

1.1.1.1.1 createLOOPExpr

This method writes the codes of loop inline expression.

Parameters	
Block block	Block of the inline expression
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram is copied from input LSC model.
LSCObject proc	Input event
Block pcBlock	Parent block of the block for nested case

1.1.1.1.1.1 existInfiniteLoopInBlock

This method determine infinite loop occurrence in the block

Parameters	
Block block	Block of the inline expression

1.1.1.1.1.2 writeExprs

This method determines emits the code statements which corresponds the model events. For example, sending method calls are emitted in it.

Parameters	
Hashtable list	Event list that holds the events.

Operand operand	If this method is called in a operand, operand model element
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram is copied from input LSC model.
String blockName	If this method is called in a operand, block model element
Block pcBlock	Parent block of the block for nested case

1.1.1.1.2 createALTExpr

This method writes the codes of alternative inline expression.

Parameters

Block block	Block of the inline expression
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
Block pcBlock	Parent block of the block for nested case

1.1.1.1.3 createChartExpr

This method writes the codes of universal and existential charts.

Parameters

Block block	Block of the inline expression
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
Block pcBlock	Parent block of the block for nested case

1.1.1.1.4 createIfThenElseExpr

This method writes the codes of “if-then-else” inline expression.

Parameters

Block block	Block of the inline expression
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
Block pcBlock	Parent block of the block for nested case

1.1.1.1.5 createIfThenExpr

This method writes the codes of “if-then” inline expression.

Parameters

Block block	Block of the inline expression
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
Block pcBlock	Parent block of the block for nested case

1.1.1.1.6 createDoWhileExpr

This method writes the codes of “do-while” inline expression.

Parameters

Block block	Block of the inline expression
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
Block pcBlock	Parent block of the block for nested case

1.1.1.1.7 createWhileDoExpr

This method writes the codes of “while-do” inline expression.

Parameters

Block block	Block of the inline expression
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
Block pcBlock	Parent block of the block for nested case

1.1.1.1.8 createPreChartExpr

This method writes the codes of pre-chart inline expression.

Parameters

Block block	Block of the inline expression
String className	Name of the instance as class name

String activeDiagramName	Name of the diagram comes from input LSC model.
Block pcBlock	Parent block of the block for nested case

1.1.1.1.9 createPARExpr

This method writes the codes of parallel inline expression.

Parameters

Block block	Block of the inline expression
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
LSCObject proc	Input event object
Block pcBlock	Parent block of the block for nested case

1.1.1.1.10 createSEQExpr

This method writes the codes of sequential inline expression.

Parameters

Block block	Block of the inline expression.
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
Block pcBlock	Parent block of the block for nested case

1.1.1.1.1.1.1 boolean sendOrReceive

This method writes the event codes. Event may be sending, receiving, or condition event. For every defined case (event type), different codes are generated. If defined event type is not found, it returns false, otherwise true.

Parameters

LSCObject prc	Input event object
String className	Name of the instance as class name
String activeDiagramName	Name of the diagram comes from input LSC model.
Block block	block of the event

1.1.1.1.1.1.1.1 insertOutOrderRelatedCode

This method emits out-order code of the event that is a source event fort the ordering.

Parameters

LSCCodeGen.LSCObject prc	*LSCObject* structure to hold the event
String activeDiagramName	Name of the diagram comes from input LSC model.

1.1.1.1.1.1.1.2 insertInOrderRelatedCode

This method emits in-order code of the event that is a target event fort he ordering.

Parameters

LSCCodeGen.LSCObject prc	*LSCObject* structure to hold the event
String activeDiagramName	Name of the diagram comes from input LSC model.

1.1.1.1.1.1.1.3 WriteEndBarrierSynCode

For multi-instance case, loops are synchronized. This method emits the synchronization related barrier codes to the generated code.

Parameters

ArrayList instList	List to hold other LSC instances to be synchronized.
String activeDiagramName	Name of the diagram comes from input LSC model.

1.1.1.1.1.1.1.4 closeCondBrackets

This method emit the condition closing code ("}") at the end of the if-clause.

Parameters

boolean isInline	Indicate whether it is in a inline or not.

1.1.1.1.1.1.1.5 LSCTimer findInLSCTimerList

This method finds the timer in the timer list.

Parameters

ArrayList list	List to hold timers
String elem	Timer name to be found in the list.

1.2. writeTimerCodes

This method writes the all timer declaration codes of instance.

Parameters

ArrayList allTimerList	List holds all timers.
String className	Name of the instance as class name

1.2.1 createTimerCode

This method writes the definition of the timer and related timer methods/properties.

Parameters

String ID	Timer name comes from input LSC model.
String className	Name of the instance as class name

1.3. writeSendMethods

This method writes the sending procedure declaration codes. It also writes related aspect codes that catches the join point of Java code (all procedure definitions of sending and receiving events are sample join points) into the aspect source file.

Parameters

ArrayList sendListClassHandlers	List holds sending events of the instance.
String libraryStr	Namespace of the *LSCObject* class
String className	Name of the instance as class name
String ExternalLib	Name of the external library for model integration
String activeDiagramName	Name of the active LSC diagram name.

1.4. writeReceiveMethods

This method writes the receiving procedure declaration codes.

Parameters

ArrayList recvListClassHandlers	List holds receiving events of the instance.
String libraryStr	Namespace of the *LSCObject* class
String ExternalLib	Name of the external library for model integration
String activeDiagramName	Name of the active LSC diagram name.

1.5. writeReceiveAspectMethods

This method writes aspect codes that catches the join point of Java code (procedure definitions are join point) into the aspect source file.

Parameters

ArrayList recvListClassHandlers	List holds receiving events of the instance.
String className	Name of the instance as class name

1.6. writeBlockDecs

This method writes the boolean flag definition for each block into the diagram code. This flag is used for the barrier synchronization of the inline expression.

Parameters

ArrayList blocksForInstance	List holds the blocks of instance
String className	Name of the instance as class name

1.7. writeChooseCondAspect

This method emits the catching code of the choose condition auxiliary method for the random condition selection.

Parameters

String className	Name of the instance as class name

1.8. writeChooseLoopCountAspect

This method emits the catching code of the choose loop count auxiliary method for the random count selection in the loop inline expression.

Parameters

String className	Name of the instance as class name
String diagramName	Name of the active LSC diagram name.

1.9. writeChooseCoregionAspect

This method emits the catching code of the choose next message selecting method randomly in the coregion. In the coregion, next sending event is selected in this method.

Parameters

String className	Name of the instance as class name

1.10. writeChooseAltAspect

This method emits the catching code of the choose alternative auxiliary method for the random selection in the alternative inline expression

Parameters

String className	Name of the instance as class name
String diagramName	Name of the active LSC diagram name.

1.11. writeVariableDeclarationsAspect

This method emits the variable declarations coming from the model in the LSC instance.

Parameters

ArrayList variablesForInstance	List holds the variables in the current instance

1.12. writeChooseCond

This method emits the choose condition auxiliary method for the random condition selection

1.13. writeChooseAlt

This method emits the choose alternative auxiliary method for the random selection in the alternative inline expression

1.14. writeChooseCoregion

This method emits the choose next message selecting method randomly in the coregion. In the coregion, next sending event is selected in this method.

1.15. writeChooseLoopCount

This method emits the loop count auxiliary method for the random count selection in the loop inline expression.

APPENDIX D

A CODE GENERATION EXAMPLE

In this section, concrete model of an example LSC is presented. Then this concrete model is modeled in GME by using LSC metamodel developed by Topçu. After the modeling, our generator is run. Generator first constructs corresponding intermediate form of the model then second it generates corresponding source codes of the intermediate form. In this section briefly all process (in four view namely concrete model view, GME model view, intermediate form view and code view) of LSC code generation is presented step by step.

