IMPLEMENTING KQML AGENT COMMUNICATION LANGUAGE FOR MULTIAGENT SIMULATION ARCHITECTURES ON HLA

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

BY

EREK GÖKTÜRK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

AUGUST 2003

ABSTRACT

IMPLEMENTING KQML AGENT COMMUNICATION LANGUAGE FOR MULTIAGENT SIMULATION ARCHITECTURES ON HLA

Göktürk, Erek
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Faruk Polat

August 2003, 70 pages

Multiagent simulation is gaining popularity due to its intuitiveness and ability in coping with domain complexity. HLA, being a distributed simulation architecture standard, is a good candidate for implementing a multiagent simulation infrastructure on, provided that agent communication can be implemented. HLA, being a standard designed towards a wide coverage of simulation system architectures and styles, is not an easy system to master. In this thesis, an abstraction layer called the Federate Abstraction Layer (FAL) is described for better engineering of software systems participating in an HLA simulation, providing lower project risks for the project manager and ease of use for the C++ programmers. The FAL is in use in project SAVMOS in Modelling and Simulation Laboratory. Discussion of FAL is followed by discussion of the study for realizing KQML for use in multiagent architectures to be built on top of HLA as the data transfer medium. The results are demonstrated with 10 federates implemented using the FAL.

Keywords: High Level Architecture, Simulation, Multiagent architecture, Agent Communication Language, KQML

ÖZ

HLA ÜZERINDE ÇALISAN ÇOK ETMENLI SIMÜLASYON MIMARILERI İÇIN KQML ETMEN HABERLEŞME DILI'NIN GERÇEKLEŞTIRIMI

Göktürk, Erek
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Faruk Polat

Ağustos 2003, 70 sayfa

İnsanın problem çözme şekline yakınlığı ve karmaşık alanların analizi ile başa çıkma konusundaki başarısı nedeniyle çok etmenli simülasyon kavramı popülerlik kazanmaktadır. Bir dağıtık simülasyon standardı olan HLA, çok etmenli simülasyon mimarilerini gerçekleştirmek için uygun bir adaydır. Ancak bu uygulamaların yapılabilmesi için etmenlerarası iletişimin gerçekleştirilmesi gerekir. çok geniş bir yelpazedeki mimari ve stillerini desteklemek üzere geliştirildiğinden HLA kolay kullanılır bir sistem değildir. Bu tezde, bir HLA simülasyonunun parçası olarak işlev görecek yazılım sistemlerinin mühendislik çalışmalarının daha iyi yapılabilmesi için bir soyutlama katmanı olan Federe Soyutlama Katmanı (FESK) anlatılmıştır. Bu katmanın proje yöneticisi bakış açısından çeşitli riskleri azaltması, C++ programcılarına da kullanım kolaylığı sağlaması öngörülmektedir. FESK, Modelleme ve Simülasyon Laboratuarı'nda sürmekte olan SAVMOS projesinde kullanılmaktadır. FESK'in tanımını KQML dilinin HLA'yı veri transfer ortamı olarak kullanan çok etmenli simülasyon sistemleri için gerçekleştirilmesi konusundaki çalışmaların anlatımı izlemektedir.

Anahtar Kelimeler: Yüksek Düzey Mimarisi, Simülasyon, Çoktu Etmen Mimarisi,

Etmen Haberleşme Dili, KQML

Dedicated to my father,

who waited so long for happiness to find him,

whom we tought to fall into depression,

whom I'm losing now from brain cancer,

and whom I miss so much...

Also dedicated to my mother,

who loved me with endless care,

who's in endless pain, but endures without showing any sign,

whom I desparately fear to lose,

whom I love so much...

ACKNOWLEDGMENTS

I would like to thank my beloved mother and Beyza Demir for their incredible in family support. I also would like to thank Mr. and Mrs. Demir for being almost a second family to me, and I was so glad to feel their support.

My thanks also goes to Dr. Faruk Polat, from whom I learnt so much as my senior project and thesis supervisor. Also I would like to thank the SAVMOS project group, who compansated my place in the team as I wrote this thesis, Murat Balcı, Erkin Çilden, Dr. Ahmet Coşar, Sertan Girgin, Volkan Koç, Evren Kapusuz, and Mehmet Tan. I would also like to thank Dr. Cem Bozşahin, Dr. Ali Doğru, Dr. Halit Oğuztüzün, and Dr. Göktürk Üçoluk for I used very frequently, in the course of this thesis, what I learned from them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis describes an abstraction layer called the Federate Abstraction Layer (FAL) for implementing simulations using C++ and the High Level Architecture (HLA) standard, and uses the implementation of this library in a study on identifying methods that may be used in enabling the use of the agent communication language KQML in multiagent simulation architectures.

1.1 Simulation and High Level Architecture

A simulation might be defined as a controlled computation of a future state of a system, depending on its model based on some assumptions about its components and their dynamics. This definition means seeing into the future, a dream of mankind. How far we fall from this dream is a question of the quality in terms of expressiveness and predictive power of our models, physical or mathematical.

A simulation is not necessarily something entirely logical. To give an example, a model made by an architect might as well be a simulation of the building to be made out of the prepared blueprints. This thesis is about software simulators, which are programs that model a model of some desired reality. The requirements of such a software simulator is a logical model, mostly prepared by using modeling paradigms such as the object-oriented or agent-oriented paradigms, and the software itself is a simulation of that model. Although the philosophical relationship of a simulation to the reality it represents is a complex one, attracting many science fiction fans to theaters, the aim is simple: A simulation is built in order to answer some predefined set of questions in the future of a given reality.

High Level Architecture (HLA) is a distributed simulation standard (IEEE 1516) that aims at interoperable and component-based reusable simulations. It was born out of a need of cutting down development costs for the USA Department of Defense funded simulation systems, and it's among the aims of the standard to be able to be used in many different kinds of simulations, like be it continuous or discrete time.

Description of the structure of HLA simulation architecture should follow on two parallel tracks: The data plane and the control plane views. On the control plane, HLA divides the computational entities cooperating in a simulation into architectural units called federates, and the whole simulation is named as the federation. On the data plane view, the HLA provides the federates necessary services for sharing data in two forms called objects and interactions. More detailed information is given in Chapter 3.

Because of the fact that HLA is a simulation standard prepared with an eye towards a wide coverage of simulation styles, it provides so many services that expertise is needed for its efficient application onto a simulation model. The Federate Abstraction Layer (FAL) developed, which is described in Chapter 4, builds upon the interface specification of HLA to provide ease of use, by employing C++'s object orientation and aims leveraging the knowledge needed to use HLA to the knowledge level of a good C++ programmer.

Another problem with the HLA's simulation architecture is that objects of HLA are quite alike the objects in the object-oriented paradigm, yet they're so different, and these similarities and differences are mentioned in greater detail in Chapter 4. The object-oriented paradigm is so popular today that it is not very likely that any designer or programmer that faces with the task of designing or producing a simulation using the HLA would not think, by reflex, about the objects in question in an object-oriented paradigm sense. The FAL is designed to act as a layer converting between the HLA data and C++ data structures, and to provide ease of use in accessing the services HLA provides.

Use of such an abstraction layer also decreases the third party software dependence risk as observed from the project manager's perspective of a simulation development effort using HLA. HLA federates, which are the target products for a HLA simulation project, should use the Run-Time Infrastructure (RTI) implementation, which is the part of the HLA simulation architecture providing the services through various libraries and application programming interfaces (APIs). There're more than one implementations of RTI, and it is a wise step to confine the RTI dependent code into a small module in case any change of the RTI implementation is required. The FAL also accomplishes such a function.

So far, the benefits of using FAL is described, yet these benefits are not gained at no loss. The FAL is designed to provide only a subset of the full range of services HLA provides, but in a more accessible way. Nevertheless, the previous version of FAL is being used in a project of Modeling and Simulation Laboratory co-established by the Middle East Technical University and Turkish Armed Forces, named SAVMOS, and adoption of the new revision is underway. From observations of the development process of SAVMOS, it might be said that FAL works in the sense that it has successfully abstracted away the HLA related concepts from attention of the rest of the members of the development team, although they work with HLA related concepts every day.

1.2 KQML over HLA

A multiagent system is composed of interacting entities called agents. This interaction between the agents is provided by communication. The agent communication in the multiagent systems may be analyzed in three layers: the content layer, communication layer and the transport layer (see Section 2.1). An agent communication language (ACL) is a language for the agents to encapsulate the content to be communicated. It also maintains a common ground for pragmatics of the communication, thereby affecting the design of overall communication infrastructure provided by the environment the agents are to work in. In this respect, an ACL, such as the KQML, is an integral part of a multiagent architecture.

A multiagent architecture is defined as one in which decisions and reasoning are distributed among the agents. This distribution of processes creates an intuitive style of modeling, when considered from a simulation perspective as well. Modeling using multiagency helps tackling with the domain complexity by allowing one to separately study the structure of components that make up the system and the interactions

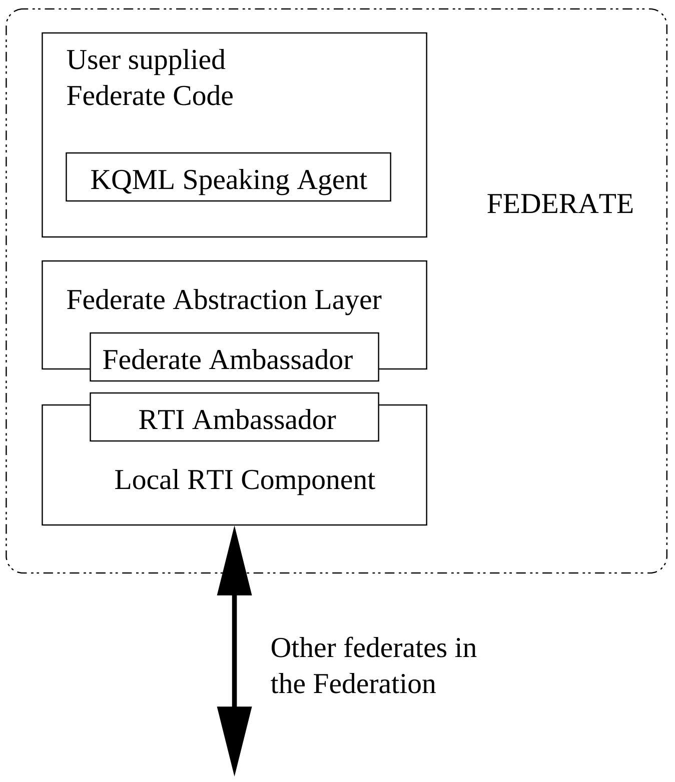


Figure 1.1: Use of FAL to demonstrate the KQML realization.

between these components which create the domain's dynamics [18].

Both HLA and the concept of multiagency have benefits for building simulation systems, and the idea of a marriage between the two is attractive. HLA can be used as an infrastructure to build multiagent simulations on, to foster its interoperability concept thereby allowing a multiagent simulation itself to act as a component in a bigger simulation. Furthermore, multiagency fits nicely into the reusability concept of HLA, since coupling of agents to the rest of the system is extremely low and the interfaces are very well defined.

To use multiagency in HLA based simulations, an agent communication language must be incorporated into the multiagent simulation architecture to be built upon HLA. In this thesis, methods for using HLA as a transport intermediate layer in a multiagent simulation with KQML talking agents is discussed, and some problems stemming from the nature of HLA are identified and some solutions proposed. KQML has strong alternatives such as the Foundation for Intelligent Physical Agents' ACL (FIPA-ACL), but the choice is not arbitrary, and is due to the fact that KQML puts less restrictions on the policies and leaves more room for the designer's preferences, when compared to its alternatives.

The FAL is used for developing some test federates for demonstration of this study of realization of KQML on HLA. This implementation of test federates have also helped identifying new needs for the FAL, thus these two studies became somewhat complementary. The structure of a federate in a multiagent simulation, simulating a KQML speaking agent through the FAL is given in Figure 1.1.

The structure of the rest of the thesis is as follows: In the second chapter, a general introduction to the concept of agent communication languages is given, followed by a brief history of ACLs and description of the KQML agent communication language. The High Level Architecture is described in the third chapter. The discussion of the motivation and structure of the Federate Abstraction Layer is in the fourth chapter. The fifth chapter is on identification of problems of implementing KQML for multiagent simulation architectures on HLA, and description of proposed solution alternatives. The thesis is concluded in the sixth chapter.

CHAPTER 2

KQML AGENT COMMUNICATION LANGUAGE

This chapter describes the agent communication language KQML, its motivation, syntax and semantics. To put this description into its context, first the concepts of multiagent systems and agent communication languages are discussed, followed by a proposal of a three layered approach to modeling agent communication languages and a brief history of agent communication languages. The last section describes the KQML.

A detailed description of a multiagent system is given by Ferber in his book "Multi-Agent Systems" [2]:

> The term 'multiagent system' (or MAS) is applied to a system comprising the following elements:
>
> 1. An environment, E, that is, a space which generally has a volume.
> 2. A set of objects, O. These objects are situated, that is to say, it is possible at a given moment to associate any object with a position in E. These objects are passive, that is, they can be perceived, created, destroyed and modified by agents.
> 3. An assembly of agents, A, which are specific objects ($A \subseteq O$), representing the active entities of the system.
> 4. An assembly of relations, R, which link objects (and thus agents) to each other.

5. An assembly of operations, Op, making it possible for the agents of A to perceive, produce, consume, transform and manipulate objects from O.
6. Operators with the task of representing the application of these operations and the reaction of the world to this attempt at modification, which we shall call the laws of the universe.

By definition of the multiagent systems, agent communication plays a crucial role as the only way of coordination between the problem solving entities, the agents. The communication between the agents take place either explicitly, that is by agents issuing communication actions and sending messages to other agents, or implicitly by agents having the results of their actions perceived by other agents. Without explicit communication, coordination becomes the very hard problem of opponent modeling, which is not a problem to be tackled for most of the practical multiagent system implementations.

2.1 A Three-Layered Approach to Modeling Agent Communication

For decades, the state of the art in modeling communication systems has been the use of layers whose functions and interfaces are more or less well-defined. In the literature, we see that agent communication is also structured in layers [5] [3]. We take an approach similar to the one in the FIPA Abstract Architecture, and propose a three layered model.

The highest layer is the *content layer*. In the content layer, the information intended to be passed between the agents is expressed in a content language such as Knowledge Interchange Format (KIF) or Semantic Language (SL), using an ontology known by both participants of the communication.

Below the content layer is the *communication language* layer, in which the content received from the upper layer is encapsulated using an agent communication language (ACL) such as KQML or FIPA-ACL [6] [7]. Usually, these agent communication languages contain performatives, which are said to stem from the speech-act theory [24]. The ACL message also contains information such as the names of the sender and receiver agents as well as interaction protocol related fields, ontology and the content language used.

As the lowest layer, just above the actual information transfer such as a network connection, IPC or a middleware such as HLA, we observe the *transport layer* whose task is to handle the specific transfer related issues, such as converting the ACL message into a form transferable over the actual connection, converting agent names to transport addresses and encapsulating the ACL message with the sender and receiver transport addresses and the ACL type.

2.2 A Brief History of Agent Communication Languages

The KQML is a language that is designed to support interactions among intelligent software agents [4]. The need for communities of intelligent agents stems from the emergence of highly distributed, heterogeneous, and extremely dynamic systems comprising a large number of autonomous nodes. KQML was developed by the ARPA supported Knowledge Sharing Effort (KSE) [23], the External Interfaces Working Group formed in 1990. The KQML draft specification was published in 1993 [3], and although many experimental and commercial systems using KQML followed, it was never a big success story. In 1997, as a response to reports of semantic inconsistencies in the language as defined in the draft specification, Labrou and Finin proposed some changes to the specifications [14] [15] [16] [17].

The Foundation for Intelligent Physical Agents (FIPA) was formed in 1996 to produce software standards for heterogeneous and interacting agents and agent-based systems, and adopted the following official mission statement, that closely resembles the motivation behind development of the KQML language:

> The promotion of technologies and interoperability specifications that facilitate the end-to-end interworking of intelligent agent systems in modern commercial and industrial settings.

To support this, FIPA has adopted and is working on specifications that range from architectures to support agents' communicating with each other, communications languages and content languages for expressing those messages and interaction protocols which expand the scope from single messages to complete transactions.

For communication issues, FIPA produced many specifications including the FIPA ACL Message Structure Specification, FIPA Ontology Service Specification, a series of interaction protocols about pragmatics of agent communication, the FIPA Communicative Acts Library Specification which defines the performatives to be used with the FIPA ACL and their semantics, and specifications for supporting various content languages such as Knowledge Interchange Format of KSE and Semantic Language (SL). Some of these specifications were declared as standards in December 2002. FIPA publishes all mentioned documents through its web site [8].

KQML-Lite should also be mentioned here, as an attempt to merge FIPA and KQML specifications [12]. Although KQML-Lite is meant to be light, it adds more performatives from the FIPA communicative acts library to the performative set of KQML. KQML-Lite also calls for standardization of the services offered by the facilitator agents, thereby emphasizing the correlation between KQML implied architecture and FIPA Abstract Architecture. Furthermore, in an attempt to establish a better separation between the transport and communication layers, KQML-Lite defines a layer above the KQML expression and calls it the transmittal layer, whose output is again in the form of a KQML message but restricted to use only the networking and facilitation performatives, and performatives related to agent execution control.

When we look at the usage of the mentioned ACLs, the FIPA ACL seems to have the biggest user community, and used both in academia and industry.

2.3 KQML

KQML is a language for programs to use in order to communicate attitudes about information, such as querying, stating, believing, requiring, achieving, subscribing, and offering. The language is at the communication language level, described in Section 2.1.

A KQML message is called a *performative*, in that the message is intended to perform some action by virtue of being sent. The term "performative" comes from the speech act theory. The set of performatives provided in KQML specification is by no means claimed to be necessary nor sufficient, and multiagent system designers are free to add definitions of their own performatives provided that they are defined precisely.

Table 2.1: KQML reserved parameters and their meanings.

Keyword	*Meaning*
:content	the information about which the performative expresses an attitude
:force	whether the sender will ever deny the meaning of the performative
:in-reply-to	the expected label in a reply
:language	the name of representation language of the :content parameter
:ontology	the name of the ontology (e.g., set of term definitions) used in the :content parameter
:receiver	the actual receiver of the performative
:reply-with	whether the sender expects a reply, and if so, a label for the reply
:sender	the actual sender of the performative

2.4 Syntax

A performative is expressed as an ASCII string using the syntax given below in BNF notation.

```
<performative>     ::= (<word> {<whitespace> :<word> <whitespace> <expression>}*)
<expression>       ::= <word> | <quotation> | <string> |
                       (<word> {<whitespace> <expression>}*)
<word>             ::= <character><character>*
<character>        ::= <alphabetic> | <numeric> | <special>
<special>          ::= < | > | = | + | -- | * | / | & | ^ | ~ | _ |
                       @ | $ | % | :  | .  | !  | ?
<quotation>        ::= '<expression> | '<comma-expression>
<comma-expression> ::= <word> | <quotation> | <string> | ,<comma-expression>
                       (<word> {<whitespace> <comma-expression>}*)
<string>           ::= "<stringchar>*" | #<digit><digit>*"<ascii>*
<stringchar>       ::= \ <ascii> | <ascii> \\\ <double-quote>
```

A KQML performative is a LISP like s-expression that starts with the performative name and is followed by parameter name-value pairs. The parameter names always start with a ":".

