ACKNOWLEDGEMENTS

I would like to acknowledge the support from my project supervisor, **Dr. Tan Soon Huat, Gary**. His valuable comments and suggestions have helped me much during the process of my Master study.

I wish to thank **Dr. Côme Raczy**. He always brings me the problem, asks me the solutions and clarifies my thinking. I really benefited from his academic advices.

Thanks also to Yu Jun and Li Yong Bo who provided much assistance in my research. They gave me much valuable discussion.

Last but not least, I also thank my dear wife, Yang Jie, for her understanding and moral support when I wrote the thesis.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programmer's Interface
CASE	Computer Aided Software Engineering
CRC	Class-Responsibility-Collaborator
DMSO	Defense Modeling and Simulation Office
FOM	Federation Object Model
GEF	Graph Editing Framework
HLA	High Level Architecture
M&S	Modeling and Simulation
NSUML	Novosoft UML API
OMG	Object Management Group
OMT	Object Model Template
OO	Object Oriented
OOM	Object Oriented Methodology
RTI	Runtime Infrastructure
SOM	Simulation Object Model
UML	Unified Modeling Language

SUMMARY

The High Level Architecture (HLA) defines a set of standards and principles for distributed simulations and promotes the reuse of simulation software and interoperability between them.

However, it is a laborious task when directly developing HLA simulations with the low level service methods. Moreover, the HLA does not address formal design approach for the simulation software. These difficulties hinder the HLA simulations design and development.

To improve the productivity of the simulation development and enhance software reuse, this project proposes a design framework based on the HLA simulation development process. It is an architectural point for the users to implement the simulation software. Moreover, a UML-based tool is implemented to support designing HLA simulations. It includes *pre-defined class diagram*, *reflection-in-action context help*, and *automatic code generation*. Concepts and notations for HLA extensions have been presented and their implementations are discussed.

Two example simulations, HelloWorld and Tank, are introduced to describe the basic process of the design and development of HLA simulations under the proposed framework with tool support.

Chapter 1. Introduction

1.1. Overview of software design

1.1.1. Object-Oriented Methodology (OOM)

Object Oriented (OO) is the technology based on objects and classes [BAUD96]. An object is a representation of a real-life entity that incorporates both data structure and behavior. A class is the abstraction of the real world based on the objects. OO design is concerned with developing an OO model of a software system to implement the identified requirements [HEND92].

Object Oriented Methodology (OOM) is a system development approach that encourages and facilitates the reuse of software components [G52A03]. With this methodology, a system can be developed on a component basis which enables the effective reuse of existing components and facilitates the sharing of its components by other systems. There are several types OOM given in [COAD90] [MART96]. In this project, one of OOM: Class-Responsibility-Collaborator (CRC) card modeling approach is introduced to describe the classes of application software.

1.1.2. Framework

The OO framework is a promising technique to support reusable software components. The description of software framework is given [GAMM95]:

> "*A framework is a set of cooperating classes that make up a reusable design for a specific class of software*... *A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and*

collaborations...*A developer customizes a framework to a particular application by adding new components to plug into it*"

From the description above, the framework is a reusable design expressed as object-oriented class hierarchy. It defines the responsibilities of the classes, the interaction of the objects derived from the class hierarchy and the thread of control.

The framework describes the architecture of the applications. All the classes in the framework provide the skeleton of applications. By subclassing and composting instances of classes to customize the framework, developers can construct and develop the particular applications.

The framework emphasizes reusable design over code reuse [FAYA99]. It can be embodied in OO programming languages, executed and reused.

1.1.3. UML and CASE tools

The Unified Modeling Language (UML) is a model language for OO design and development. It helps users specify, visualize, and document models of software systems. It facilities the development process of the framework [OMG003] [DSOU99] [ODELl98]. The development process of framework needs automated tools in some way. The Computer Aided Software Engineering (CASE) tools are the automation of step-by-step tools for software development to reduce the amount of repetitive work the developers need to do [DANE96] [HERZ94] [SINA02]. They provide the automated engineering discipline for software development, maintenance, and project management. It plays an assistant role in the design phases of system. CASE tools are introduced to enforce a standard development methodology.

In the OO software development process, UML and CASE tools can help users design and implement more flexible and reusable systems. This project focused towards the framework technique and UML-based CASE tool for modeling and simulation domain.

1.2. Overview of modeling and simulation

As the size and complexity of real systems are increasing, modeling and simulation techniques have been widely used to analyze their behaviors and communication. The modeling of a relevant real system or subsystem is the first step in the OO software design. A model is an abstract representation of a real system or subsystem. A simulation is the program execution of a model to give information about the system [BOEH96].

Modeling and Simulation (M&S) is the discipline of designing a model, and executing the model [ZEIG00]. Distributed simulations refer to the technology concerned with executing simulations over computing systems containing a collection of loosely coupled distributed processors [DOD994].

There are two desirable properties for a simulation: reuse and interoperability [SISO03].

Simulation reuse means simulations, which are constructed for the purpose application, can support the different applications and no need of re-coding for reuse. Simulation interoperability implies that the simulations on the distributed computing platform can provide service to other simulations and accepts services from them. Simulations use the exchanged services to enable them to operate effectively together.

1.3. Overview of HLA

Based on the premise that no one simulation can satisfy all applications and users, the High Level Architecture (HLA), which was initially introduced by Defense Modeling & Simulation Office (DMSO) of U.S, defines a set of standards for the distributed simulations and promote the reuse of simulations and interoperability between the simulations [DMSO97].

The HLA has a variety of benefits for distributed simulation. It is possible to construct a larger and more complex simulation using existing simulations.

Figure 1-1 shows a logical view of the main components of the HLA infrastructure: simulation, Runtime Infrastructure (RTI) and runtime interface.

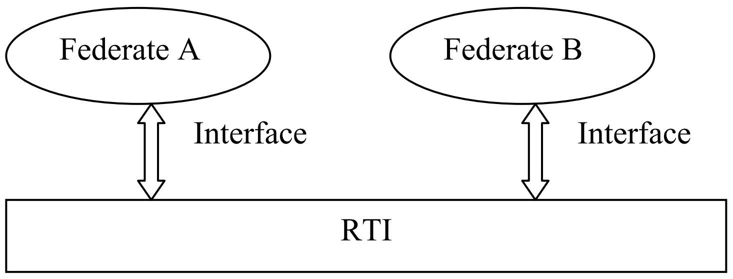


Figure 1-1: The logical view of HLA components

In the context of the HLA, a simulation is also generally referred to as a federate. A federation is the particular group of interoperating federates. The federate interacts and exchange data with other federates supported by the services in the Runtime Infrastructure (RTI). RTI is a software implementation of communication service to support federates interaction and federation management. Runtime interface specification, which contains a set of Application Programmer's Interface (API), provides a way for communication between federate and RTI. A federation is the particular group of interoperating federates. All federates and the RTI are connected through a distributed network and together are made up of a federation execution.

1.4. Problem statement

The HLA provides a set of standards and principles for distributed simulations. But it does not address any detail of common approach to facilitate federate design and development. For the HLA programming beginners, the challenges are to effectively design and develop HLA federates. There are some general problems that relate to the federate development based on experiences:

1) Laborious coding with HLA

To date, most federates are manually programmed in popular object oriented languages, such as C++, Java, through RTI. RTI provides a number of flexible functional interfaces (APIs) and the interface specification consists of over a hundred methods names and descriptions. For example, even for the very simple federate software, it includes thousands of lines of source code and most are just the HLA RTI basic services. But the number of lines of the user defined data type and relevant operations are not more than one hundred. Without tools support or pre-defined template, federate developers face the relative steep learning curve and boring work in the HLA programming.

2) No formal design approach

HLA does not address formal federate design approach. It also does not address any reuse guideline between the different design paradigms. Developers may create their federate software in their own favorite manner [TOLK02] [RADE02] [COX998]. The different design paradigms results in implementation incompatibilities between the federates. It leads to ad hoc fashion. It is difficult to reuse the existing federates. Thereby, this conflicts with the HLA goals: reuse and interoperability.

1.5. Project objectives

To solve the above problems, there is a great need of techniques and tools to improve the productivity of the creation of the HLA simulation and enhance software reuse. The project objectives include:

1) To provide a more formalized description of HLA simulations' design and development process.
2) To offer HLA developers the reusable design and source code of HLA simulations.
3) To decrease the HLA programming complexities by encapsulating a number of HLA low level service methods.
4) To allow HLA developers to focus on application-specific simulation fields rather than on the basic RTI services.
5) To implement some supporting functionalities in a UML-based CASE tool for federate design and development.

1.6. Project contributions

Within OO software design's concepts and principles, this project proposes a federate framework and a UML-based CASE tool to achieve the research objectives. Several contributions are made in this project:

1) Research and design a basic federate framework. The proposed framework restricts to HLA concepts and is compatible with the HLA principles. It is the general form of various kinds of HLA simulation applications and initial set-up for HLA programmers to develop a new federates and supports to reuse the existing federate for multi applications. The framework includes:

   ➢ The collaborating classes represent the HLA simulations. In these classes, the standard data structures and behaviors are well-structured defined to meet the necessary requirement of the federate execution capability.

   ➢ The general higher level simulation services, which are required by typical federates, encapsulate a set of flexible functional interfaces of HLA low level service methods (APIs).

2) Design and implement the HLA extensions' functionalities in the open source CASE tool: ArgoUML. They considerably support the proposed framework. The main functionalities can be summarized into the areas listed below:

   ➢ Extension to the standard UML profile to represent a federate;

   ➢ Some usable help during the federate design process;

   ➢ Automatic process of generating the HLA simulation in C++ source code skeleton;

1.7. Structure of thesis

The research described in this thesis involves:

Chapter 1: OO design concepts, HLA, and some problems related to the federate software development are briefly overviewed. It proposes the project objectives and contributions.

Chapter 2: CRC cards methodology, framework, UML and CASE tools are discussed. One CASE tool, ArgoUML, is introduced. Because ArgoUML is still a research project and there is insufficient documentation about it, its structure and implementation are thoroughly analyzed. It is the ground to support the federate design framework.

Chapter 3: The technique of the modeling and simulation are described. The HLA infrastructure, its components and the general federate development process are investigated. Before focusing towards the federate design framework, the federate implementation problems have been determined.

Chapter 4: It proposes a basic federate design framework. The classes in this framework and their relationship are discussed by CRC cards. The framework components' structure and functions are thoroughly discussed.

Chapter 5: Some problems of the HLA extensions to support federate design are discussed. This is the foundation of the functionalities implementation of ArgoUML.

Chapter 6: The UML-based tool: ArgoUML is implemented to support HLA federate design. The implementations of the HLA extensions are presented.

Chapter 7: Under the proposed federate design framework, two example federates are presented to describe the basic process of the design and development of a federate in ArgoUML environment.

Chapter 8: It concludes the main research work and the further works are discussed.

Chapter 2. Software design and related work

2.1. CRC cards methodology

The Class-Responsibility-Collaborator (CRC) cards approach is one of OOM. It usually includes the users, analysts, and developers in the modeling and design process, bringing together the entire development team to form a common understanding of an OO development project [BECK89]. There are a lot of materials about how to use this technique in the OO software [DOUG99] [WILK95] [BELL97].

The CRC modeling technique includes a collection of cards which are divided into three sections:

- Class: A Class represents a group of similar objects. Objects are described by their attributes and their operations. The class name appears across the top of the CRC card.
- Responsibility: The responsibilities are things that the class has knowledge about itself or what the class can do. For example, a customer class might have responsibility for its name, address, credit level, and phone number. The Responsibilities of a class appear along the left side of the CRC card.
- Collaborator: A collaborator is another class that is used to get information or perform actions for the class at hand. It often works with a particular class to complete steps in a scenario. The collaborators of a class appear along the right side of the CRC card.

Table 2-1 is an example CRC card:

Class	
Responsibility	**Collaborator**

Table 2-1: An example CRC card

CRC cards can be created for any identified class. After CRC cards are created; they are often placed on a centralized table. In this project, CRC cards are used to describe the classes of the federate framework.

2.2. Framework

A framework is a reusable software architecture made of both design and code. It represents a partial design and implementation for an application in a given problem domain. Johnson and Foote have developed the most frequently used definition in [JOHN88]:

> *"A framework is a set of classes that embodies an abstract design for solutions to a family of related problems."*

Frameworks are a promising technology for the proven software designs and implementations in order to reduce the cost and improve the quality of software [SCHM97].

The primary benefits of the framework are reuse and extensibility.

- Reuse: Frameworks achieve reuse by defining generic components that can be reapplied to create new applications. Reuse of framework components can substantially improve software productivity, as well as the quality, performance, reliability and interoperability of software.
- Extensibility: Extending the framework can provide the functionality unique to the application. Framework extensibility ensures customization of new application services and features.

A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. The framework typically consists of a mixture of abstract and concrete classes. The abstract classes usually reside in the framework, while the concrete classes reside in the application. A framework, then, is a half-complete application that contains certain fixed aspects common to all applications in the problem domain.

The framework development has been successful in many domains. For example, JUnit [JUNI03] is a test framework for Java program [MARC03]. The architecture of JUnit is shown as a UML class diagram in figure 2-1. Each rectangular box represents a class. The upper section holds its name and the lower holds its methods. Each relationship between these classes is represented by the bars that connect them. This JUnit framework contains the abstract class *Test*, class *TestCase* and class *TestSuite* and can be instantiated by the concrete class myTestCase and myTestSuite.

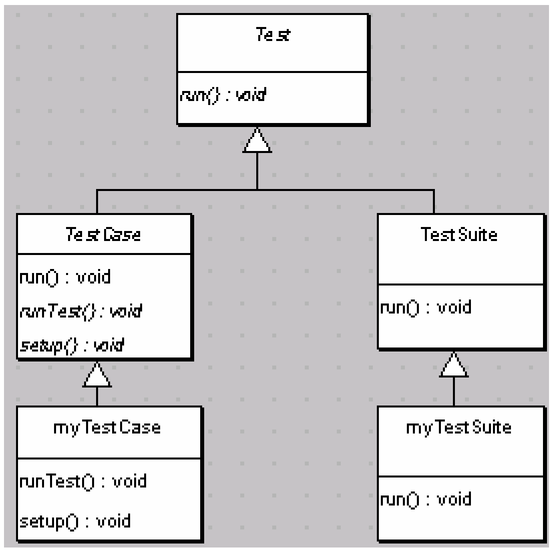


Figure 2-1: The architecture of the JUnit framework

Frameworks that are extended are classified w*hitebox* frameworks and *blackbox* frameworks.

Black-box frameworks are easier to use since the internal mechanism is hidden from the developers. Blackbox frameworks are more difficult to develop since they require the framework to anticipate a wider range of potential application scenario. Whitebox frameworks require application developers to have basic knowledge of the frameworks structure. The JUnit framework is just an example of Whitebox framework.

There are three major steps to develop a framework: domain analysis, framework design, and framework instantiation [MARK03].

The domain analysis phase discovers the domain's requirements and possible future requirements. The framework design phase defines the framework's abstractions. Finally, in the instantiation phase, the classes of the framework are implemented, generating a software system.

There are some examples of the OO frameworks in [PATE96] [WALD96] [SRID96].

2.3. UML

2.3.1. Overview of UML

The core work of OO problem solving is the construction of a model. The model abstracts the essential detail of the underlying problem from its real world. The modeling languages encourage more developers to model their software systems before starting them. The standard modeling languages improve the developers' communities. The Unified Modeling Language (UML) is a standard language for object-oriented analysis and design facilities, which was set by the Object Management Group (OMG) in 1997, and now the standard for communicating OO concepts.

UML specification defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of software systems. UML models systems using object-oriented concepts. The models consist of objects that interact with each other by exchanging message.

UML provides different views of the abstraction level in the design process. It defines different types of diagram to describe and model the real world. These diagrams include:

- Class Diagram
- Use Case Diagram
- Sequence Diagram
- Collaboration Diagram
- Statechart Diagram

- Activity Diagram
- Component Diagram
- Deployment Diagram

In the next section, the concepts of the class and its stereotype are overviewed. In the appendix A, some necessary parts of UML notation: class diagram, sequence diagram, and classes relationships are introduced to provide background understanding. More detail and guide material can be found in [OMG003].

2.3.2. Class and stereotype

The class is one of the most important items of UML. It is a collection of objects with common structure, common behavior, common relationships and common semantics. A class is the "blueprint" for objects. It wraps attributes (data) and behaviors (methods or functions) into a single distinct entity. Objects are instances of classes.

A class is represented as a rectangle with three compartments. It wraps name, attributes and behaviors into a single distinct entity. Figure 2-2 shows an example class.

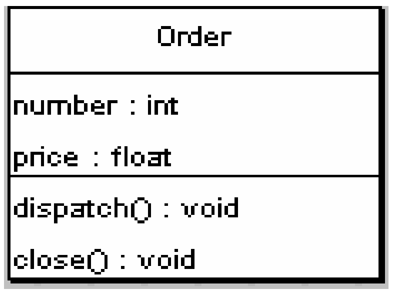


Figure 2-2: An example class

Classes should be named using the vocabulary. For example, a name, like “Order”, is a string that is used to identify a class. The structures of the classes can be represented by their attributes such as “number” and “price”. Operations such as “dispatch” and “close” are the representation of the behavior of the classes, which an object may be requested to perform. Users can assign access levels such as private, public, protected to a class, attributes, and operations.

A stereotype is a model element that defines additional values (based on tag definitions), additional constraints, and optionally a new graphical representation. Figure 2-3 shows an example class with stereotype.

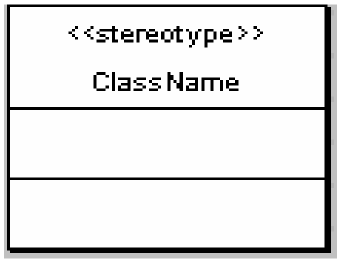


Figure 2-3: An example class with stereotype

Stereotypes are one of the extension mechanisms of UML. User defined class can be associated with specific stereotype name. For example, a new stereotype name <<stereotype>> could be defined that can be attached to classes.