1. Concrete Model View

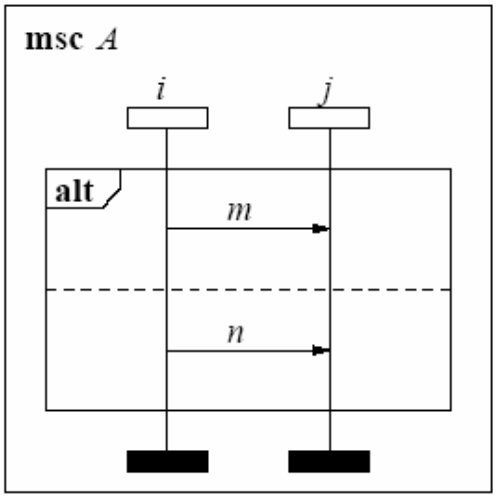


Figure D.1. B29 in Z120 AnnB

2. GME Model View

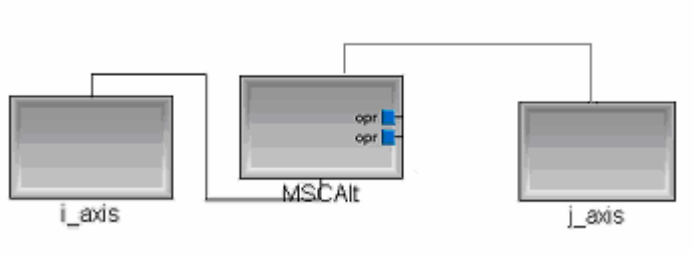


Figure D.2. GME Model of Instance Alt Inline Expression

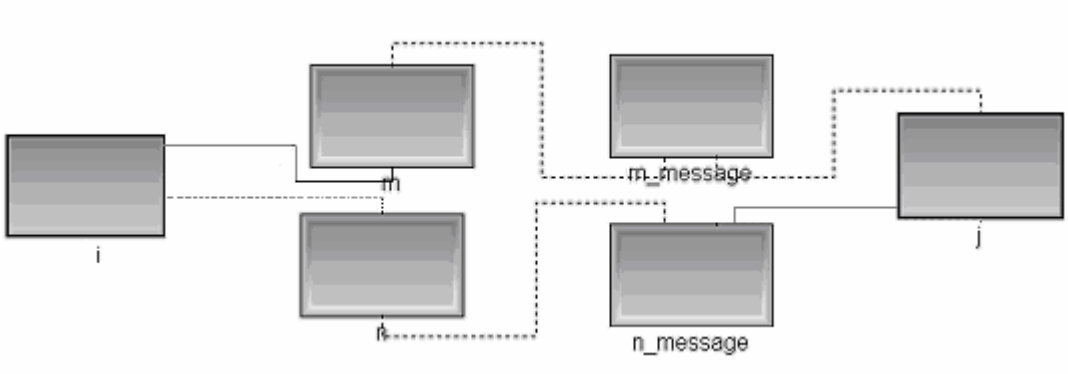


Figure D.3. GME Model of First Operand of the ALT Inline Expression

3. Intermediate Form View (Data Structure/Memory Heap)

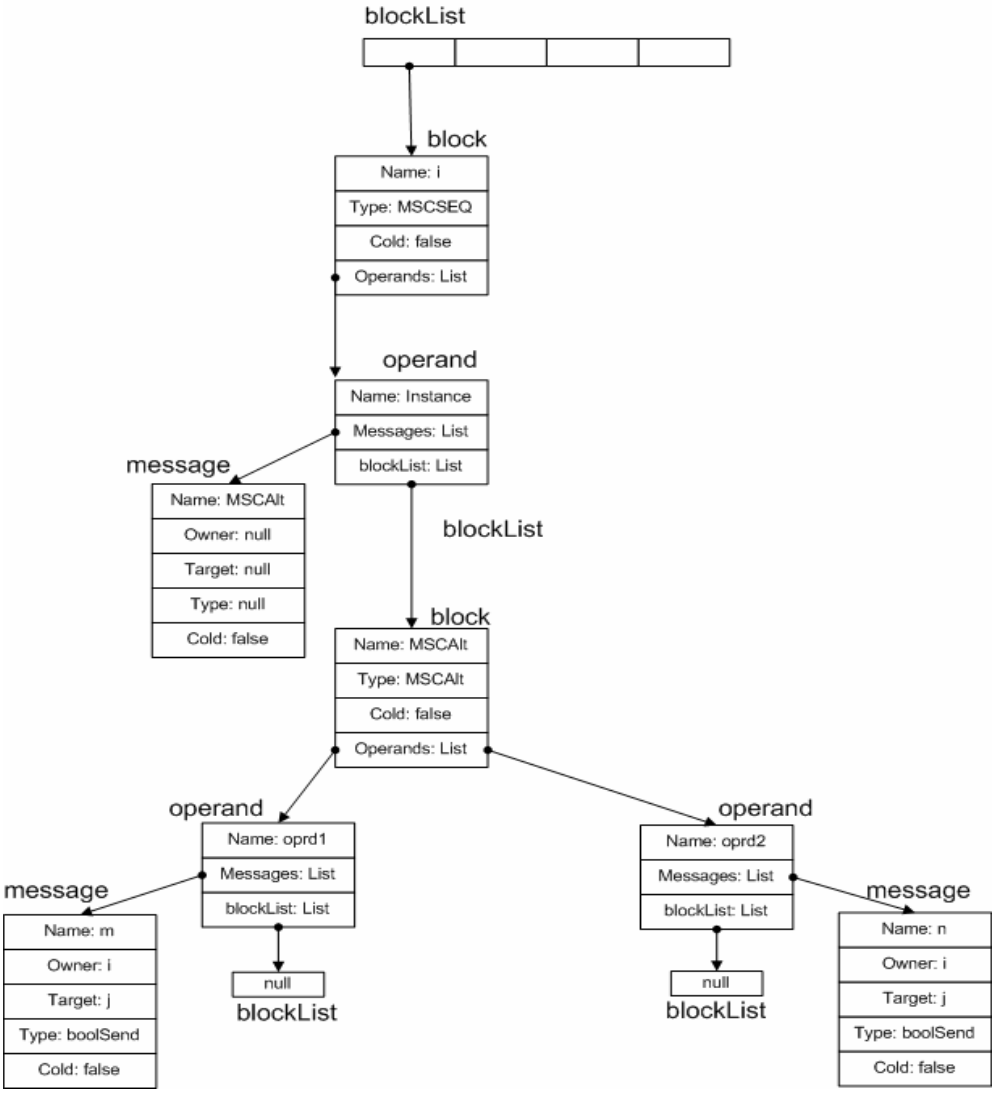


Figure D.4. Intermediate Form of Instance i

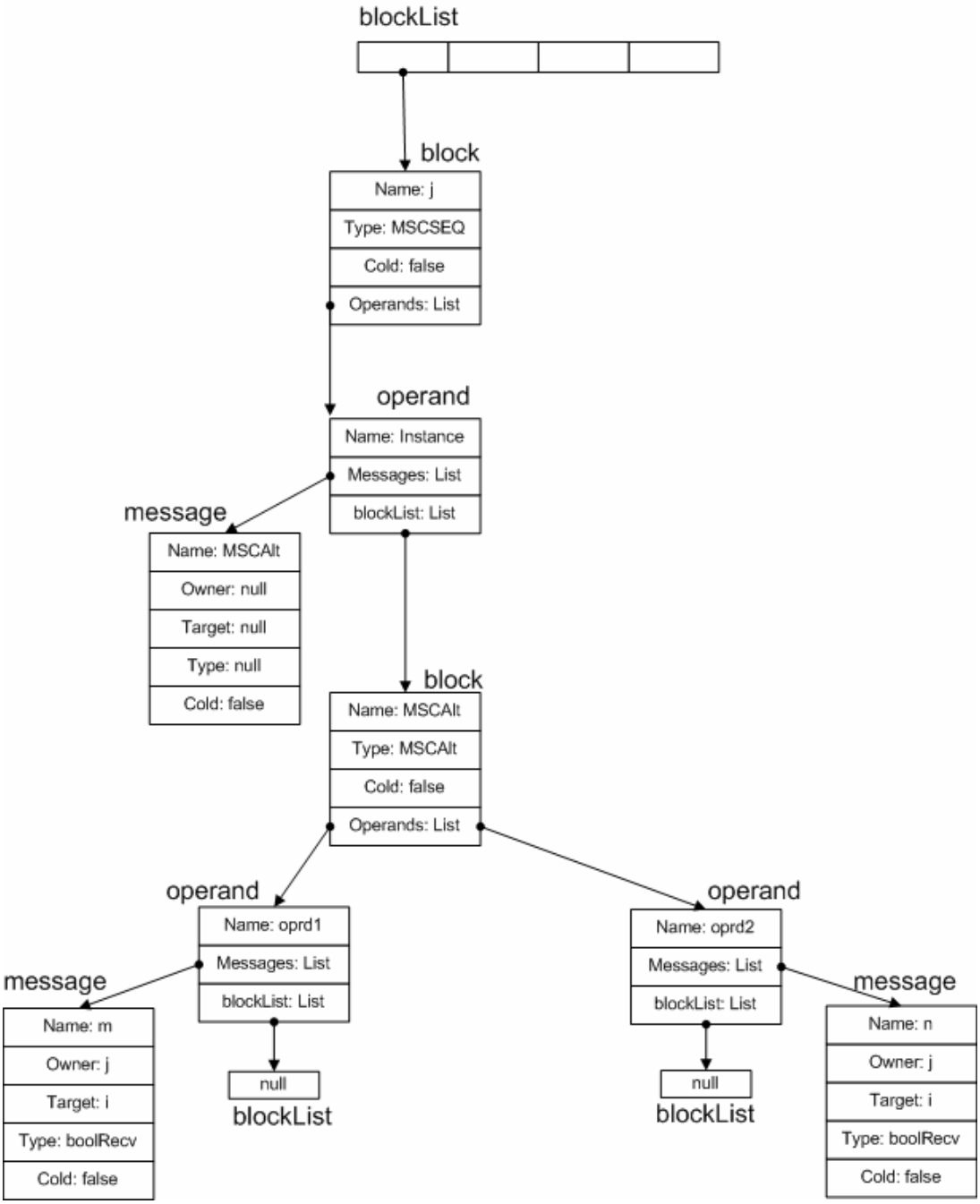


Figure D.5. Intermediate Form of Instance j

4. Code View

```
public static void iMainLoop(){
        int MSCAlt=((Integer)altChoices.get("MSCAlt")).intValue();
        switch(MSCAlt){
        case 0:
                Sendmj(new Integer(0));
                break;
        case 1:
                Sendnj(new Integer(0));
                break;
        }
}
```

Figure D.6. Generated Main Loop Code of Instance i

```
public static void jMainLoop(){
        int MSCAlt=((Integer)altChoices.get("MSCAlt")).intValue();
        switch(MSCAlt){
        case 0:
                condRecvmi();
                break;
        case 1:
                condRecvni();
                break;
        }
}
```

Figure D.7. Generated Main Loop Code of Instance j

APPENDIX E

LSC EXAMPLES AND THEIRS CODE EQUIVALENCY

In this appendix, LSC/MSC models which are retrieved from the literature and their corresponding source code is presented. All variety of MSC/LSC constructs is included such as ALT, LOOP, SEQ inline expressions, Gate, general ordering.

ALT (Alternative)

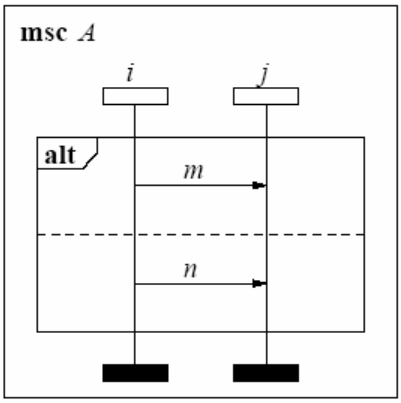


Figure E.1. B29 in Z120 AnnB

```
public static void iMainLoop(){
                int MSCAlt=((Integer)altChoices.get("MSCAlt")).intValue();
                switch(MSCAlt){
                case 0:
                        Sendmj(new Integer(0));
                        break;
                case 1:
                        Sendnj(new Integer(0));
                        break;
                }
        }
```

Figure E.2. Code of instance i

```
public static void jMainLoop(){
        int MSCAlt=((Integer)altChoices.get("MSCAlt")).intValue();
        switch(MSCAlt){
        case 0:
                condRecvmi();
                break;
        case 1:
                condRecvni();
                break;
        }
}
```

Figure E.3. Code of instance j

PAR (Parallel)

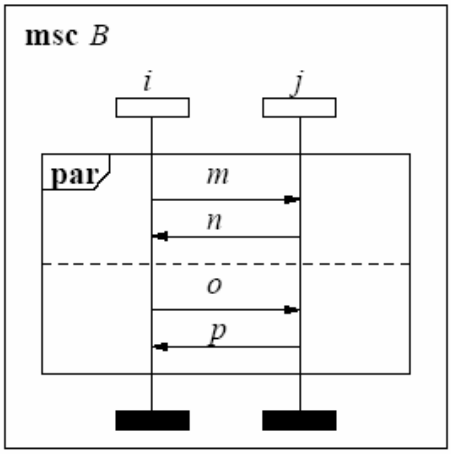


Figure E.4. B29 in Z120 AnnB

```
public static void iMainLoop(){
        class MSCOperand1 extends Thread {
            MSCOperand1() {}
            public void run() {
                Sendmj(new Integer(0));
                condRecvnj();
                stop();
            }
        }
        MSCOperand1 p0 = new MSCOperand1();
        p0.start();
        class MSCOperand2 extends Thread {
            MSCOperand2() {}
            public void run() {
                Sendoj(new Integer(0));
                condRecvpj();
                stop();
            }
        }
        MSCOperand2 p1 = new MSCOperand2();
        p1.start();
        while(p1.isAlive()||p0.isAlive());
    }
```

Figure E.5. Code of instance i

```
public static void jMainLoop(){
        class MSCOperand1 extends Thread {
            MSCOperand1() {}
            public void run() {
                condRecvmi();
                Sendni(new Integer(0));
                stop();
            }
        }
        MSCOperand1 p0 = new MSCOperand1();
        p0.start();
        class MSCOperand2 extends Thread {
            MSCOperand2() {}

            public void run() {
                condRecvoi();
                Sendpi(new Integer(0));
                stop();
            }
        }
        MSCOperand2 p1 = new MSCOperand2();
        p1.start();
        while(p1.isAlive()||p0.isAlive());
    }
```

Figure E.6. Code of instance j

LOOP (Loop)