2.5 Semantics

Each agent in the KQML semantic model appears as having and managing a knowledge base. This does not mean that the agent actually has some kind of a knowledge base, so in the specifications this knowledge base is referred as a *virtual* knowledge base (VKB). Although agents communicate about the contents of their or other agents' VKBs using

Table 2.2: KQML predefined performatives and their meanings.

Name	*Meaning*
achieve	S wants R to do make something true of their environment
advertise	S is particularly-suited to processing a performative
ask-about	S wants all relevant sentences in R's VKB
ask-all	S wants all of R's answers to a question
ask-if	S wants to know if the sentence is in R's VKB
ask-one	S wants one of R's answers to a question
break	S wants R to break an established pipe
broadcast	S wants R to send a performative over all connections
broker-all	S wants R to collect all responses to a performative
broker-one	S wants R to get help in responding to a performative
deny	the embedded performative does not apply to S (anymore)
delete	S wants R to remove a ground sentence from its VKB
delete-all	S wants R to remove all matching sentences from its VKB
delete-one	S wants R to remove om matching sentence from its VKB
discard	S will not want R's remaining responses to a previous performative
eos	end of a stream of responses to an earlier query
error	S considers R's earlier message to be mal-formed
evaluate	S wants R to simplify the sentence
forward	S wants R to route a performative
generator	same as a standby of a stream-all
insert	S asks R to add content to its VKB
monitor	S wants updates to R's response to a stream-all
next	S wants R's next response to a previously mentioned performative
pipe	S wants R to route all further performatives to another agent
ready	S is ready to respond to R's previously-mentioned performative
recommend-all	S wants all names of agents who can respond to a performative
recommend-one	S wants the name of an agent who can respond to a performative
recruit-all	S wants R to get all suitable agents to respond to a performative
recruit-one	S wants R to get another agent to respond to a performative
register	S can deliver performatives to some named agent
reply	communicates an expected reply
rest	S wants R's remaining responses to a previously-mentioned performative
sorry	S cannot provide a more informative reply
standby	S wants R to be ready to respond to a performative
stream-about	multiple-response version of ask-about
stream-all	multiple-response version of ask-all
subscribe	S wants updates to R's response to a performative
tell	the sentence in S's VKB
transport-address	S associates symbolic name with transport address
unregister	a deny of a register
untell	the sentence is not in S's VKB

KQML, this does not restrict their preferences about representation languages that might be used in encoding statements in their VKBs.

From the KQML point of view, the statements in the VKB is classified into beliefs and goals. The set of beliefs contain statements about the agent itself and its external environment, including its knowledge about the VKBs of other agents. A performative of KQML might refer to either or both of agent's goals and beliefs, e.g. an agent may want another agent to achieve something or it might want the other to send a certain kind of information.

KQML reserves some parameter names and defines a predefined set of performatives. The KQML reserved parameters and a short description of their meanings is given in Table 2.1. The list of predefined performatives is given in Table 2.2. For more detailed information, the reader is referred to [3] and [14].

CHAPTER 3

High Level Architecture

This chapter presents, as a background material, the High Level Architecture (HLA). It starts with a definition of what HLA is, and what it is not, then proceeds with a small history of the standard. In the next section, HLA's view of the structure of a simulation is presented. The last section tells about the details of the standard, the HLA rules, the Object Model Template (OMT) and the Run-Time Infrastructure (RTI).

So what is HLA? First of all, HLA, the High Level Architecture, is a software architecture. So we must first define what is an architecture to conceive its limitations. Shaw and Garlan defines software architecture as follows [25]:

> Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns.

When analyzed as a software architecture, the following elements and their categorization is identified [13]:

- *Elements.* The elements of HLA are the federation, federate, RTI, and the common object model.
- *Interactions.* The interactions between the federates and the RTI, and between federates is defined. The federation object model (FOM) defines the kinds of data carried between the federates, and the OMT serves as a meta-model that defines the structure of every valid FOM.

- *Patterns.* The HLA rules defined in Section 3.3.1 sets the allowed patterns of composition for building HLA federations.

A more inclined toward the functionality definition for HLA might be given as a standard framework that supports simulations composed of different simulation components [22].

The motivation behind formulation of the HLA standard is providing the desirable properties of reusability and interoperability. In the context of HLA, reusability means the ability of breaking away federations, whose run constitute a simulation, and using the resultant components, the federates, in building new federations, and do all these decomposition-recomposition with minimal effort. On the other hand, interoperability implies an ability to combine component simulations on distributed computing platforms of different types, often with real-time operation.

3.1 History

In the early 1990s, there were so many military simulations in USA that no one had an accurate count of them. Each organization that saw a need for a simulation created their own, with little reuse of what had existed. The cold war was clearly over by then, and the political situation was a lack of interest in increased defense expenditures. The solution was seen in information technology in general, and simulation in particular. This provided the management and economic push towards the development of HLA.

The Director of Defense Research and Engineering assigned the Defense Modeling and Simulation Office (DMSO) the objective of assuring interoperability and reuse of military simulations. This resulted in a substantial change in the DMSO's mission, and DMSO integrated the need for development of a common simulation framework into the strategic plan [19].

First of all, an underlying approach was formulated. This started with a call to industry and in the summer of 1995, three technical contractors were commissioned for six month investigations of the technical options and approaches. Another team, called the program team, worked closely with these three teams to explore the options for meeting the development objectives. The latter team was composed of government, laboratory and university people and the structure of this team meant to a gathering of all people related to the prepared standard. The result was an initial technical architecture in March 1996.

The baseline development of the standard continued with formation of an Architecture Management Group (AMG), which consisted of technical teams from sixteen major defense programs that covered the range of uses for defense simulation. The AMG developed four prototype federations, each targeting a different foreseen use of simulations that will be developed using HLA in the future. The baseline HLA definition, HLA 1.0, which was completed in August 1996, was approved as the standard technical architecture for all DoD simulations on September 10^{th}, 1996 [21].

RTI software was a very critical part of the development process. The first RTI prototypes were developed by a team consisting of FFRDCs (The MITRE Corporation and MIT Lincoln Laboratory) and industry (SAIC and Virtual Technology Corporation). The initial RTI software, known as the 0.X series was developed with an OMG Interface Definition Language (IDL) application programmer's interface (API) and CORBA. With the acceptance of HLA baseline, the development of the 1.0 version started. A familiarization release was published in December 1996, known as the F.0 release, and it was followed by the 1.0 release in May 1997. This release did not include implementation of the DDM service group, and the concept was tested in another implementation called the RTI-S. The RTI-S was used with an advanced concept technology demonstration called the Synthetic Theater of War (STOW), and guided further developments in the RTI to the release RTI 1.3 (there was no , the first full-service RTI implementation. The RTI 1.3 implemented the HLA specification released in March 1998.

RTI 1.3NG is a full implementation of RTI services based on competitive industry designs and development. The Phase I RTI 1.3NG software design contract began immediately following the HLA Baseline Definition in August 1996, and culminated in the award of a Phase II RTI 1.3NG software development contract to SAIC in September 1997. PEO STRI (Program Executive Office for Simulation, Training and Instrumentation, formerly known as STRICOM) has been the procurement agent for the design and development effort, and a technical advisory team, which includes representatives from various DoD user organizations, is supporting this activity. RTI 1.3NG also supports HLA Specification 1.3. The RTI 1.3NG was the first RTI implementation distributed freely to the public until recently, and it is commercialized in September 2002. Today, RTI implementations can be provided from private industry vendors. The vendors that has commercial RTI implementations as of this thesis' preparation are MaK Technologies, MA, USA; Mitsubishi Space Software Company Ltd., Japan; Pitch AB, Sweeden. The DMSO RTI 1.3NG is handed over to Science Application International Corporation (SAIC) and Virtual Technology Corporation (VTC).

The HLA standard today is an IEEE open standard (IEEE 1516, 2000). The standard is adopted by Object Management Group (1998) and NATO (1998, [20]), and is expected to penetrate into the commercial simulation software industry as much as the military industry in the following years.

3.2 The Concept of Simulation From HLA's Perspective

The HLA conceives the concept of simulation as communicating entities cooperating in computational work in order to satisfy the requirements of a simulation.

This view of simulation calls for composable simulation models and run-time establishment of cooperation among the architectural entities, and reinforces the HLA's reusability. A suitable simulation model to be implemented using the HLA is one in which the coupling between the computational entities in the model is confined to the communication schemes provided by the HLA. The standard seems to suit best to object oriented models of the reference reality for the simulation to be built.

The HLA perspective on simulation, as analyzed from the data plane and control plane points of view, is the subject of this section.

3.2.1 Control Plane

Any software, or more generally any simulation model, can be analyzed from the data and control division perspectives. The control plane view shows how the computation is distributed among the entities in the model.

The HLA divides the computational work onto software called *federates*. The simulation, resulting from the cooperative execution of the federates is called a federation. Each run of a federation is called a federation execution and represents the execution of the simulation model. A federation might also take part in a bigger simulation by acting as a federate in another federation execution, by means of a bridging federate that transfers data from the bigger federation to the federation acting as a federate.

The federates are encapsulated by RTI, that is the sole source of communication between the federates is the RTI. By this design, it becomes possible to move a properly designed federate from one federation to another, with no need for changing communication mechanisms and communication based protocols such as time management.

Time management is an important distributed mechanism that is controlled by the HLA. HLA gives freedom to federates in choosing their way of managing time, and provides ways of establishing control over sharing the time among federates or synchronization of local times of federates creating an illusion of a federation time.

3.2.2 Data Plane

The data plane analysis of HLA's simulation perspective includes how the data in the model is represented, and how it flows between the entities in the model.

For the HLA, the only data relevant is the data that is to be shared between the federates taking part in a federation. There may be more data present in the run of a simulation, but unless any federate shares it with the rest of the simulation, that data remains as the "private data" of the federate and HLA executes no control on that. When data is to be shared between federates, or will be open to use in the federation, that data has to conform to the HLA. The structure of this shared data is defined using a meta-model called the Object Model Template (OMT) and is kept in a file used by the RTI implementation which is called the Federation Object Model (FOM). In this object model, data is represented in two distinct forms: objects, and interactions.

Using interactions is just like sending a broadcast (or multicast if Data Distribution Management (DDM) is used) message to all federates in a simulation. The interactions have persistent structure that is defined in the FOM, yet their presence in the federation is transient in that they do not persist as entities after being delivered to all the interested federates. An interaction may be defined to occur at a point in simulated time. A federate is said to send an interaction; the other federates receive it. Each interaction carries a set of named data, whose names are static and defined in FOM, and each of these name-value pairs is called a parameter.

An HLA object has a persistent presence, as contrary to the interactions, and come into existence with a federate creating an object and is deleted when a federate (not necessarily the one created it) communicates a deletion request to the RTI. An HLA object is an instance of the class description given in the FOM for the federate. This class description includes the names of the attributes the object of the class may have, yet it's not mandatory that every attribute of an HLA object instance is updated by a federate in any point of time in federation execution. The HLA classes may have a hierarchical structure, which should again be defined in the FOM file of the federation.

The HLA objects are more or less resemble the objects of the object-oriented paradigm, yet fundamental differences exists. The similarities are that the HLA objects are instances of previously declared object classes, and that HLA object classes also are structured under a class hierarchy that allows inheritance of class attributes. The main difference is that HLA object classes do not include any description of behaviors associated with the object instances created from them in their declarations in FOM. This means that the HLA objects aren't exactly what is called an "Abstract Data Type". So HLA objects resemble more to the structures of C than to the objects of C++. The structures of C bind together relevant data and provide named components for reaching and updating the data, whereas the HLA objects bind relevant communication channels and provide named components to address them. Furthermore, HLA has a concept of a federate owning an attribute. A federate owning an attribute means that the federate is the only one in the given wall clock time instance that have the privilege of updating the attribute value, or put in other words the federate controls the entrance to the communication channel represented by the attribute. The ownership is transferable between the federates through the services of RTI. This ownership concept clearly has no direct counterpart in object-oriented paradigm.

Taken together, objects and interactions provide a logical division in the communication channel provided to a federate by the RTI, and the addressing semantics necessary for carrying out this communication. The addressing defined in HLA prevents federates from being aware of each other when this is not explicitly established by some protocol implemented by them over the communication channel provided. This is established through a publication-subscription mechanism, which is widely applied in information systems for decoupling the producers from data consumers. HLA relieves the producer of a certain kind of data of having to know (and manage) the set of recipients. This functionality, or lack of functionality for some purposes, provides seamless integration between federates and supports the HLA's goal of composable simulations.

The bad side of decoupling producers from consumers is that it decreases the data routing efficiency. Routing efficiency becomes a major concern as the simulations get distributed, especially over heterogeneous networks including slower links. The HLA standard presents the Data Distribution Management (DDM) as the solution. DDM is a pack of services provided by the RTI which enables the producers and consumers from narrowing their description of interests from HLA class level to include a federation-wide but user defined details level. The DDM services work over multi-dimensional routing spaces. The structure of these spaces, namely the names and number of dimensions is defined in the FOM for the federation. Each dimension is defined, in a generic way, to be a consecutive set of integers with RTI implementation defined maximum and minimum limits. Each producer or consumer defines the region of distribution for any data they produce or consume, be it an object attribute update or an interaction, and DDM provides routing of data only if the producer's and consumer's regions of interest for the data being distributed intersect.

3.3 Details of the Standard

The HLA standard is composed of three parts:

- HLA rules,
- Object Model Template (OMT),
- and the Run-Time Infrastructure (RTI).

Each of these parts is described in this section below.

3.3.1 HLA Rules

At the highest level, HLA consists of a set of ten rules which a federate or federation must obey in order to be regarded as HLA compliant. The 10 rules are divided into two: five of these are for federations, and the remaining five are for federates.

The federation rules establish ground rules for creating a federation:

1. Federations shall have a FOM, documented in accordance with the OMT. (Documentation Requirements)
2. All representation of objects in the FOM shall be in the federates, not in the RTI. (Object Representation)
3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI. (Data Interchange)
4. During a federation execution, federates shall interact with the RTI in accordance with the HLA interface specification. (Interfacing Requirements)
5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time. (Attribute Ownership)

The federate rules deal with individual federates and describe the federate responsibilities:

6. Federates shall have a Simulation Object Model (SOM), documented in accordance with the OMT. (Documentation Rule)
7. Federates shall be able to update and/or reflect any attributes of objects in their SOM, and send and/or receive SOM interactions externally, as specified in their SOM. (Control of Relevant Attributes)
8. Federates shall be able to transfer and/or accept ownership of attributes of objects, as specified in their SOM. (Transfer of Relevant Attributes)
9. Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM. (Control of Relevant Attributes)
10. Federates shall be able to manage local time in a way which will allow them to coordinate data exchange with other members of a federation. (Time Management)

3.3.2 Object Model Template

In order to provide reusability and interoperability, all the objects and interactions managed by the federate visible to the federation should be specified in detail and documented with a common format. The Object Model Template (OMT) provides a standard for generating such specifications and documentation.

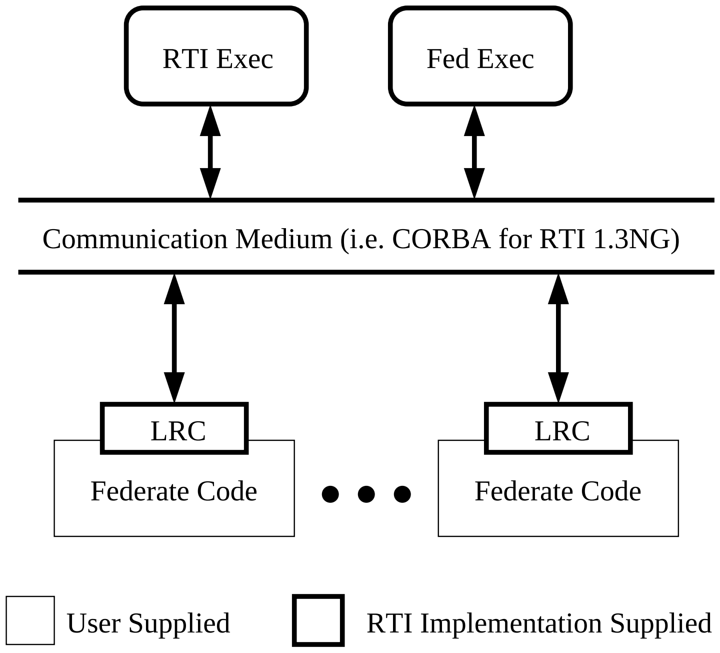
The OMT defines structure of two important documents: The Federation Object Model (FOM) and the Simulation (Federate) Object Model (SOM). The FOM is created per federation, and it's mandated by the first HLA rule. It specifies all the shared information about the interactions, objects, DDM routing spaces and some more information. The FOM contemplates the inter-federate issues.

The Simulation Object Model (SOM) should be defined one per federate. Unlike FOM, its definition is not mandatory for running a federation, the main purpose of SOM is documenting the federate's interaction capabilities. The SOM describes the salient characteristics of a federate and presents objects and interactions that can be used externally. The SOM can be said to focus onto the effects that a federate creates as a result of its internal operation.

OMT also defines a third structure called the Management Object Model (MOM) which specifies the details of the objects and interactions used to manage a federation.

3.3.3 Run-Time Infrastructure

The Run-Time Infrastructure specification defines what services are provided to the federates taking part in a federation. RTI is itself not a software but a list of services, but it's common practice to refer to the RTI implementation as the RTI and following this I will use RTI for both the specification and RTI implementation. This section describes the services provided by the RTI specification, and admittingly it may include some information about RTI 1.3NG of DMSO as it is the RTI implementation used throughout the work done for this thesis, and this implementation is built upon HLA 1.3 so the reader might find differences between the services described in this text and services described in the IEEE 1516 standard.



RTI Exec
Fed Exec
Communication Medium (i.e. CORBA for RTI 1.3NG)
LRC
Federate Code
LRC
Federate Code
User Supplied
RTI Implementation Supplied

Figure 3.1: The architecture of a simulation run.

3.3.3.1 Architecture

The architecture of a running simulation is given in figure 3.1. There are three entities in that figure that I have not defined yet. These are the RTI Executive (RtiExec), Federate Executive (FedExec) and the Local RTI Component (LRC).