2.4. CASE tools

Computer Aided Software Engineering (CASE) tools provide computer based support for the design and development of software, mostly through the provision of a diagram editor with underlying functionality for the development and analysis of the design [OMAN90] [COST94].

[IEEE96] gives a formal definition of CASE tools:

> *"CASE tool: A software tool that aids in software engineering activities, including, but not limited to requirements analysis and tracing, software design, code production, testing, document generation, quality assurance, configuration management, and project management."*

CASE tools are introduced for modeling the design and automating repetitive development tasks, such as visual design diagram representation, source code skeleton generation from designed model, and task list management. A CASE tool may provide support in only selected functional areas or in a wide variety of functional areas.

Regardless of the category and features of the tool, CASE tools users usually claim significant gains from successfully adoption of the CASE tools. These benefits include the following:

- Increased development productivity;
- Improvements in the quality of the delivered software;
- Improved consistency and uniformity of the development approach;

CASE tools provide assistance to the software developers [KEIT02] [BANK91]. There are many CASE tools for software modeling in [CASE03].

During the adoption of CASE tools phase, it involves acquiring an understanding of the needs of the project and the technology available [BOLO98]. In this project, the examples capabilities of the CASE tools are:

- Support standard UML notation like create class diagrams;
- Create text specifications, such as class specifications like attributes, operations and relationship of the model elements;
- Support the UML extension mechanism like stereotype;

- Support context sensitive help for the designer;
- Generate object-oriented language source code like C++;

2.5. ArgoUML

2.5.1. Overview of ArgoUML

ArgoUML is a research CASE tool for use in the analysis and design of object-oriented software systems [ARGO03]. It supports the standard UML diagrams like class diagram, state class diagram, use case class diagram, activity class diagram, and collaboration class diagram. In addition, it can generate Java source code skeleton from the class diagram.

Compared to many other UML modeling tools in industry like MagicDrawUML, COOL:Jex, GDPro, Visual Modeler (from Microsoft), Objecteering (with great support for repositories), Together (round trip engineering), ArgoUML has some unique key features:

- ArgoUML cognitive help system provides the knowledge supports for the object-oriented software designers and architects. It includes some specific cognitive features like context sensitive help, reflection-in-action and comprehension and problem solving. ArgoUML cognitive support system includes a number of design critics, which are active agents that continually check the design materials for errors or design areas needing improvement. A design critic is an intelligent user interface agent embedded in a design process. It independently analyzes a design in the context of decision-making and produces one piece of feedback to help the designer improve the design. The designer can easily and timely view the related feedback. The designer can only see feedback produced by the critics.

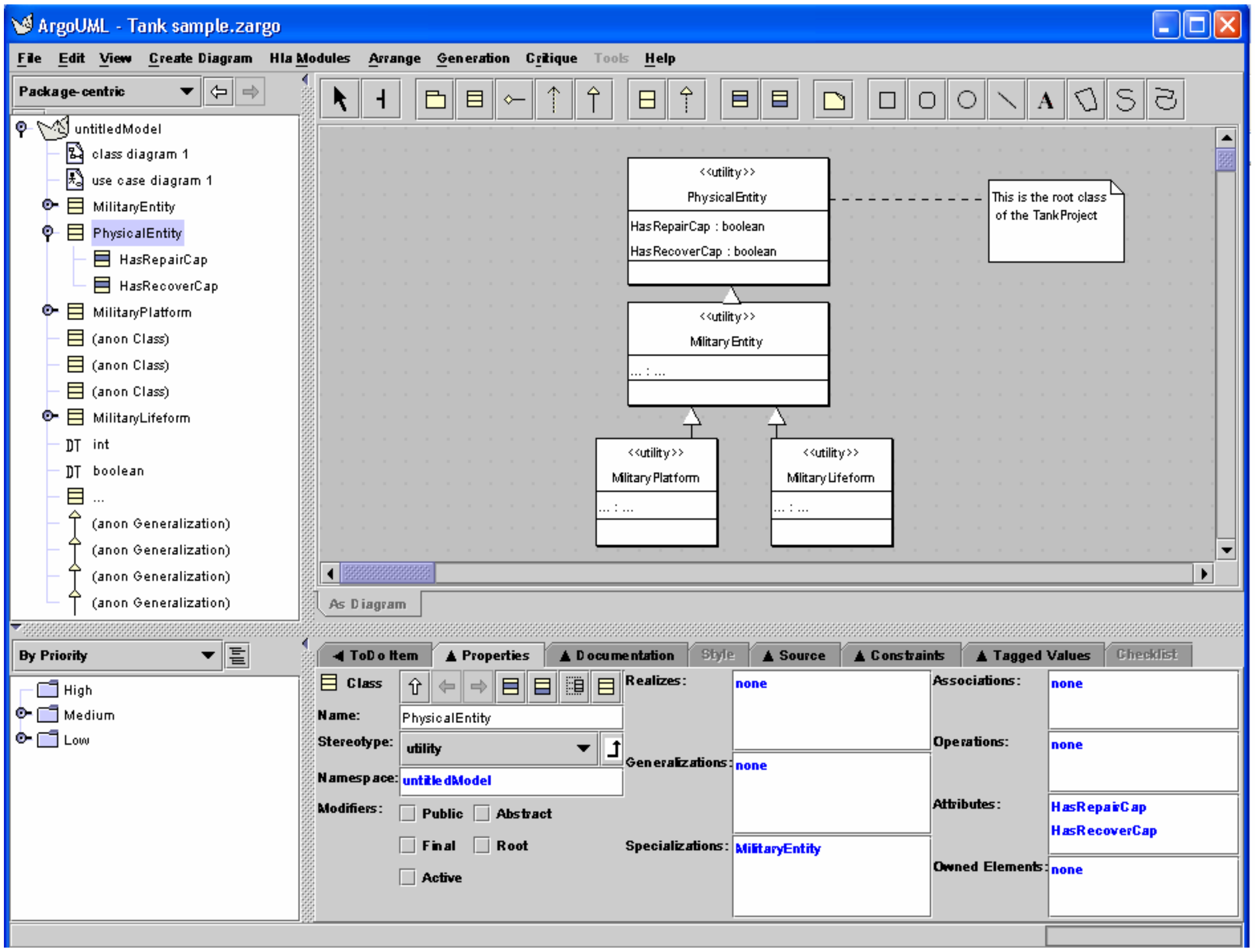
- ArgoUML is a 100% Pure Java application and open source. It does not depend on the particular platform. It promises Write Once, Run Anywhere in the Java2 platform. It allows users to design and implement their object-oriented software in the preferred Operating System environment. ArgoUML is an open source code product. Since the source code is available, it can be customized and to meet the user particular requirement.

ArgoUML makes use of some of existing open-source projects in order to achieve its goals.

- Graph Editing Framework (GEF) - a graph editing library that can be used to construct many, high quality graph editing applications. It provides a library making for the Java applications publish diagrams [GRAP03].
- Novosoft UML API (NSUML) – a representation of the UML meta-model by java classes. It records the elements of the model in a static structure copied on meta-model UML [NOVO03].

The main window of ArgoUML contains five parts: menu and tools bars, navigator pane, editing pane, details pane and To-Do pane. The To-Do Pane is located on the lower left part of the window. It includes a ToDo list of UML-specified cognitive critics for the design phase. The design critics continuously track the design process. When a potential problem is found, the critic produces a "to do" item and adds it to the To-Do Pane. When the use highlighted the critics, more detail description will appear in the ToDoItem tab of the Details pane, which is located on the lower right part of the window. When an identified problem is fixed, this ToDoItem is removed from the ToDo list. Appendix B.1 shows more details of other parts of the user interface.

Figure 2-4 shows a screenshot of ArgoUML window.



Name: PhysicalEntity
Stereotype: utility
Namespace: untitledModel
Modifiers: ☐ Public ☐ Abstract ☐ Final ☐ Root ☐ Active
Realizes: none
Generalizations: none
Specializations: MilitaryEntity
Associations: none
Operations: none
Attributes: HasRepairCap HasRecoverCap
Owned Elements: none

Figure 2-4: The screenshot of ArgoUML window

2.5.2. ArgoUML's architecture

The current version of ArgoUML includes sixteen key top-level packages, which are depicted in the package diagram in figure 2-5.

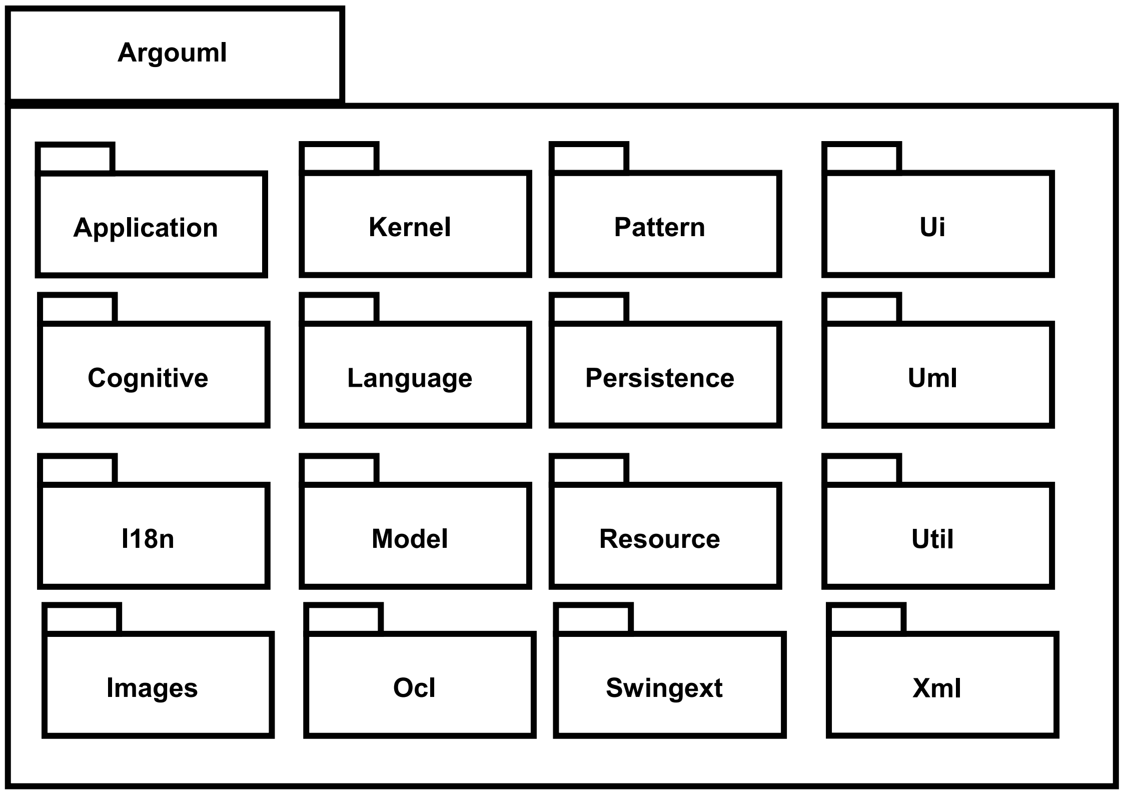


Figure 2-5: The package diagram of ArgoUML

The main components of ArgoUML include the kernel, graphic user interface, cognitive critics system and code generation. They are:

- **org.argouml.model**

This package contains various UML metamodel implementations or facades used within ArgoUML. Such metamodels include Foundation component, ModelManagement component, BehavioralElements in UseCases, StateMachines, and Collaborations component.

- **org.argouml.ui**

This package manages the principal graphic interface window, with through the singleton of ProjectBrowser class.

- **org.argouml.cognitive**

This package defines the fundamental elements of the cognitive support system, such as the Designer class, Poster class and ToDoItem class.

Appendix B.2 shows more details of other packages.

2.5.3. ArgoUML's expert critiquing system

The expert critiquing system is one of the key features of ArgoUML.

During the design process, decision making is an essential activity. Sound design in this phase is important for the success of the software system. Any errors and faults in the high level design become more expensive to overcome at later development process [GUIN87] [STAC95].

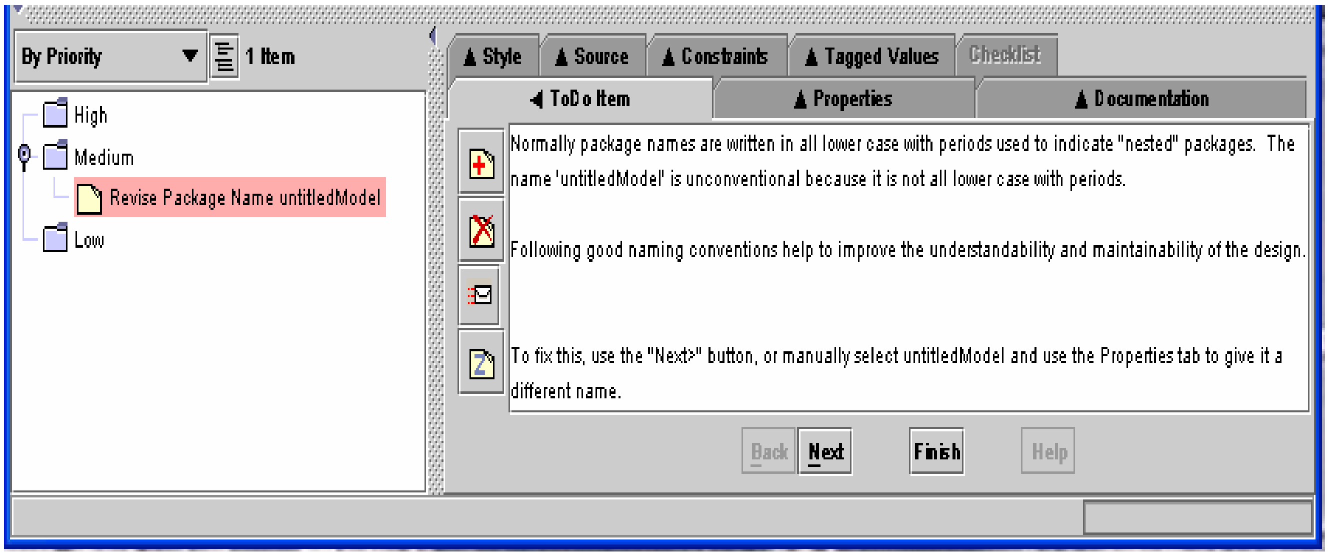
A designer may not have the comprehensive knowledge about how to build the specific process related to a particular task. In addition, the analysis technologies do not support the design decision and provide feedback timely.

ArgoUML expert critiquing system is based on the reflection-in-action theory. This approach provides the designers the results in the context of their decision making timely. Compared to the traditional analysis technologies, the users' cognitive needs are continuously considered. It supplies the design knowledge to designer when they are needed.

ArgoUML expert critiquing system is a fine-grained and real time mechanism to support design decision making. It uses critics to perform analysis on a particular design architectural model. Critics, which are also considered as context sensitive help, are always associated with the states of the user edit pane. A critic can identify the problem of the design process and produce the feedback: ToDoItem posting on the ToDo list to give the explanation of the underlying issues.

For example, when a designer selects a class and puts it in a class diagram by using ArgoUML, a critic is fired and produces one ToDoItem like: the name of the new package has not been named yet.

Figure 2-6 shows the screenshot of ArgoUML Argo issues some critics.



Normally package names are written in all lower case with periods used to indicate "nested" packages. The name 'untitledModel' is unconventional because it is not all lower case with periods.

Following good naming conventions help to improve the understandability and maintainability of the design.

To fix this, use the "Next>" button, or manually select untitledModel and use the Properties tab to give it a different name.

Figure 2-6: An example critic in ArgoUML

The ArgoUML critiquing system includes critics, criticism control mechanisms, feedback management, corrective automations, and design history. [ROBB97] [ROBB98]

- ***Critics*** are active agents that support decision-making by continuously and pessimistically analyzing partial architectures. It is embedded in the design environment to monitor the specified design problem, stylistic violation and incomplete sections. When the error is founded, it timely and automatically produces feedback that is relevant to design decisions.

ArgoUML has defined some types of critics [ROBB98]. These critics are related to the particular design problems. Table 2-2 shows more detail of the critics.

Name	Functionality
Correctness critics	Detects syntactic and semantic flaws in the partial design.
Completeness critics	Detects when a design task has been started but not yet finished.
Consistency critics	Detects contradictions within the design.
Presentation critics	Detects awkward use of the notation.
Alternative critics	Reminds the designer of alternatives to a given design decision.
Optimization critics	Suggests better values for design parameters.

Table 2-2: The category of critics

- ***Criticism control mechanisms*** determine whether the related critic is active continuously and control the execution of the critic.
- ***Feedback management*** allows the designer to control the presentation of the design ToDoItems, which is directly linked to the elements of the architecture. The ToDoItems on the ToDo list are grouped by catalog as priority, decision type, knowledge type, offending design elements, posting package. When the designer select one ToDoItem, the details of problem description and suggested solution can be viewed through the ToDoItem Tab in the Details pane.

Once a critic produces the ToDoItem, it is timely presented and posted in the ToDo list. When the critic is not valid, it is removed from the ToDo list.

- ***Corrective automations*** aid the architect to improve the design by correcting the specific problem automatically. In ArgoUML, Critic may provide a Wizard to fix specific problem. The designer can follow the wizard steps: "Next" and "Back" buttons to finish the solution.
- ***Design history*** records the earlier significant design activities. Currently, ArgoUML uses the design history only to ensure that previously resolved ToDoItems are not produced in the future.

2.5.4. ArgoUML's code generation structure

ArgoUML can generate the skeletal Java source code for class or interface based on the UML class diagrams.

ArgoUML provides the abstract class: ***Generator*** and one subclass: ***GeneratorJava*** to generate Java source code. Another subclass ***GeneratorDisplay*** generates simplified Java code to be displayed in the "Source" tab.

The source code skeleton includes the class name, member attributes and the empty body of the member operations. ArgoUML is only used in the design phase. Java IDE is needed to edit an existing the source during the development phase.