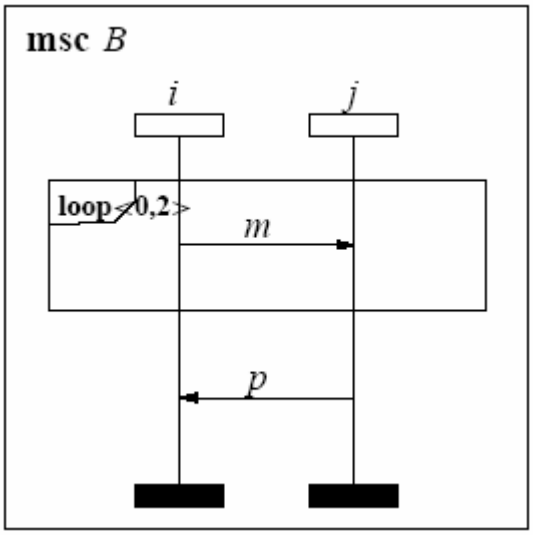


Figure E.7. B31 in Z120 AnnB

```
public static void iMainLoop(){
                boolean loopCond=false;
                int countMSCLoop=0;
                int loopCount = getLoopCount("0","2");
                if(loopCount==-1)
                        loopCond=true;
                while(countMSCLoop<loopCount || loopCond)
                {
                        Sendmj(new Integer(0));
                        countMSCLoop++;
                }//end of loop
                condRecvpj();
        }
```

Figure E.8. Code of instance i

```
public static void jMainLoop(){
        boolean loopCond=false;
        int countMSCLoop=0;
        int loopCount = getLoopCount("0","2");
        if(loopCount==-1)
                loopCond=true;
        while(countMSCLoop<loopCount || loopCond)
        {
                condRecvmi();
                countMSCLoop++;
        }//end of loop
        Sendpi(new Integer(0));
    }
```

Figure E.9. Code of instance j

SEQ (Sequential)

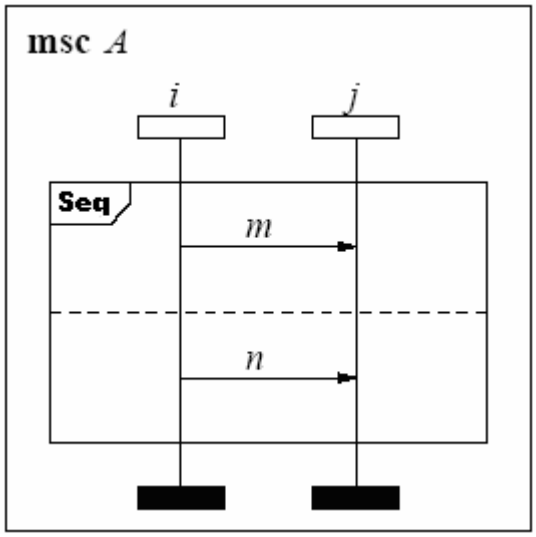


Figure E.10. Derived from B29 in Z120 AnnB

```
public static void iMainLoop(){
                Sendmj(new Integer(0));
                Sendnj(new Integer(0));
        }
```

Figure E.11. Code of instance i

```
public static void jMainLoop(){
                condRecvmi();
                condRecvni();
        }
```

Figure E.12. Code of instance j

EXC (Exclusion)

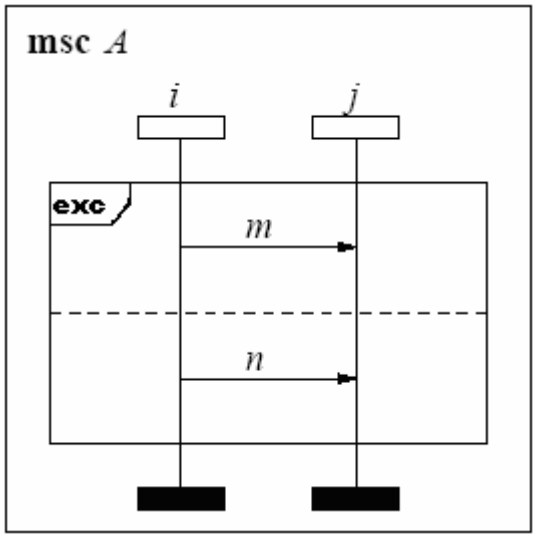


Figure E.13. Derived from B29 in Z120 AnnB

```
public static void iMainLoop(){
        try{
            Sendmj(new Integer(0));
        }
        catch(Exception ex)
         {
            Sendnj(new Integer(0));
        }
    }
```

Figure E.14. Code of instance i

```
public static void jMainLoop(){
        try{
            condRecvmi();
        }
        catch(Exception ex)
        {
            condRecvni();
        }
    }
```

Figure E.15. Code of instance j

OPT (Optional)

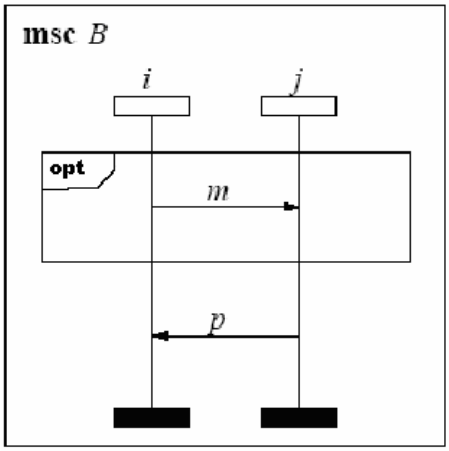


Figure E.16. Derived from B31 in Z120 AnnB

```
public static void iMainLoop(){
        if(((Boolean)coldChoices.get("MSCOpt")).booleanValue()){
            {
                Sendmj(new Integer(0));
            }
        }
        condRecvpj();
    }
```

Figure E.17. Code of instance i

```
public static void jMainLoop(){
        if(((Boolean)coldChoices.get("MSCOpt")).booleanValue()){
            {
                condRecvmi();
            }
        }
        Sendpi(new Integer(0));
    }
```

Figure E.18. Code of instance j

Chart

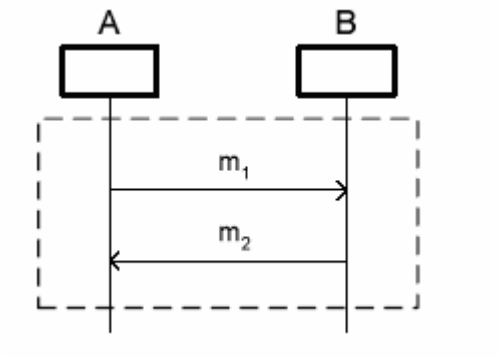


Figure E.19. Madsen paper – (Existential chart Figure2.9)

```
public static void AMainLoop(){
        if(((Boolean)coldChoices.get("A")).booleanValue()){
            {
                Sendm1B(new Integer(0));
                condRecvm2B();
            }
        }
    }
```

Figure E.20. Code of instance A

```
public static void BMainLoop(){
        if(((Boolean)coldChoices.get("B")).booleanValue()){
            {
                condRecvm1A();
                Sendm2A(new Integer(0));
            }
        }
    }
```

Figure E.21. Code of instance B

Pre-chart

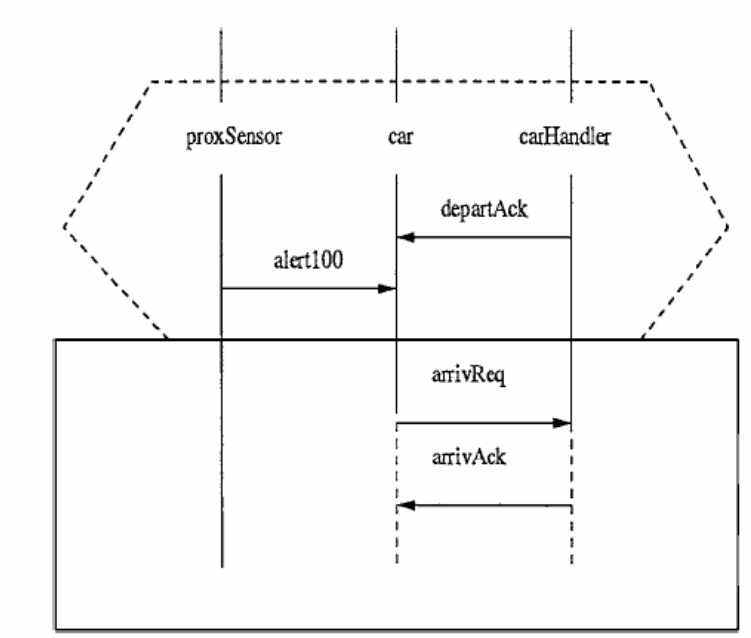


Figure E.22. Damm paper (Figure5)

```
public static void proxSensorMainLoop(){
        boolean condPreChart=false;
        condPreChart= Sendalert100car(new Integer(0)) || condPreChart;
        if(condPreChart){
                //if clause start
        }////if closed end
}
```

Figure E.23. Code of instance proxSensor

```
public static void carMainLoop(){
        boolean condPreChart=false;
        condPreChart= condRecvdepartAckcarHandler() || condPreChart;
        condPreChart= condRecvalert100proxSensor() || condPreChart;
        if(condPreChart){
                //if clause start
                if(((Boolean)coldChoices.get("arrivReq")).booleanValue()){
                        SendarrivReqcarHandler(new Integer(0));
                }//cold
                if(((Boolean)coldChoices.get("arrivAck")).booleanValue()){
                        condRecvarrivAckcarHandler();
                }//cold
        }////if closed end
}
```

Figure E.24. Code of instance car

```
public static void carHandlerMainLoop(){
        boolean condPreChart=false;
        condPreChart= SenddepartAckcar(new Integer(0)) || condPreChart;
        if(condPreChart){
                //if clause start
                condRecvarrivReqcar();
                if(((Boolean)coldChoices.get("arrivAck")).booleanValue()){
                        SendarrivAckcar(new Integer(0));
                }//cold
        }////if closed end
}
```

Figure E.25. Code of instance carHandler

DoWhile

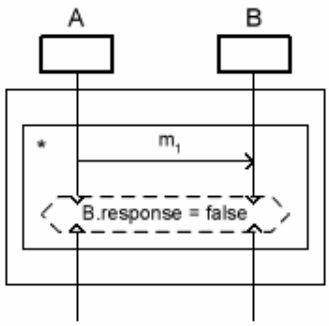


Figure E.26. Madsen paper (Figure2.13)

```
public static void AMainLoop(){
        do {
            Sendm1B(new Integer(0));
        } while(!((Boolean)coldChoices.get("BResponse")).booleanValue());
    }
```

Figure E.27. Code of instance A

```
public static void BMainLoop(){
        do {
            condRecvm1A();
        } while(!((Boolean)coldChoices.get("BResponse")).booleanValue());
    }
```

Figure E.28. Code of instance B

WhileDo

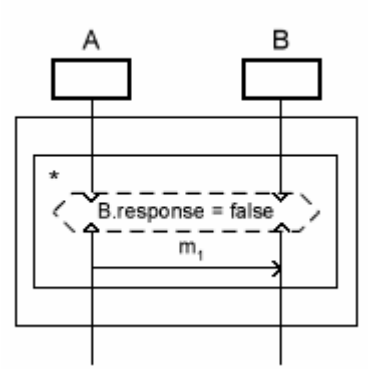


Figure E.29. Madsen paper (Figure2.14)

```
public static void AMainLoop(){
        while(((Boolean)coldChoices.get("BResponse")).booleanValue()){
            Sendm1B(new Integer(0));
        }
    }
```

Figure E.30. Code of instance A

```
public static void BMainLoop(){
        while(((Boolean)coldChoices.get("BResponse")).booleanValue()){
            condRecvm1A();
        }
    }
```