The RTI executive is a global process that manages the creation and destruction of federation executions and related federation executives. Each of the federation executions created and managed by an RTI execution has a unique name. The RTI executive is run on one platform and it listens to a well-known port for federation management.

The Federate Executive manages a federation execution. It controls joining and resigning of federates, giving them unique handles, and facilitates the data exchange between the participating federates in the federation execution. The federation execution is created by the RTI executive, by the first federate to join the federation execution.

The Local RTI Component (LRC) makes RTI services methods available to federates in terms of various Application Programming Interfaces (APIs). The APIs provided by the DMSO RTI 1.3NG are in C++, Java, CORBA IDL and Ada.

The interface between the federate code and the LRC is realized by two entities:

the RTI Ambassador and the Federate Ambassador, which are implemented by two classes in the C++ API. The RTI Ambassador provides the service access points for the federate code to access the RTI services defined in the interface specification. The Federate ambassador is created by the user, conforming to its specification in the interface specifications, and provides the callback functions for the LRC to federate code information flow.

3.3.3.2 Services

In this section, the major services of the RTI is described. The services described here is not an exclusive list. An exclusive list of RTI services and information about their implementation in the DMSO RTI 1.3NG can be found in [1].

Objects Related The first major group of services in that is related with the objects are about declaration management and include the "Publish Object Class", "Subscribe Object Class Attributes", "Unpublish Object Class", and "Unsubscribe Object Class" services. As their name implies these services are used by a federate for communicating the data production or consumption interests to the RTI. The data in question is to be produced or consumed in the form of HLA objects.

Another group of services is about object management. The services "Register Object Instance", which is used for creating a new HLA object instance , "Delete Object Instance", which is used for deleting an object instance as the name says, and "Update Attribute Values", which is used for communicating data through given attributes of an object, all belong to this group.

The last group of services contains the services used for ownership management. This group includes the attribute ownership related services including the "Attribute Ownership Acquisition" service used for acquiring the ownership and update privilege of an attribute, and various forms of ownership divestiture services for giving away such privileges.

Interactions Related The services related to interactions may be grouped under the same titles that ware used in the description of the object related services, except that there are no services related to ownership management and far less number of services related to object management, due to the fact that interactions do not persist as the objects does. The services about declaration management include "Publish Interaction Class", "Subscribe Interaction Class", "Unpublish Interaction Class", and "Unsubscribe Interaction Class" services. As in the case of object related services, these services are used for communicating the data production and consumption interests to the RTI. The only difference is that in the case of these services, the subject data is to be produced or consumed in the form of HLA interactions.

In the group of object management related functions, the most important service related to the interactions is the "Send Interaction" service, which is used for sending an interaction to other federates in the federation.

Time Management The time management services in the interface specification of HLA is designed by a group led by Fujimoto and the details of techniques used in implementing the services described here and their usage may be found in [10] and [9].

The RTI supports two delivery modes for events: the receive order and time-stamp order. In the receive order delivery, the events that arrive at the LRC is queued and the federate receives the latest (in wall clock time) event in the queue first. In the time-stamp order delivery mode, the LRC delivers the event with the least time-stamp, which denotes the sending agent's preference of the event occurrence simulation time. The federate has chance of enabling time constrained and/or time regulation time management modes, which control the delivery of the events created or received, and federates ability of incrementing its simulation time. In the time constrained mode, the federate is bound to the simulation time of the time regulating federates, and can generate only receive-order events, whereas it is able to receive time-stamp ordered ones in correct simulation time order. In the time regulating mode, a federate binds the incrementation of simulation time for the time constrained federates in the federation, and may receive and generate events in both receive order and time-stamp order modes. It's also possible that a federate be in neither modes, or in both. When the federate is neither in time constrained or time regulating, it can only receive or generate events in receive order mode, and is free to increment its simulation local time as it wishes. On the other hand, when the federate enables both the time constrained and time regulation modes, it may receive or generate both receive order and time-stamp order events, and it both is bound by and binds the local time incrementation of the other federates in the federation.

The services “Enable Time Constrained”, “Disable Time Constrained”, “Enable Time Regulation”, and “Disable Time Regulation” enables and disables the time constrained and time regulation modes described in the previous paragraph.

A federate may attempt incrementing its simulation time using the services “Time Advance Request”, “Next Event Request”, and “Flush Queue Request” services defined in the interface specification. Using the first service, the federate requests its simulation time incremented to the specified instant. In the next event request case, the federate requests its time incremented to the minimum of the time of the next time-stamped event it should receive and the maximum time it specifies in the service request. The last service, flush queue request, is used for forcing the LRC to deliver all events in the queues waiting to be delivered, even when they belong to the federates future (in simulation time), and LRC acts like an time advance request is made to increment the federate time to a given simulation time instant.

One should also mention the service “Query Federate Time” as a means for a federate to learn its local simulation time.

Data Distribution Management The DDM services defined for objects and interactions differ substantially. For the objects, the “Register Object Instance With Region”, “Subscribe Object Class Attributes With Region”, “Unsubscribe Object Class With Region”, “Associate Region For Updates”, and “Unassociate Region For Updates” services are defined. Using first service, as its name implies, a federate may create a new object instance, and at the same time define the publication regions for the attributes. The subscription and unsubscription services provides the functionality of subscribing to an object class attributes in the given distribution regions, and unsubscribing them. The association and unassociation services bounds the distribution of an object instance attribute update to the given region, and unbounds and unpublishes a specified object instance attribute, respectively.

The services “Subscribe Interaction Class With Region” and “Send Interaction With Region” provides the DDM functionality for the interactions. The subscription service is used by a federate to consume the interactions produced by other federates within a region that intersects with the given region. On the other side, the send service is used by the producer to create an interaction whose distribution is bounded by the specified region.

It must be stressed at this point that the regions for HLA object instance attribute publications stays in effect until changed by the federate using the appropriate service call, whereas the publication region for an interaction is transient just like the interaction itself. It's possible to send each interaction of a class to different regions, but doing so with an object instance attribute takes explicit calls to modify the publication region of it. The importance of this difference in the context of this thesis will be discussed further in the chapters 4 and 5.

Other Services Other services defined in the HLA interface specification include the federation management services and services related to types and ancillary services.

The federation management services include services for creating federation executions ("Create Federation Execution"), joining the federate to a previously created federation execution ("Join Federation Execution"), resigning the federate from a federation execution ("Resign Federation Execution"), and killing a federation execution ("Destroy Federation Execution"), as well as services for introducing federation-wide synchronization points and coordinating federation-wide save and restore operations.

The services related to types and ancillary services are various, and deal mostly with the conversions between the logical names of entities in the FOM and implementation defined federation execution specific handles.

CHAPTER 4

HLA FEDERATE ABSTRACTION LAYER

This chapter describes the HLA Federate Abstraction Layer (FAL) implemented in the scope of this thesis. The first version of FAL was implemented for supporting the SAVMOS project. Then this first version was completely redesigned and recoded to support additional features such as interactions or DDM support. This chapter is organized as follows: The motivation behind designing such an abstraction layer is described. The next section is on the difference between the C++ and HLA models, and is followed by assumptions needed to make a realization possible. In the last section the structure of FAL is described.

The Federate Abstraction Layer is designed for keeping the programmer away from peculiarities of the HLA standard and its RTI implementations. The HLA standard is not a software, but defines services to be provided by a software called RTI to the federates, and some supporting architectural elements. This presents a not-so-small developmental risk due to third party software. Good project management rules of thumb suggest that such third party software be abstracted behind a local (in-project) layer, in case use of standard continued, but the third party implementation is changed. Federate Abstraction Layer provides such a layer between the federates of the SAVMOS project, and the HLA-RTI implementation used, which is the DMSO HLA-RTI NG 1.3. The reader must be reminded here is that in the course of the project, DMSO decided to stop distributing its RTI implementation freely, and commercialized the project. This leaves the HLA based federation designers in Turkey with two alternatives: to obtain a commercial implementation of RTI, or to build one. Both alternatives may lead to change of RTI implementation used in the SAVMOS project, and my hope is that the FAL design will ease the job of maintainers.

4.1 C++ Model and HLA Model

To understand why it is not so straightforward to build HLA federates using C++, especially when both are using some entities called *objects* to model the data plane, we should look deeper into what C++ and HLA objects are. In C++, the structure of the program is a composition of data entities, and their encapsulation using functions that define transformations of these data. This provides the designer a data oriented meta-model for building applications. From this point of view, a running program is a bunch of active data encapsulations that are triggering functions of one-another (see Figure 4.1). When we distribute a system, the C++ model provides no additional support. The subject of distributed systems is simply out of C++'s application context.

HLA, on the contrary, is a native distributed system. The system is based on the idea of distribution and seamless composition, so the system encapsulates the execution units and prevents them from being aware of each other. The federates may be written in any programming language, provided that the RTI implementation has API for that language. Due to the object oriented structure of the HLA standard itself, it might be expected that programming a federate with C++ would be more convenient because of its programming model's similarity.

Although the programming models are similar, the objects of HLA are not exactly like the objects of C++. To understand the difference, its instructing to look at the source of the HLA objects. The objects in the simulation is distributed to the federates for computing state changes through the execution of the simulation. Some of these objects, may be shared among the federates. This forms the basis for data flow in the federation, namely the execution of the simulation, along with interactions. Not all of these objects are to be shared between the federates. The ones that need not be shared, simply live inside the boundaries of particular federates, and their simulation is encapsulated from the other federates in the federation. Figure 4.2 provides a diagrammatic example showing the C++ and HLA objects in a federation.

The things are even more complicated. The fact that federates can observe only a subset of all attributes of a class of HLA objects, and the fact that the set of updated

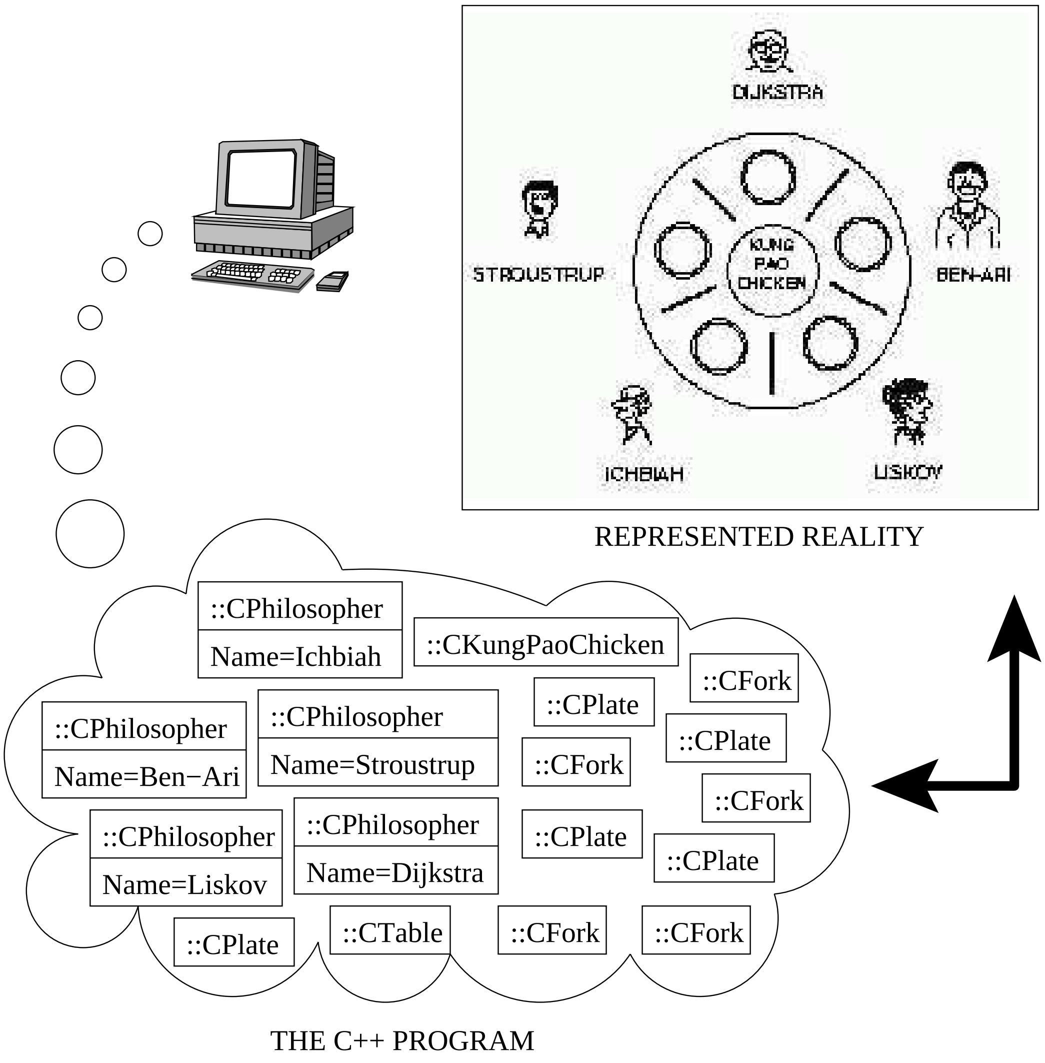


Figure 4.1: The C++ objects in a monolithic object-oriented simulation of the dining philosophers problem. All the objects of the simulation model resides in the same program and this structure is very convenient for using C++ efficiently.

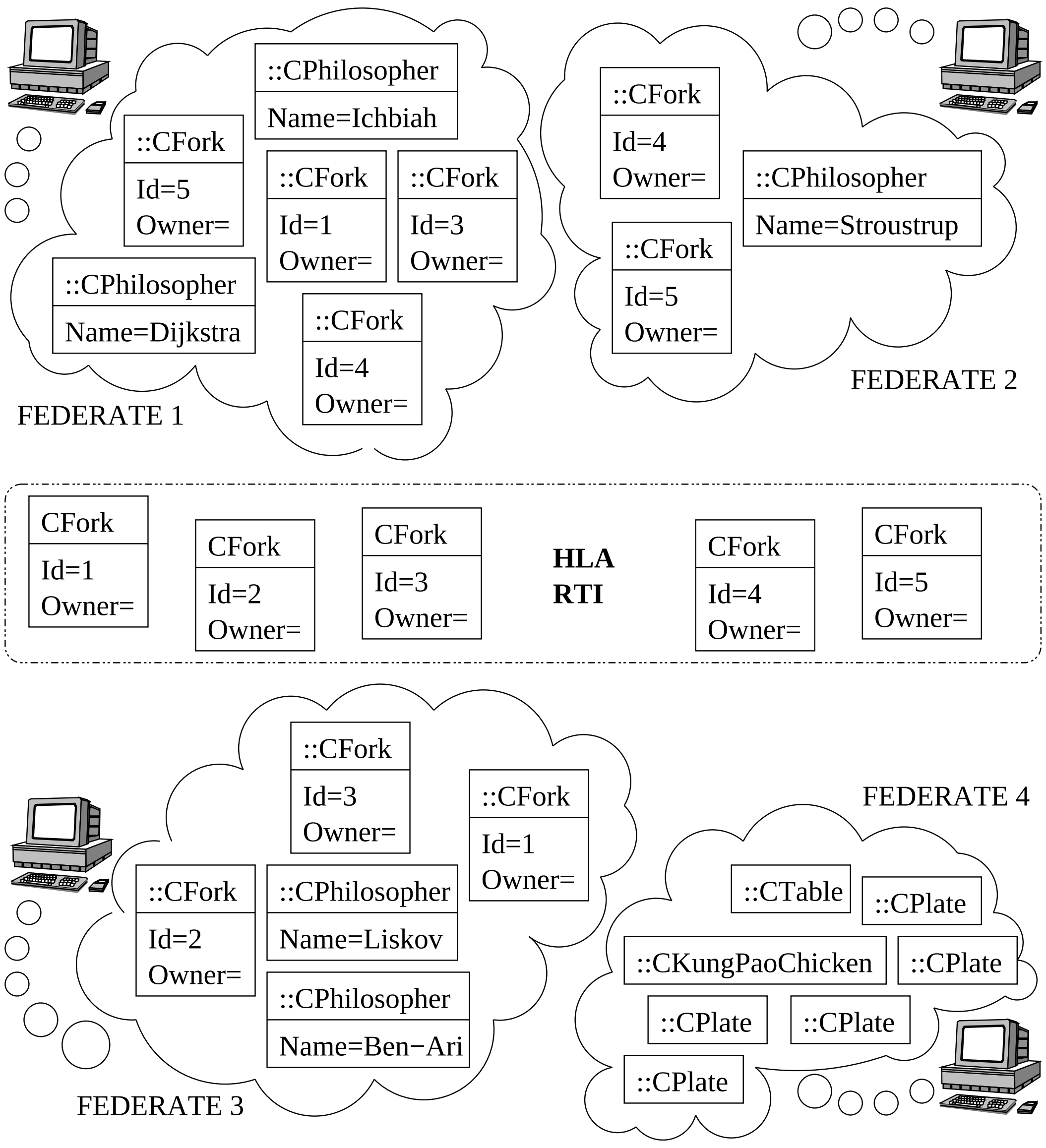


Figure 4.2: A possible simulation of the dining philosophers problem using the HLA. As shown, when the simulation gets distributed, the HLA objects are only the objects whose states is to be shared between federates. These shared objects have mirror copies in all federates that inform the RTI of their interest in subscribing or updating the attributes of them. The other objects of the simulation model live in the vicinity of federates in the federation.