Chapter 3. Modeling, Simulation and HLA

3.1. Modeling

A model is a simplified representation of a system over some time period or spatial extent intended to promote understanding of the real system. [IEEE89] gives the formal definition of a model:

"(1) Model is an approximation, representation or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concepts, or system.

(2) To serve as a model as in (1)

(3)To develop or use a model as in (1)"

There are five basic model types [TOLK03] [FISH95]:

1) Conceptual Models: define a physical system at a very high abstraction level. All static and dynamic knowledge about the physical system must be encoded in some forms.
2) Functional Models: are usually useful to expose some system behavior to help in exceptional situations.
3) Declarative Models: concentrate on the form of the current system state and a subsequent system state after a state transition occurs.
4) Constraint Models: denote a coherent set of application independent constraints defined over all the software used in an application system.

5) Spatial Models: describe the geographical distribution of scatterers around the transmitter and receiver and predict the distribution of delay, amplitude, and direction of arrival of multi path components at the receiver.

Models provide ways to think and reason about real systems. A model is intended to promote the development of understanding of the real system. Models always involve a trade-off as to what levels of details of the real system are included in the models. If there are too little details in the model, such model may miss the relevant information and interaction of the real system. But if the model includes too many details, it may become overly complicated and actually preclude the development of understanding. A good model is constructed with suitable information to answer a specific set or class of questions about a system. There are much more resources about models in [MODE03].

3.2. Simulation

A simulation is the manipulation of a model in such a way that it operates on time and/or space to compress it, thus enabling one to perceive the interactions that would otherwise not be apparent because of their separation in time or space [RUMB91] .

[IEEE89] gives the formal definition of a simulation:

> *"(1) A model that behaves or operates like a given system when provided a set of controlled inputs. Synonymy: simulation model.*
>
> *(2)The process of developing or using a model as in (1)"*

From the definition of simulation, it is a tightly coupled and iterative three component process composed of

1) Model design: According the objective of simulation and basic assumption of characteristics of the real system, Model builders formalize details of specification of conceptual and functional model for a simulation.

2) Model execution: Model builders translate the functional model to an executable program. They can use general purpose programming language or a special simulation language.

3) Execution of a model: After the model execution, validation of a model can be done by comparing the simulation output with output generated by the real system or analytical model.

Distributed simulation systems contain a number of simulations that execute on multiple processing units in a geographically distributed system.

The distributed simulation architectures support the execution of simulation process in a distributed way by connecting different distributed simulation models of various functional areas [BRAT87]. These simulation components collaborate and communicate in order to realize the functionality of the system as a whole. In order to achieve the interoperability between the distributed simulation models architecture, the Department of Defense in the United States introduces the High Level Architecture (HLA) for modeling and simulation activities.

3.3. HLA

The HLA is a software infrastructure for heterogeneous distributed simulations. It was developed to support reuse and interoperability for simulation models. It provides a concept framework for development of distributed simulations [KUHL99].

The HLA consists of:

- Rules;
- Interface Specification;
- Object Model Template;

3.3.1. Rules

The HLA consists of 10 rules, which must be obeyed if a federate, or federation is to be regarded as the HLA compliant. All these rules are divided into two groups: federation rules and federate rules [DMS98A].

There are five rules for federation:

1) Federations shall have an HLA Federation Object Model (FOM), documented in accordance with the HLA Object Model Template (OMT).
2) In a federation, all representation of objects in the FOM shall be in the federates, not in the runtime infrastructure (RTI).
3) During a federation execution, all exchange of FOM data among federates shall occur via the RTI.

4) During a federation execution, federates shall interact with the runtime infrastructure (RTI) in accordance with the HLA interface specification.
5) During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.

There are five rules for federate:

6) Federates shall have a HLA Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT).
7) Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.
8) Federates shall be able to transfer and/or accept ownership of attributes dynamically during a federation execution, as specified in their SOM.
9) Federates shall be able to vary the conditions (e.g., thresholds) under which they provide updates of attributes of objects, as specified in their SOM.
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.

The HLA rules govern the behaviors of the federation and federate.

The federation rules establish the ground rules for creating a federation. The federate rules deal with the individual federates. Under the HLA, all federates must document the public information. In addition, all data representation takes place in the federates (not in the RTI) with only one federate owning any given attribute of a HLA object instance at any given time. Federate rules do not allow explicit communications between federates and all the information exchange among the federates takes place via the RTI as specified in the HLA interface specification.

3.3.2. Interface specification

The HLA Interface specification defines the methods between each federate and the Runtime Infrastructure (RTI). These methods allow the federates to communicate and cooperate with each other. [DMS98B] provides the details of the definition of these services.

From the viewpoint of abstraction services, the interface specification consists of six types of services listed in table 3-1.

Category	Functionality
Federation Management	Creates dynamic control, modifies and deletes a federation execution, and to allow simulations to join or resign from existing federations, and to control checkpoint, pause, resume and restart an execution.
Declaration Management	Establishes intent to publish object attributes and interactions that produce and subscribe to attributes and interactions produced by other federates.
Object Management	Creates and deletes object instances, control attribute and interaction publication, and to produce and receive individual attribute updates and interactions.
Ownership Management	Allows a federate to transfer the ownership of object attribute to other federates during the federation execution.
Time Management	coordinates the advance of logical time and its relationship to real time, so it allow the federates interoperable in despite of different time management schedule
Data Distribution Management	supports efficient routing of data by applying some technique to filter data

Table 3-1: The basic Runtime Infrastructure services

3.3.3. Object Model Template (OMT)

Currently, most of the HLA federates are constructed using Object-Oriented (OO) programming languages like C++ through RTI API. Prior to federate development, one should identify the difference between the basic concepts defined in the OO programming language and HLA. “Object” defined in terms of HLA have only a degenerate relation with the concept of “object” in the OO programming language. HLA objects do not define any behavior or operation and do not even have a type. They are merely hierarchical sets of attributes (data member) declarations (In the thesis, the HLA object is referred to as “HLA object” and “object” in OO is referred to as “OO object”).

The HLA object models describe the sharable elements between federates. HLA puts no constraints on the specific data type in these HLA objects models. It requires federate and federation to use the standard HLA Object Model Template (OMT) to document their HLA object models [DAHM97].

The OMT defines two types of classes:

- HLA object classes
- HLA interaction classes.

The main difference between them is that HLA object class associated with attributes persists for some interval time. By contrast, HLA interaction class is a collection of the data which is called parameters, sent and forgotten through the RTI to other federates. [KUHL99].

HLA object models shall contain at least one HLA object class or one HLA interaction class.

The OMT are classified to two types of the HLA object models: Federation Object Models (FOM) and Simulation Object Model (SOM). OMT format is applicable for both FOM and SOM.

The FOM describes sharable elements across the federation. The SOM describes these elements of the individual federate, which are available in the future federation. The FOM can be considered a superset of a set of SOMs of the participating federate. The federate interacts with other federates with a compatible FOM in the federation. The SOM and FOM design and development are not in the scope of this project.

The OMT consists of the different tables, more detail about these tables can be found in [DMS98C].

3.3.4. HLA infrastructure

HLA infrastructure contains the following main components:

- A number of federates which are HLA compliant;
- Runtime Infrastructure (RTI);
- Federate and RTI runtime communication interface;

Figure 3-1 shows the logical view of HLA infrastructure [DMSO00].

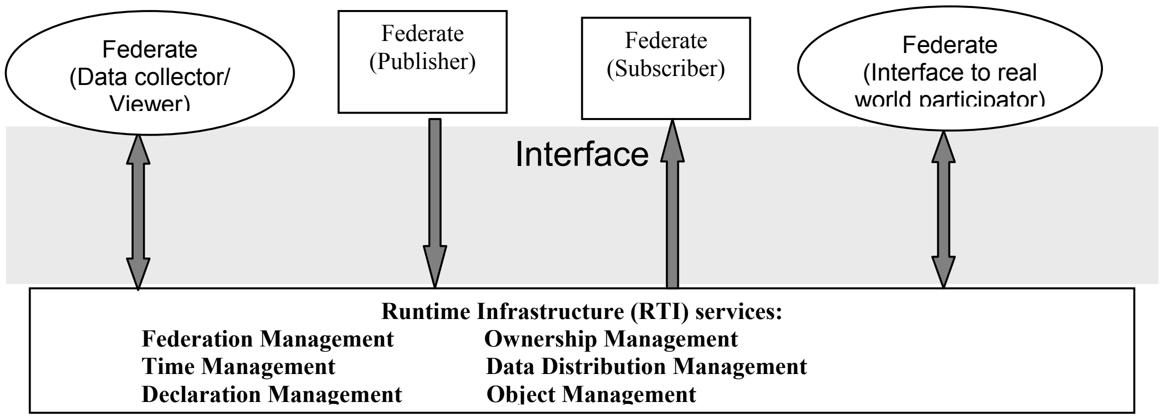


Figure 3-1: HLA federation

A federate can be a computer simulation, a manned simulator, a supporting utility (such as a viewer or data collector), or a live player or instrumented range. All the data representation are in the federates. Each federate maintains and controls a collection of sharable elements. These sharable elements have a number of the HLA object class, attributes of the HLA object class, HLA interaction class and parameters of the HLA interaction class that defined in the SOM.

3.3.5. RTI components

The RTI software is comprised of the RTI Executive process (RtiExec), the Federation Executive process (FedExec) and the libRTI library.

The RtiExec process manages the creation and destruction of federation executions. The FedExec manages federates joining and resigning the federation. The libRTI provides the federate communication service with RtiExec, FedExec and other federates. The libRTI library includes two classes: class RTIambassador and abstract class FederateAmbassador.

The class RTIambassador bundles the services provided by the RTI. All requests made by a federate on the RTI take the form of the RTIambassador method call. The HLA programmers can directly call these methods.

The abstract class FederateAmbassador, which provides a set of callback methods, is connected locally to the federate, and is responsible for the communication to the RTI. The HLA programmers should implement functionalities of these abstract callback methods in the derived classes.

3.3.6. Federation execution

A federation consists of a collection of related federate and RTI services with a FOM. Federates interact with other federates through the publishing/subscribing, sending/receiving services, which are provided by the RTI. A federate may simultaneously participate in more than one federation.

Figure 3-2 shows the federation execution life cycle with UML sequence diagram.

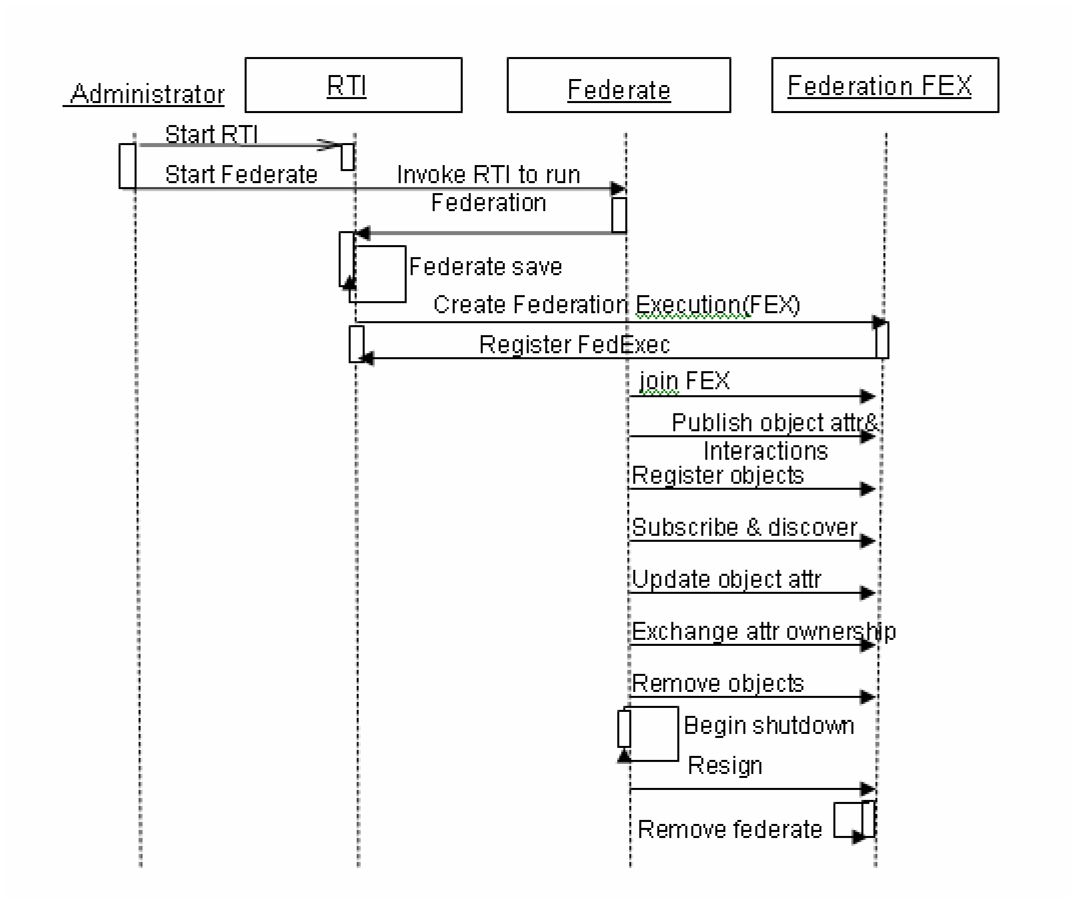


Figure 3-2: The sequence diagram of the federation execution life cycle

The federation execution sequence is described below:

1) When a federation is started, the administrator first starts the RTI execution process (RTIExec).

2) Then one federate creates a federation execution process (FEDExec), which is supposed to be given the name “FEX”, by invoking the RTI method: create Federation Execution. If FEX does not exist, the RTI process create it, otherwise the “FederationExecutionAlreadyExists” exception will be issued.

3) The RTIambassador reserves a name with RTIExec, and spawns a FEX, and that FEX registers its communication address with RTIExec. The federation FEX execution is underway.

4) When a federate joined the FEX, it can publish the sharable elements, register HLA objects, subscribe and discover the HLA object, update the HLA object attributes, exchange the attribute ownership and so on. Finally, it will shutdown and remove the federate from the FEX.

5) Once a federation FEX execution exists, other federates can join it. That RTI Ambassador consults RTIExec to get the address of FEX, and invokes “joinFederationExecution” on FEX. Additional federates can join via the same process.

3.4. Federate development process

From the viewpoint of the functionality abstraction, the basic functionalities of a federate should contain sharable elements of SOM/FOM representation and cover basic RTI services such as Federation Management, Declaration Management, and Object Management. For federate developers, they should implement these basic functionalities for a federate [DMSO00]. Figure 3-3 shows the basic federate functionalities.

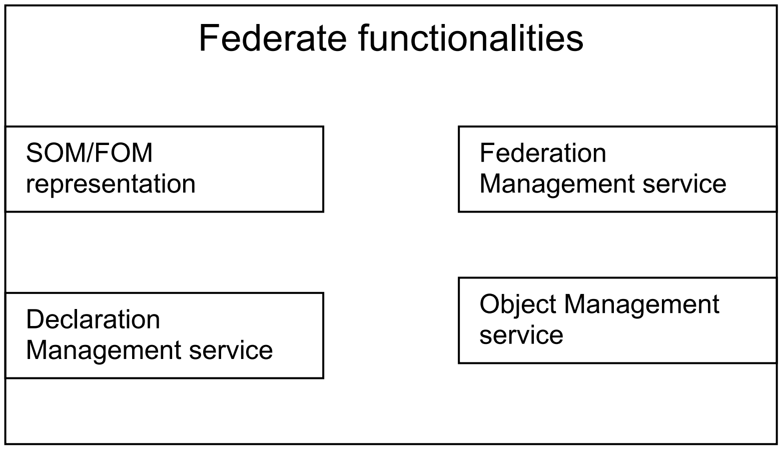

Federate functionalities
SOM/FOM representation
Federation Management service
Declaration Management service
Object Management service

Figure 3-3: Overview of the basic federate functionalities

3.4.1. SOM/FOM representation

The HLA rules for a federate constrain that a federate should be written in accordance with a SOM before it joins in a federation. When the federate is deployed in a federation, the executable federate should be in accordance with a FOM.

In the RTI specification, RTI and federation execution process use unique handles to indicate these sharable elements. The federate developers should define these sharable elements handles in the federate software.

3.4.2. Relevant Federation Management

The basic functionalities of Federation Management services are the services to create, join, resign and destroy a federation. Before a federate may join a federation execution, the federation execution must exist. Once a federation execution exists, federates may join and resign from it in any sequence. The relevant activities about Declaration Management and Object Management occur between the federate join and resign a federation.

Figure 3-4 shows a general view of basic life cycle of federation management service.