Figure E.31. Code of instance B

IfThen

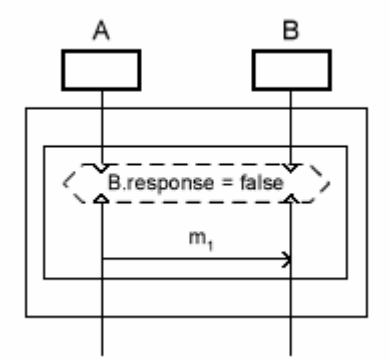


Figure E.32. Madsen paper (Figure2.15)

```
public static void AMainLoop(){
        if(((Boolean)coldChoices.get("BResponse")).booleanValue()){
            Sendm1B(new Integer(0));
        }
    }
```

Figure E.33. Code of instance A

```
public static void BMainLoop(){
        if(((Boolean)coldChoices.get("BResponse")).booleanValue()){
            condRecvm1A();
        }
    }
```

Figure E.34. Code of instance B

IfThenElse

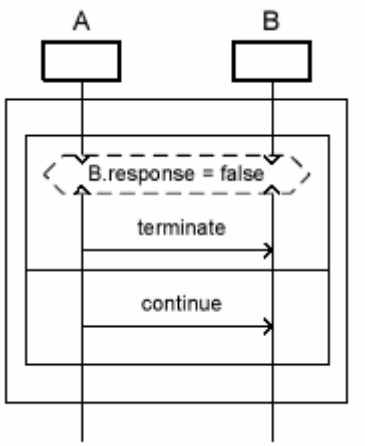


Figure E.35. Madsen paper (Figure2.16)

```
public static void AMainLoop(){
		if(((Boolean)coldChoices.get("BResponse")).booleanValue()){
			SendterminateB(new Integer(0));
		}
		else {
			SendcontinueB(new Integer(0));
		}
	}
```

Figure E.36. Code of instance A

```
public static void BMainLoop(){
		if(((Boolean)coldChoices.get("BResponse")).booleanValue()){
			condRecvterminateA();
		}
		else {
			condRecvcontinueA();
		}
	}
```

Figure E.37. Code of instance B

Dynamic Instance Creation

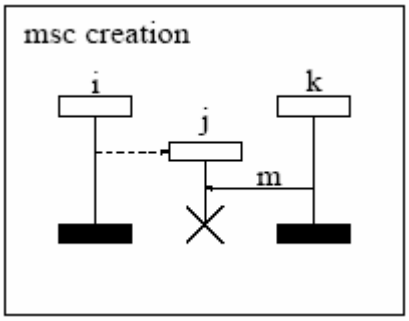


Figure E.38. B6 in Z120 AnnB

```
public static void iMainLoop(){
                B6Diagram.pj.start();
        }
```

Figure E.39. Code of instance i

```
public static void jMainLoop(){
                condRecvmk();
                B6Diagram.pj.stop();
        }
```

Figure E.40. Code of instance j

```
public static void kMainLoop(){
                Sendmj(new Integer(0));
        }
```

Figure E.41. Code of instance k

Condition

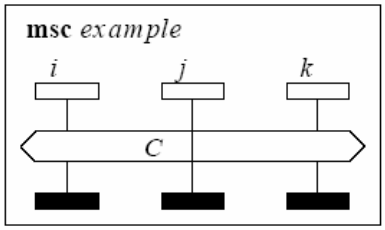


Figure E.42. B13 in Z120 AnnB

Example condition is hot condition.

```
public static void iMainLoop(){
        if(((Boolean)B13Diagram.coldChoices.get("MSCCondition")).booleanValue())
         {//condo start
           }//condo end
        else//Hot condition
                return;//Hot condo
        }
```

Figure E.43. Code of instance i

```
public static void kMainLoop(){
        if(((Boolean)B13Diagram.coldChoices.get("MSCCondition")).booleanValue())
        {//condo start
        }//condo end
        else//Hot condo
                return;//Hot condo
        }
```

Figure E.44. Code of instance k

Timing

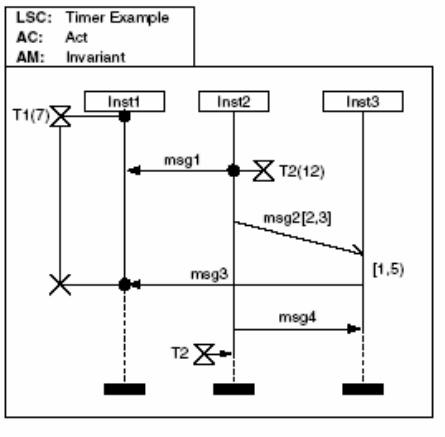


Figure E.45. Brill Paper

```
public static void inst1MainLoop(){
                doLaterT1(7);
                condRecvmsg1inst2();
                condRecvmsg3inst3();
                calcelT1();
                doLaterT1(7);
        }
        static Timer timerT1= new Timer();
        static boolean timerFlagT1=false;
        public static void  doLaterT1( long delayInMillis )
        {
                ScheduleRunnerT1 scheduleRunner = new ScheduleRunnerT1();
                timerT1.schedule( scheduleRunner, delayInMillis );
        }
```

Figure E.46. Code of instance inst1

```
public static void  calcelT1()
{
        timerT1.cancel();
}
static class ScheduleRunnerT1 extends TimerTask
{
        public void run()
        {
                timerFlagT1=true;
        }
}
```

Figure E.46. Code of instance inst1 (continue)

```
public static void inst2MainLoop(){
        doLaterT2(12);
        Sendmsg1inst1(new Integer(0));
        Sendmsg2inst3(new Integer(0));
        Sendmsg4inst3(new Integer(0));
        if(timerFlagT2)
        {
                RecvT2Timeout();
        }
}
```

Figure E.47. Code of instance inst2

```
static Timer timerT2= new Timer();
static boolean timerFlagT2=false;
public static void  doLaterT2( long delayInMillis )
{
        ScheduleRunnerT2 scheduleRunner = new ScheduleRunnerT2();
        timerT2.schedule( scheduleRunner, delayInMillis );
}
public static void  calcelT2()
{
        timerT2.cancel();
}
static class ScheduleRunnerT2 extends TimerTask
{
        public void run()
        {
                timerFlagT2=true;
        }
}
```

Figure E.47. Code of instance inst2 (Continue)

Lost/Found Messages/Action

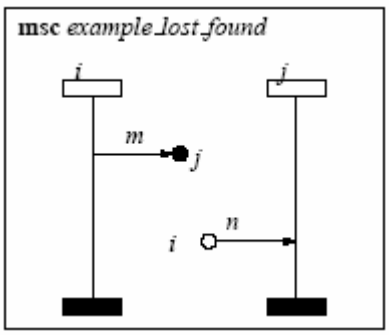


Figure E.48. B11 in Z120 AnnB

```
public static void iMainLoop(){
                SendmLOST(new Integer(0));
}
```

Figure E.49. Code of instance i

```
public static void jMainLoop(){
                condRecvnFOND();
}
```

Figure E.50 Code of instance j

General Ordering

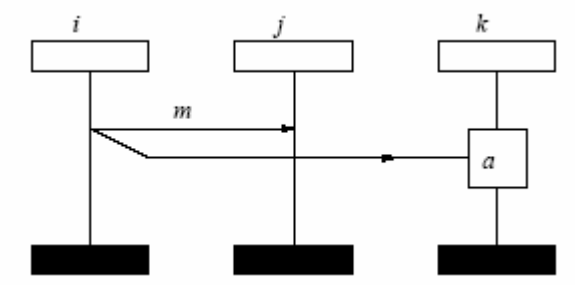


Figure E.51. B18 in Z120 AnnB

```
public static void iMainLoop(){
                Sendmj(new Integer(0));
                B18Diagram.setMSCGeneralOrder=true;
        }
```

Figure E.52. Code of instance i

```
public static void jMainLoop(){
                condRecvmi();
        }
```

Figure E.53. Code of instance j

```
public static void kMainLoop(){
                while(!B18Diagram.setMSCGeneralOrder);
                SendaACTN(new Integer(0));
        }
```

Figure E.54. Code of instance k

Coregion

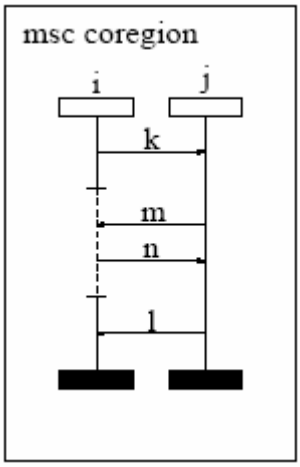


Figure E.55. B16 in Z120 AnnB

```
public static void iMainLoop(){
                Sendkj(new Integer(0));
                ArrayList selectedList = new ArrayList();
                int n;
                int i;
                selectedList.add(new Integer(2));
                selectedList.add(new Integer(3));
                n=2;
                i=1;
                while(i<=n)
                {
                        int choice=chooseOne(selectedList,orderList);
                        switch(choice)
                        {//switch
                        case 2:
                                if (boolmj())
                                {
                                    condRecvmj() ;
                                 }
                                 break;
                        case 3:
                                 Sendnj(new Integer(0));
                                 break;
                        }//switch
                        i++;
                }
                selectedList.clear();
                condRecvlj();
        }
```

Figure E.56. Code of instance i

```
public static void jMainLoop(){
                condRecvki();
                Sendmi(new Integer(0));
                condRecvni();
                Sendli(new Integer(0));
        }
```

Figure E.57. Code of instance j

Temperature Property

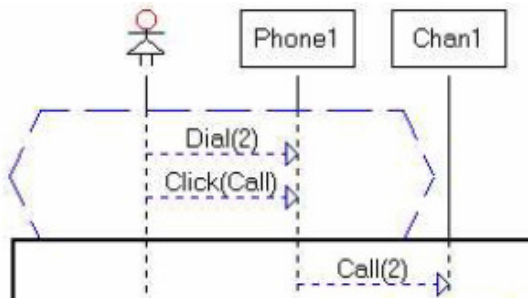


Figure E.58. Harel Paper Figure-5

```
public static void UserMainLoop(){
        boolean condPreChart=true;
       if(((Boolean)coldChoices.get("Dial")).booleanValue(){
        condPreChart= SendDialPhone1(new Integer(0)) || condPreChart ;
        }
       if(((Boolean)coldChoices.get("Click")).booleanValue()){
        condPreChart= SendClickPhone1(new Integer(0))||condPreChart;
        }
        if(condPreChart){
                //if clause start
        }////if closed end
}
```

Figure E.59. Code of instance User

```
public static void Phone1MainLoop(){
        boolean condPreChart=true;
        for (int i=0;i<50;i++) {
           if (boolDialUser()) {
                    condPreChart= condRecvDialUser() ||condPreChart;
                     break;
              }
              SleepThread(100);
            }//cold
          for (int i=0;i<50;i++) {
            if (boolClickUser ()) {
                     condPreChart= condRecvClickUser() || condPreChart;
                    break;
              }
              SleepThread(100);
            }//cold
        if(condPreChart){
                //if clause start
                if(((Boolean)coldChoices.get("Call")).booleanValue()){
                        SendCallChan1(new Integer(0));
                }//cold
        }////if closed
}
```

Figure E.60. Code of instance Phone1

```
public static void Chan1MainLoop(){
        boolean coldLoc=false;
       for (int i=0;i<50;i++) {
        if (boolCallPhone1()) {
                condRecvCallPhone1();
                coldLoc=true;
               break;
          }
         if(coldLoc)//for cold receive message in cold location
           return;
}
```

Figure E.61. Code of instance Chan1

An example of cold inline expression is presented in Existential-Chart part and related code segment is shown below.

```
if(((Boolean)coldChoices.get("A")).booleanValue()){
                {
                        Sendm1B(new Integer(0));
                        condRecvm2B();
                }
          }
```

Figure E.62. Code of cold chart (Existential Chart) example

An example of cold location is presented in Pre-Chart part and related code segment is shown in Figure E-61.