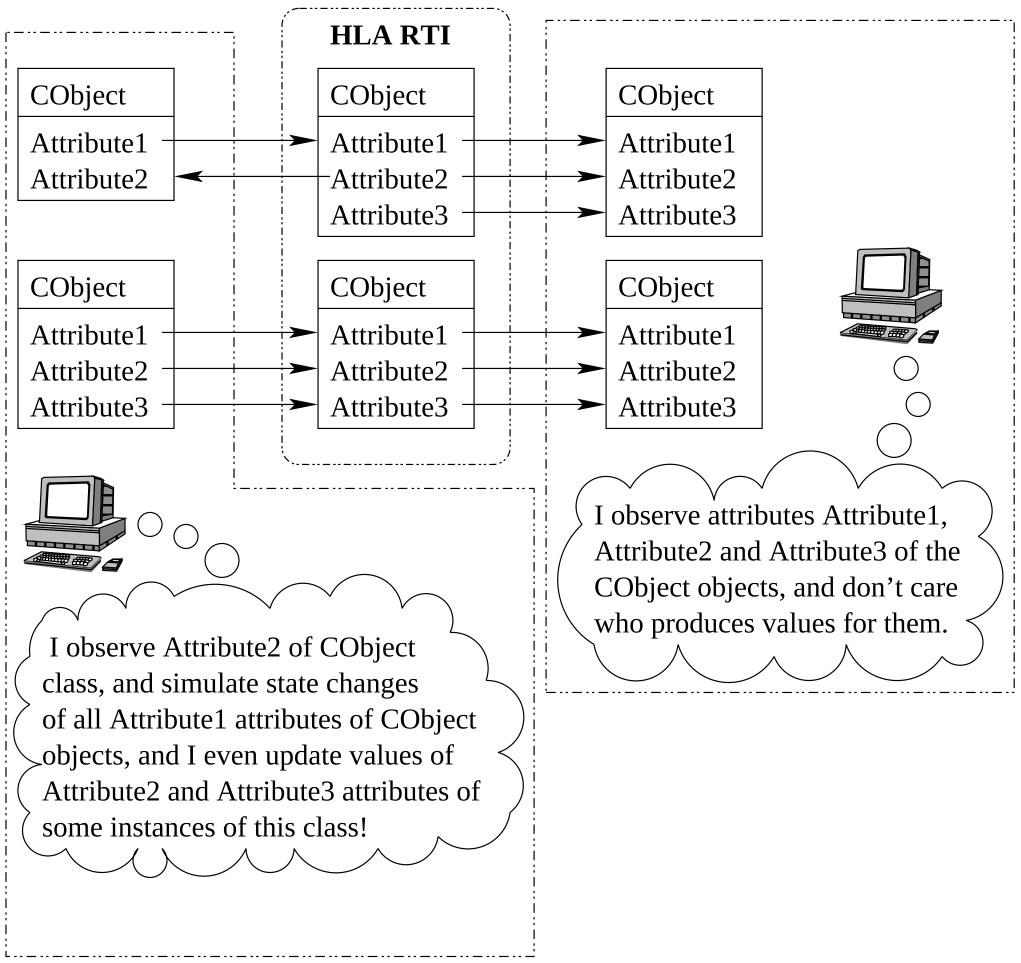

HLA RTI
CObject
Attribute1
Attribute2
CObject
Attribute1
Attribute2
Attribute3
CObject
Attribute1
Attribute2
Attribute3
CObject
Attribute1
Attribute2
Attribute3
CObject
Attribute1
Attribute2
Attribute3
CObject
Attribute1
Attribute2
Attribute3
I observe attributes Attribute1, Attribute2 and Attribute3 of the CObject objects, and don't care who produces values for them.
I observe Attribute2 of CObject class, and simulate state changes of all Attribute1 attributes of CObject objects, and I even update values of Attribute2 and Attribute3 attributes of some instances of this class!

Figure 4.3: The HLA objects are in the simulation execution for the purpose of data sharing. These objects only act as a binder that puts together the relevant data paths, namely the attributes. A federate may subscribe to all or some of the attributes of an object class, and it may choose to update some or all attributes of an object. So HLA objects are quite different than C++ objects.

attributes may differ from one object instance to another makes HLA objects quite different than C++ objects (see Figure 4.3).

The FAL is designed to provide a C++ programming model to cope with the complex mechanisms of HLA. The need is to implement federates so that the federates are keeping mirrors of the HLA object data in the simulation according to their interests, and they must be able to co-operate in state updates. The federate abstraction layer provides these two mechanisms.

4.2 The Underlying Assumptions

HLA provides many mechanisms, in order to support many different simulation implementations. If an abstraction layer to be implemented over HLA does not put some limitations on the possible implementable capabilities, it quickly becomes the API itself, if the standard itself is sufficiently evolved. The FAL does not aim to be complete in the sense that every program implementable using the API's of HLA-RTI, is not implementable in FAL. This section mentions the assumptions about the structure of simulation models to be implemented using FAL.

HLA's time management services are defined in terms of some number of services, that is HLA does not mandate any time management schemes to the programmer/designer of federates. To provide ease of use, FAL operates in some time management modes, implemented using the HLA's primitives. These time management modes of FAL are *no management*, *constrained*, *restricted*, *synchronized*, and *optimistic*. In no management mode, as the name implies, the federate neither does participate in any way to the updating of the simulation, nor the federation time has any effects on the federate's time. The constrained and restricted modes are the same as using the HLA's SET TIME CONSTRAINED and SET TIME REGULATING services (not their combination). The synchronized mode is the combination of these two modes, and in this mode the federate is both bound to and binds the simulation time of all federates in the federation. In optimistic time management mode, the federate is in the synchronized mode, but FAL permits usage of HLA's FLUSH QUEUE REQUEST service to get the events in federate's future before the federate time is advanced to the time of these events.

Time advancing mechanism is also less complex than HLA. FAL supports three time advancement methods of the HLA, the TIME ADVANCE REQUEST, GET NEXT EVENT, and FLUSH QUEUE REQUEST. These three time advancement methods block federation until a time advance is granted. This restriction, which is not present in the HLA services, eases programming of federates considerably. The reader must be reminded here is that it's not an objective of FAL to provide the freedom HLA-RTI standard provides to the federate designer.

In FAL, the HLA object class subscriptions and publications are determined when the object class is registered to the RTI, and FAL does not let the federate change its subscriptions of publications. This means that the subscription information for an object class C whose attributes A_c1 and A_c2 is given to the FAL at the registration time of the class. It's not possible for the federate to unsubscribe to the attribute A_c1 or A_c2 after this registration. The same is true for publications. When DDM is used, setting the region of the subscription to an impossible or unused region provides a turn-around solution to the unsubscription problem. The solution for not publishing is to release the ownership of the created object instances whose attributes will not be updated in the course of simulation.

The HLA allows more than one attributes to be grouped under a DDM region. This behavior is to be used for attributes that have the same distribution. Such a behavior is not supported in the FAL. In FAL, all attributes that are managed according to the DDM have their own distribution regions defined. This increases the number of regions necessary, but provides an implementable general mechanism for DDM region management. One of the problems that lead me to such a design is described in the following paragraph.

Changing the subscription region of an attribute means unsubscribing and resubscribing to an the attribute with the new region. This mechanism is not the best possible way to implement region changes, but DMSO RTI 1.3-NG's region implementation is somewhat restrictive in the sense that one needs to specify the maximum number of extents, the continuous sub-regions, in the region at the region object creation time. Furthermore, the class provides no way of deleting an extent in a region object. With the only possible action of adding extents, the region object class of RTI 1.3NG is not adequate for efficiently implementing region changes. To get around this inadequacy, region change mechanism in the FAL had to be implemented by unsubscribing to the attribute and destroying the old subscription region object, then creating a new region object and resubscribe to the attribute with the new region. This unsubscribe-resubscribe mechanism has some implications of its own. The RTI 1.3NG local RTI library discards any relevant events received prior (in terms of wall clock time) to the unsubscription, even in the case that the event belongs to the federate's future (in terms of simulation time). In Figure 4.4 this is diagrammed. When the federate with time simulation time F unsubscribes to an attribute, the LRC discards the relevant events in its future, the events $e1$, $e2$ and $e3$. The same is true for

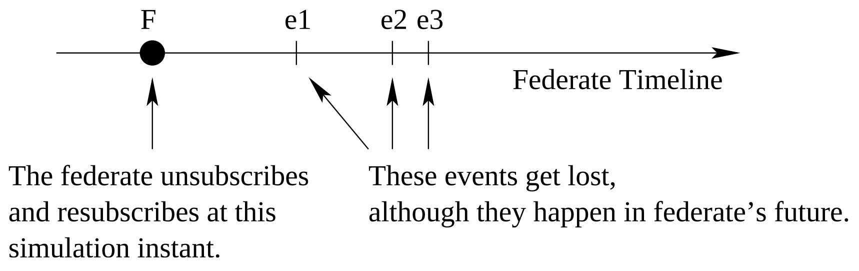

F
e1
e2 e3
Federate Timeline
The federate unsubscribes and resubscribes at this simulation instant.
These events get lost, although they happen in federate's future.

Figure 4.4: The unsubscription causes discarding of future events problem.

interactions, that is unsubscribing to an interaction causes interactions received and waiting for delivery in the LRC event queue be discarded. This discarding of events poses no problems for implementing KQML, as will be described in the next chapter.

The FAL also ignores the publish relevance advisories, which RTI creates to inform the federate about presence or absence of consumers for an attribute or interaction it provides. Such functionality can easily be implemented into the FAL library, but is ignored for the moment and not implemented as of this date.

The assumptions in this section are made in order to simplify the framework, as the framework should be as simple as possible to increase its usability. RTI's service structure is too unrestrictive for most of the simulation implementations, and its very natural since HLA aims to be a very general and adequate simulation standard.

4.3 Structure of FAL

This section describes the structure of the FAL designed. Below, the section starts with a look at the architectural place of the library. Then in the next subsection, the description of the public interfaces of classes is given. How to use the classes, including diagrammatic information on how the FAL works internally is given in the subsection 4.3.2.

Where the FAL fits in the federate architecture is given in Figure 4.5. The federate code, that is the code written by the user for the simulation purpose, lies above the FAL and is completely abstracted from the Local RTI Component (LRC). The FAL includes all the code that is to interact with the RTI, either by calling the services of the RTIAmbassador class or by acting as the FederateAmbassador to the LRC and receiving the callbacks generated. So all events received from the RTI is routed to the appropriate places in the user code by the FAL. This way, the single communication

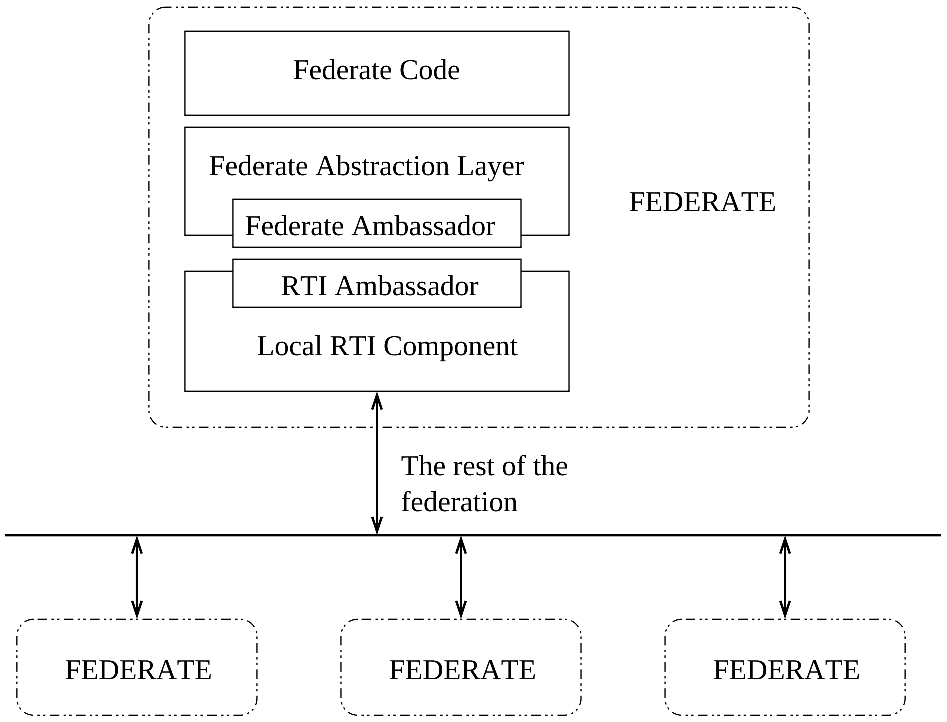


Figure 4.5: The figure shows where the federate abstraction layer fits in the overall architecture of the federation.

channel of RTI which has multiple logical channels (attributes and interactions) is reflected to the user code as multiple communication channels that end up in the appropriate places in code (see Figure 4.7.

4.3.1 Classes

The class hierarchy of the component of FAL is given in Figure 4.6. As seen from the figure, the CFederate object instance is where the information about almost all the object instances is collected. The object instances are created in a hierarchical factory design in which CFederateClass, CFederateInteractionClass and CDdmSpace object instances are created from the CFederate object instance, and CFederateObject, CInteraction and CDdmRegion object instances are created from them. With this mechanism, necessary run-time information is efficiently distributed to all the objects in the FAL.

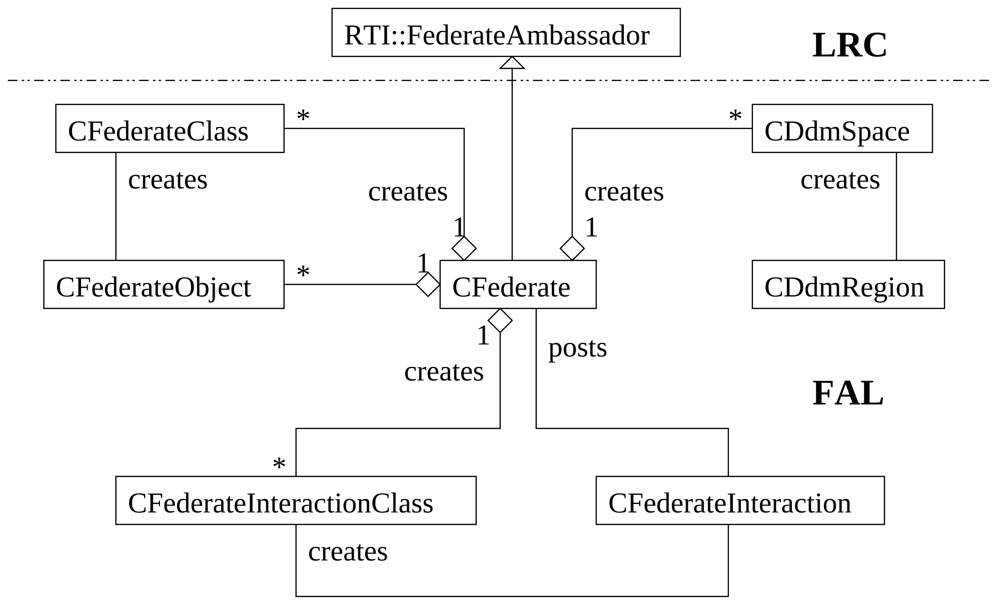


Figure 4.6: The class hierarchy of the classes in the federate abstraction layer library.

4.3.1.1 CFederate

The CFederate class is the federate's doorway to the rest of the federation. In a federate implementation, there should be exactly one instance of this object. Having more than one instance of this federate might work, but FAL is not designed for such a use and it's not tested this way. Furthermore, RTI 1.3NG documents explicitly say that having two different federates implemented in one program might create problems.

The CFederate class encapsulates all RTI related function calls and data structures, and derives from the RTI's FederateAmbassador class in order to be able to register itself as the target object instance of RTI's callbacks. Other classes of FAL does not include any code related to the RTI library. This way, the RTI risk is confined into the boundaries of this class, and this class is the smallest unit that must depend on the actual RTI implementation. The time, federate, object, interaction, DDM management services are all obtained by FAL classes or by the user code of the federate through calling the methods of the instance of this class.

CFederate object instance acts as a data router between the object instances created in the federate and the RTI, for both mirrors of external objects whose updates are observed and the objects whose attributes are updated by the federate, and even the objects with a combination of subscribed and updated attributes. It also routes

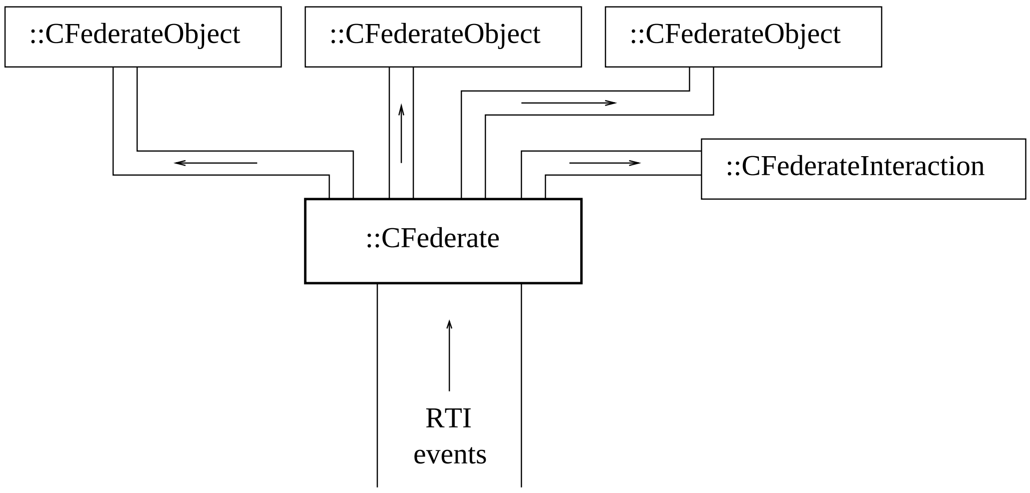


Figure 4.7: The events from the RTI arrive in a logically divided single communication channel. This single physical communication channel ends up in a method of the CFederate object instance of the federate, and this method is called by the LRC to inform the federate about the incoming events. The CFederate objects acts as a software demultiplexer to route the incoming events to the relevant CFederateObject and CFederateInteraction classes, creating them if needed.

the interactions created by the federate and creates appropriate interaction classes and informs the federate of their arrival.

CFederate object instance also acts as a factory class for creation of CFederateClass and CFederateInteractionClass object instances. By this means, these object instances get initialized at their creation with the dynamic information obtained, which might change from one execution to another.

Types

timeHandlingStyle

The timeHandlingStyle enumeration is used in setting the time management mode of the federate. The valid values for the enumeration are *timeNoHandling*, *timeRegulating*, *timeConstrained*, *timeSynchronized*, and *timeOptimistic*, and these correspond to the time management schemes described in section 4.2.

Methods

CFederate()

The constructor of the federate, contrary to the intuitive expectation, does not call any RTI services. It merely initializes the state of the newly created CFederate object instance and sets the federate name to be used by the federate.

createFederation()

This method attempts at creating a new federation execution. The preferred method of joining a federation includes attempting to create a federation execution as well, in the case that the federate is the first one to start execution. To allow such a use, this method ignores the error of an already existing federation execution. So the net effect of calling this function is that either the call fails, or the federation execution exists after this method returns.

joinFederation()

This method attempts at joining the given federation execution given number of times. The reason that it may be needed to try several times is that the createFederation function call might return successfully before the federation execution gets fully initialized and interactable, and it's possible that in the course of its initialization, the federate might receive failure indicating the unexistance of the specified federate execution.

resignFromFederation()

This method resigns the federate from the federation execution.

killFedExec()

This method attempts at killing the federation execution the federate was in. This method should be called after resigning from the federation, otherwise it has no effects since it ignores the failure case of other federates being still in the federation executions. This behavior is provided for implementing the suggested method of resigning from a federation execution, which also includes killing the federation execution if the resigning federate is the last one.

createClass()

The CFederate object instance also acts as a factory for the CFederateClass objects. This method creates a new CFederate object instance allocated in the heap, initializes and returns it. This method takes as a parameter the list of attributes to be subscribed and the list of attributes to be published, and as a result of this call, the subscription and publication information is communicated to the RTI.

setTimeManPolicy()

Sets the time management policy for the federate, as described in Section 4.2. This method is where the federate lookahead is set, if it is necessary for the time management mode.

timeAdvanceNextEvent()

This method advances the simulation time of the federate to the minimum of the next event the federate is to receive and the time given as parameter, and returns the granted simulation time. This method blocks until a time is granted.

timeAdvanceToInstant()

This method advances the simulation time of the federate to the given time, and causes events whose time is smaller than the given time to be delivered to the federate. These events arrive at the CFederate object instance in the form of method callbacks, and the data in these events are routed to the appropriate objects in the federate. This method also returns the granted simulation time.

timeFlushQueue()

This method is used only when the federate is in optimistic time management mode. This method causes all events waiting in the LRC to be delivered to the federate. Furthermore, the simulation time is advanced to the given time.

getEvents()

This method give LRC computation time to obtain any events waiting for delivery to the federate.

getSimulationTime()

This method returns the current simulation time.

getLookahead()

This method returns the current lookahead of the simulation.

createDdmSpace()

The CFederate object instance act as a factory of the CDdmSpace objects. This method creates a new CDdmSpace object for use later in DDM related services.