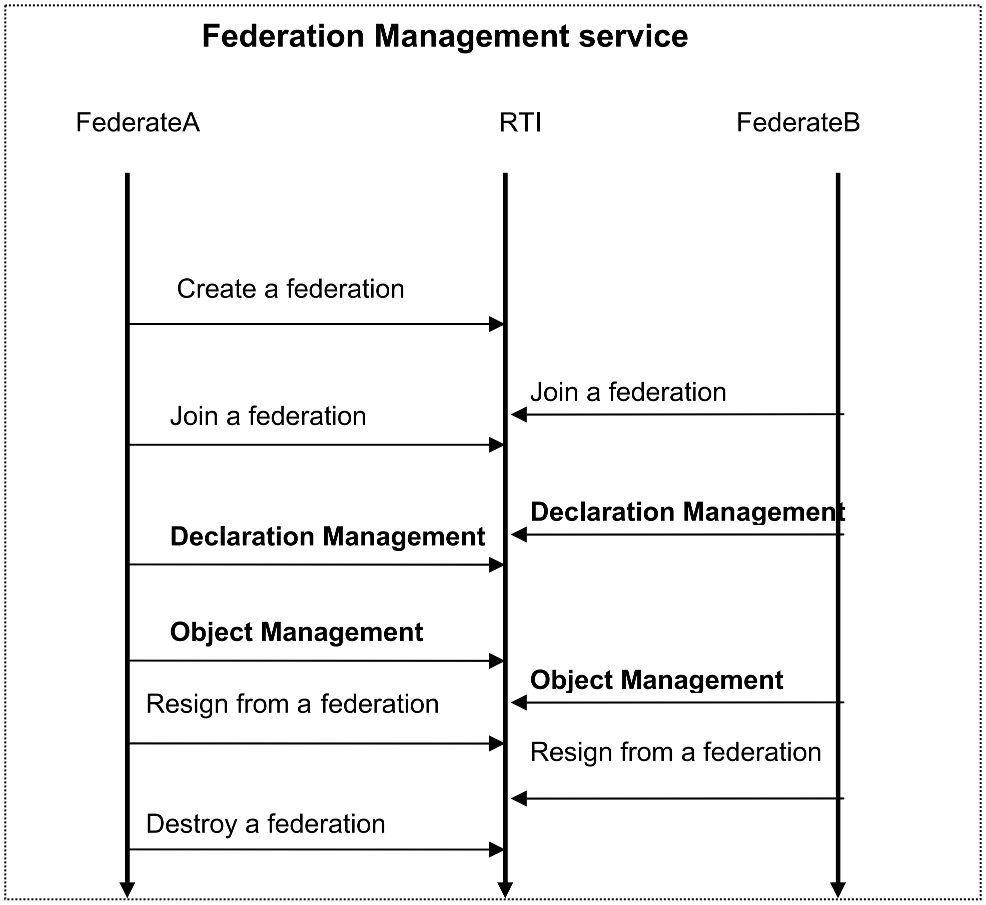


Figure 3-4: The basic Federation Management life cycle

In the DMSO RTI implementation, RTIambassador method *createFederationExecution*() can create a specified federation.

Similarly, by calling an RTIambassador method *joinFederationExecution*(), a federate can join a specified federation. RTIambassador method *resignFederationExecution*() terminates a federate's participation in a specified federation. RTIambassador method *destroyFederationExecution*() attempts to terminate an executing federation.

(Most of RTIambassador methods may raise exceptions. For simplification, the exception handling parts of relevant RTIambassador methods are omitted here although they should be included in the federate software.)

The following is the C++ code of these methods:

```
rti_ambassador.createFederationExecution(federation_name, "federation_FEDfile_name");
rti_ambassador.joinFederationExecution(federate_name,federation_name,fedaeteambassdor);
rti_ambassador.resignFederationExecution(RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES);
rti_ambassador.destroyFederationExecution(federation_name);
```

3.4.3. Relevant Declaration Management

The basic functionalities of Declaration Mmanagement services are the services to get RTI handles of HLA sharable elements to publish/ subscribe and unpublish/unsubscribe sharable elements of HLA object model. The relevant service about Object Management occurs between the federate publish/subscribe and unpublish/unsubscribe activities. Figure 3-5 shows a general view of basic life cycle of declaration management.

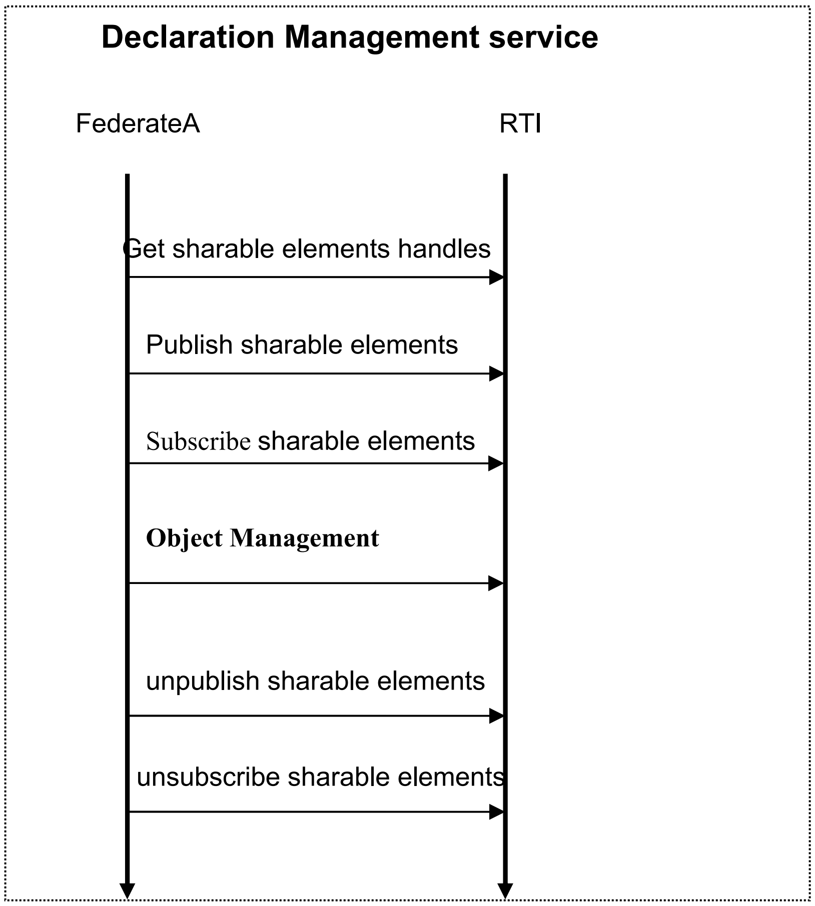

Declaration Management service
FederateA
RTI
Get sharable elements handles
Publish sharable elements
Subscribe sharable elements
Object Management
unpublish sharable elements
unsubscribe sharable elements

Figure 3-5: The basic Declaration Management life cycle

By calling a RTIambassador method *getObjectClassHandle*(), a federate can get the relevant handles. The method returns an RTI::ObjectHandle. For example, “country” is an HLA object class. The following is the C++ code of this method:

```
RTI::ObjectClassHandle ms_objectClassHandle;

char* const ms_objectClassStr = "Country";

ms_objectClassHandle = rtiAmb->getObjectClassHandle(ms_objectClassStr);
```

Similarly, a federate can get other sharable elements handles by relevant RTIambassador methods: *getAttributeHandle*(), *getInteractionClassHandle*() and *getParameterHandle*().

By calling a RTIambassador methods: *publishObjectClass*(), *unpublishObjectClass*(), *subscribeObjectClassAttributes*(), and *unsubscribeObjectClass*() a federate can publish/unpublish this HLA object class and subscribe/unsubscribe attributes of the HLA object class. For example, "country" is a HLA object class and "name" is an attribute, the following is the C++ codes of these methods:

```
objectClassAttributes = RTI::AttributeHandleSetFactory::create(1);
objectClassAttributes ->add( this->ms_attributesHandle );
rtiAmb->publishObjectClass(this->ms_objectClassHandle,* objectClassAttributes);
rtiAmb->subscribeObjectClassAttributes(this->ms_objectClassHandle,* objectClassAttributes);
rtiAmb->unpublishObjectClass(this->ms_objectClassHandle);
rtiAmb->unsubscribeObjectClass(this->ms_objectClassHandle);
```

Similarly, by calling RTIambassador methods: *publishInteractionClass*(), *unpublishInteractionClass*(), *subscribeInteractionClass*() and *unsubscribeInteractionClass*() a federate can publish/unpublish this HLA interaction class and subscribe/unsubscribe parameters of the HLA interaction class.

3.4.4. Relevant Object Management

The basic functionalities of Object Management services are the services to register/discover and delete/remove HLA object instances, update/reflect instance attribute values, and send/ receive HLA interactions.

The main difference between HLA object attributes and interaction data is that attributes associated with a registered object instance persist before the object is deleted, but an interaction data is sent then forgotten. Figure 3-6 shows a general view of HLA object instance life cycle of object management.

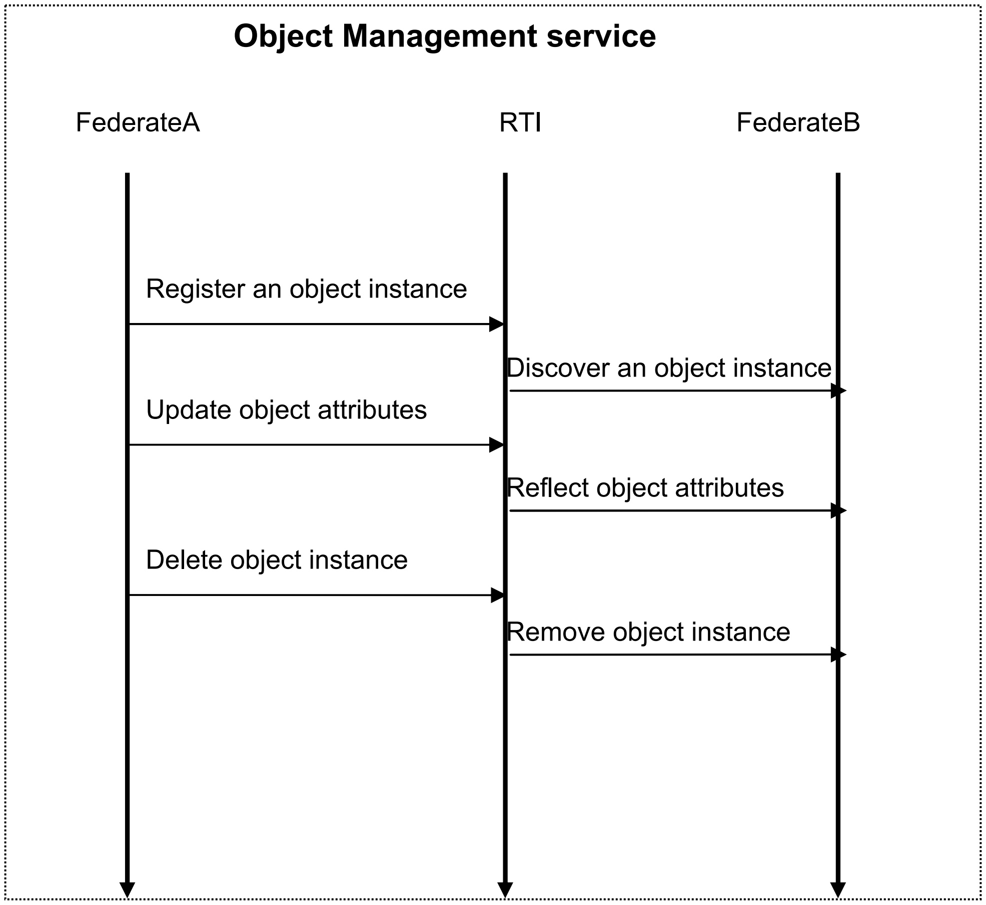

Object Management service
FederateA
RTI
FederateB
Register an object instance
Discover an object instance
Update object attributes
Reflect object attributes
Delete object instance
Remove object instance

Figure 3-6: The basic Object Management life cycle for an HLA object instance

By calling RTIambassador methods: *registerObjectInstance*(), a federate can register a HLA object instance. The method returns an RTI::ObjectHandle. Similarly, by calling RTIambassador methods: *updateAttributeValues*() and *deleteObjectInstance*(), a federate can update attributes values and delete the object instance.

The following is the C++ code of these methods:

```
m_instanceHandle =rtiAmb->registerObjectInstance(this->ms_objectClassHandle);
rtiAmb->updateAttributeValues(m_ instanceHandle, objectClassAttributes, NULL );
rtiAmb-> deleteObjectInstance (m_ instanceHandle);
```

When a federate object instance registers a federation, the FederateAmbassador *discoverObjectInstance*() callback informs a local federate with a handle indicating this HLA object instance. The developers must derive some subclass from the abstract class FederateAmbassador and implement this functionality. The following is the C++ code of this method:

```
void discoverObjectInstance(RTI::ObjectHandle theObject, RTI::ObjectClassHandle theObjectClass, const char * theObjectName)
{
// Implementation of how to do when find a object instance;
}
```

Similarly, the federate developers must implement other callback functions: *reflectAttributeValues*() and *removeObjectInstance*() which are declared in the abstract class FederateAmbassador.

A federate sends HLA interaction data in a similar way to the HLA attribute updates. Figure 3-7 shows a general view of HLA interaction class life cycle of object management.

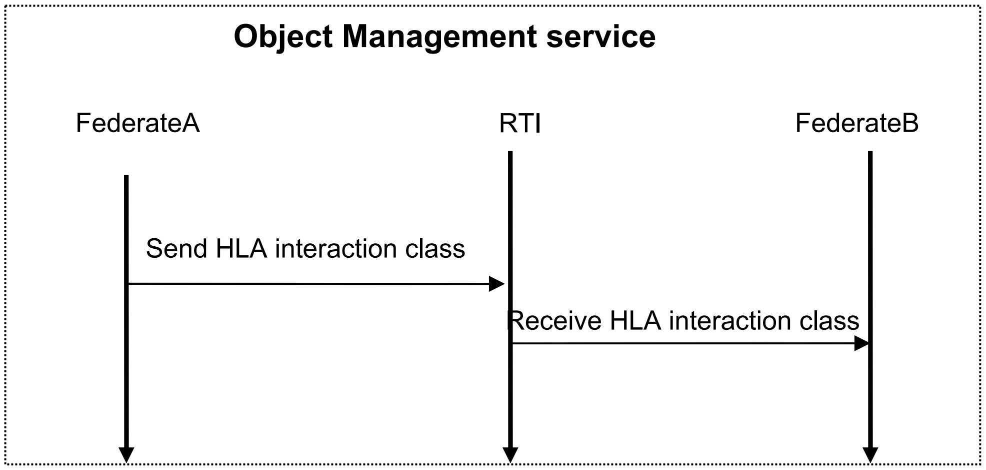

Object Management service
FederateA
RTI
FederateB
Send HLA interaction class
Receive HLA interaction class

Figure 3-7: The basic Object Management life cycle for an HLA interaction

By calling RTIambassador methods: *sendInteraction*(), interactions data are sent and forgotten. The following is the C++ code of this method:

```
RTI::ParameterHandleValuePairSet* pParams = NULL;
```

```
pParams = RTI::ParameterSetFactory::create( 1 );
pParams->add( this->ms_parameterHandle,cmdline,strlen(cmdline)+1);
rtiAmb->sendInteraction( this->ms_interactionClassHandle, *pParams,NULL );
```

Similarly, the federate developers must implement the other callback methods: *receiveInteraction*(), which is declared in the abstract class FederateAmbassador.

3.5. Federate implementation problems

3.5.1. General problems

When developers construct the HLA federate software, some common problems should be considered:

1) The HLA is only a conceptual framework for the distributed simulations. It does not address any approach to design a federate structure. It is users' responsibility to design federate software structure. If the members of the developing team design different federate structures, it may result in incompatibilities among the federates software. There is often little opportunity for reuse within an organization, much less between organizations.
2) The HLA RTI structure is monolithic. A supporting infrastructure has to be provided for each method of the RTI API. As a result, much effort is invested to construct the supporting implementation. Without suitable management of these implementations, the federate software seems as ad hoc basis and increases the cost of software maintenance.

3) The RTI provides low level and fine granularity functions. The developers need to further develop complicated implementation of relevant RTI APIs respectively even for a very simple concept. The developers should manage the correlating RTI APIs.

4) The HLA lacks the support for the management of the RTI concurrent event errors. In the RTI specification, any attempt to re-enter servicing RTI event causes a concurrent error that result in an exception being thrown. The developers should invest more effort to handle these problems.

5) Federate source codes usually involve many RTI APIs and their supporting implementation. It is difficult to track the bugs in the development phase. If such federate software is shared by other federates, all these software meet the same problems.

Based on the HLA programming experience, there are also some specific problems related to Federation Management, Declaration Management, and Object Management.

3.5.2. Problems related to Federation Management

When a federate creates, joins, resigns and destroys a federation, the developers should think about some strategies for the following problems:

1) **Who creates a federate execution?** A federation consists of a set of related federates. Before a federate can join a federation, the federation execution must exist. Otherwise a federate must create the given federation.

2) **How to get federation run time parameters?** A federate need some initialization data for the execution such as the run-time parameters like federate name, the federation name that a federate should join.

3) **What will a federate do if it fails to create or join a federation?** When the attempt to create a federation execution fails or a federate fails to join a federation, the developers should consider whether to continue or stop the simulation.

4) **What will a federate do after resigning from a federation?** When a federate terminates its participation in a given federation, the developer should decide what must be done by a federate.

5) **How to destroy a federation?** When a federate tries to terminate an executing federation, if the invoking federate is not the last federate to have resigned and there are still other federates joined in the targeted federation, an exception will be raised. Thus, this destroying effort must fail.

3.5.3. Problems related to Declaration Management

In the HLA, there is no functionality to represent the HLA object model and define their handles. Thus, when developers want to publish, subscribe sharable elements and use their handles. The developer should think about some strategies about mapping the HLA object model and their handles into object-oriented programming languages.

1) **How to represent HLA attributes?** When a federate publishes or subscribes to a HLA object class, it must indicate explicitly which attributes it can produce. Thus, the attributes of HLA object class are associated with a registered object instance of the HLA object class. These attributes should be constructed to support each HLA object instance.

2) **How to represent HLA parameters?** When a federate publishes or subscribes the HLA interaction class, interactions are sent or received as "all or nothing." It is not

possible to specify which parameters of an HLA interaction class will be published or subscribed. Thus, these parameters are usually associated with the HLA interaction class.

3) **How to represent handles of HLA sharable elements?** The RTI converts names of the HLA sharable elements to the handles that are used by the various RTI services to refer to them. Thus, before a federate publishes or subscribes HLA sharable elements, it requires their relevant handles. The handles of the HLA sharable elements should be access by the entire registered HLA object instance. These handles should have global scope for a federate.

The developers should carefully consider how to construct the HLA object model and their handles as the different scopes in the federate software.

3.5.4. Problems related to Object Management

When a federate updates HLA instance attributes or sends HLA interactions, the developers should think of some strategies about:

1) **What implementation of the mechanisms for updating attributes values?** Before a federate updates its attributes values associated with a registered HLA object instance in a federation, the developers should decide how to update them. When the effort of updating attributes fail, the developers should consider whether to continue or stop the simulation.