Composite LSC Structures

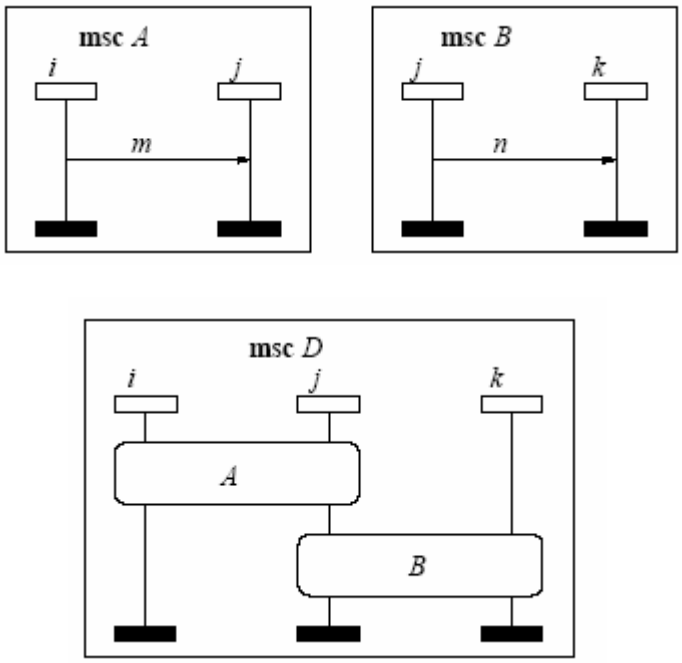


Figure E.63. B33 in Z120 AnnB (D=A seq B)

Diagram A codes:

```
public static void iMainLoop(){
                Sendmj(new Integer(0));
        }
```

Figure E.64. Code of instance i

```
public static void jMainLoop(){
                condRecvmi();
        }
```

Figure E.65. Code of instance j

Diagram B codes:

```
public static void jMainLoop(){
                Sendnk(new Integer(0));
        }
```

Figure E.66. Code of instance j

```
public static void kMainLoop(){
                condRecvnj();
        }
```

Figure E.67. Code of instance k

Diagram D codes:

```
public static void iMainLoop(){
                Sendmj(new Integer(0));
        }
```

Figure E.68. Code of instance i

```
public static void iMainLoop(){
                condRecvmi();
               Sendnk(new Integer(0));
        }
```

Figure E.69. Code of instance j

```
public static void kMainLoop(){
                condRecvnj();
        }
```

Figure E.70. Code of instance k

Gate

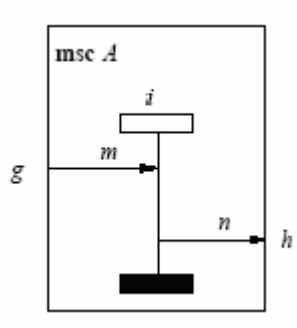


Figure E.71. B38 in Z120 AnnB

```
public static void iMainLoop(){
                A.condRecvmg();
                Sendnh(new Integer(0));
        }
```

Figure E.72. Code of instance j

APPENDIX F

INTEGRATION OF HLA METHODS WITH LSC MODEL: FRONT END

Intermediate form generation continued in the external library's input model. In our example case, generator also walks on the HFMM, HOMM models in the FAMM. In this appendix, traversing methods and their explanation is given. This is the extension of intermediate from generation module (front end) in Figure 3.3.

LSCCodeGen.LSCObject generateExternalLibAspectCodes

This method generates the federation execution aspects to be used for receiving call-back events from the RTI for every federation execution

Parameters

ArrayList targetList	List holds the federation executions
String InstanceName	Current LSC instance name.
String diagramName	Current diagram name in which to be traversed
String PATH	Path of the generated aspect.

LSCCodeGen.LSCObject traverseExternalObject

This method traverses the HSMM method's one by one. It is called from the LSCCodeGen. This method is the integration point for external library and LSC. Instead of the *LSCMessage* model element in the LSC, this method is called. This HSMM method (also called *HLAMethod*) is derived from the *LSCMessage*. *HLAMethod* information is retrieved from FAMM especially HSMM, FSMM and HOMM.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getSuppliedArguments

This method traverses the HSMM method's supplementary argument model element and retrieves the supplementary argument's model element.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getReturnedArguments

This method traverses the HSMM method's returned argument model element and retrieves the returned argument's model element.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getIndicator

This method traverses the HSMM method's argument which is an indicator model element and retrieves the indicator parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getOrderType

This method traverses the HSMM method's argument which is an order model element and retrieves the order parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getStringType

This method traverses the HSMM method's argument which is a string model element and retrieves the string parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

FEDERATION MANAGEMENT

LSCCodeGen.LSCObject getCreateFederationExecutionData

This method traverses the Create Federation Execution HSMM method and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getFederate

This method traverses the federate model element in FSMM and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getFederateSet

This method used in the methods of HSMM whose supplementary arguments have a federate set. It traverses the set and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.

JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getFederateResponsePairs

This method used in the methods of HSMM whose supplementary arguments have a federate responses pairs set. It traverses the set and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getFederateSavePairs

This method used in the methods of HSMM whose supplementary arguments have a federate save pairs set. It traverses the set and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getJoinFederationExecutionData

This method traverses the Join Federation Execution HSMM method and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getResignFederationExecutionData

This method traverses the Resign Federation Execution HSMM method and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getSynchronizationData

This method used in the methods of HSMM whose supplementary arguments have a synchronization label. It traverses the label and retrieves the method parameter information.

Parameters	
LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name

DECLARATION MANAGEMENT

LSCCodeGen.LSCObject getInteractionClassFromMessage

This method used in the methods of HSMM whose supplementary arguments have an interaction class. It traverses the class and retrieves the method parameter information.

Parameters	
LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getObjectClassFromMessage

This method used in the methods of HSMM whose supplementary arguments have an object class. It traverses the class and retrieves the method parameter information.

Parameters	
LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getInteractionClassFromRetraction

This method used in the methods of HSMM whose arguments have a retraction. It traverses the interaction class and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getObjectClassFromMessage

This method used in the methods of HSMM whose arguments have a retraction. It traverses the object class and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getOnlyObjectAttributesFromMessage

This method used in the methods of HSMM whose supplementary arguments have only object attributes. It traverses the attributes and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getObjectAttributesFromMessage

This method used in the methods of HSMM whose supplementary arguments have object class and its object attributes. It traverses the object class and its attributes and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getObjectAttributesAndRegionsFromMessage

This method used in the methods of HSMM whose supplementary arguments have object attribute sets and region sets. It traverses the sets and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getOnlyAttributeFromMessage

This method used in the methods of HSMM whose supplementary arguments have only an object attribute. It traverses the attribute and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
isObjName	Label is used as a name for the synchronization object name
boolean regionIsEnabled	Region is enabling or not.

OBJECT MANAGEMENT

LSCCodeGen.LSCObject getChangeInteractionTransportationType

This method used in the methods of HSMM whose supplementary arguments have an transportation type to be changed. It traverses the interaction class and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getChangeAttributeTransportationType

This method used in the methods of HSMM whose supplementary arguments have an transportation type to be changed. It traverses the object class and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getObjectAttributesOrderData

This method used in the methods of HSMM whose supplementary arguments have an order type to be applied. It traverses the order model element and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getRemoveObjectInstance

This method used in the methods of HSMM whose supplementary arguments have an object class to be removed. It traverses the object class and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getReflectAttributeValues

This method used in the methods of HSMM whose supplementary arguments have an object class to be reflected. It traverses the object class and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getReceiveInteraction

This method used in the methods of HSMM whose supplementary arguments have an interaction class to be received. It traverses the interaction class and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element
boolean regionIsEnabled	Region is enabling or not.

LSCCodeGen.LSCObject getDeleteObjectInstance

This method used in the methods of HSMM whose supplementary arguments have an object class to be deleted. It traverses the object class and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

DATA DISTRIBUTION MANAGEMENT METHODS

LSCCodeGen.LSCObject getRegionsFromMessage

This method used in the methods of HSMM whose supplementary arguments have a region set. It traverses the set and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getRegionDataFromRef

This method used in the methods of HSMM whose supplementary arguments have a region reference. It traverses the reference and retrieves the method parameter information.

Parameters

LSCCodeGen.LSCObject obj	Input *LSCObject* of the message.
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject addRegionDimension

This method used in the interaction methods which has a dimension. It traverses the dimension and retrieves the method parameter information and adds to the interaction.

Parameters

JBuilderModel InteractionClass	Input interaction class.
JBuilderModel Message	Message model element
LSCObject obj	Input *LSCObject* of the message

LSCCodeGen.LSCObject addRegionDimension

This method used in the object methods whose attribute has a dimension. It traverses the dimension and retrieves the method parameter information and adds to the attribute.

Parameters

JBuilderModel attribute	Input object attribute.
JBuilderModel Message	Message model element
LSCObject obj	Input *LSCObject* of the message

LSCCodeGen.LSCObject getDimData

This method traverses the dimension and retrieves the dimension information.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getRegions

This method traverses the region set and retrieves the region on it. This is used to *createregion* method.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getRegion

This method traverses the region and retrieves the region information. This is used in region related method.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getRegion

This method traverses the region reference and retrieves the region information. This is used in region related method.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element
boolean isObjName	Region is used as a LSC object name

TIME MANAGEMENT METHODS

LSCCodeGen.LSCObject getTimeData

This method traverses the time model element in the supplementary arguments of the HSMM method and retrieves the time stamp information.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element

boolean isObjName	Region is used as a LSC object name

LSCCodeGen.LSCObject getLookahead

This method traverses the lookhead model element in the supplementary arguments of the HSMM method and retrieves the lookhead information.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element
boolean isObjName	Region is used as a LSC object name

SUPPORT SERVİCE

LSCCodeGen.LSCObject getMultipleCallbacksData

This method traverses the multiple callback model element in the supplementary arguments of the HSMM method and retrieves the callback information.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getCallbackData

This method traverses the callback model element in the supplementary arguments of the HSMM method and retrieves the callback information.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element

LSCCodeGen.LSCObject getBoundsData

This method traverses the region set bounds model element in the supplementary arguments of the HSMM method named *setregionbounds* and retrieves the bounds information.

Parameters

LSCObject obj	Input *LSCObject* of the message
JBuilderModel Message	Message model element

APPENDIX G

INTEGRATION OF HLA METHODS WITH LSC MODEL: BACK END

In this appendix, code generator gets external library related intermediate form information and emits the corresponding code segments. This is the extension of Java code generation module (back end) in Figure 3.3.

LSCCodeGen.LSCObject writeExternalLibAspect

This method simply emits the federation execution aspect code for each federation execution.

Parameters

String fedName	Federate name
String RTILib	External library name to reach external library in our case, LscRTILib name to reach actual pRTI.
String rtiName	Federation execution name
String diagramName	Current LSC diagram name
String LSCLib	Library in which *LSCObject* is declared.

LSCCodeGen.LSCObject SendExternalMethods

This method emits the external sending method's definitions. In our case, RTI Ambassador method definitions are generated in the federate base code.

Parameters

LSCCodeGen.LSCObject proc	Input *LSCObject* of the message
String RTILib	External library name to reach external library.

LSCCodeGen.LSCObject writeReceiveExternalMethods

This method emits the external receiving method's definitions. In our case, federate Ambassador method definitions are generated in the federate base code. But, method bodies are emitted by *writeReceiveMethods* method in the following.