4.3.1.2 CFederateClass

CFederateClass class represents an HLA object class. This class holds information about the object class attribute subscriptions and publications. Through its methods, DDM subscription ranges might be set for individual attributes. This class also act as a factory class for creating instances of the class CFederateObject.

Types

There are no public types of this class.

Methods

CFederateClass()

The constructor of the class, which is called only by the CFederate class methods and never by the user of the FAL library, initializes the internal data structures and registers the class to the CFederate object.

getAttributeIndexes()

The attributes of the HLA object class represented by the CFederateClass object are kept in an indexed structure for fast access. These indexes, which are also called the local handles, can be queried through this function. This method returns an associative list of attribute local handles indexed by attribute names. The user code is expected to keep these attribute handles and use them for communicating with the FAL library.

isPublished()

This method queries if the given attribute is published or not.

isSubscribed()

This method queries if the given attribute is subscribed or not.

createFederateObject()

This method creates a CFederateObject object instance, and initializes the created object. It uses a function obtained from the user at CFederateClass object instance creation time to create an instance of the right subclass of the CFederateObject class.

setAttrSubsDdmRegion()

Sets the DDM region for subscription to the given attribute. This call may cause an unsubscribe-resubscribe call pair as described in the section 4.2.

4.3.1.3 CFederateObject

The CFederateObject class represents an instance of an HLA object present in the federation execution. This class is abstract, and is expected to be subclassed by the user of the FAL library to create appropriate object with data serialization-deserialization functions for attribute updates and reflections.

Types

There are no public types of this class.

Methods

This class has no public methods since it is designed to be subclassed by the user and the user is to provide necessary methods in the subclass for accessing the member variables of the subclass, be them reflections of HLA object attributes or some other variables used only internally. The methods to be overloaded are protected members of CFederateObject class.

4.3.1.4 CFederateInteractionClass

The CFederateInteractionClass represents an HLA interaction class. This class holds information on the publication and subscription status of the interaction class, as well as the attribute names and is used as a factory class for creating CFederateInteraction object instances, setting DDM subscription region information for the HLA interaction class subscription, and sending an interaction of this class distributed to all federates or to a given DDM region in the appropriate space.

Types

There are no public types of this class.

Methods

CFederateInteractionClass()

The constructor initializes the newly created CFederateInteractionClass object instance and registers the instance to the CFederate object. This constructor is to be called only by methods of CFederate class, and never by the user of the FAL library.

isPublished()

This method returns if the interaction class is published or not.

isSubscribed()

This method returns if the interaction class is subscribed or not.

sendInteraction()

Sends the given interaction. This method acts as a mediator between the CFederate object and the user code.

sendInteractionWithRegion()

This method sends the given interaction with distribution limited to the given DDM region.

createInstance()

Creates a new CFederateInteraction object and initializes it. This method uses a function set at the CFederateInteractionClass object instance creation time for creating the right subclass of the CFederateInteraction class. The user code is expected to manipulate the parameter values of the created interaction and send it using either sendInteraction or sendInteractionWithRegion methods.

getParamHnd()

Returns an associative list of the local handles of the parameters of the HLA interaction class, indexed by parameter names. The user code is expected to communicate with the FAL library using these local handles.

getDdmSpace()

This method returns the DDM space object which is specified in the FED file of the federation for the represented HLA interaction class. The DDM space should have been created before a call to this method using appropriate methods of CFederate.

setSubscriptionRegion()

This method sets or modifies the subscription region of the interaction class. Calling this function causes the federate to unsubscribe and resubscribe to the interaction class, and causes the interactions already received and queued by the LRC but not delivered to the federate be discarded as described in Section 4.2.

4.3.1.5 CFederateInteraction

CFederateInteraction class represents an HLA interaction and is created either to be sent or as response to an arriving interaction. This class is abstract, and should be subclassed for each interaction class that is to be used by the federate. The protected members of the class provide necessary serialization-deserialization interface for the FAL, and they must be overloaded by the user of the FAL library.

Types

There are no public types of this class.

Methods

setTime()

This method sets the simulation time to be put in the event for the interaction as the time stamp.

getTime()

This method is used to query the time stamp information of the interaction.

4.3.1.6 CDdmSpace

This class represents a DDM space defined in the FED file of the federation and keeps necessary information such as the names of the dimensions. An instance of this class is created using a CFederate object instance as a factory. This class acts as a factory class for creating new regions of the space represented.

Types

There are no public types of this class.

Methods

CDdmSpace()

The constructor initializes the object members.

getDimensionHandle()

This function returns the local handle of the dimension of the DDM space represented. The dimension whose local handle will be returned is denoted by its name as a string argument.

createRegion()

Creates a new region of the represented space.

4.3.1.7 CDdmRegion

The CDdmRegion class represents a region of a DDM space. Instances of this class is to be created using the appropriate CDdmSpace object instance. Instances of this class is used as arguments in the DDM related services provided by the FAL. This class also act as a factory for creating its extents as CExtent objects.

Types

CExtent()

CExtent object represents one continuous sub-region of a region. It is created through calling the appropriate functions of a CDdmRegion object instance.

Methods

CDdmRegion()

Initializes a new CDdmRegion object. This method is never to be called directly by user code, since CDdmRegion objects are to be created by using appropriate functions of the relevant CDdmSpace object instance.

createExtent()

Create a new extent object initialized with the necessary data from the creating CDdmRegion object instance.

addExtent()

Add an extent to the list of extents for the region.

getNumOfExtents()

Returns the number of extents used in the definition of the region.

getExtent()

Return the extent with the given index. The list of extents can be accessed as an integer indexed array through this function.

4.3.2 Workings of FAL

In this section, how the FAL is used for obtaining some primary services from the RTI and what happens in the library and between the library, user supplied code and the LRC is described in a manner that is free of unnecessary details. The FAL library provides slightly more functionality that is described in this thesis, yet the major functionality without details is considered to be enough for presenting the general outline and flavor of the design. Most of the description in this section is given in terms of UML like figures, and the reader must be reminded that the author did not attempted presenting diagrams in a way that fully conforms to the UML standard, whenever the diagrams would be more comprehendable otherwise.

4.3.2.1 Creating and Destroying a Simulation

The lifecycle of a federate is given in the Figure 4.8. The federate starts by attempting a new federation execution in case one does not exists, and joins the federation execution that is just created or was already present. Then the federate carries out computations in the federation. When the simulation ends, or the federate has no more to do in the course of simulation, the federate first resigns from the federation, and attempts at killing the federation execution. This kill federation execution service call fails if there are still federates connected to the federation execution, so this last call succeeds only for the federate that resign the latest.

What happens internally between the CFederate class, user code and RTI ambassador in the execution of a federate is shown in Figure 4.9.

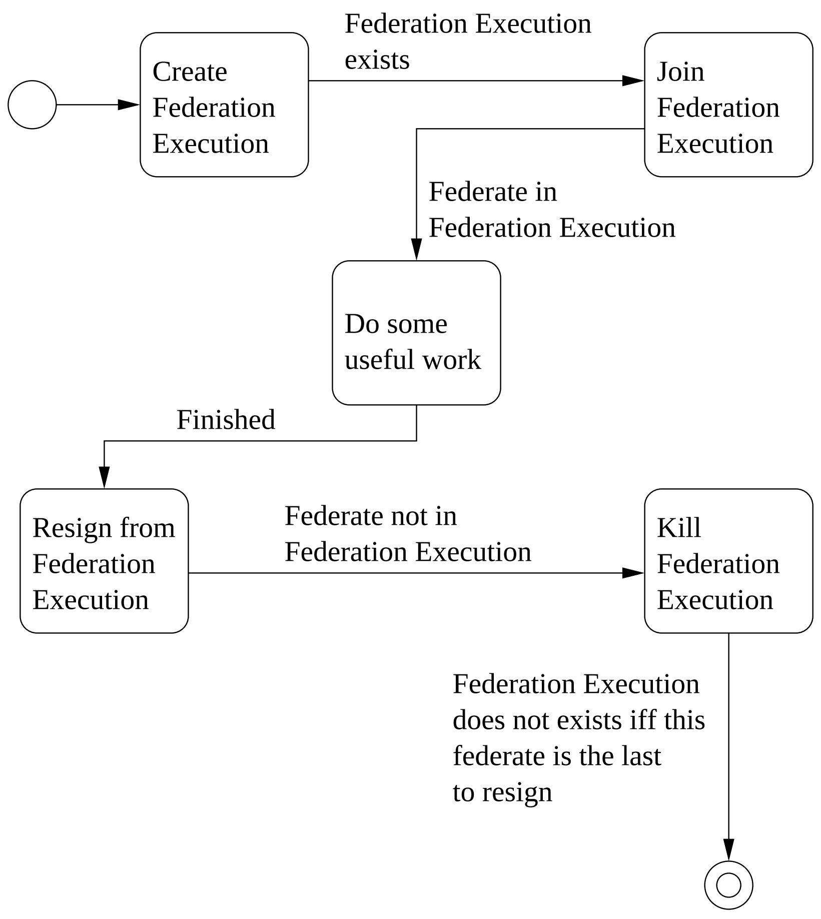

Create Federation Execution
Federation Execution exists
Join Federation Execution
Federate in Federation Execution
Do some useful work
Finished
Resign from Federation Execution
Federate not in Federation Execution
Kill Federation Execution
Federation Execution does not exists iff this federate is the last to resign

Figure 4.8: The federate lifecycle. Federate starts by attempting at creating a federation execution and then joins the federation execution. Then the federate does some simulation work, most probably interacting with other federates in the federation. When the federate finishes doing its job, it resigns from the federation execution and attempts at stopping the federation execution as well in case it is the last federate left in that federation.

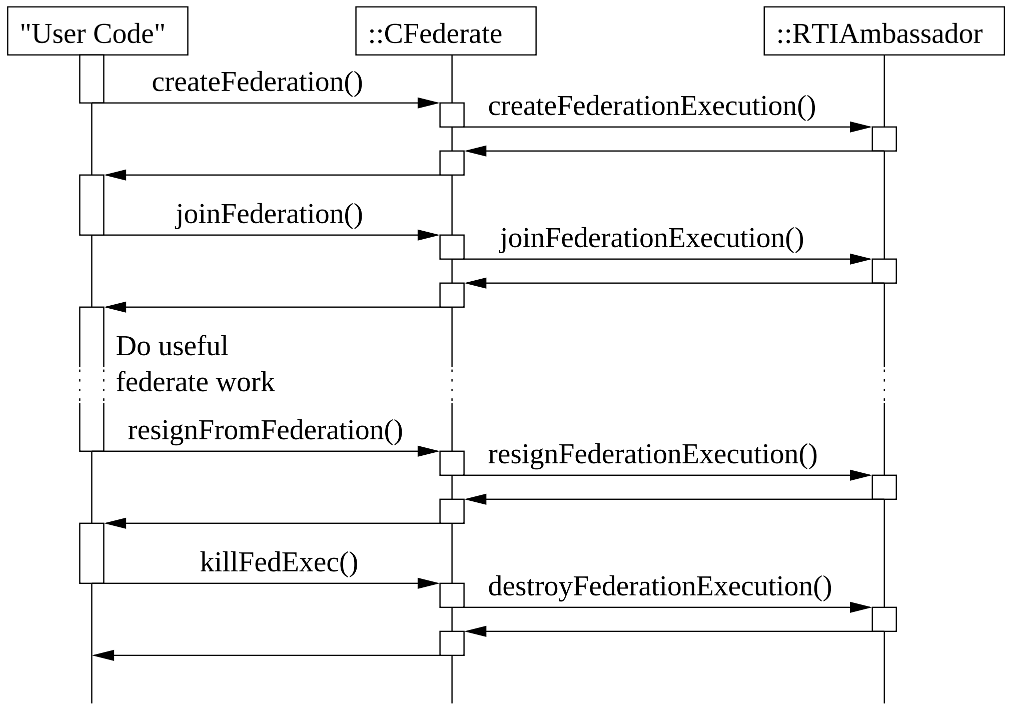


Figure 4.9: Lifecycle of a federate shown in a function calls level.

4.3.2.2 Interacting via Objects

To interact with an HLA object instance, a federate must first create a representation of the class of the HLA object. HLA object classes are represented by CFederateClass objects in FAL and created by calling a method of the CFederate class, as shown in Figure 4.10. At CFederateClass object instance creation, the user also supplies as an argument an object instance creation function using which instances of appropriate subclass of the CFederateObject class will be created for representing the HLA object instances.

The HLA objects are represented by instances of subclasses of the CFederateObject class, as described in Section 4.3.1.3. An instance of a subclass of the CFederateObject class is created by calling a method of the CFederateClass object instance representing its HLA class, as shown in Figure 4.11.

The instances of the subclasses of CFederateObject class, according to the subscription and publication concerns of the federate, receive or send updates of the attributes in the HLA object instances they represent. Figures 4.12 and 4.13 show the internal mechanisms in receiving and sending these updates respectively.

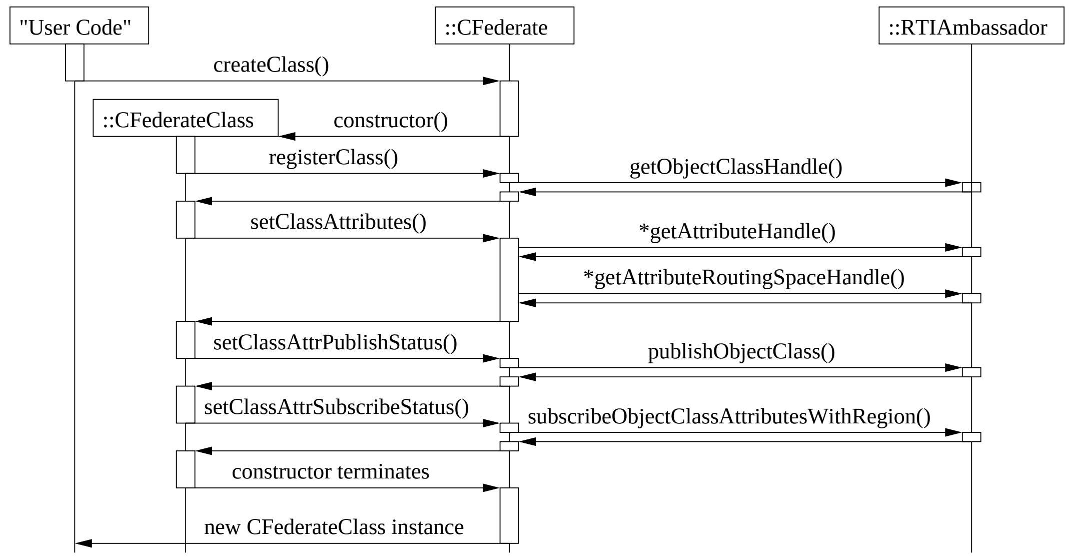


Figure 4.10: Creation of a representation of an HLA object class.

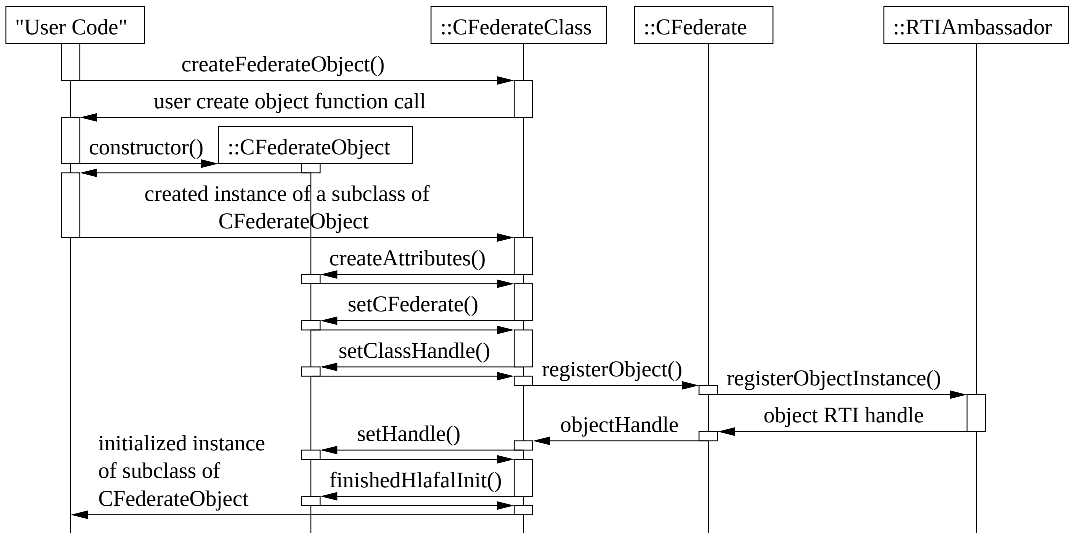


Figure 4.11: Creation of a representation of an HLA object instance. The object instance created is an instance of a subclass of CFederateObject FAL class. This subclass is to be provided by the use of FAL and includes functions for serializing the attributes in a network transferable byte stream and for deserialization.

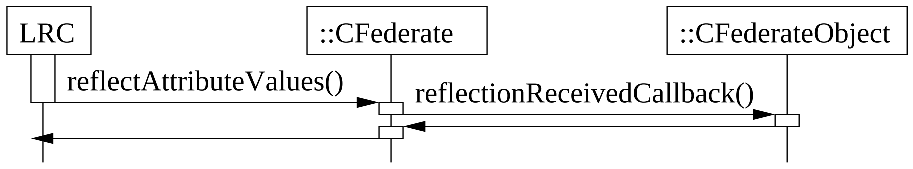


Figure 4.12: Reception of an attribute update. The user overloaded protected member reflectionReceivedCallback of the CFederateObject FAL class provides the deserialization functionality for the received attribute.