2) **What implementation of the mechanisms for reflecting attributes values?** When the remote object instance updates its attribute values, the developers should consider how the local subscribing federates reflect these attributes values.

3) **What are the implementation mechanisms for sending HLA interactions?** Before a federate sends HLA interactions into a federation, the developers should also decide what are the mechanisms for sending interactions. In addition, the remote federate sends HLA interactions, the developers should consider how the subscribing federate correctly receives these interactions. When the effort of sending HLLA interactions fail, the developers should consider whether to continue or stop the simulation.

4) **What are the implementation mechanisms for receiving HLA interactions?** When a federate interacts with other federates by calling RTIambassador methods, RTI will issue some response by callback methods that are declared in the abstract class FederateAmbassador. Because the RTIambassador methods and FederateAmbassador callback methods are used as the request and response interface respectively, the developers should consider how to arrange these relevant RTIambassador methods and FederateAmbassador callback methods in the federate software.

5) **What to do after a federate registers an object instance?** When a federate registers an HLA object instance in a given federation by calling RTIambassador method: *registerObjectInstance*() . RTI will inform other federates that a new object instance has come into existence by the FederateAmbassador callback method: *discoverObjectInstance*(). These two methods are naturally related in a federate.

   Similarly, RTIambassador method: *updateAttributeValues*() and *sendInteraction*() are naturally related to FederateAmbassador callback method: *reflectAttributeValues*() and *receiveInteraction*(). For typical federate software, the developers should consider what to implement these three callback methods.

Chapter 4. Federate software framework

4.1 Overview of complete federate framework

As mentioned in section 3.4, there is a significant amount of work in developing federate software. Thus, the developers must handle many relevant problems. Without supporting tools or pre-defined federate templates, a federate must be developed more or less from scratch. For this reason, federate developers require tools and techniques to improve the productivity of the development.

Using a framework is one promising solution. The framework includes a set of classes and embodies an abstract design for solutions. The complete federate framework can cover all the federate development process and solve the relevant problems.

The complete framework provides the following functionalities:

1) This framework contains a federate design guide or generic structure template to construct the federate software. This results in reuse of the implementations between the federates software.
2) This framework provides a set of implementations for each RTI API to save developers effort in constructing the supporting infrastructure in the federate software.
3) This framework provides a set of higher level general simulation service implementing functionalities which are required by all the federates. The verbose and complex low level RTI APIs is hidden from the designers.

4) The framework components can be reused to substantially improve software productivity. The federate software errors can be fixed once in the framework rather than in each federate independently.

5) This framework supports the error management for HLA federate software. The developers can easily track the bugs to avoid the common programming pitfalls associated with RTI programming.

The complete federate framework works for the common simulation domain problems regardless of any purpose or technique implementation. It consists of the complete default implementations of RTI APIs, a set of general simulation service functionality, and error management. However, this universal framework should be based on the thorough analysis of the entire application requirements. In addition, designing such framework needs the relevant HLA programming experience on the simulation domain and deeper understanding of the future application evolution trend.

A practical work is to construct a basic design framework to solve some open problems which the typical federates developers need to consider. It can be extended and configured for a particular simulation.

4.2 Proposed federate design framework

4.2.1. Overview of design framework

The proposed framework restricts to HLA concepts and principles. Its scope covers the first four parts of the complete federate framework. To solve the parts of the general problems in the section 3.5.1, the following describes the key features of the proposed framework:

1) For problem (1): **there is no any approach to design a federate structure**, the proposed framework is a reusable design template which is composed of a set of cooperating classes. The template consists of five collaborating classes:

    - Base class ObjectClassBase for all the HLA object classes;
    - User defined subclass of the class ObjectClassBase for each HLA object class;
    - User defined interaction class for each HLA interaction class;
    - Base class DefaultFederateAmbassador and its subclass for the abstract class FederateAmbassador;

2) For problem (2) and problem (3): **the HLA RTI structure is monolithic and complicated implementations of RTI low level services are needed**, each class defines the standard data structure to represent the HLA sharable element. In addition, it also provides the general higher level simulation services found in typical HLA simulations. All these services cover Declaration Management and Object Management.

4.2.2. Components' responsibilities and collaborators

In this section, CRC card describes the responsibilities and collaborators of each class in the framework. The collaborating classes solve the federate development problems in the sections 3.5.3 and 3.5.4.

1) Class ObjectClassBase

To solve the problem (1) in the section 3.5.4: **how to implement updating of all HLA object classes' attributes values**, an abstract class ObjectClassBase is defined as the base class for the different types of the HLA object class. The abstract class supports the dynamic subclass loading at run time through virtual functions. For example, an object belonging to a derived class acts as the subclass by calling the virtual function. Using CRC card, table 4-1 shows the responsibilities and collaborators of the class ObjectClassBase.

class ObjectClassBase	
Responsibility	**Collaborator**
Declares the virtual functions to reflect the data updating of the remote object instances.	➢ The class RTIambassador; ➢ Its subclass for each HLA object class; ➢ The user defined subclass of the class DefaultFederateAmbassador ;

Table 4-1: CRC cards describing the class ObjectClassBase

2) User defined object class

To solve the problems (1), (3) in the section 3.5.3 and problem (1) in the section 3.5,4:

How to represent HLA attributes, handles of HLA sharable elements, and **update attributes values,** for each HLA object class, its attributes and the relevant operations should be modeled as an individual user defined object class. Using CRC card, table 4-2 shows the responsibilities and collaborators of the user defined object classes.

User defined object class	
Responsibility	**Collaborator**

➢ Map names of HLA object class and data types of its attributes;
➢ Construct the relevant handles;
➢ Publish and subscribe HLA object class associated with the attributes;
➢ Register the instance of HLA object class;
➢ Update attributes of the HLA object;
➢ Override the virtual functions;
	➢ The user defined subclass of the class DefaultFederateAmbassador ;
	➢ The class RTIambassador;
	➢ The class ObjectClassBase;

Table 4-2: CRC cards describing the user defined object class

3) User defined interaction class

To solve the problems (2), (3) in the section 3.5.3 and problem (3) in the section 3.5.4: **how to represent HLA parameters, handles of HLA sharable elements,** and **send interaction class**, for each HLA interaction class, its parameters and the relevant operations are also modeled as an individual user defined interaction class. Using CRC card, table 4-3 shows the responsibilities and collaborators of the user defined interaction class.

User defined interaction class	
Responsibility	**Collaborator**
➢ Map names and handles of HLA interaction class; ➢ Map data types of the parameters (optional); ➢ Construct the relevant handles;	➢ The user defined subclass of the class DefaultFederateAmbassador ; ➢ The class RTIambassador;

Table 4-3: CRC cards describing the user defined interaction class

4) Class DefaultFederateAmbassador and the user defined subclass

To solve the problems (2), (4) and (5) in the section 3.5.4: **how to reflect attributes values**, **receive HLA interactions and what to do after registering a federation**, these two classes contain implementations for the abstract class FederateAmbassador. The class DefaultFederateAmbassador provides the default implementations. Developers can override the necessary callback methods in its subclass to support the desirable functions. Using CRC cards, table 4-4 shows the responsibilities and collaborators of these classes.

class DefaultFederateAmbassador	
Responsibility	**Collaborator**
It includes a set of dummy implementations providing the basic function to handle HLA concurrent exception including: *discoverObjectInstance* (), *updateAttributeValues*() and *receiveInteraction*().	➢ The abstract class FederateAmbassador; ➢ The user defined federate ambassador class;
User defined FederateAmbassador class	
Responsibility	**Collaborator**

It implements the user specific functions by overriding the functions which are inherited from the class DefaultFederateAmbassador.	➢ The class ObjectClassBase;
	➢ The user defined object class;
	➢ The user defined interaction class;
	➢ The class DefaultFederateAmbassador;

Table 4-4: CRC cards describing the classes for the class FederateAmbassador

4.2.3. Components' structure

This section describes the data structure and member functions of each component in the proposed framework.

1) C++ class ObjectClassBase

The class ObjectClassBase is fixed in this framework. It is an abstract class for all the HLA object classes. It declares two virtual functions:

- Function ***getInstanceHandle***() supports to get the object handle of the remote federate.
- Function ***updateValueFromRTI***() supports to reflect the data updating of the object instances of the remote federates.

The user defined subclass of HLA object class should override these two functions. Figure 4-1 shows the relationship between the class ObjectClassBase and its subclass.

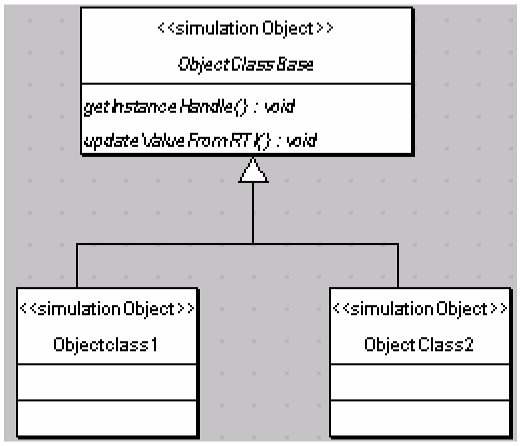


Figure 4-1: The class diagram for class ObjectClassBase and its subclasses.

2) C++ class for an HLA object class

A user defined object class, which is called “ObjectClass”, directly maps the HLA object class and their handles as member attributes. In addition, it can encapsulate basic RTIambassador methods and their relevant implementation as member functions. This class provides the services which cover the Declaration Management services and the Object Management services.

For example, a federate, “Tank”, includes an HLA object class called “Position” and it has two attributes called “pos_x” and “pos_y” with float data type.

Figure 4-2 shows an example of the user defined object class “Position”.

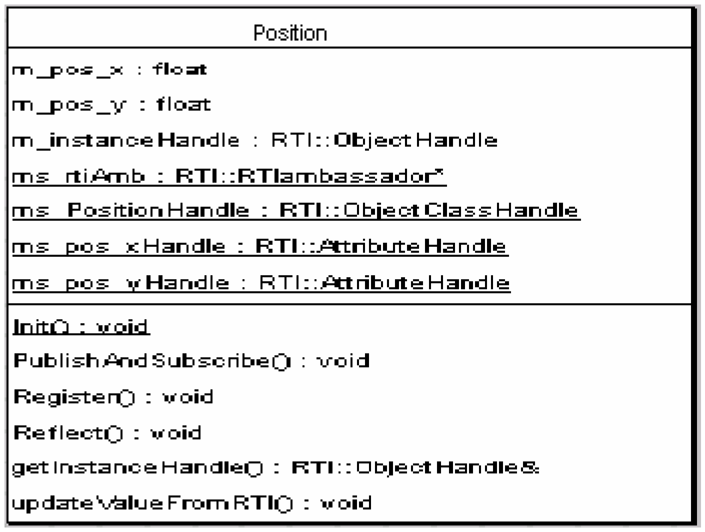

Position
m_pos_x : float
m_pos_y : float
m_instanceHandle : RTI::ObjectHandle
ms_rtiAmb : RTI::RTIambassador*
ms_PositionHandle : RTI::ObjectClassHandle
ms_pos_xHandle : RTI::AttributeHandle
ms_pos_yHandle : RTI::AttributeHandle
Init() : void
PublishAndSubscribe() : void
Register() : void
Reflect() : void
getInstanceHandle() : RTI::ObjectHandle&
updateValueFromRTI() : void

Figure 4-2: An example of the user defined object class “Position”

Member attributes:

- The “m_pos_x”, “m_pos_y” and “m_instanceHandle” can be defined as the ***private*** member attributes to only associate with each registering object. Each object instance can update its attributes values respectively. These attributes can be constructed and destroyed in the construction functions and deconstruction of the class “ObjectClass”.

- The relevant handles are defined as ***public static*** member attributes such as "ms_PositionHandle", "ms_pos_xHandle" and "ms_pos_yHandle" to support the global scope in a federate. The developers construct them to publish and subscribe the HLA object class.
- To call RTIambassador methods, a pointer instance of the class RTIambassador is defined as a ***public static*** member attribute like "ms_rtiAmb".

Member functions:

The class "ObjectClass" should override the virtual functions: ***getInstanceHandle***() and ***updateValueFromRTI***().

- ***getInstanceHandle***()***:*** This function returns the m_instanceHandle value of class Tank;
- ***updateValueFromRTI***()***:*** This function supports to update the attribute values of the remote object instance of class Tank;

The class "ObjectClass" can also define some functions to encapsulate the basic RTIambassador methods related to HLA sharable elements.

According to their functionalities of abstraction services, some functions below show the possible encapsulation.

- ***Init***()***:*** This function is declared as a ***public static*** function. It is a class level member function with one parameter: ***RTI::RTIambassador* rtiAmb***. This parameter can pass the RTIambassador pointer to the static attribute: "ms_rtiAmb". The developer can use this attribute to call RTIambassador methods. In addition, this function constructs the handles for publishing and subscribing HLA object class. It encapsulates the necessary RTIambassador methods related to getting the HLA handles. These methods include: *getObjectClassHandle*(),*getAttributeHandle*(), *getInteractionClassHandle*() and *getParameterHandle*(). They are described in Declaration Management.

- ***PublishAndSubscribe*()*:*** This function is declared as a ***public*** function. It is an object level member function. This function constructs the HLA attributes, publishes, and subscribes HLA object classes by using their relevant handles. It encapsulates the necessary RTIambassador including *publishObjectClass*() and *subscribeObjectClassAttributes*(). They are described in Declaration Management.
- ***Register*()*:*** This function is declared as a ***public*** function. It is an object level member function. This function registers an HLA object stance. It encapsulates the necessary RTIambassador methods related to creating an instance of the HLA object class and registering it with the federation. This method is *registerObjectInstance*(). It is described in Object Management.
- ***Reflect*()*:***This function is declared as a ***public*** function. It is an object level member function. This function updates the attributes values associated with the registering HLA object instances. It encapsulates the necessary RTIambassador methods and the implementation mechanisms of updating attributes values. This method is *updateAttributeValues*(). It is described in Object Management.

The user defined object class can be used after instantiation. The calling sequence is fixed in some degree. Functions: ***Init*()** and ***PublishAndSubscrib*()** must be first called, and the function ***PublishAndSubscribe***() should be called after the function ***Init***(). Both of them are class level member functions, so they can be directly used.

After that, the developers can use functions: ***Register***()and ***Reflect***(). The function *Register()* must be invoked before the function *Reflect*(). Because all these functions are the object level member functions, they should be called through one object instance.

3) C++ class for an HLA interaction class

A user defined interaction class, which is called "InteractionClass", directly maps the HLA interaction class and their handles as member attributes. In addition, it can encapsulate basic RTIambassador methods and their relevant implementation as member functions. This class provides the services which cover Declaration Management and Object Management.

For example, a federate, "Tank", includes an HLA interaction class called "Communication" and its parameter called "Message" with a string type.

Figure 4-3 shows an example of the user defined interaction class "Communication"

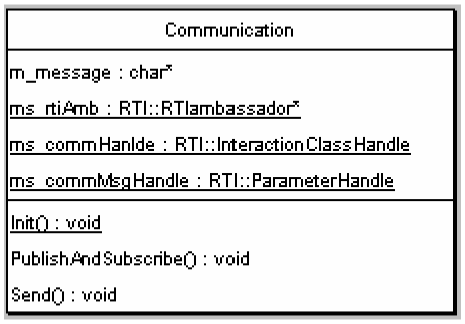

Communication
m_message : char*
ms_rtiAmb : RTI::RTIambassador*
ms_commHanlde : RTI::InteractionClassHandle
ms_commMsgHandle : RTI::ParameterHandle
Init() : void
PublishAndSubscribe() : void
Send() : void

Figure 4-3: An example of the user defined interaction class "Communication"

The member attributes and member functions are similar to class "ObjectClass". The main differences are:

Member attributes:

The handles of the HLA interaction class and its parameters are defined as ***public static*** member attributes such as "ms_commHandle" and "ms_commMsgHandle" to support the global access scope in a federate. The developers can construct them to publish, subscribe the HLA interaction class. Sometimes, data type of HLA parameters need not be defined as member attributes.

Member functions:

The member functions also include ***Init***(), ***PublishAndSubscribe***() and ***Send***(). The key feature of "InteractionClass" is the function: ***Send***()***:***This function is declared as a ***public*** function. It is an object level member function. This function sends HLA interactions. It encapsulates the necessary RTIambassador methods and the implementation mechanisms of sending interactions. This method is ***sendInteraction***(). It is described in Object Management. In the DMSO, the method *tick*() is used to let a late arriving federate join an existing federation and pass the information to the existing federates. So, this method is encapsulated in the member function.

The user defined interaction class can be used after instantiation. The calling sequence is similar to the user defined object class. The function *Send*() must be invoked after the functions *Init*() and *PublishAndSubscribe*().

4) C++ classes for FederateAmbassador

The HLA constrains that the developers should define a subclass, which is derived from the abstract FederateAmbassador. In this subclass, the developers can override the callback methods to discover the remote object instance, reflect updating values of the remote object instance attributes and receive HLA interactions. All these functions describe the remote federate data information.

During the federate development process, the developers should derive subclass from the abstract class FederateAmbassador to respond to the requests that RTI issues.

In this framework, such services can be implemented by two inheritance classes. Such inheritance structure allows the federate developers to focus on the simulation domain problems.

The base class DefaultFederateAmbassador can play a role of “Basic” implementation and user defined classes can derive from this base class. Figure 4-4 shows the relationship between these classes.