Parameters	
ArrayList recvListClassHandlers	List holds the receiving events.
boolean ExtLibEnabled	Indicate whether external library is enabled or not.
String libraryStr	Library in which *LSCObject* is declared.
String RTILib	External library name to reach external library
String className	Instance name, in our case federate name

LSCCodeGen.LSCObject writeReceiveAspectMethods

This method emits the catching aspect codes into the federation execution aspect to handle callback forwarding to the base code.

Parameters	
ArrayList recvListClassHandlers	List holds the receiving events.
String className	Instance name, in our case federate name

LSCCodeGen.LSCObject writeReceiveMethods

This method emits the external receiving method's bodies.

Parameters	
ArrayList recvListClassHandlers	List holds the receiving events.
String libraryStr	Library in which *LSCObject* is declared.
String ExternalLib	External library name to reach external library
String activeDiagramName	Current LSC diagram name

APPENDIX H

CODE GENERATOR USER GUIDE

This documents presents information for the practical use of the code generator. Following the provided guidance the user can download and install the code generator and all required supplementary programs. After installation, user can run the generator and produce the source code of his/her input model. User may edit the generated code to reflect his/her computation logic into generated code. User then can run the generated code. Finally, critical modeling points for code generation are listed at the end.

1. How to Download and Install GME, Java JRE, Eclipse, Aspectj Plug-in, and Pitch-RTI

Downloading resources

- Download GME 6.11.9 from the `http://www.isis.vanderbilt.edu/projects/gme/`
- Download Java JRE (Java Runtime Environment) version 5 or later from `http://www.java.com/en/download/manual.jsp`
- Download Eclipse 3.x from the `http://www.eclipse.org/downloads/`
- Download AspectJ Plug-in `ajdt.1.4 for Eclipse 3.2` from `http://www.eclipse.org/aspectj`.
- Download Pitch-RTI evaluation version from `http://www.pitch.se`
- Download code Generator from `http://www.ceng.metu.edu.tr/~e73883`

Installing GME

Run `GME 6.11.9`'s setup file and install GME into the desired folder, e.g. `c:\Program Files\GME`.

Installing Java JRE

Extract the Java JRE compression file into the specified folder, e.g. `C:\Program Files\Java\jre1.5.0`.

Installing Eclipse

1. Extract Eclipse compression file into the desired folder, e.g. `c:\eclipse-SDK-3.0.1-win32`.
2. Run `Eclipse.exe` file in the extracted folder and start Eclipse.
3. Choose workspace folder e.g. `c:\eclipse-SDK-3.0.1-win32\workspace`, while eclipse is started.
4. Select `Windows->Preferences...` menu item and add previously installed Java JRE into the Eclipse (see Figure H-1)

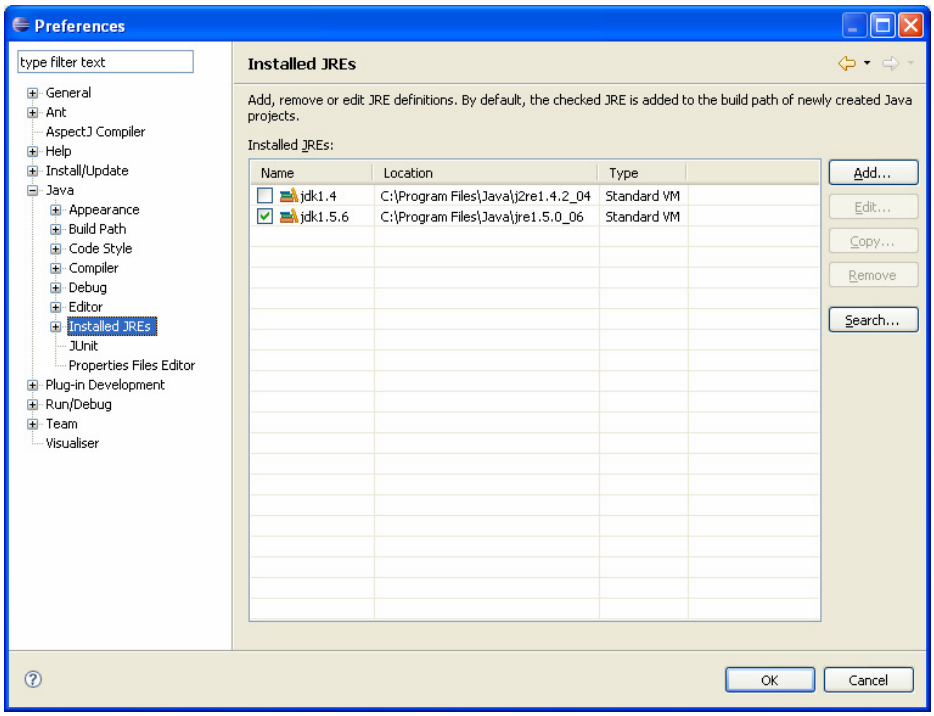


Figure H.1 Eclipse Java JRE Installation Window

Installing Aspect J

1. Extract the AspectJ compressed file (`ajdt.1.4.zip`) into the Eclipse folder (`C:\eclipse-SDK-3.0.1-win32`).
2. Alternatively, if extract the compressed file into another folder, `plug-ins` and `features` folder is created in it. Copy these two folders and paste them into the eclipse folder (`c:\eclipse-SDK-3.0.1-win32`).

Installing RTI middleware

1. Run the downloaded RTI setup. In our example pitch-RTI setup namely `prti1516le_v3.1.1.exe` is run.
2. Installed RTI libraries must be added into the Eclipse project. Select the project in which RTI is included and open properties of the project (see Figure H-2).
3. Select `Java Build Path` and add jar files of the RTI libraries e.g. `c:\Program Files\prti1516le\lib` by pressing `Add External JARs...` button.

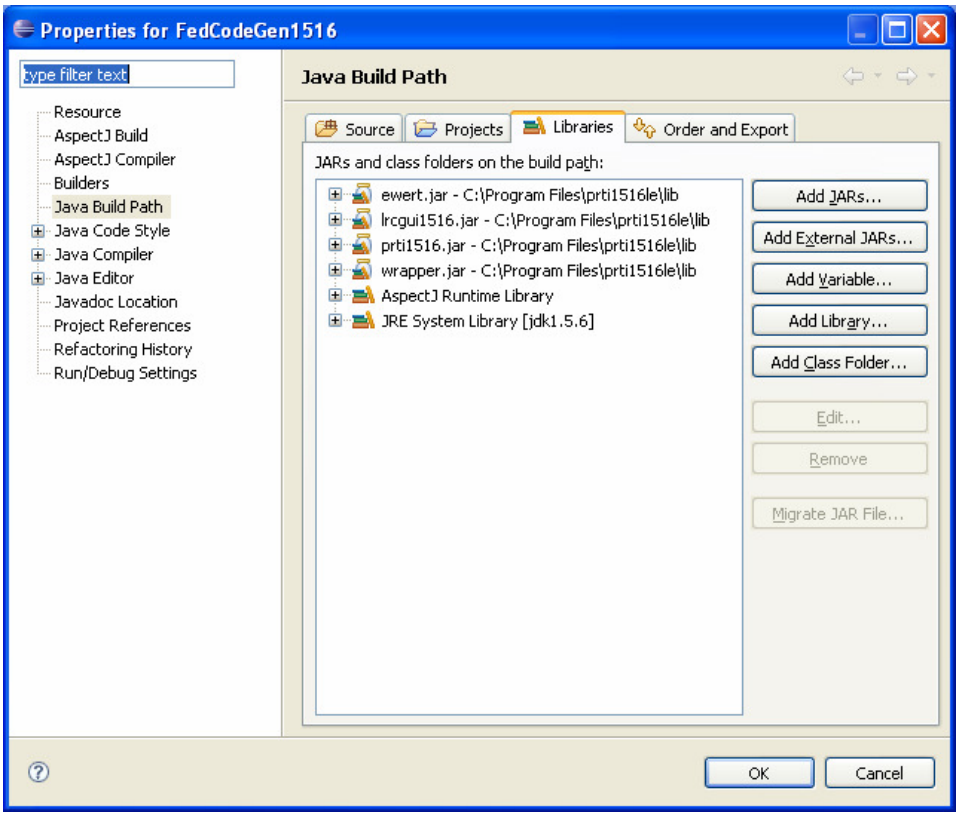


Figure H.2. Adding External Library Jar Files into the Eclipse

2. How to Install Code Generator as a GME Model Interpreter

1. Extract the code generator compression file namely `code_generator.zip` into the desired folder, e.g. `C:\eclipse-SDK-3.0.1-win32\eclipse\workspace\NewCodeGenProject`.
2. Open `Program Files\GME\SDK\Java` folder.
3. Run `JavaCompRegister.exe` model interpreter register program (Figure H-3).

4. Fill `Name`, `Description`, `Menu/Tooltip` fields as desired. An example filling is given in FigureH- 3.
5. `ClassPath` must be the generator extracted folder (`C:\eclipse-SDK-3.0.1-win32\eclipse\workspace\NewCodeGenProject`).
6. Fill the `Class` field (`org.isis.gme.bon.LSCCodeGen`) exactly as in the Figure H-3.
7. Then press the `Register` button.

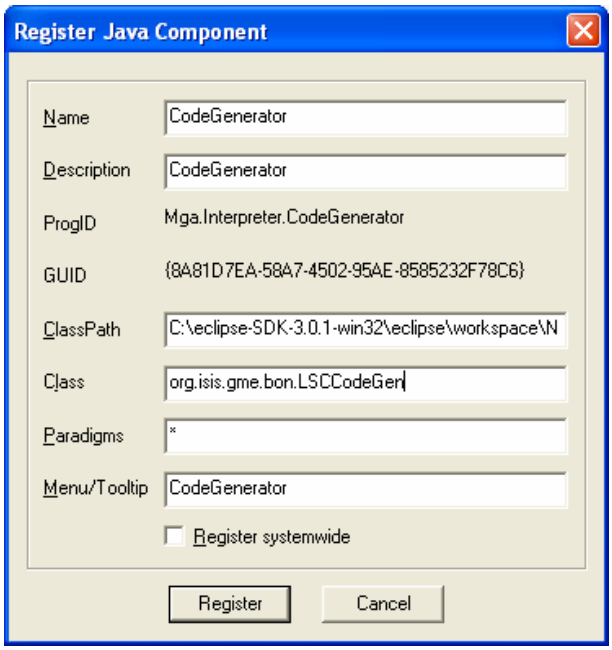


Figure H.3 Component Register Window

8. Select input GME model (`stms.mga` in our example) by double-clicking on the input file so GME is started and opened with the input model.
9. Select `File->Register Components...` menu item in GME.
10. Select the code generator (`LSCInterpreter` in our example) in the components window in Figure H-4.
11. Press the toggle button and enable the registered code generator in the GME environment toolbar. When mouse is moved on the toolbar element, name of the interpreter is shown. (Available components are indicated by an exclamation mark)

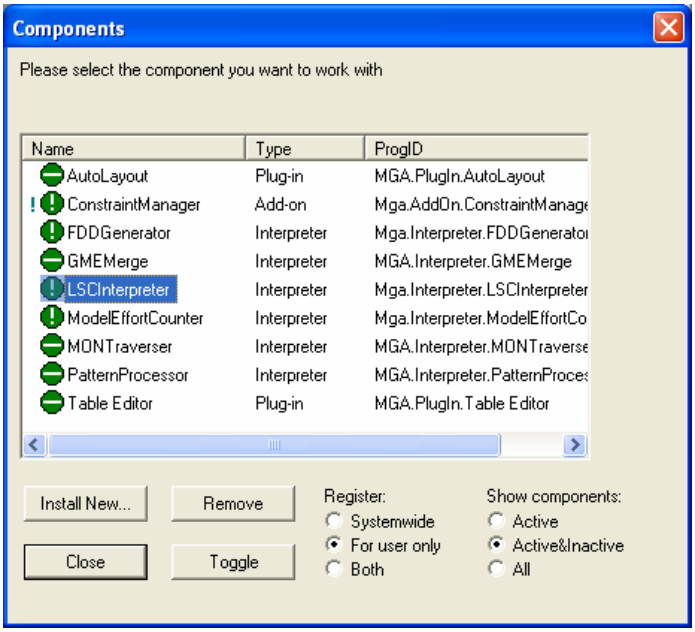


Figure H.4. Registered Components window

3. How to Configure the Generator

Generator provides a configuration document based on XML, called `GeneratorConf.xml`, with initial values for configuration parameters for the users.