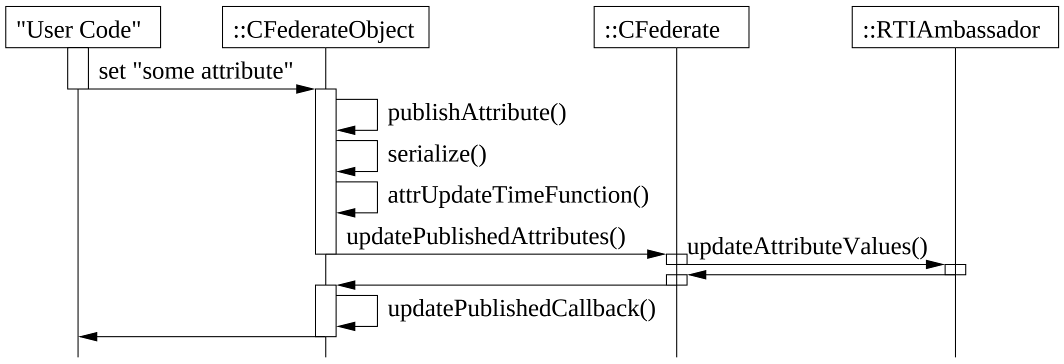


Figure 4.13: Sending of an attribute update. The user overloaded protected members serialize and attrUpdateTimeFunction of the CFederateObject FAL class provides the serialization and time stamp setting functionality for the attribute to be updated. The protected member publishAttribute of CFederateObject is the single function call made by the user through the methods of the appropriate subclass.

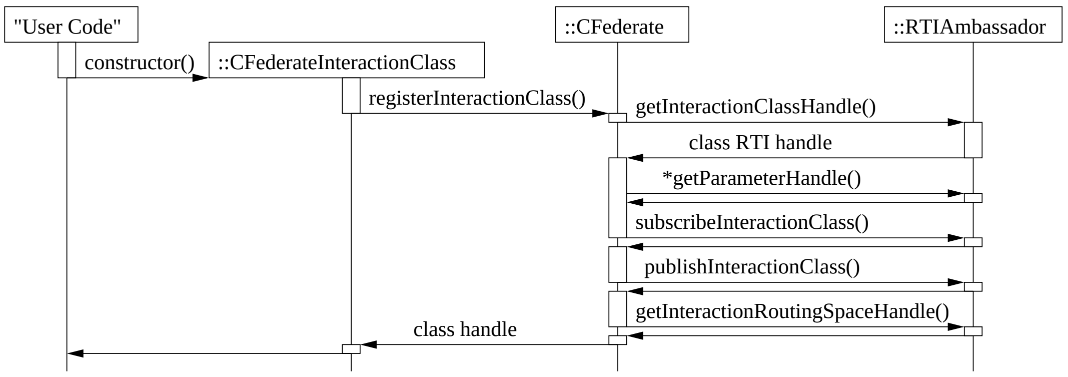

"User Code"
::CFederate
::RTIAmbassador
constructor()
::CFederateInteractionClass
registerInteractionClass()
getInteractionClassHandle()
class RTI handle
*getParameterHandle()
subscribeInteractionClass()
publishInteractionClass()
getInteractionRoutingSpaceHandle()
class handle

Figure 4.14: Creating a representation of an HLA interaction class. Note that * is used to indicate possibility of multiple calls.

4.3.2.3 Interacting via Interactions

The interaction classes defined in the federation's FED file are represented by CFederateInteractionClass object instances in the FAL. What happens internally when the user creates an instance of the CFederateInteractionClass class is given diagrammatically in Figure 4.14. As in the CFederateObject creation process, user must supply a function for creating instances of the correct subclass of CFederateInteraction class that represents the interaction instances of the HLA interaction class represented by the created CFederateInteractionClass, and an additional function for notification when a new interaction of the created class is received.

When an interaction of a subscribed HLA interaction class is received, a new CFederateInteraction object instance is created and the data is loaded onto this new

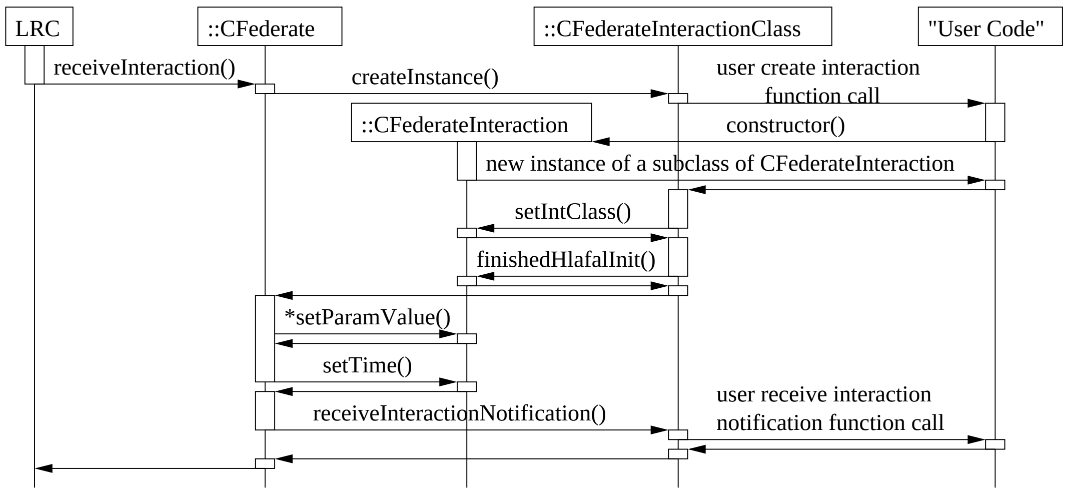

LRC
::CFederate
::CFederateInteractionClass
"User Code"
receiveInteraction()
createInstance()
user create interaction function call
::CFederateInteraction
constructor()
new instance of a subclass of CFederateInteraction
setIntClass()
finishedHlafalInit()
*setParamValue()
setTime()
receiveInteractionNotification()
user receive interaction notification function call

Figure 4.15: Receiving an interaction. The user supplied interaction instance creation function is used for creating an instance of the appropriate subclass of the CFederateInteraction class of FAL, thereby mostly eliminating the need for dynamic run time type information. Note that $*$ is used to indicate possibility of multiple calls.

object. Then this new object is delivered to the user notification function as seen in Figure 4.15.

To send an interaction, the user code must call a method of the CFederateInteractionClass that is representing the appropriate HLA interaction class to obtain a new CFederateInteraction object instance. Then the user code might set the parameters of the instance and send the interaction with the appropriate method of the relevant CFederateInteractionClass. The sequence is given in Figure 4.16.

4.3.2.4 Using DDM with Interactions

The interaction subscription region is set by calling the appropriate function of the CFederateInteractionClass object representing the HLA interaction class of the interaction. The Figure 4.17 shows what happens when setting the subscription DDM region of a newly created CFederateInteractionClass object instance. A new CFederateInteractionClass object instance, if subscription is requested, is subscribed without any DDM subscription region. When a DDM subscription is set, the RTI is informed of an unsubscription and a resubscription with the given region. This unsubscription-resubscription also happens when the interaction class is already subscribed with a DDM region, but the RTI services invoked change.

For sending an interaction into a DDM region, first an interaction is created from

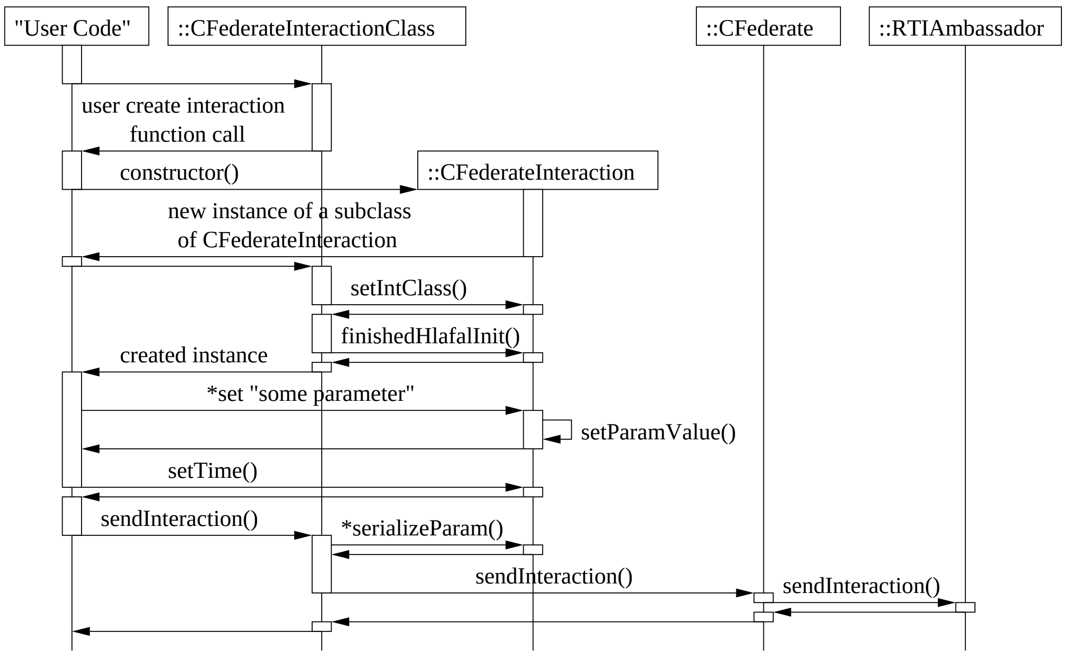
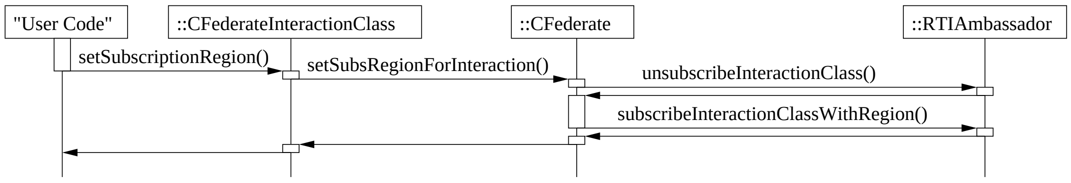


Figure 4.16: Sending an interaction. The user defined subclass of the CFederateInteraction FAL class is assumed to have individual value set methods for each of the attributes the federate is interested in publishing. The "set some parameter" call indicates such a call to set a value of a parameter. Note that * is used to indicate possibility of multiple calls.



"User Code"
::CFederateInteractionClass
::CFederate
::RTIAmbassador
setSubscriptionRegion()
setSubsRegionForInteraction()
unsubscribeInteractionClass()
subscribeInteractionClassWithRegion()

Figure 4.17: Setting the subscription region for an interaction class.

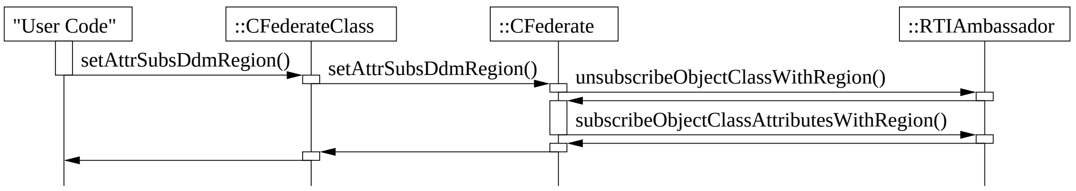


Figure 4.18: Changing subscription region for an HLA object class attribute.

the CFederateInteractionClass object representing the appropriate HLA interaction class. Then the instance of the subclass of the CFederateInteraction class is modified and data of parameters of the interaction to be sent is set, and a new CDdmRegion object is created and set according to the region boundaries that the interaction is to be sent. Finally the prepared interaction instance is sent by calling the sendInteractionWtihRegion method of CFederateInteractionClass, instead of the sendInteraction method in Figure 4.16.

4.3.2.5 Using DDM with HLA Objects

The use of DDM in object subscription and publication mechanisms of RTI are quite different in the sense that while subscription is class-wide and the regions are determined for the attributes of all objects in the class, the publication regions are defined per instance of the class. Because of this, use of DDM in object attribute subscription and publications is handled quite differently in the FAL.

Change of regions for both publication and subscription turns out to be problematic. The subscription is problematic due to the discarding of future events problem described before. When a new HLA object class is subscribed, all attributes are associated with the default region, as this is said to create the same effect of subscribing without a region, in the RTI-NG 1.3 programmer's manual. Whenever a region change is requested, the attribute subject to the region change is unsubscribed and then resubscribed. The details of how to change the subscription region and what happens is given in Figure 4.18.

The change of region for publication of an attribute is also problematic due to the fact that RTI does not guarantee delivery of future events when the update region of the attribute is changed. In the programmers manual, it is said that RTI takes into consideration both the update and subscribed regions at the time the event is delivered to the receiving federate, which may mean that the update and subscribed

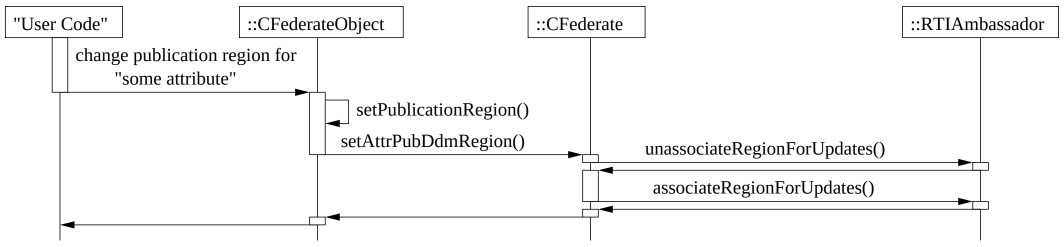

"User Code"
::CFederateObject
::CFederate
::RTIAmbassador
change publication region for "some attribute"
setPublicationRegion()
setAttrPubDdmRegion()
unassociateRegionForUpdates()
associateRegionForUpdates()

Figure 4.19: Changing publication region for an HLA object instance attribute.

regions may no longer intersect, causing LRC to discard the event. The regions for publication is set per HLA object instance attribute as shown in Figure 4.19.

CHAPTER 5

IMPLEMENTING KQML

This chapter describes the proposed solution to the problem of providing KQML agent language for use by agents in a multiagent simulation that is to be run on top of the HLA architecture.

The title of this chapter is deliberately chosen to remind the reader that an agent communication language (ACL) is not an interpreted or a compiled language, and cannot have an implementation per se [17]. What we mean by implementing KQML is implementing an architecture in which agents may communicate respecting the semantics of KQML performatives as defined in [16] and [3], building on top of HLA as the transport medium.

5.1 KQML Architectural Assumptions

Like every ACL, KQML makes its own assumptions about the environment the language is going to be used. These assumptions have to be satisfied for the language to be used in a consistent manner. The outset point for making KQML work in an HLA environment is to identify these assumptions and then try mapping them to appropriate HLA architectural elements and mechanisms.

First of all, agents exist in a KQML spoken environment. According to KQML designers, these agents might not be quite alike, the system might be heterogeneous. This implies that agents might have different capabilities and it is not mandated by KQML that these agents know of each other, but they must have ways of locating each other so that they may query each other in order to learn their capabilities. The fact that agents may be quite dissimilar is acknowledged by mentioning of a special kind of agent, the facilitator, in the specifications of KQML. The capability queries are done by exchanging KQML messages, the discovery is left to the multiagent system designer.

Another concept is the concept of a message. The message is an encoded form of the KQML message generated. All agents are assumed to know how to encode and decode a KQML message.

KQML designers have also made some assumptions about the transport layer that will carry the KQML messages. Although the specification of KQML mostly follows the distinction between the layers that is defined in Chapter 2, the "register" and "transport-address" performatives in KQML require the agent to record the transport address the KQML message come from, introducing the transport details into their semantics and blur the distinction between the transport and the communication layers.

The transport assumptions KQML designers make are as follows:

1. Agents are connected by unidirectional communication links that carry discrete messages. These links may have a non-zero message transport delay associated with them.

2. When an agent receives a message, it knows from which incoming link the message arrived. When an agent sends a message, it may direct the message to a particular outgoing link.

3. Messages to a single destination arrive in the order they were sent.

4. Message delivery is reliable.

It should be observed that the first assumption, combined with the second one, calls for point-to-point links between the agents. Furthermore, the third assumption calls for send-order reception of messages. As will be discussed in the next section, these assumptions turn out to be the most problematic ones.

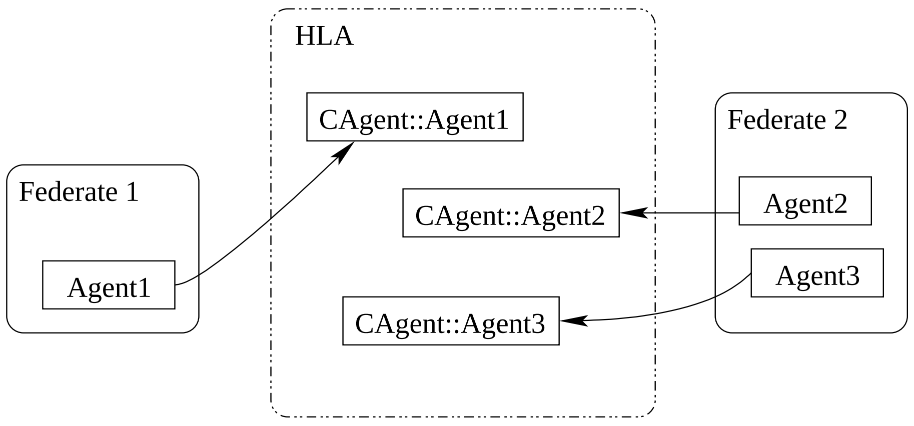


Figure 5.1: The agents in a KQML architecture is to be mapped onto federates.

5.2 Implementing KQML on HLA

Implementing KQML on HLA means accomplishing the functions of the architectural elements defined in the precious section on HLA. This section discusses how these architectural entities might be mapped onto HLA forming a multiagent simulation architecture in which agents communicate using KQML.

5.2.1 Agents

The first architectural entity to be mapped onto HLA is the concept of agents. From the KQML perspective, agents are computational entities carrying unique names and causing some KQML messages be sent, and this description makes it pretty obvious that the agents are to be mapped onto federates as shown in Figure 5.1. The agents should be able to find out about at least the presence of other agents in the runtime environment, thus we need to design a way so that agents discover other agents. This can be accomplished by making agent executing federates publish some information about their agents in the form of instances of an HLA object class named *Agent*, and subscribe to this class as well so that the RTI will inform the federate of new agents introduced into the environment, or of deleted ones (see Figure 5.2). Discovering in this sense does not mean that federates can discover abilities of agents executed by remote federates. To do so would require sending of perhaps several KQML messages for querying agent abilities.