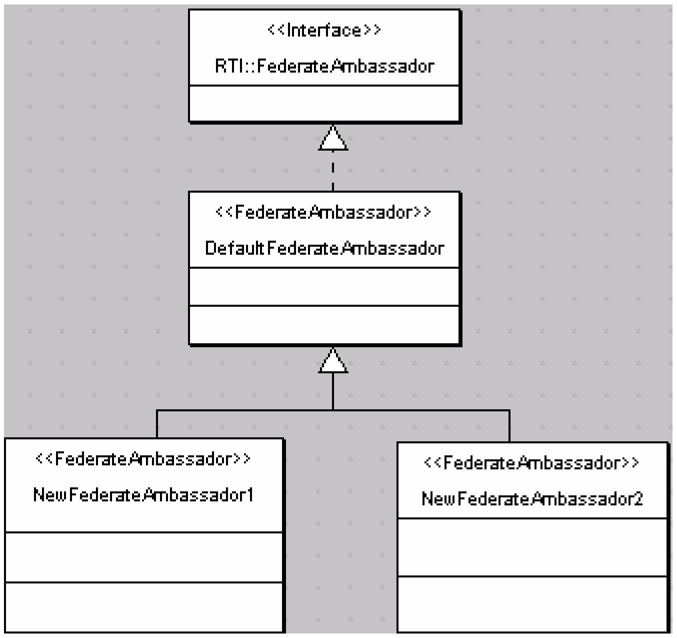


Figure 4-4: The class diagram for abstract class federate ambassador

The class DefaultFederateAmbassador derives directly from the abstract class FederateAmabssador. It is an “abstract” base class for federate software. It provides the default implementation for each abstract callback method, especially for these three virtual callback methods: ***discoverObjectInstance***(), ***reflectAttributeValues***(), and ***receiveInteraction***(), which the developers usually need to override.

The default implementation contains basic functionalities to handle HLA exception to avoid concurrent problems. For example, when RTI tries to re-invoke the updating federate attributes values, an exception "FederateInternalError" will be caught and output a stream.

The following C++ codes of ***discoverObjectInstance*** ():

```
        void DefinedFederateAmbassador:: discoverObjectInstance (
                        RTI::ObjectHandle            theObject,
                        RTI::ObjectClassHandle       theObjectClass,,
                        const char *          theObjectName)
        {
        //a default implementation of class DefinedFederateAmbassador;
        {cout << "Find a new HLA object instance  " << endl;}
}
```

This class is fixed in this framework. It can exchange information with the class "ObjectClassBase", the class "ObjectClass" and the class "InteractionClass". They can work together to complete the basic functionality covering Declaration Management and Object Management.

All the operations declared in this class are virtual operations. So they can be overridden by their subclasses.

The user defined subclass of the class DefaultFederateAmbassador implements the desirable abstract callback methods. The users can override the virtual methods which are inherited from the class DefaultFederateAmbassador to do its particular work.

4.2.4. Instantiation of framework

The federate design framework provides a configurable federate template. The subclass of the class DefaultFederateAmbassador should be associated with the class ObjectClassBase, the user defined object class, and the user defined interaction class.

This framework can be represented as an HLA specific class diagram. Figure 4-5 shows the structure of the federate design framework.

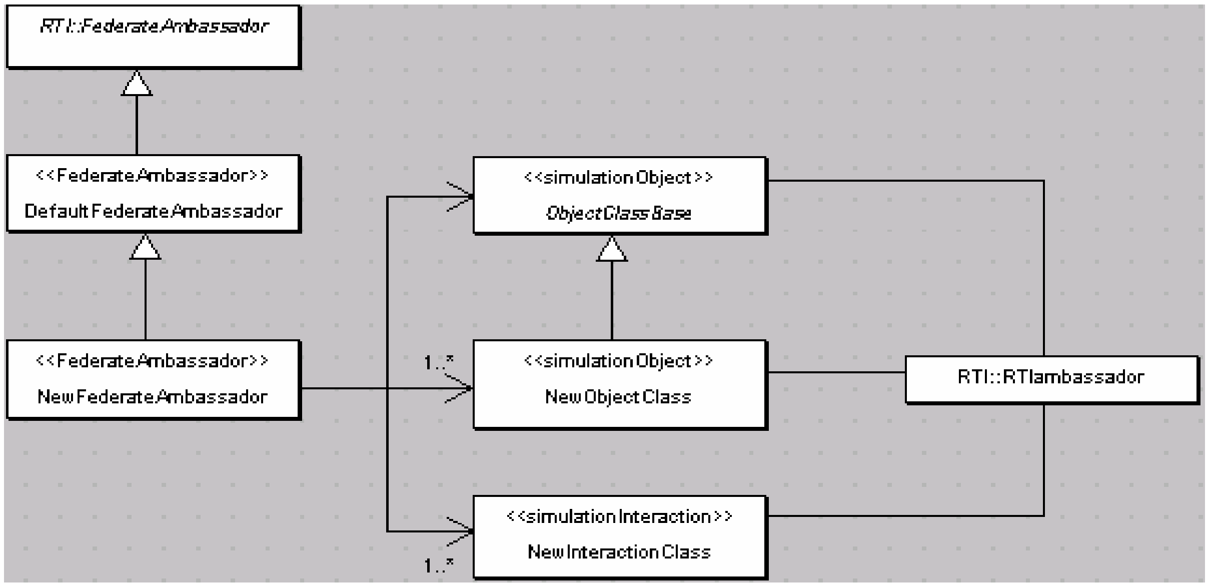


Figure 4-5: A pre-defined HLA federate class diagram

To use this framework, developers should instantiate all these classes in a federate executable program. This program usually solves the problems mentioned in the section 3.5.2 such as reading federation name, creating a federation, joining federates to a given federation, resigning federates from a specific federation, and destroying the federation. All these functions are described in Federation Management. Because all these works are tightly coupled to a particular simulation scenario, it is difficult to provide the common solution covering the different simulation application. This project provides an example simulation execution program for the developers' reference.

At the beginning of instantiation of federate framework, the federate execution program instantiates the object instances of the class RTIambassador, the user defined object class, the user defined interaction class and the subclass of the class DefaultFederateAmbassador and creates a federation and lets a federate join the given federation. Then, it can directly call the functions of the user defined object class and the user defined interaction class in the fixed sequence.

4.2.5. Benefits of framework

The proposed framework is a reusable design for HLA simulations. All these classes construct the skeleton of HLA federate software. It simplifies the task of HLA federate design and development in some ways:

1) The intention of the framework is to ease the creation of the federate. Given the FOM of a particular federate, developers can construct the federate software following the structure and functions of framework's components. For example, as mentioned in the 4.2.3, the designer can build a C++ class Position to map the HLA object class in accordance with in the FOM. Similarly, the C++ class Communication maps the HLA interaction class.

2) Through the pre-defined federate template, all the HLA simulations have the similar structure. That facility to reuse the existing design for the compatible federates in the similar application. It does not need to design and develop anew for each individual application. For example, there is an existing class Position in some federate and the FOM of another federate has the compatible definition of this class. The designer can totally reuse the design over OO language code.

3) The proposed framework encapsulates a number of HLA low level service methods (APIs). It hides much of the complexity of directly interfacing with RTI. For example, in the user defined object class, its member function ***PublishAndSubscribe***() provides the basic encapsulation of a set of complex low level RTIambassador methods. The following C++ code shows the RTI methods to implement this function:

```
// Implementation of publishing and subscribing by low level RTI methods
// to publish and subscribe
 RTI::AttributeHandleSet *PositionAttributes;
 PositionAttributes = RTI::AttributeHandleSetFactory::create(2);
  PositionAttributes->add( ms_pos_xHandle );
  PositionAttributes->add( ms_pos_yHandle );
 ms_rtiAmb->subscribeObjectClassAttributes( ms_PositionHandle,
                                   *PositionAttributes );
 ms_rtiAmb->publishObjectClass( ms_PositionHandle,
                             *PositionAttributes);
 PositionAttributes->empty();
 delete PositionAttributes;   // Deallocate the memory
```

Similarly, the framework defines other member functions such as ***Init***(), ***Reflect***(), ***Register*** ()and ***Send***(). These encapsulation functions allow the designers centralize the simulation domain rather than on meeting HLA RTI services.

4) The proposed framework reduces the development cost by the class DefaultFederateAmbassador. This class provides some basic implementation for most HLA simulations. For example, the class DefaultFederateAmbassador provides the default implementation of the function ***reflectAttributeValues***(). It can be reused for all the federate software without any change. The following C++ code shows the main body of this function:

```
reflectAttributeValues (){......
//After the federate discover the object instance, it can update its values
    if( theObject == pos->getInstanceHandle() ) {
        cout << "An intnstanceId found! " << theObject << endl;
        pos->updateValueFromRTI(theAttributes);
        break;}
  }
```

Chapter 5. Analysis of HLA extensions

5.1. Overview of HLA extensions

The federate design framework is useful only if it is supported by a set of tools. It needs supporting tools to facilitate the federate design and implementation process. A CASE tool, ArgoUML, is introduced because of its cognitive support mechanism and open source features. It can be customized with the HLA extensions to support federate design under the framework.

The main services of HLA extensions include:

1) The federate design framework should be supported by default in ArgoUML. The designers can use pre-defined structure to rapidly design a new federate and reuse the existing federate.
2) The standard UML semantics is extended to describe the federate framework. The designer can differentiate the HLA specific class from the ordinary class in the class diagram.
3) Cognitive help system is implemented for federate design. Reflection-in-action context help will appear when the system predicts some problems related to federate framework elements like user defined federate ambassador class and user defined object class. It helps architects on how to use the framework and reduce the design time.

4) Automatically generates C++ source code skeleton for the federate implementation phase.

5.2. Extension 1: UML extensions

During the federate design phase, one desirable HLA extension for designer is to differentiate the HLA specific class and ordinary class. It provides a way to produce HLA specific critics and generate the federate source code skeleton for the HLA specific class diagram.

As mentioned in section 2.3.2, the stereotype is one of the mechanisms for extending existing UML modeling elements. In predefined UML stereotype, there is no concept related to HLA. Thus, the HLA specific stereotype can be defined and represented by placing quotes around the new stereotype name, e.g. <<HLA_Name>>. This project proposes the possible extensions or specializations of the UML notation as part of the UML profile. An HLA specific class can be built through the new stereotypes names.

A new stereotype of class, which is marked by using <<FederateAmbassador>> keyword, is used to present a class federate ambassador.

Similar to the class federate ambassador, a new stereotype of UML extension, the object class and interaction class of the federate can be represented by using <<simulationObject>> and <<simulationInteraction>>.

Table 5-1 provides an example of stereotype for HLA extensions. It describes classes' representation involving a federate design.

Metamodel Element	HLA Component	Stereotype Name
Class	Federate Ambassador	FederateAmbassador
Class	HLA Object Class	SimulationObject
Class	HLA Interaction Class	SimulationInteraction

Table 5-1: Proposed stereotypes for the HLA extensions

5.3. Extension 2: Cognitive help for federate

During the design phase, the cognitive support feature can be implemented for the federate designer. The HLA extensions for the cognitive help include two parts: *HLA specific critics* and *criticism control mechanism*.

HLA specific critics predicate the HLA specific design decision. These HLA specific critics' predictions cover the following basic federate design problems:

- ➢ It reminds the users to consider changing the name of the user defined object class and the user defined interaction class like "Position".
- ➢ It reminds the users to add the HLA sharable elements and their handles as the member attributes in the user defined object class and user defined interaction class.
- ➢ It reminds the users to add the member functions in the user defined object class and user defined interaction class to encapsulate basic RTIambassador methods

and their relevant implementation like ***Init***(), ***PublishAndSubscribe***(), ***Register***(), ***Reflect***() and ***Send***().

- It reminds the users to override the virtual functions: ***getInstanceHandle***() and ***updateValueFromRTI***()in the user defined object class.
- It reminds the users to consider changing the name of the user defined federate ambassador class like "HwFederateAmbassador".
- It reminds the users to consider overriding the callback in the user defined federate ambassador class like ***discoverObjectInstance***(), ***updateAttributeValues***() and ***receiveInteraction***().

A new catalog, ***By HLA design,*** contains all the HLA specific critics. Meanwhile, these critics can also show in the existing catalogs so it will not break the original ArgoUML cognitive help system.

For the detailed classification, these critics can be divided into groups: ***HLA Federate Ambassador***, ***HLA Simulation Object,*** and ***HLA Simulation Interaction***.

The HLA specific criticism control mechanism manages and control the HLA related critics. It controls whether the HLA specific critics are active. When they are active, these critics produce ToDoItem and posted ToDo list. If they are not valid, such ToDoItems should be removed from the ToDo list.

5.4. Extension 3: Automatic C++ code generation for federate

A useful extension is to allow ArgoUML to generate the C++ source code skeleton from the federate framework.

In ArgoUML with stereotype extension, the classes of the federate framework can be differentiated from the ordinary classes. When designer selects the targeted federate classes, the C++ source template codes for these classes should be generated.

The code generation mechanism generates the C++ source code skeleton for the user defined federate ambassador C++ file, the user defined interaction class C++ file and the user defined object class C++ file and their related header files. The operations bodies in these files are just empty in the design phase.

Because the class ObjectClassBase and the class DefaultFederateAmbassador are the fixed classes, their completed source can be generated by default.

Chapter 6. Implementation of HLA extensions

6.1. Overview of HLA extensions architecture

The HLA extensions are packaged into a new package: ***hla***.

A series of packages are designed to support HLA extensions. The new package contains four main packages. It is seamlessly integrated with ArgoUML. Figure 6-1 shows the HLA extensions package structure.

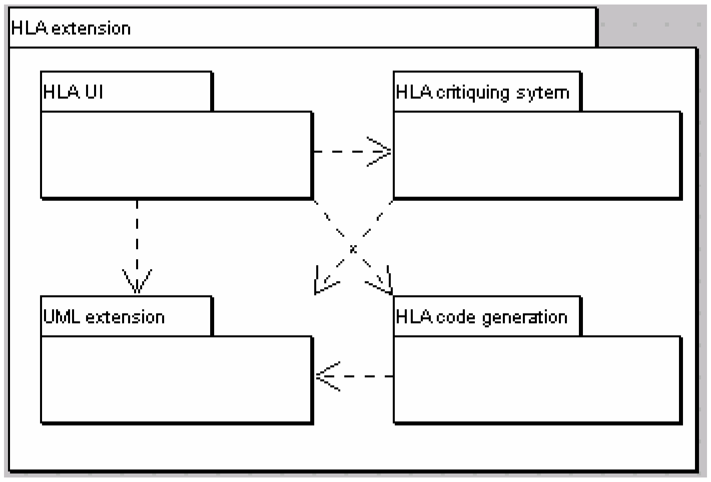


Figure 6-1: HLA extensions package structure

The package HLA UI contains the new user interface for federate designer. The designer can select the class with the UML extensions to differentiate an HLA specific class. The package of HLA critiquing system and the package HLA code generation also depend on this package to provide the access points for the federate designer.

The package HLA critiquing system provides the cognitive help for the federate designer. It can produce the ToDoItems posting on the ToDoList.

The package HLA code generation produces the C++ code skeleton for a federate when designer selects the HLA specific classes.

6.2. User interface: new menu item for HLA

For the HLA extensions to be accessible in the ArgoUML, a new menu item is added into the ArgoUML menu. Figure 6-2 shows the new menu item for the HLA extensions.

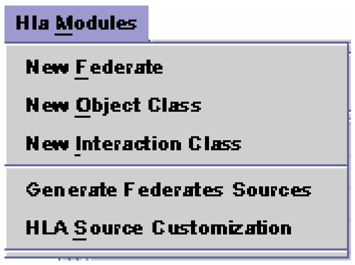


Figure 6-2: A screenshot of new menu items for the HLA extensions

org.argouml.ui.ProjectBrowser is the main class in ArgoUML that needs to be modified in order to access the HLA extensions. This class handles the user interface functions of ArgoUML.

Through the new HLA menu items, some classes are added to support HLA federate design action. All these action classes extend the super class: UMLAction. Figure 6-3 shows the class diagram of the actions classes.

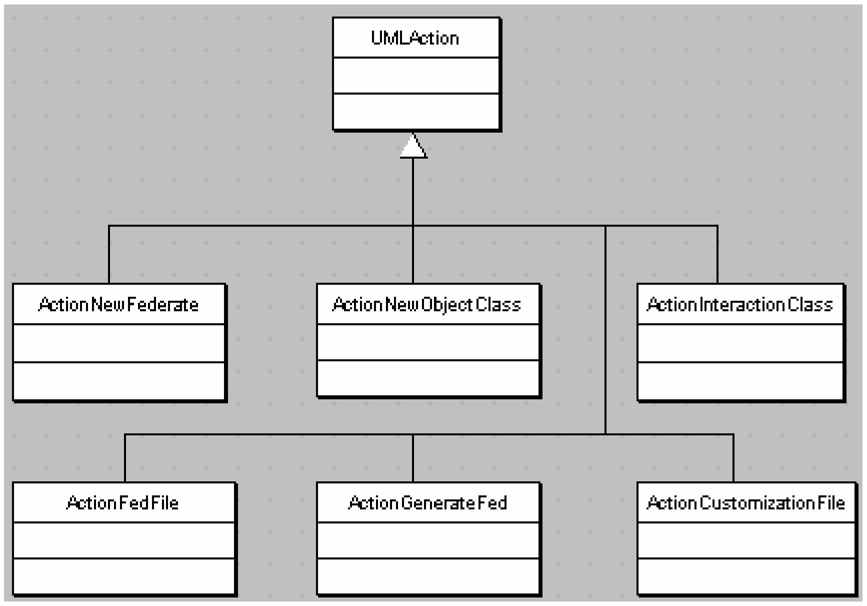


Figure 6-3: A class diagram of action classes

The following collection of the classes support the actions related to the user menu items.

- **org.argouml.uml.ui.ActionNewFederate:** This class carries out the action related to user interface item: **New Federate.** It can load a pre-defined class diagram for federate.
- **org.argouml.uml.ui.ActionNewObjectClass:** This class carries out the action related to user interface item: **New Object.** It produces a new class with HLA specific stereotype name "ObjectClass".
- **org.argouml.uml.ui.ActionInteractionClass:** This class carries out the action related to user interface item: **New Interaction.** It produces a new class with HLA specific stereotype name "InteractionClass".