1. Define a path variable for the code generator namely `GeneratorPath` in the `Environment Variables` of the Windows XP operating system. `Environment Variables` are accessed in the control panel. This path presents the configuration file path that initially can be code generator path (`C:\eclipse-SDK-3.0.1-win32\eclipse\workspace\NewCodeGenProject`). Note that for the other operating system this definition can be in different way.
2. Configure our example STMS by setting values of the following parameters:
   - Seed for the random number generator,
   - The path for the generated code,
   - The path of the code generator
   - Maximum poll count for receiving an optional (cold) message, and
   - Waiting (sleep) time between two successive polls.
   - External library name and its prefix.

So final configuration XML file can be presented (Figure H-5) as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Confs>
<Random seed="123456"> <!— for random number generator -->
</Random>
<Sleep time="100" passes="50">
<!—sleep time and number of passes for cold message receiving-->
</Sleep>
<PATH>
<Generated path="c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\FedCodeGen1516\">
</Generated> <!—destination path for the generated code-->
<Generator path="c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\NewCodeGenProject
\">
</Generator> <!— path of the generator code-->
</PATH>
<External-InstanceLibs>
 <InstanceLib name="RTILib" prefix="RTI"> <!— external library used in the generator-->
</InstanceLib>
</External-InstanceLibs>
</Confs>
```

Figure H.5. XML Configuration File for the Code Generator for ShipFd

4. How to Run the Generator

1. After the GME input model is opened, run the code generator by clicking on the generator's toolbar button in the GME.
2. The generated code files are placed in the folder (`c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\FedCodeGen1516` for our example) specified in the configuration file. In our example, `Ship_MSC` (Diagram class), *`ShipFd`* (Ship federate class), `User` (Live entity class), `ShipFdAspect` (computation aspect of ship federate), `UserAspect` (computation aspect of user) and `BosporusFederationLibAspect` (federation execution aspect) are generated. Generated three classes and three aspects are shown with a class diagram in Figure H-6.
3. Copy the generated code folder into the Eclipse workspace folder (`c:\eclipse-SDK-3.0.1-win32\eclipse\workspace`). If generated path is set to Eclipse workspace (in our example: `c:\eclipse-SDK-3.0.1-`

`win32\eclipse\workspace\ FedCodeGen1516` is set in the configuration file), it is not required.

4. Open Eclipse and select `File->New->AspectJ Project` menu item.
5. Give generated project folder name as a project name e.g. `FedCodeGen1516`. After that generated codes are appeared in the Eclipse.
6. Add vendor specific RTI library (pRTI) into the project as described above (in our example `Ship_MSC` and `Station_MSC`).
7. Copy the generated FDD file into the project folder (`c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\FedCodeGen1516`) described in the configuration file. In our example `StraitTraffic.xml` is copied.
8. Copy LscRTILib library into the project folder. (Copy LscRTILib folder named `RTIlib` into the `c:\eclipse-SDK-3.0.1-win32\eclipse\workspace\FedCodeGen1516` in our example).

Figure H.6. Class Diagram of the Ship Federate

a) Base Codes for our example

The `Ship_MSC`, `ShipFd` and `User` classes constitute the base code of the ship federate application as shown in Figure H-6. `Ship_MSC` is a diagram code in which the `ShipFd` and `User` threads are defined and run. `ShipFd` is an instance code where federate RTI methods and LSC-specific auxiliary methods are generated. User is also an instance code in which user sends ship name, direction and speed to the ship federate.

```
public static void ShipFdMainMethod (){
 (…) // prechart 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 named MainThread.
 MainThread_02ee() {}
 public void run() {
 do { //loop is repeated until the ReserveObjectInstanceName (ROIN) is succeeded.
  condRecvMessageInput_03e0User(); // ship's name comes from the user
  // Reserve Object Instance Name is sent to RTI:
  SendReserveObjectInstanceNameROINBosporusFederation("s0");
  // "s0" is to be overridden by the computation aspect which will take ship name from user
  // Object Instance Name Reserved (OINR) is received from RTI
  condRecvObjectInstanceNameReservedOINRBosporusFederation();
  (…) //If OINR succeeds leave the loop
  } while (!((Boolean)Ship_MSC.coldChoices.get("until_0300")).booleanValue());
  (…)// Other Inputs: direction and speed come from the user.
  SendRegisterObjectInstanceRegisteredShipObjectBosporusFederation(...);
  // Register Object Instance is sent to RTI for the Ship object
  SendUpdateAttributeValuesRegisteredShipObjectBosporusFederation(…);
  // Update Attribute Values is sent to RTI for the Ship object
  SendRequestAttributeValueUpdateDiscoveredShipObjectBosporusFederation(…);
  // Request Attribute values Update is sent to RTI for the Ship object
  doLaterMessageTimer_03c6(100); //timer is started for periodically send interactions
  // While-Do Main Simulation Loop begins
  while (((Boolean)Ship_MSC.coldChoices.get("ExitCondition_040f")).booleanValue()) {
   //loop is repeated until the federate is resigned.
```

Figure H-7. Excerpts from the Generated Java Code of Ship Federate

```
   (…) // The code generated for SendRadioMessage chart is inserted here.
   // when a timeout occurs radio message interactions are sent and timer is restarted
   // Time Management  methods begin
    SendTimeAdvanceRequestTARBosporusFederation(new Double(55.0));
   // Timestamp type Double comes from FAM.  Timestamp value (55) should be overridden.
  condRecvTimeAdvanceGrantTAGBosporusFederation();
   // Time Advance Grant is received from RTI.
 }//end of main simulation loop.
 (…) // The code generated for Exit Federation chart goes in here.
 //federate is resigned and federation is destroyed.
}//end of the main thread
```

Figure H.7. Excerpts from the Generated Java Code of Ship Federate

(Continue)

To `give` a sense of the generated code, a part of the ship federate's (see Figure H-7) and a sample RTI Ambassador Method (`sendinteraction` in Figure H-8a) and a federate Ambassador method (`receiveinteraction` in Figure H-8 b) are shown in the figures. The first operand (main thread) of the parallel inline expression (see Figure 3.2) of the generated `shipFd` code is exemplified in Figure H-7. For every operand in a parallel inline expression occurring in the LSC, a thread (e.g. `MainThread_02ee` and `CallbackThread_032c`) is generated. For loop idioms, `while-do` or `repeat-until` code statements are generated. Values of loop conditions are retrieved from the dictionary (implemented as *hashtable* named `coldChoices`) defined in the computation aspect. In place of the chart references in the LSC model, the referenced charts' codes are generated. For example, for `CreateRegions` reference, `CreateRegion` and `SetRangeBounds` methods are generated. In Figure H-8a, interaction parameters are packed into an object of `LSCObject`. Then the corresponding `LscRTILib` method (in this case, `sendInteraction`) is called. In Figure H-8b, a federate Ambassador method (in this case, `receiveinteraction`) example in the federate base code is shown.

```
public static boolean
SendSendInteractionWithRegions_0536RadioMessageBosporusFederation(…)
//parameters
{
      LSCLib.LSCObject proc= new LSCLib.LSCObject();
      //interaction class information comes from HOMM.
      proc.name="RadioMessage"; //interaction class name
      proc.pars=new ArrayList(); //parameter list of the interaction class
      LSCLib.LSCAttribute parNew0 =new LSCLib.LSCAttribute();
      //parameter1 is declared
      parNew0.name="CallSign"; //parameter1's name
      parNew0.type="Object"; //parameter1's type in Java
      parNew0.objClass="HLAASCIIstring"; //parameter1's type in HLA datatype
      parNew0.objVal=CallSign; //parameter1's value
      proc.pars.add(parNew0); //parameter1 is added to the parameter list
      (…)//parameter2 is added.
            //dimension and region data is added to the parameter list
            //time stamp data is added to the parameter list
      BosporusFederationRTILib.sendInteractionWithRegion(proc);
      //same named LscRTILib method is called
}
```

Figure H.8 a. A Sample SendInteraction RTI Ambassador Method in Federate Base Code (ShipFd)

```
public static void RecvReceiveInteractionRadioMessageBosporusFederation
(LSCLib.LSCObject iClass,String TimeStamp,int SentOrderType,int ReceiveOrderType,String
TransportationType)
{}//received interaction parameter values are held in iClass.
```

Figure H.8 H. 8 b. A Sample ReceiveInteraction Federate Ambassador Call-back Method in Federate Base Code (ShipFd)

b) Codes for Aspects for our Example

Two computation aspects and a federation execution aspect are generated, namely `ShipFdAspect`, `UserAspect`, and `BosporusFederationLibAspect`. `ShipFdAspect` overrides all RTI methods in the `ShipFd` federate base code. In `ShipFdAspect`, dictionaries and LSC-specific auxiliary methods' (i.e. `chooseOne`, `getLoopount)` advices are also generated.

Two sample advices, namely, RTI Ambassador Method's (send interaction) advice and a federate Ambassador method's (receive interaction) advice, are shown in Figure H-9a and Figure H-9b, respectively. In Figure H-9a, federate send interaction method (cf. Figure H-8a) is caught in the `ShipFd` base code and preliminary logic (in italic) is filled in. The developer can edit this advice as described in the subsequent "Editing the Computation Aspect" section.

In Figure H-9b, federate receive interaction method (cf. Figure H-8b) is found on the `ShipFd` base code and received data is placed in its advice in the `ShipFdAspect`. This received data is the values of all parameters of the interaction class. In this example, the interaction class is `RadioMessage` with parameters `callsign` and `message`.