The RTI assigns names to the created objects, names which are unique in the scope of a federation. Using these names as the agent symbolic names relieves the designer

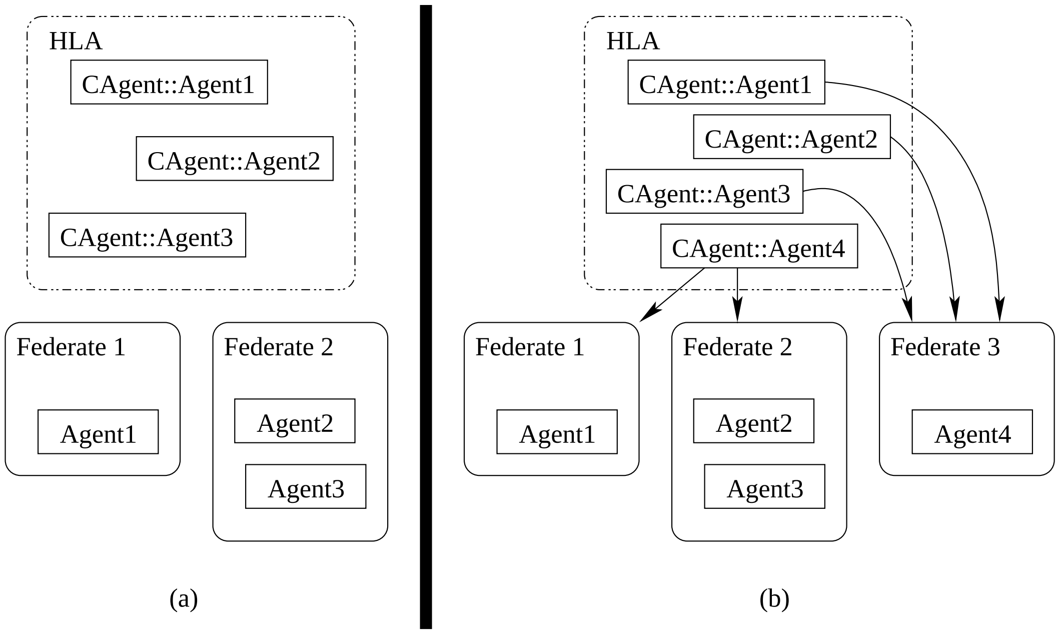


Figure 5.2: Agent discovery on HLA. In part a, the initial situation is shown. I part b, a new agent 4, simulated by the federate 3, is introduced into the system.

of solving the problem of ensuring unique agent names to be used for referrals in the messages transported.

5.2.2 Messages

The other architectural entity to be mapped is a KQML message. A KQML message is nothing but a piece of transient data to be transferred between the agents. Since the agents are to be executed by federates, the KQML messages are to be perceived as data passed between the federates, although semantically this data is sent from an agent to an agent. It is assumed for the purposes of this thesis is that the exchanged KQML data is a somewhat encoded form of the KQML message, and that all the federates know how to decode and parse it.

5.2.3 Satisfying the Transport Assumptions

In this section, problems caused by satisfaction of KQML's transport assumptions, and a list of solutions is discussed. The section starts with the more easily satisfiable assumptions, and continues with the description, source and solutions of major problems posed.

The first assumption listed in Section 5.1 is about the ability of transferring data between agents. In the discussion of mapping the agents onto HLA constructs, I proposed mapping agents onto federates. This architecture takes advantage of HLA's data sharing mechanisms for trivially satisfying the first assumption.

The answer how the fourth assumption, which is about reliable delivery of KQML messages, might be satisfied lies in a setting of correct parameters of the HLA RTI for a federation execution. In the declaration of HLA class in the Federate Object Model (FOM, see Section 3.3.2), it is also possible to set the preferred method of actual data transmission as either reliable, in which case the HLA uses reliable transport protocols at the network layer such as the TCP, or unreliable, in which case the unreliable but fast network protocols is used. Therefore, the fourth assumption is a matter of correctly setting these parameters.

The two major problems that arise from the transport assumptions are the point-to-point links and send-order reception problems, which are discussed in their own sections below.

5.2.3.1 Point-to-Point Links Problem

Providing point-to-point links between the agents in the described multiagent architecture defined over the HLA is not easy to establish due to HLA's policy of preventing federates from being aware of each other's existence directly. The KQML messages to be transferred between federates creates a need of a federate being able to refer to the other federates in the federation, which is something not provided by services of HLA. This implies that the links we talk about are simulated links between the agent HLA objects, wherever their execution may be taking place (see Figure 5.3. It turns out that one may use Data Distribution Management (DDM) quite effectively for the purpose of limiting the distribution of the messages transported over the HLA to target only the relevant federates. The use of DDM is postponed to the end of this subsection.

Having described the problem, we may have a look at the solutions proposed. The first way of transporting KQML messages between the federates that execute the communicating parties in a way that simulates point-to-point links between the agents as demanded by the transport assumptions of KQML is using the interactions described in Section 3.2.2. In this method, an interaction class for the purpose is declared in the

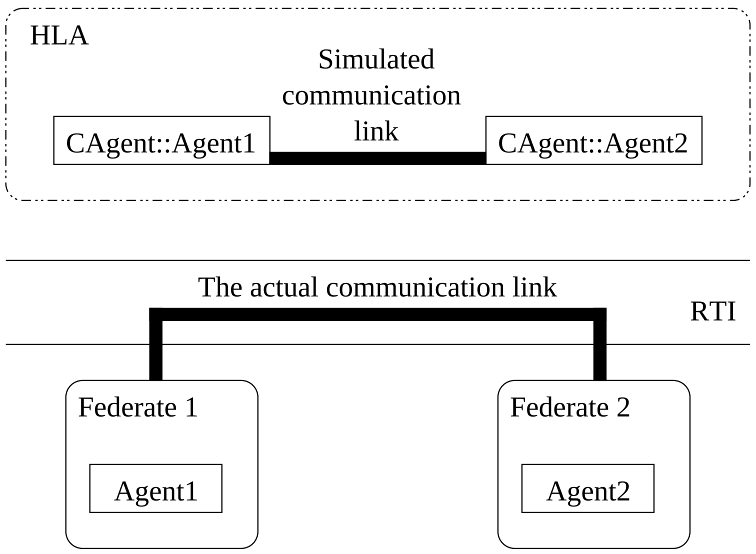


Figure 5.3: The simulated links between the agents are really links between the federates simulating them.

FOM of the federation, say an interaction class named *KQMLMessage*, and all federates publish and subscribe to this interaction class. Message sending is established by the federate which wants to send a message on behalf of an agent it simulates creating a new instance of the KQMLMessage interaction class, loading it with an encoded form of the KQML message to be sent and sending this interaction using the HLA services. All federates, having subscribed to the KQMLMessage interaction class, receive this interaction created, and decode the KQML message loaded onto the interaction to extract the target agent id to see if that is the id of one of the agents it simulates. If so, the message is elaborated further, and if not, it ends up in the trash bin along with some wasted computation time (see Figure 5.4). This waste of time can be prevented, or somehow delegated to the RTI, by the use of DDM as will be described after a second proposal for a solution to the point-to-point links problem.

The second way of transporting KQML messages between agent simulating federates is using the object attribute update mechanisms of HLA. There are actually two ways of implementing such logical point-to-point links between the agents using the HLA objects: the pull and the push communication schemes. In both of these schemes, an attribute, which is actually a transfer channel, of the class used by the federates for publishing the information about the agents they simulate, is reserved for transferring the KQML. What differs is that either the agent is assumed to uttered

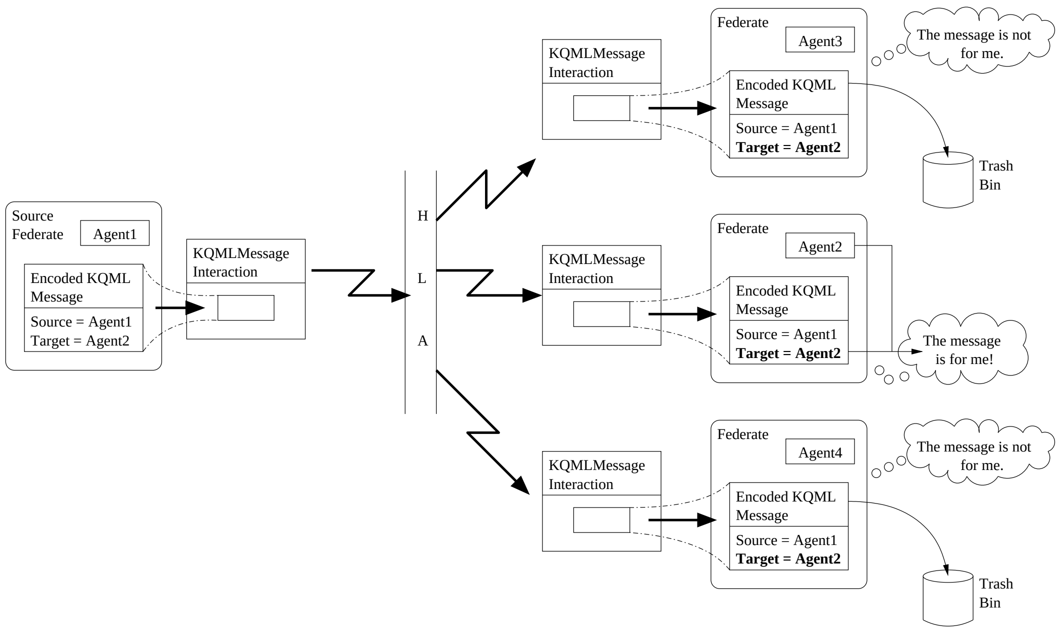


Figure 5.4: Using interactions to solve point-to-point links problem.

the message published through this attribute of the object instance that acts as its representation in the HLA, or is assumed to hear the message that appears as an update of that attribute.

The pull communication scheme is relatively more simple than the push scheme. In this schema, the communication attribute of the agent HLA object class, named say *MessageUttered* for the purpose of discussion of this schema, is used for publishing a message of the source agent. To obtain any messages that is targeted at any agents they simulate, all agent simulating federates in the federation subscribe to this attribute. Whenever a federate wants to send a message, it encodes the message and updates the MessageUttered attribute of the agent instance that it sends the message on behalf of, and having subscribed, all other federates receive this attribute update and decode the message. If the target agent id in the decoded KQML message is among the agents the federate simulates, than the message is further processed, otherwise it is discarded, again with a loss of computational time (see Figure 5.5.

In the push scheme, the communication attribute defined in the HLA class representing a simulated agent, appropriate to name this time as the *MessageReceived* attribute, is used by the simulating federate for reception of the KQML messages targeted at an agent it simulates. This method also requires transfer of HLA object instance attribute ownership from federate to federate. All federates in the federation

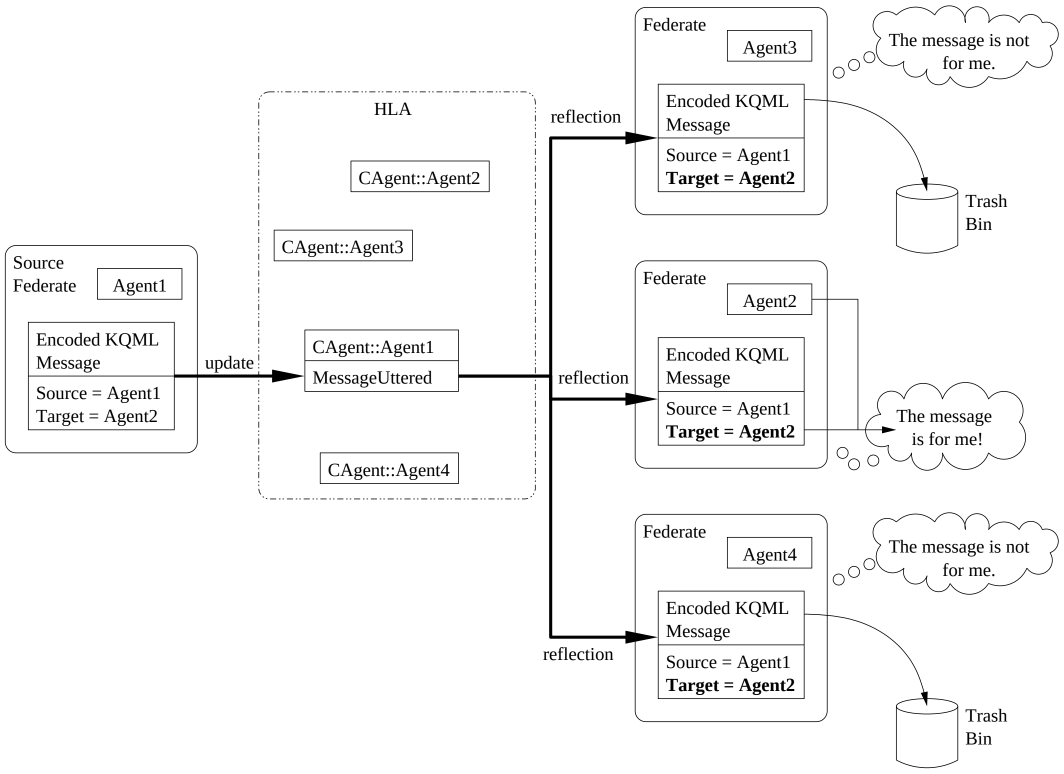


Figure 5.5: Using HLA objects in a pull scheme to solve point-to-point links problem.

only subscribe to the MessageReceived class attribute, thereby leaving the attribute unowned for any future attribute ownership acquisitions. This subscription also enables a federate to obtain any updates to the attribute of the agent HLA objects, including updates for any agents it simulates as well as all others. This inefficiency con be prevented using DDM as described below. In the push scheme, when a message is to be sent on behalf of an agent, the originating federate first acquires the ownership of the MessageReceived attribute of the HLA object instance representing the target agent of the message to be transferred. When acquisition is completed, the message is sent by updating the value of the attribute, thereby causing all federates to receive the message and decode to see if it is for an agent they simulate (see Figure 5.6).

As it's most easily seen in the last scheme, these methods of transporting the KQML message are not very efficient due to the fact that all federates receive all messages, whether it is relevant for them or not. This is surely not desirable, instead we conceive of point-to-point links as being between the corresponding parties, and not causing any load for others that may be present in the communication network. As said before, this inefficiency is due to HLA's design. The solution proposed by the standard

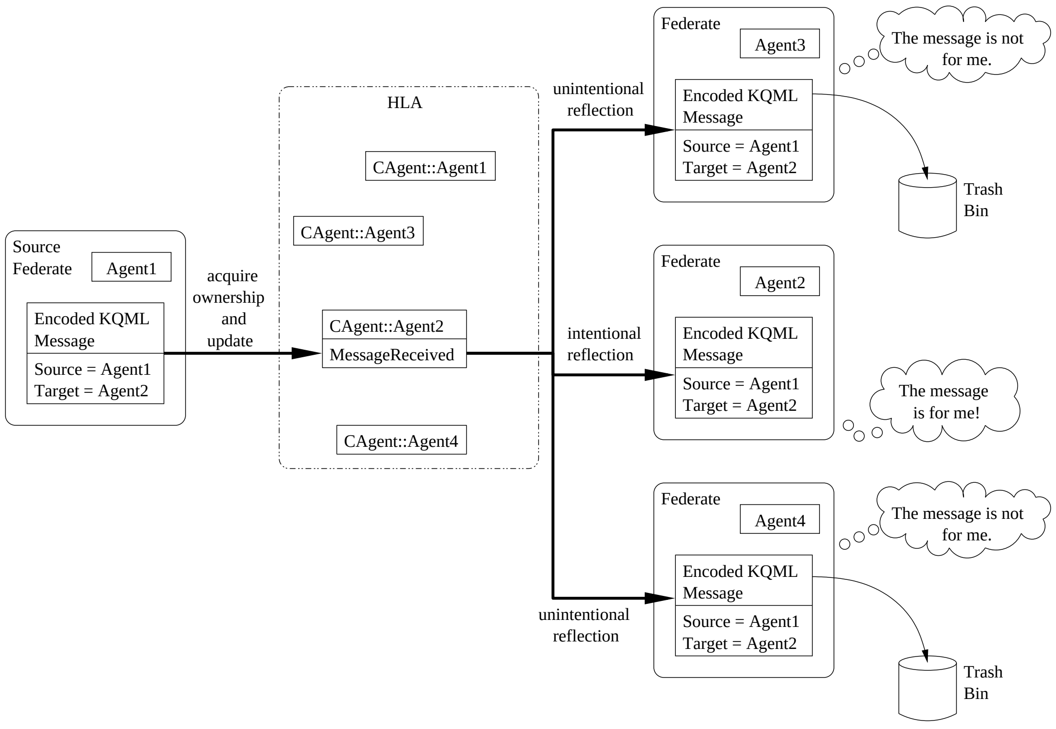


Figure 5.6: Using HLA objects in a push scheme to solve point-to-point links problem. HLA does not permit object instance attribute level subscriptions, what a federate can do is to subscribe to the class attribute, receiving the attribute updates of every object instance of the class created. In push scheme, all federates subscribe to the MessageReceived attribute of the agent HLA class, named in the figure as CAgent. This class level subscription causes reception of unintentional attribute updates.

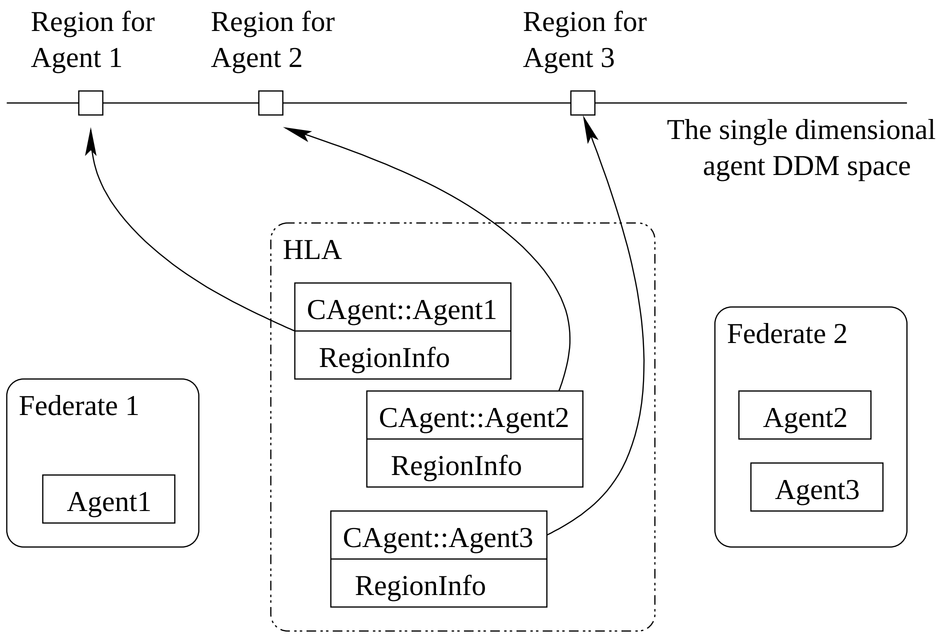


Figure 5.7: The agents DDM space and sample agents given different regions.

is use of DDM services, and this solution is applied to the methods described above. To use DDM, a routing space is created and every agent in the simulation is given a different region in this space as shown in Figure 5.7. All the federates simulating agents publish, for every agent they simulate, the region of the agent through an attribute of an HLA object instance of agents class. Since all federates also subscribe to this class to get informed about the agents in the simulation, all federates can access the regions of any agent.