- **org.argouml.uml.ui.ActionGenerateFed:** This class carries out the action related to user interface item: **Generate Federate Components.** The designer can target the HLA specific class to generate C++ source code skeleton.
- **org.argouml.uml.ui.ActionCustomizationFile:** This class carries out the action related to user interface item: HLA Customization. It is a simple text editor for source code edition.

6.3. Cognitive support for HLA

6.3.1. HLA specific criticism control mechanism

The HLA specific criticism control system involves the class Hla, the class HlaModel, the class GoListToHlaToItem and the class ToDoByHla. Figure 6-4 shows the structure of the HLA specific control system.

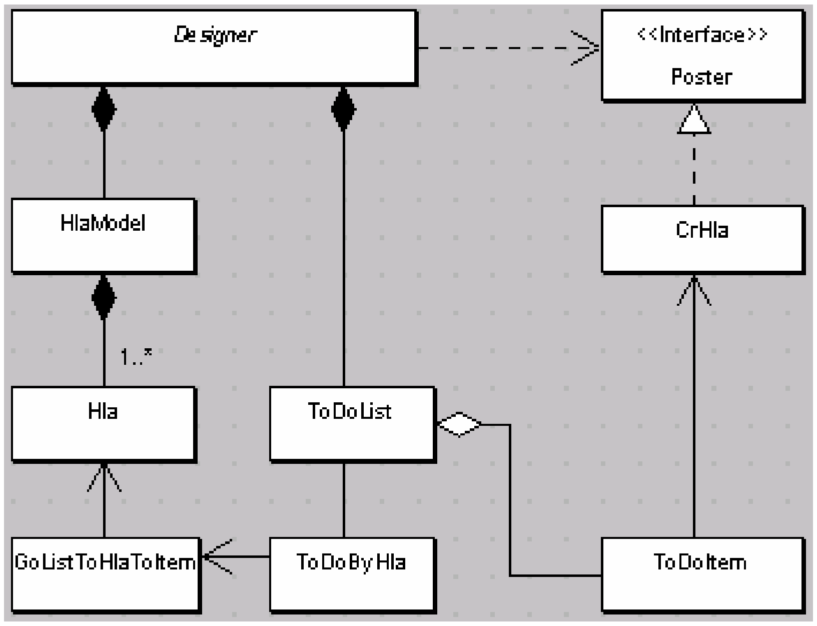


Figure 6-4: A class diagram of criticism control system for the HLA extensions

Class Hla: This class defines basic cognitive decisions related to HLA design.

Class HlaModel: it is a part of class Designer. It describes what types of HLA related design issues should be active. The main functions include:

```
Boolean isConsideringHla(String decision)
void setHlaPriority(String hla, int priority)
void defineHla(String hla, int priority)
void startConsideringHla(String decision)
void stopConsideringHla(String decision)
Hla findHla(String decName)
```

Class ToDoByHla: This class launches GoListToHlaToItem to build the tree view pane. It controls and monitors the HLA specific critics' changes. HLA specific critics produce ToDoItems when they detect design improvement issues. ToDoItems are stored in the designer's ToDo list. When this ToDoItems are not valuable, they are removed from the ToDo list.

Class GoListToHlaToItem: This class implements the interface *TreeModel* to build the HLA extensions tree model.

The original ArgoUML classes: class Designer, class ToDoList, and class ToDoItem are modified to support HLA specific control system. The following are the major modification to these classes:

The class Designer is composed of the basic types of cognitive decisions. The HLA related decision should be added in this class.

The function, boolean ***supports*** (Hla h), is added in the class ToDoItem support the HLA extensions.

The function, Vector ***getHla***() is added in the class ToDoList to access the HLA related critics.

6.3.2. HLA specific critics

HLA specific critics are implemented as the subclass from the super class CrHla.

This class defines three types of HLA related critics:

```
Hla HLA_FederateAmbassador=new Hla("hla.FederateAmbassador",5);
Hla HLA_SimulationOjbect = new Hla("hla.SimulationObject",5);
Hla HLA_SimulationInteraction = new Hla("hla.SimulationInteraction",5);
```

The subclasses override the methods in the base class CrHla to customize a new critic. The method **predicate (Object dm, Designer dsgr)** is overridden to determine which critic should be valid and this critic can appear in the given ToDoItem and be kept in the ToDo list.

The method **setResource(String key)** is overridden to lookup the relevant textual help description posting on the ToDo list.

HLA specific wizards are implemented as the subclasses of the class Wizard. They implement these actions with code specific to each wizard. Designer presses the "Next>" button to move on to the next step of the wizard.

The main classes related to the HLA specific critics involve:

```
Class CrHLAFedAmbChangeName;
Class CrHLAFedAmbNoOverridingOperation;
Class CrHLAFedAmbOperationsMatched;
Class CrHLAFedAmbOperationsNotMatched;
Class CrHLAFedAmbOperationsNewAttributes;
Class CrHLAObjectClassChangeName;
```

Class CrHLAObjectClassNoAttribute;
Class CrHLAObjectClassNoOperation;
Class CrHLAInteractionClassChangeName;
Class CrHLAInteractionClassNoAttribute;
Class CrHLAInteractionClassNoOperation;

All the critics will be posted in the ToDo list pane according to their attributes.

According to the ArgoUML categories of critics, the HLA specific critics can be classified as completeness and consistency critics:

1) Completeness critics

Class CrHLAFedAmbChangeName, Class CrHLAInteractionClassChangeName and ***class CrHLAObjectClassChangeName***: These critics will be invoked if they predict that the default name of the user defined subclass of the class DefaultFederateAmbassador, the user defined interaction class or the user defined objects class need to be changed like "Tank" and "HwFederateAmbassador".

Class CrHLAFedAmbNoOverridingOperation: This critic will be invoked if it predicts that the designer does not override any callback methods which are defined subclass of the class DefaultFederateAmbassador.

Class CrHLAFedAmbOperationsMatched: This critic will be invoked if it predicts that the designer overrides some functions such as ***discoverObjectInstance***(), ***updateAttributeValues***(), and ***receiveInteraction***(), which are inherited from the class DefaultFederateAmbassador.

Class CrHLAObjectClassNoAttribute and ***Class CrHLAObjectClassNoOperation***: These classes will be invoked if they predict that the user defined object class has no member attributes or member functions such as ***Init***() and ***PublishAndSubscribe***().

Class CrHLAInteractionClassNoAttribute and ***Class CrHLAInteractionClassNoOperation***: These classes will be invoked if they predict that the user defined interaction class has no member attributes or member functions such as ***Init***() and ***PublishAndSubscribe***().

2) Consistency critics

Class CrHLAFedAmbOperationsNotMatched: This critic will be invoked if it predicts that the designer adds some new methods which are not defined in class DefaultFederateAmbassador.

Class CrHLAFedAmbOperationsNewAttributes: This critic will be invoked if it predicts that the designer adds some new attribute in the user defined subclass of the class DefaultFederateAmbassador.

Class CrHLAObjectClassNoVirtualOperations: This critic will be invoked if it predicts that the designer does not override the virtual functions in the user defined object class.

6.4. Code generation

In the standard version of ArgoUML, it can generate Java source codes mapping the ordinary class or interface. With the UML stereotype extensions, the HLA specific classes can be differentiated from the ordinary classes. When these HLA specific classes are targeted, ArgoUML can generate HLA C++ source code skeleton.

The class Generator is an abstract base class that defines code generation framework. All the classes related to code generation are the subclass of it.

All the HLA specific classes are implemented using the Singleton pattern to avoid the overhead of being allocated and freed each time new classes are needed.

6.4.1. Class ActionGenerateFed

This code generation class should be valuable only when the targeted class is represented by the user defined object class, the user defined interaction class, the class ObjectClassBase, class DefaultFederateAmabassador or its subclass. The main function is implemented in ***shouldBeEnabled***()

Variable ***nodes*** is used to store all the classes nodes in the active UML class diagram. For storing the default federate ambassador class, object federate ambassador class and object class are used to create the vectors:

```
Vector nodes;
Vector federateObjectClasses;
Vector defaultFedAmbClasses;
Vector objectFedAmbClasses;
Vector federateInteractionClasses;
```

After it collects all the HLA specific classes, it will call class FedGenerationDialog to allow designers to select the federate ambassador, user defined object class and user defined interaction class.

6.4.2. Class FedGenerationDialog

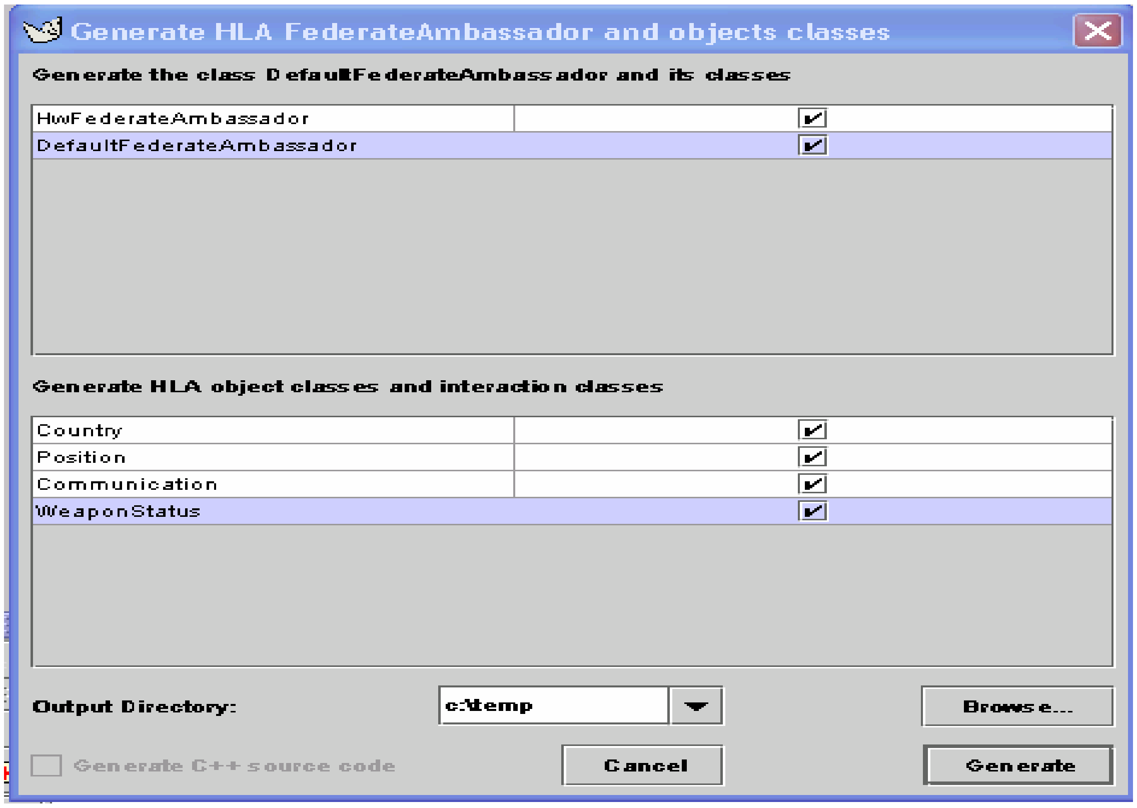
This class constructs the dialog for code generation.

Class FedGenerationDialog builds the main dialog for the designer. Federate ambassador classes, user defined object classes and user defined interaction classes will be located in the different tables.

Class TableModelClassChecks implements table model in a subclass of the AbstractTableModel class. The AbstractTableModel class specifies the methods which the JTable will use to interrogate a tabular data model.

JTable _classFedAmbTable and _classFedObjTable display federate nodes in the table.

JScrollPane _classFedAmbScrollPane and _classFedObjScrollPane provide scrollable view of federate ambassador and object classes. Figure 6-5 shows the code generation dialog for HLA federate.



Generate the class DefaultFederateAmbassador and its classes

HwFederateAmbassador		✓
DefaultFederateAmbassador		✓

Generate HLA object classes and interaction classes

Country		✓
Position		✓
Communication		✓
WeaponStatus		✓

Output Directory: c:\temp

Figure 6-5: A screenshot of the code generation dialog for an HLA federate

6.4.3. Classes of generating C++ source code

There are some main classes to generate the C++ source code: GeneratorObjectClass, GeneratorInteractionClass, GenObjectFederateAmbassador and GenDefaultFederateAmbassador.

Figure 6-6 shows the class diagram of code generation for HLA extensions.

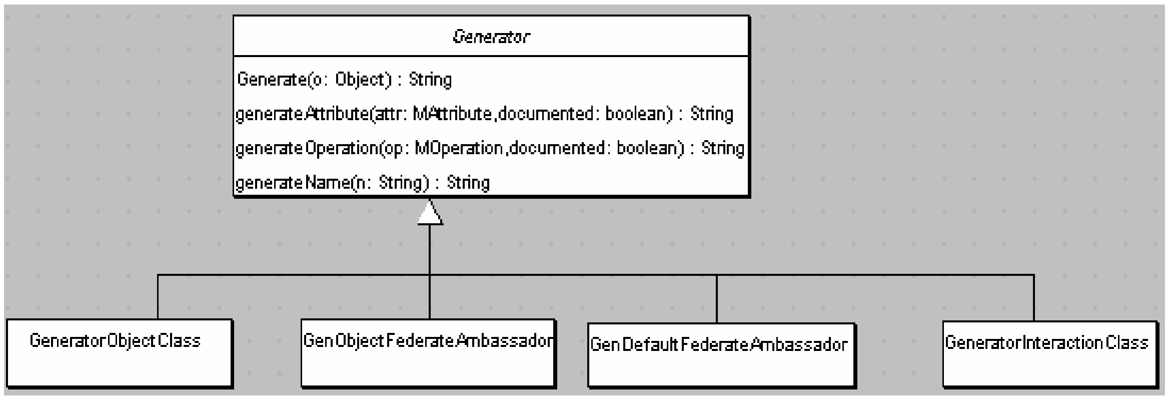

Generator
Generate(o: Object) : String
generateAttribute(attr: MAttribute,documented: boolean) : String
generateOperation(op: MOperation,documented: boolean) : String
generateName(n: String) : String
GeneratorObjectClass
GenObjectFederateAmbassador
GenDefaultFederateAmbassador
GeneratorInteractionClass

Figure 6-6: A class diagram of code generation for the HLA extensions

The class GeneratorObjectClass generates the C++ source code for the class ObjectClassBase and the user defined object class. It contains:

SINGLETON: It creates a single instance of the class GeneratorObjectClass. Using the Singleton Pattern can save run time and have advantages in memory management like garbage collection.

GenerateObjectFed (MClassifier cls, String path): It builds the object class file and object head file.

generateHeader (MClassifier cls, String name,String packagePath): It builds the header for the object class file.

generateAttribute (MAttribute attr, boolean documented): It builds the single object class attribute.

generateOperation (MOperation op, boolean documented): It builds the single object class operation.

generateClassifierBody (MClassifier cls): It builds the main body of the user defined object class file including all the attributes and operations in accordance with C++ grammar. The attributes and operations are classified according to their visibility: Public, Private and Protected. The strings of the attributes and operations are transferred to the class GeneratorObjectClassHead to build the header file.

The class GeneratorObjectClass uses SINGLETON, the static instance of the class GeneratorObjectClassHead, to generate the main body of the object class C++ header file.

The class GenObjectFederateAmbassador, the class GenDefaultFederateAmbassador and the class GeneratorInteractionClass have the similar structure with the class GeneratorObjectClass. They can generate the relevant C++ files and their header files.

The class CustomizationDialog is similar to traditional text editor tool. It can read, write and save the existing federate file. Designers need not leave the ArgoUML environment to edit their federate source code.

Chapter 7. Case studies

7.1 Overview of design and development process

Under the proposed framework and the UML-based CASE tool, ArgoUML, the general federate design and development process is:

Step 1: Through the pre-defined structure in ArgoUML, a designer creates some user defined classes in a class diagram to represent HLA object models and their relationships. These user defined classes extend the fixed classes (class ObjectClassBase and class DefaultFederateAmbassador) for a particular application.

Step 2: For each user defined class, the designer adds the member attributes to map the sharable elements and member operations to encapsulate the HLA low level APIs. During the design phase, ArgoUML issues some cognitive help for federate designers.

Step 3: By using the code generation mechanism in ArgoUML, the designer can generate some source code skeleton from the designed visual models.

Finally, the HLA developers can insert the application-specific logic code into the stub code and compile the complete code in the programming environment with HLA support. The actually executable federate software is deployed in the federation.

Figure 7-1 shows a simplified life cycle of the design and development process.

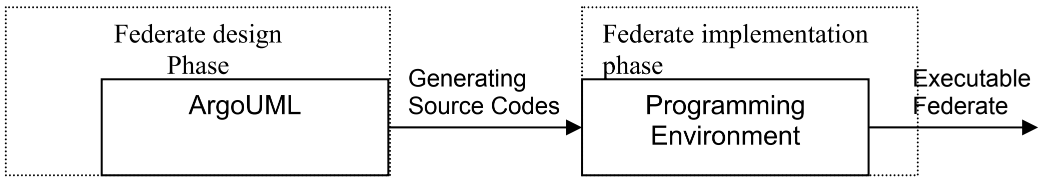


Figure 7-1: A simplified life cycle of the federate design and development process

Two examples, HelloWorld and Tank, have been exercised in the following sections.

The example HelloWorld is a basic federate. It only has the necessary RTI services and no specific operations. The federates just communicate by sending a set of fixed string "Hello World". This example shows the challengeable work of directly manually designing and coding. Thus, it can be improved under the framework with tool support.

Another example: Tank, which is a realistic and simplified example, covers not only the fundamental aspects of the RTI functionality but also the specific logic for actual operations. The tanks can exchange the information belonging to the sharable elements in accordance with the HLA object models. This example shows the complete the design and development process. It also illustrates how to the reuse the components of the existing federate: HelloWorld.

7.2 Example 1: HelloWorld

7.2.1. Overview of HelloWorld's specification

The federate HelloWorld only covers the necessary RTI services including Declaration Management and the Object Management. The simple operation is to send and receive character string “Hello World”. For the purpose of reuse in the following example, it defines two classes: Country and Communication.