```
pointcut pcSendSendInteractionWithRegions_0536RadioMessageBosporusFederation()
//pointcut definition
{
    CallSign=new Boolean(true);
    //call sign is given  as preliminary computation in the computation aspect
    Message="Radio Message Sample"; //message is given as preliminary computation
    (…) //declaration detail of dimension  is get outed
    parChannelDimension2_0.strVal="ChannelDimension"; //dimension comes from FAM
    (…) //declaration details of region
    parChannel13_0.strVal="Channel1"; // region comes from FAM
    (…) //other  details of dimension and region
    TimeStamp=new Double(2.0);//must be overridden
    proceed(CallSign,Message,RadioMessagewithRgnsDims,TimeStamp);
    return true;
}
```

Figure H.9 a. A sample RTI Ambassador Method (advice) in Computation Aspect (ShipFdAspect)

```
pointcut pcRecvReceiveInteractionRadioMessageBosporusFederation (…)
//pointcut definition
{
    Object CallSign= (Object)((LSCLib.LSCAttribute)iClass.pars.get(0)).objVal;
    System.out.println("Received CallSign Parameter:"+CallSign);
    //callsign interaction class parameter is printed.
    Object Message= (Object)((LSCLib.LSCAttribute)iClass.pars.get(1)).objVal;
    System.out.println("Received Message Parameter:"+Message);
    //Message interaction class parameter is printed.
    System.out.println("Received TimeStamp:"+TimeStamp);
    System.out.println("Received SentOrderType:"+SentOrderType);
    System.out.println("Received ReceiveOrderType:"+ReceiveOrderType);
    System.out.println("Received TransportationType:"+TransportationType);
    proceed(iClass,TimeStamp,SentOrderType,ReceiveOrderType,TransportationType);
}
```

Figure H. 9 b. A sample Federate Ambassador Method (advice) in Computation Aspect (ShipFdAspect)

`BosporusFederationLibAspect` (federation execution aspect) is mainly used to catch call-back methods from the Bosporus federation execution. A `BosporusFederationRTILib` object is instantiated from `LscRtiLib` in this aspect and it is used to reach actual RTI. A sample `LscLibRTI` definition (`BosporusFederationRTILib`) and a sample (`ReceiveInteraction`) advice are presented in Figure H-10.

In Figure H-10, `ReceiveInteraction` call-back method is caught by the federation execution aspect (`BosporusFederationLibAspect`) and forwarded to the federate (`ShipFd.ReceiveInteraction`).

```
public static RTILib ShipFd.BosporusFederationRTILib= new RTILib(); // LscRtiLib declaration for the federate
(…)//unrelated code
pointcut ReceiveInteraction(…)//pointcut definition
{
    RTILib rtiLib = (RTILib)thisJoinPoint.getThis();
    // compare received callback with federation name as there might be other federations
    if (rtiLib.federatename.compareTo("BosporusFederation")==0)
      ShipFd.ReceiveInteraction(proc); //federate method in the base code is called
}
```

Figure H.10. A LscRTILib Definition and a Sample Advice in Federation Execution Aspect (BosporusFederationLibAspect)

5. How to Edit and Navigate on the Generated Code: Especially the Computation Aspect

- After running the generator, user can edit advices of `ShipFdAspect` and `UserAspect` (generated preliminary computation) in order to effect the desired computation. Pointcut definitions must be same otherwise advice codes are not weaved on the base code. Consider, for example, how ship name is retrieved from the user to send a `reserveobjectinstance` event to the federation. In the automatically generated preliminary computation, a `sample string` is sent to the `ShipFd` as a ship name by `UserAspect`. Naturally we would like the name to be entered by the user. User types in a name in the advice. The corresponding edited code is illustrated in italic font in Figure H-11.
- Generator marks the mandatory editing points in the computation aspect by giving comments such as `must be edited`. Specially, randomization logic and randomly generated variable values must be edited.
- When user saves the edited aspect, (By pressing the save button in the Eclipse toolbar.), Eclipse automatically compiles and builds the project. If an error is occurred, Eclipse presents it by red markers on the code.

```
pointcut pcSendMessageInput_03e0ShipFd(…)//pointcut definition
{
   System.out.print("Name:> ");
  try {
      g_Name = in.readLine(); //name is read from console
  } catch (Exception ignored)   { }
  Name=g_Name;
  proceed(Name);
  return true;
}
```

Figure H.11. Adding a Computation to User Ship Name Selection Method in User's Computation Aspect

(Modifications to the generated preliminary advice are in italic)

6. How to Run the Generated Code

- Having compiled the ship federate application, the ship federate is ready to be run. Select the generator project and activates the popup menu by clicking right mouse button.
- In this menu, select `Run As->AspectJ Java Application` menu item.
- `Select Java Application` window appears. In this window (Figure H-12), select the diagram name class which contains main function. In our example `ShipFd-Ship_MSC` is selected. And finally ship federate code is run.

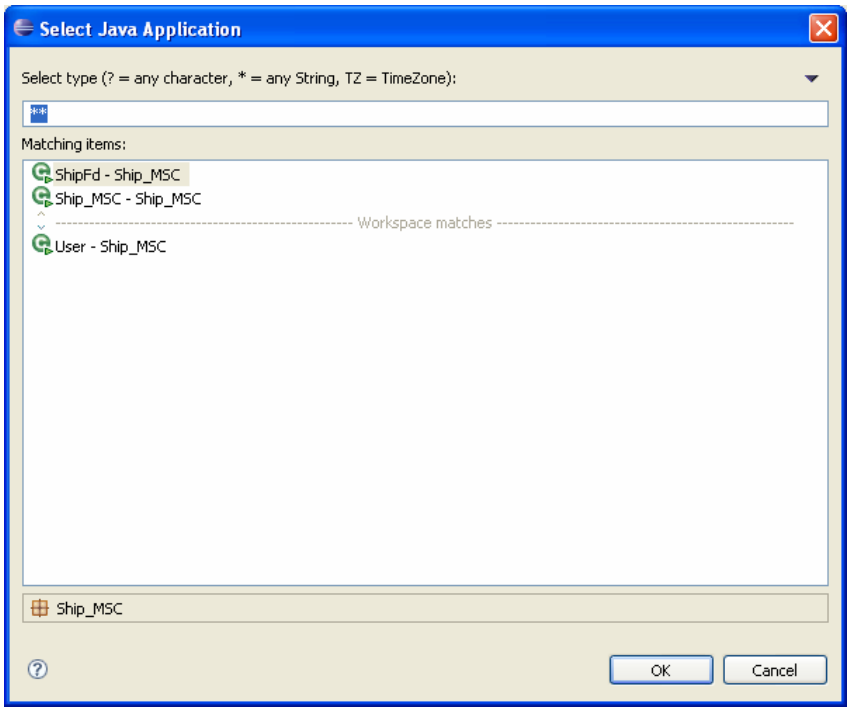


Figure H.12. Select Java Application Window

- Then the ship federate runs and joins the Bosporus federation with the station federate joined as well. Preparation of the station federate follows the same steps. A view from the running federation is presented in Figure H-13.

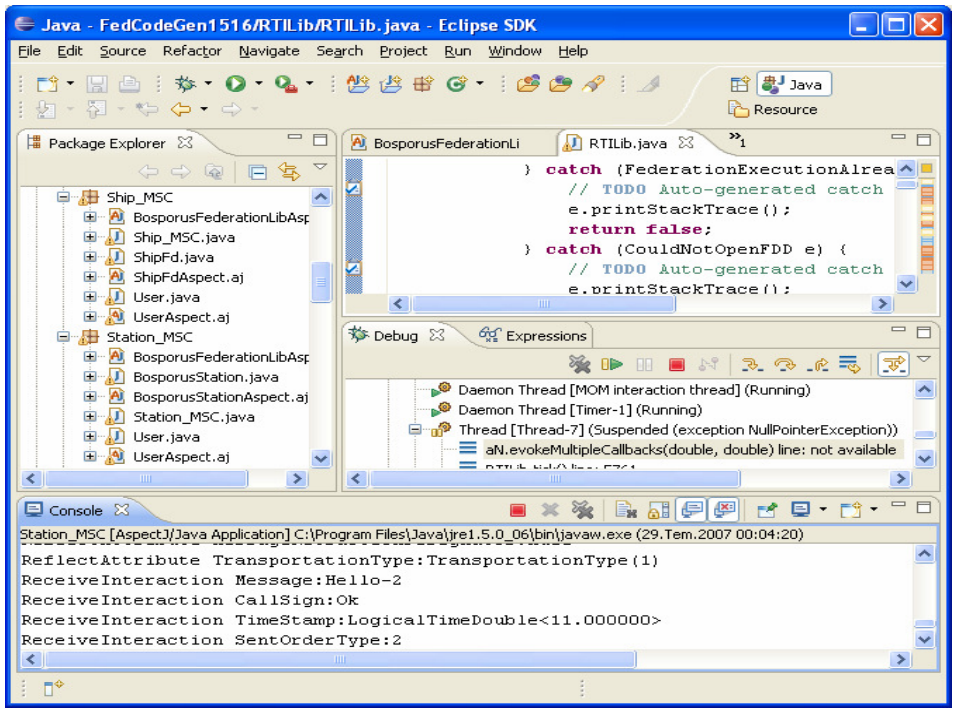


Figure H.13. A View of the Ship Federate Running (pRTI snapshot)

7. What are the Critical Modeling Points for Code Generation

- Model, atom, and reference names are used for variables in the code generation so names must be variable identification form. For example, they do not contain blank, slash, minus sign, bracket, etc and do not start with numbers.
- Model must be complete and correct for syntactically and semantically. For example events' precedence values must be ordered and correct. References in the model must be referred to the correct model elements.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Adak Bülent Mehmet
Nationality: Turkish (TC)
Date and Place of Birth: 05 December 1974, Ereğli
Marital Status: Married, one daughter.
Phone: +90 312 315 55 87
Email: bmadak@yahoo.com

EDUCATION

Degree	Institution	Year of Graduation
MS	METU Computer Engineering	2000
BS	METU Computer Engineering	1997
High School	Ereğli Cumhuriyet High School	1992

WORK EXPERIENCE

Year	Place	Enrollment
1997-Present	Aselsan Inc. Software Engineering Department	Software Engineer

FOREIGN LANGUAGES

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 **Adak Mehmet**, Topçu Okan, and Oğuztüzün Halit, "Model-based Code Generation for HLA Federates", ACM Transactions on Modeling and Computer Simulation (under review).

3 Topçu Okan, **Adak Mehmet**, and Oğuztüzün Halit, "Metamodeling Live Sequence Charts for Code Generation", Software and Systems Modeling (under review).

4 **Adak Mehmet,** Topçu Okan, and Oğuztüzün Halit, "Code Generation for Live Sequence Charts and Message Sequence Charts", Journal of Systems and Software (under review).

International Conferences

1. Molla Ayhan, Sarıoğlu Kaan, Topçu Okan, **Adak Mehmet**, and Oğuztüzün Halit, "Federation Architecture Modeling: A Case Study with NSTMSS", In Proceedings of 2007 Fall Simulation Interoperability Workshop (SIW), Orlando, Florida, USA, September 16-21, 2007.

2. **Adak Mehmet** and Oğuztüzün Halit, "A Web-based Source Code Browser for Pascal". 16th International Symposium on Computer and Information Sciences (ISCIS XVI), Antalya, pp.87-96, 2001.

National Conferences

1. Efe Osman, **Adak Mehmet** and Oğuztüzün Halit, "Davranış Belirtimlerinin Kod Üretimi Yoluyla Canlandırılması ve Bir Uygulama", UYMS, Ankara, 2007

Technical Reports

1. **Adak Mehmet** and Oğuztüzün Halit, "A Code Generator for Live Sequence Charts (LSC) and Message Sequence Charts (MSC)", Technical Report (METU-CENG-07-04), Middle East Technical University, May 2007

2. Sarıoğlu Kaan, **Adak Mehmet** and Oğuztüzün Halit, "Modeling and Code Generation for Federation Federate Monitor (FedMonFd)", Technical Report (METU-CENG-07-07), Middle East Technical University, May 2007

Thesis

1. **Adak Mehmet**, "A Web-based Source Code Browser for Pascal", MSc Thesis, The Department of Computer Engineering, The Graduate School of Natural and Applied Sciences, Middle East Technical University (METU), Ankara, Turkey, 2000.

HOBBIES

Painting