In the case of employing DDM when interactions are used for delivery, all federates subscribe to the interaction class KQMLMessage with a region that covers the union of all regions of the agents it is simulating. When a message is to be sent, the sending federate sends the interaction to the target agent's region, and DDM routes the interaction to the only relevant federate (see Figure 5.8).

When the pull scheme is used with DDM, a federate subscribes to the communication attribute with a region that is the union of all regions of the agents it simulates, as in the interaction case. Whenever a federate, on behalf of an agent, wants to send a message, it first makes the source agent's MessageUttered attribute visible to the federate simulating the destination agent by changing the publication region of the MessageUttered attribute to cover the destination agent's region. Then the source

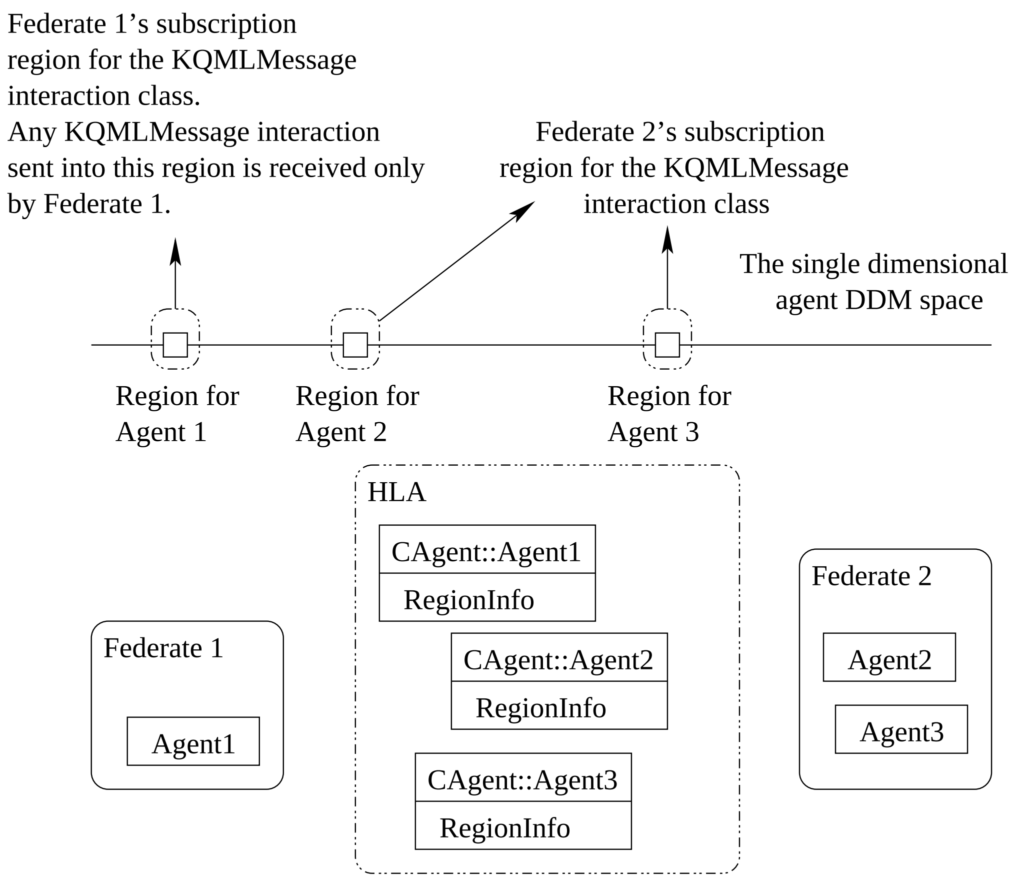

Federate 1's subscription region for the KQMLMessage interaction class.
Any KQMLMessage interaction sent into this region is received only by Federate 1.
Federate 2's subscription region for the KQMLMessage interaction class
The single dimensional agent DDM space
Region for Agent 1
Region for Agent 2
Region for Agent 3
HLA
CAgent::Agent1
RegionInfo
CAgent::Agent2
RegionInfo
CAgent::Agent3
RegionInfo
Federate 1
Agent1
Federate 2
Agent2
Agent3

Figure 5.8: Use of DDM with interactions for sending KQML messages.

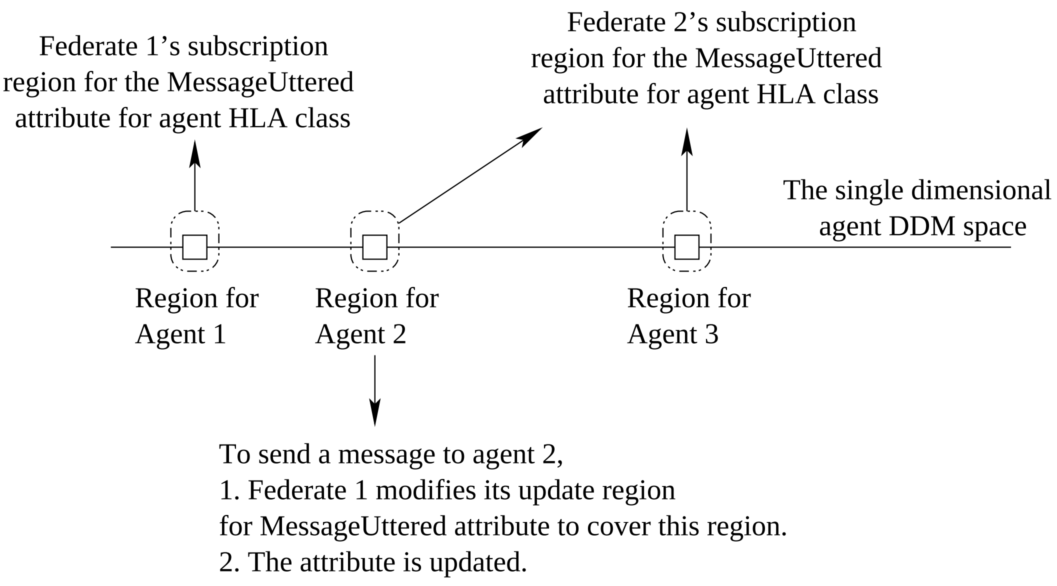

Federate 1's subscription region for the MessageUttered attribute for agent HLA class
Federate 2's subscription region for the MessageUttered attribute for agent HLA class
The single dimensional agent DDM space
Region for Agent 1
Region for Agent 2
Region for Agent 3
To send a message to agent 2,
1. Federate 1 modifies its update region for MessageUttered attribute to cover this region.
2. The attribute is updated.

Figure 5.9: Use of DDM with objects for sending KQML messages.

agent's object instance's attribute is updated and the message gets transferred (see Figure 5.9). The push scheme is similar with the only extra step of acquiring the attribute ownership.

5.2.3.2 Send-Order Reception Problem

The third transport assumption of KQML calls for send-order reception of messages. This mechanism has no immediate equivalent in the HLA architecture: The HLA architecture supports receive-order or time-stamp-order delivery of events to the federates. Two solutions are devised, both using simulation time to make messages arrive in send-order, one defining the flow of time in terms of the messages created by all agents (or federates we may say) in a federation, and the other relying on the physical nature of message generation.

The first solution defines the flow of simulation time in terms of messages generated throughout the federation. All federates work in optimistic time management mode, thereby observing all the events in their simulated future by issuing FLUSH QUEUE REQUEST service of RTI. When a federate generates a message, it increments its simulation time by a fixed amount. When it receives messages, it tries to increment its time to the time of the latest message it received. This solution creates problems with DDM though, since it depends on all federates receiving all events in the future

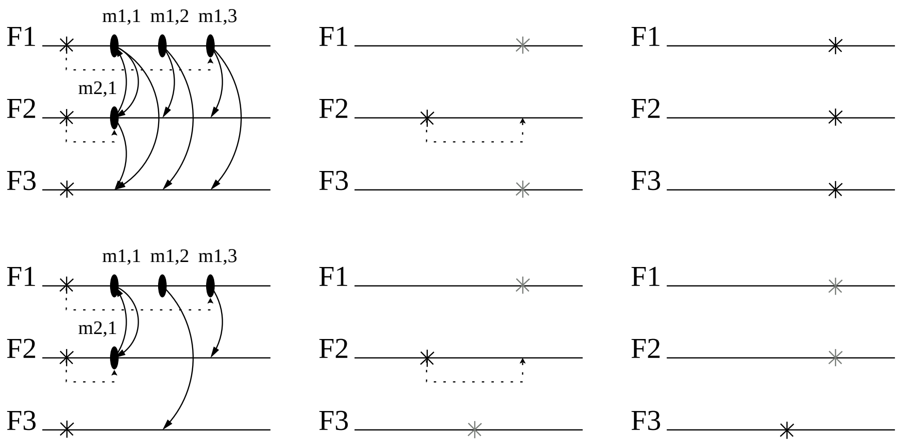


Figure 5.10: The upper three simulation timeline diagrams shows how the federates advance their time when FLUSH QUEUE REQUEST service is used without DDM. Messages $m1, 1$ and $m1, 3$ are from federate $F1$ to federate $F2$, $m1, 2$ is from $F1$ to $F3$ and $m2, 1$ is from $F2$ to $F1$. $F1$ and $F2$ attempts advancing their time to the latest time stamp on their own messages, and $F3$ attempts to advance its time to the latest message it receives. When $F2$ is granted its time, it detects that there are other messages created by $F1$, and re-increments its time. This way all three federates end up at the same simulation time. The lower three diagrams show the case when DDM is used. DDM effectively prevents federate $F3$ gaining the necessary information about $m1, 3$, therefore federates $F1$ and $F2$ get stuck in time advancing state.

to update their time, and DDM effectively prevents this by providing true point-to-point links (Fig. 5.10 shows diagrammatically). It is possible to use a separate event for the coordination purpose, and using such a small event may be justified taking the size of messages to be delivered into consideration.

The other solution is to assume that in a simulation, time has a physical meaning and using time as merely a mechanism for synchronization would hinder the federation's reuse and interoperability. In such a view of time, creating a message is a physical action that takes some amount of time to accomplish. This implies that either all the messages created by an agent have different time stamps, or the number of messages created with the same time stamp is limited, and that not receiving the messages until an agent's local time, which may considerably differ from agent to agent in a distributed simulation, passes the message creation simulated time.

CHAPTER 6

CONCLUSION

In this thesis, an abstraction layer for better engineering of HLA based simulations is described. This is followed by a discussion of problems of implementing KQML speaking agent societies on HLA for the purpose of realizing multiagent simulation models.

The Federate Abstraction Layer acts as an intermediary between the local RTI component and the user code implementing the simulation model and doing useful computational work. There exists more than one RTI implementation vendors, thus the manager of a simulation project which uses HLA as the transport medium may be forced to change the RTI implementation in the lifetime of the simulation produced. As the intermediary layer, the FAL abstracts away the user code of the federate and confines the HLA related code to a single object, decreasing the coupling between the user code and HLA, yet providing the benefits HLA provides. To change the RTI implementation, only a small portion of FAL library is to be recoded, and the rest of the federate remains the same.

Furthermore, the HLA's concept of an object is quite different than the object concept in object-oriented sense. This difference increases learning time of the HLA's simulation architecture for C++ programmers. The FAL library automates the reception and publication of data through HLA data entities, and provides the C++ programmer C++ object instance representations of the HLA data entities. The conversion of two object concepts is done by the FAL. This eases the job of the designer and the programmer of a federate, by freeing of designing the internal data flow between the simulation code and the HLA, and by freeing of the task of converting and routing information in the code, respectively.

The HLA standard is designed with care and is neat, but the target range of simulation types is quite wide. Since the standard is to support a variety of simulation flavors, there are many services provided by HLA. Not all federates use all types of services. The FAL is designed to satisfy the needs of the project SAVMOS in the Modelling and Simulation Laboratory, and the needs of the described study on realizing the KQML for multiagent simulation architectures on HLA.

FAL was described in Chapter 4.

In Chapter 5, the results of my study on identification of methods for implementing multiagent simulations on HLA in which agents use the agent communication language KQML in order to carry out interactions. To accomplish this use of KQML on HLA, the architectural elements of KQML and their possible mappings onto the HLA constructs are identified.

The solutions proposed for solving the point-to-point links and send-order reception problems have been demonstrated by implementing ten federates. Using these ten federates, the combinations of the six proposed solutions to the point-to-point links problem and two proposed solutions to the send-order reception problem are tested. It was observed that the use of DDM with optimistic time management mode created problems, so two of the twelve combinations are not implemented any further. As an emprical statement, it may be said that the most appropriate method to use in simulations which use time with a physical meaning is the use of interactions with DDM to solve the point-to-point links problem, and the physical meaning of time to solve the send-order reception problem. For implementing multiagent systems in which time has no physical meaning, the optimistic time management scheme provides immediate message transfer efficiently.

The results of this study on realizing KQML on HLA has been disseminated through a conference paper to appear in LNCS [11].

Both of these studies started as part of the SAVMOS project in Modeling and Simulation Laboratory. Although use of KQML is virtually eliminated from the project in the course of evolution of requirements for the project, the previous revision of FAL has successfully been applied in the software development process. The new revision of FAL, described in this thesis is expected to be adopted in the very near future.

6.1 Future Research Directions

It's possible to define some follow up studies of work described in this thesis.

The work that would be done on the FAL is mostly technical work. The FAL currently lacks an internal error reporting mechanism. This does not prevent it from accomplishing its necessary functions, but a good error recovery and reporting mechanism eases the job of program developers. Otherwise FAL is fairly complete and in working condition.

The work on FAL helped identifying some shortcomings of the DMSO RTI 1.3NG, which was the only free RTI until December 2002, afterwise which also have been commercialized. Unfortunately, these shortcomings cannot be attacked since the distribution of all commercial RTI implementations are in binary form. It is desirable to build an open-source RTI implementation for RTI is a very good software to use as a testbed for distributed systems research. An open-source RTI would increase the research on distributed methods that in the long term will cause a better RTI, thereby increasing trust in the standard and the thrust behind building HLA compliant simulation systems.

KQML is not the only agent communication language. FIPA-ACL is even more popular today, both in academic and commercial worlds. The results of the study described in this thesis may be used as a foundation for implementing the FIPA multiagent system architecture on top of the HLA standard. This would surely enable us to harvest the best of both worlds for the purpose of developing multiagent simulation systems. Implementation of the FIPA architecture is expected to lead a master thesis on its own.

Besides the agent communication languages, it's possible to make studies for enabling more effective use of other agent technologies in HLA based simulations. Balancing the computational work in the simulation using mobile task-exchanging agents is one possible subject. It is important to be able to describe the simulation in computational terms interpretable, and doing so for a range of simulations means some work is needed to be carried out on interpretable/simulatable domain ontologies.

REFERENCES

[1] Department of Defense, Defense Modeling and Simulation Office. *RTI 1.3-Next Generation Programmers Guide, Version 5*, 2002.

[2] Jacques Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence.* Addison-Wesley, 1999.

[3] Tim Finin, Jay Weber, Gio Wiederhold, Michael Genesereth, Richard Fritzson, James McGuire, Richard Pelavin, Stuart Shapiro, and Chris Beck. Draft specification of the KQML agent-communication language plus example agent policies and architectures. ARPA Knowledge Sharing Initiative, External Interfaces Working Group working paper, June 1993. Available in web site http://www.cs.umbc.edu/kqml/.

[4] Time Finin, Richard Fritzson, Don McKay, and Robin McEntire. Kqml as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management.* ACM Press, 1994.

[5] FIPA. FIPA abstract architecture specification. Foundation for Intelligent Physical Agents standard, December 2002.

[6] FIPA. FIPA ACL message structure specification. Foundation for Intelligent Physical Agents standard, December 2002.

[7] FIPA. FIPA communicative act library specification. Foundation for Intelligent Physical Agents standard, December 2002.

[8] Foundation for Intelligent Physical Agents (FIPA). Official web site. http://www.fipa.org/.

[9] Richard M. Fujimoto. Time management in the high level architecture. *Simulation*, 71:388–400, 1998.

[10] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wiley & Sons Inc., NY, USA, 2000.

[11] Erek Göktürk and Faruk Polat. Implementing agent communication for a multi-agent simulation infrastructure on hla. In *Proceedings of the* 18th *International Symposium on Computer and Information Sciences, LNCS, to appear.* Springer-Verlag, 2003.

[12] Paul Kogut. KQML lite specification, revision 0.3. Technical Report ALP-TR/03, Lockheed Martin Mission Systems, September 1998.

[13] Frederick Kuhl, Richard Weatherly, and Judith Dahnmann. *Creating Computer Simulation Systems*. Prentice Hall PTR, Upper Saddle River, NJ, US, 1999.

[14] Yannis Labrou. *Semantics for an Agent Communication Language*. PhD thesis, University of Maryland, August 1996.

[15] Yannis Labrou and Tim Finin. A semantics approach for kqml-a general purpose communication language for software agents. In *Proceedings of the third international conference on Information and knowledge management*, pages 447–455. ACM Press, 1994.

[16] Yannis Labrou and Tim Finin. A proposal for a new kqml specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, UMBC, 1997.

[17] Yannis Labrou and Tim Finin. Semantics and conversations for an agent communication language. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 235–242. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[18] Pierre Marcenac and Sylvain Giroux. Geamas: A generic architecture for agent-oriented simulations of complex processes. *Applied Intelligence*, 8:247–267, 1998.

[19] Defense Modeling and Simulation Office, U.S. Department of Defense. Department of defense modeling and simulation master plan, October 1995. See also http://www.dmso.mil/dmso/docslib/mspolicy/msmp/.

[20] NATO. Modeling and simulation master plan, 1998. Available in digital form from
ftp://ftp.rta.nato.int/Documents/MSG/NMSMasterPlan/NMSMasterPlan.pdf.

[21] U.S. Department of Defense, Under Secretary of Defense for Acquisition and Technology, USD (A&T), memorandum. Dod high level architecture (hla) for simulations, September 1996. See also http://www.dmso.mil/hla/policy/.

[22] California State University, McLeod Institute of Simulation Sciences. Hla lecture notes, April 1999. Available in http://www.ecst.csuchico.edu/ hla/courses.html.

[23] R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The darpa knowledge sharing effort: Progress report. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, November 1992.

[24] John R. Searle. *Speech Acts*. Cambridge University Press, Cambridge, UK, 1969.

[25] Marry Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.