- **Object class**: Country, which includes the attributes “Name” and “Population”.
- **Interaction class**: Communication, which includes the parameter, “Message”. It has the string “Hello World”.

Table 7-1 shows the FOM of the federate HelloWorld.

FOM	
Object class	**Interaction class**
Country	**Communication**
Name: String	Message: String
Population: int	

Table 7-1: The FOM of the federate HelloWorld

7.2.2. Challenge without framework

As discussed in the section 3.3 and 3.4, the HLA adds a great amount complexity to a compatible simulation. Even for this simple federate, which does not contain any specific logic code; it should contain hundreds of lines of HLA functionalities’ C++ source code to complete the basic federate execution capability. The HLA developers should face the following challenge during the design and development process without the framework:

1) There is no formal way to design the classes: Country and Communication and the structure of HelloWorld should be described from scratch ;

2) It is a laborious task to build a big monolithic simulation system: HelloWorld because of directly interfacing with HLA RTI low level APIs;
3) The development effort tends to ad-hoc in nature. It is difficult to reuse the components of HelloWorld in the other applications;

7.2.3. Under federate framework

The proposed framework in ArgoUML improves the design process by formal descriptions and graphic user interface environment. The relationships and interactions in the federate HelloWorld are visualized.

1) Structure of HelloWorld

At first, the designer invokes the pre-defined class diagram: selecting "New Federate" item to design this federate. The designers may name the new class Country, the class Communication and the class HwFederateAmbassador. For reusing the structure and code, the abstract class ObjectClassBase is also included in the federate software.

Figure 7-2 shows the example class diagram of the federate HelloWorld. This class diagram is stored as the name "HelloWorld.argo" for the reuse in the future.

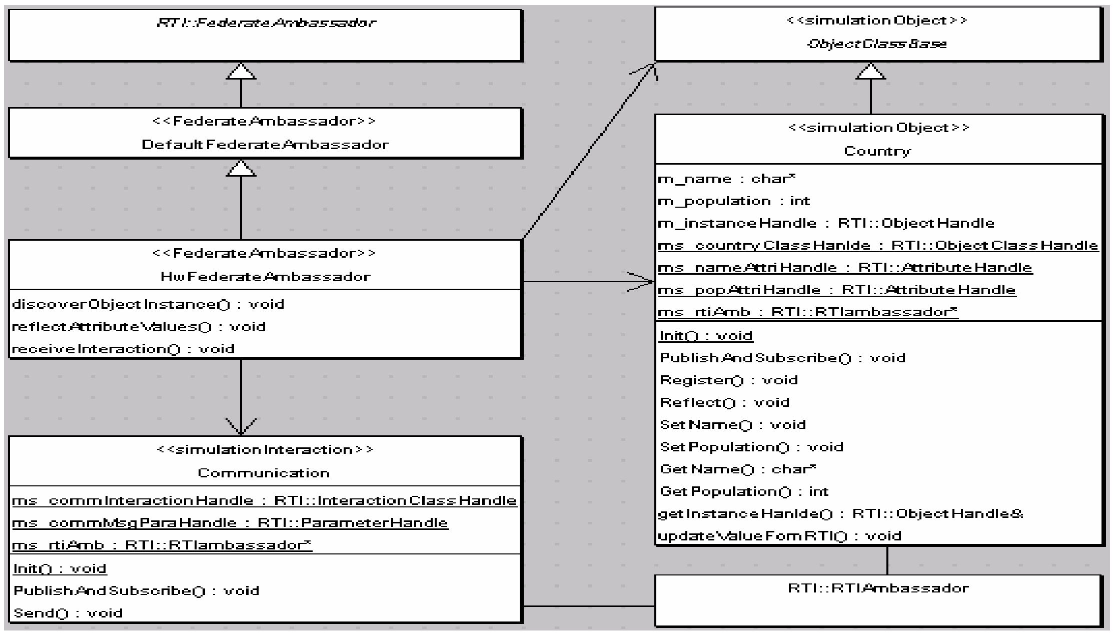

RTI::FederateAmbassador
<<FederateAmbassador>>
DefaultFederateAmbassador
<<FederateAmbassador>>
HwFederateAmbassador
discoverObjectInstance() : void
reflectAttributeValues() : void
receiveInteraction() : void
<<simulationInteraction>>
Communication
ms_commInteractionHandle : RTI::InteractionClassHandle
ms_commMsgParaHandle : RTI::ParameterHandle
ms_rtiAmb : RTI::RTIambassador*
Init() : void
PublishAndSubscribe() : void
Send() : void
<<simulationObject>>
ObjectClassBase
<<simulationObject>>
Country
m_name : char*
m_population : int
m_instanceHandle : RTI::ObjectHandle
ms_countryClassHanlde : RTI::ObjectClassHandle
ms_nameAttriHandle : RTI::AttributeHandle
ms_popAttriHandle : RTI::AttributeHandle
ms_rtiAmb : RTI::RTIambassador*
Init() : void
PublishAndSubscribe() : void
Register() : void
Reflect() : void
SetName() : void
SetPopulation() : void
GetName() : char*
GetPopulation() : int
getInstanceHanlde() : RTI::ObjectHandle&
updateValueFomRTI() : void
RTI::RTIAmbassador

Figure 7-2: A class diagram of federate HelloWorld

2) User defined classes in the diagram

• Class Country

It is the subclass of the class ObjectClassBase. It defines the member attributes, the "name", "population", and relevant handles as member attributes.

The class Country defines some member functions to encapsulate the necessary RTIambassador methods and access the member attributes including *Init*() *PublishAndSubscribe*(), *Register*(), *Reflect*(), *GetName*(), *GetPopulation*(), *SetName*(), *SetPopulation*() and *getInstanceHandle*() .

• Class Communication

It defines the handles of Communication and Message as member attributes. It does not define Message as the member attribute because the string "Hello World" is sent and forgotten. It also defines some member functions to encapsulate the necessary RTIambassador methods including *Init*(), *PublishAndSubscribe*() and *Send*().

- **Class HwFederateAmbassador**

The designer considers overriding some callback methods in the subclass HwFederateAmbassador. According to the HelloWorld specification, three methods should be overridden including: *discoverObjectInstance*(), *reflectAttributeValues*() and *receiveInteraction*().

The designer can generate the necessary C++ source code skeleton from the class diagram. Once the stub code is generated, all that remains is to add the necessary implementations. As mentioned in the section 4.2.5, federate HelloWorld needs a simulation execution program. This program initializes the class Country, the class Communication and the class HwFederateAmbassador and invokes the RTI service like creating the federation execution, joining, resigning and deleting the federation.

In the simulation execution environment, a federate HelloWorld can join a given federation and just sends or receive the “Hello world!”.

7.3 Example 2: federate Tank

7.3.1. Application scenario

The federate Tank is abstracted from the real battlefield system. In the simulation scenario, each tank belongs to a specific country which is selected by the user. A tank is armed with some ammunition and moves in the 2D dimension.

The simulation system can contain more tanks. A tank can know the positions and country's names of any others. The tank can send and receive the friendly information from others. In addition, it may fire at another enemy tank.

The proposed federate Tank is a simple simulation and just used for analysis studies. The complete implementation of the Tank simulation product such as 2D View Displays, motion platforms and fully populated control panels are beyond this project.

7.3.2. Overview of Tank's specification

The federate Tank is a more complicated federate than HelloWorld. For illustrating the design and code reuse of the proposed design framework, this federate has one new HLA object class Position and HLA interaction class WeaponStatus based on the existing federate HelloWorld.

Table 7-2 shows the FOM of the federate Tank.

FOM	
Object class	**Interaction class**
Country	**Communication**
Name: String	Message: String
Population: int	
Position	**WeaponStatus**
pos_x: double	FireLevel: int
pos_y: double	

Table 7-2: The FOM of the federate Tank

In this form, the existing HLA object class: Country and its attribute Name represent the nationality of a tank. The interaction class: Communication, which includes the parameter, “Message”, represents the exchangeable friendly information between the tanks.

The additional sharable elements of the FOM in the federate Tank are:

- **Object class**: Position, which includes the attributes “pos_x” and “pos_y”. These values indicate the physical position of the tank.
- **Interaction class**: WeaponStatus, which includes one parameter, “FireLevel”. This value represents the ammunition of the tank.

The relevant operations of a tank are described below:

- An instance of the class Country set the name of the tank. A tank can know country’s names of other tanks by reflecting the attribute value “Name”.
- An instance of the class Position changes its position “pos_x” and “pos_y” value according to user control from the PC console. A tank can also view the positions of other tanks.
- The tanks can exchange the friendly information “Hello World” by sending and receiving the HLA interaction value class Communication and its parameter Message.
- The federate Tank sends its HLA interaction class WeaponStatus and parameter FireLevel, and receives by other federates. The ammunition is a fixed integer value “100”. When a tank fires at another one, this value will be sent and forgotten.

7.3.3. Tank design process

1) Design the structure of Tank

The designer can invoke the existing class diagram "HelloWorld.argo" and add the new classes: the class Position, the class WeaponStatus. Designing process is similar to the federate HelloWorld.

Figure 7-3 shows the example class diagram of the federate. This class diagram is stored as the name "Tank.argo" for reuse in the other federation.

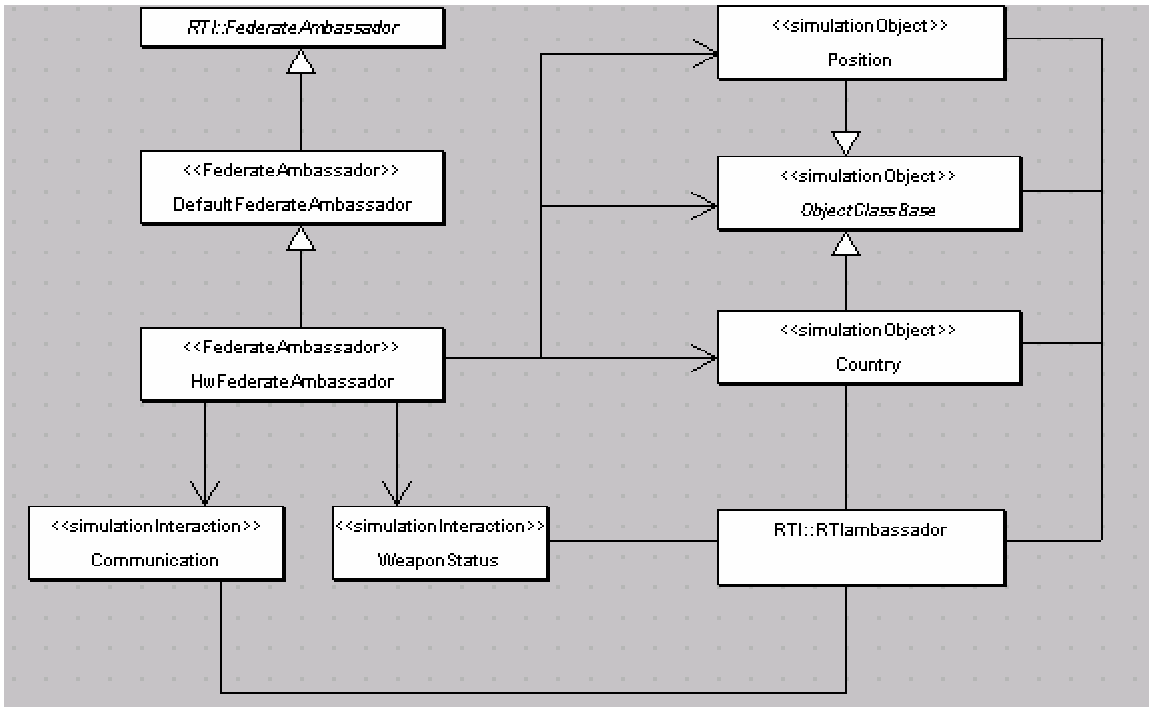


Figure 7-3: A class diagram of federate Tank

2) Design each user defined class in the diagram

• Class Position

It is inherited from the abstract class ObjectClassBase. It defines the member attributes: “m_pos_x”, “m_pos_y”, and object instance handle “m_instanceHandle”.

It defines some member functions to encapsulate the necessary RTIambassador methods. Particularly, it also defines some member functions to read and write the member attributes. For example, functions *Getpos_x*(), *Getpos_y*(), *Setpos_x*() and *Setpos_y*() are used to reflect and update the location of a tank.

- **Class WeaponStatus**

It defines the handles of WeaponStatus and FireLevel as member attributes. It need not define its parameter FireLevel as the member attribute because the ammunition is sent and forgotten. It also defines some member functions to encapsulate the necessary RTIambassador methods covering Declaration Management and Object Management. They are *Init*(), *PublishAndSubscribe*() and *Send*(). For example, when a tank fires at another one, the function *Send*() will be invoked.

- **Class HwFederateAmbassador**

In the design phase, designers can totally reuse this class in the existing federate HelloWorld. Three callback methods are still kept including: *discoverObjectInstance*(), *reflectAttributeValues*() and *receiveInteraction*().

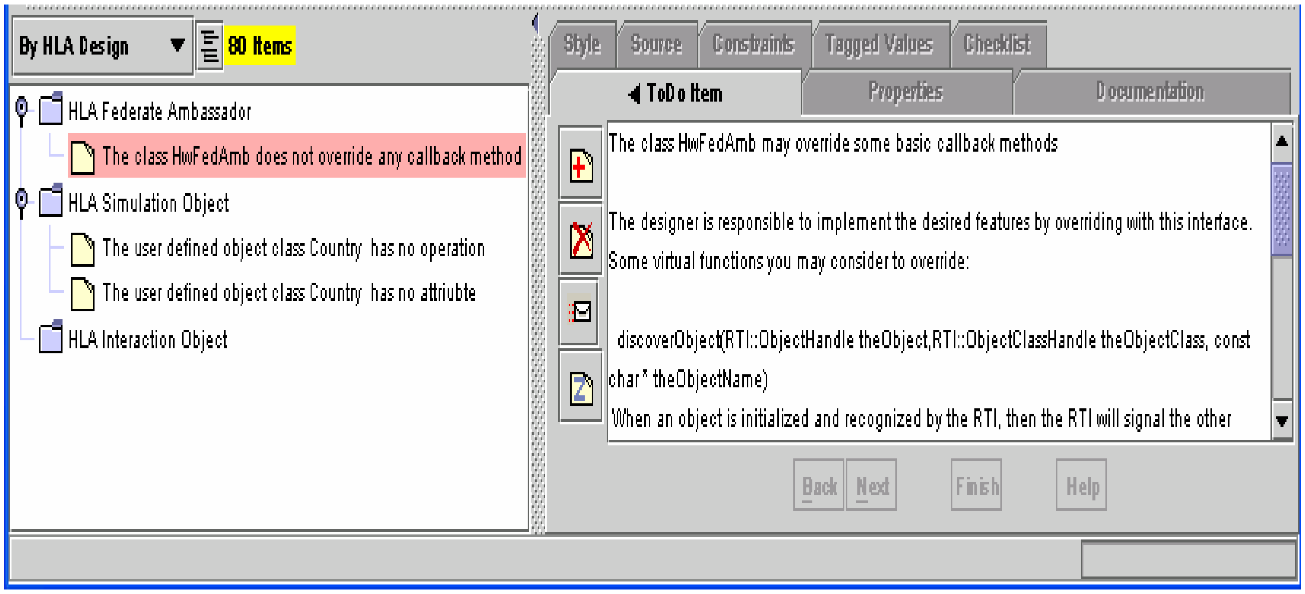
- ➢ Function *discoverObjectInstance*() will be invoked when another federate Tank joins the simulation system.
- ➢ Function *reflectAttributeValues*() will be invoked to inform some tank changes its name or position.

- Function *receiveInteraction*() will be invoked if a friendly tank send "Hello World" or an enemy tank fire the ammunition.

7.3.4. Cognitive help for federate Tank

As discussed in the section 5.3, during the federate design phase, the cognitive support helps the design decision making. When the HLA specific critic predicts some design problems, it will produce the ToDoItems and post on the ToDoList. For example, if the users override the callback methods: *updateAttributeValues*() and *receiveInteraction*() in the class HwFederateAmbassador, it produces some reminding information to help users to use these methods.

Figure 7-4 shows an example HLA specific critic.



HLA Federate Ambassador
The class HwFedAmb does not override any callback method
HLA Simulation Object
The user defined object class Country has no operation
The user defined object class Country has no attriubte
HLA Interaction Object

The class HwFedAmb may override some basic callback methods

The designer is responsible to implement the desired features by overriding with this interface.
Some virtual functions you may consider to override:

discoverObject(RTI::ObjectHandle theObject,RTI::ObjectClassHandle theObjectClass, const char* theObjectName)
When an object is initialized and recognized by the RTI, then the RTI will signal the other

Figure 7-4: A screenshot of the an HLA specific critic

7.3.5. Tank development process

1) Code generation for Tank

The designer targets any component in the class diagram and generates necessary C++ source code skeleton for the federate Tank. The files:

WeaponStatus.cpp and **WeaponStatus.hh**

Position.cpp and **Position.hh**

DefaultFederateAmbassador.cpp, and **DefaultFederateAmbassador.hh,**
HwFederateAmbassador.cpp and **HwFederateAmbassador.hh**

Communication.cpp and **Communication.hh**

Country.cpp and **Country.hh**

The existing files Communication.cpp, Communication.hh, Country.cpp and Country.hh can be totally reused in the federate Tank. All these files contain the C++ source template codes for each class. The developers need to complete the files: Position.cpp and WeaponStatus.cpp

For example, the following C++ code shows the main body of file Position.cpp:

```
//main stub code for class Position
RTI::ObjectHandle&      GetPositionHandle() { return m_instanceHandle; };
void Position::Setpos_x( const double& pos_x ){
//Need to be implemented the function for class Position in the future
}
void Position::Setpos_y( const double& pos_y ){
//Need to be implemented the function for class Position in the future
}
```

Once the source code is generated, all that remains is to add the necessary functionality into the skeleton implementations.

2) Code implementation process

- **Classes Position and WeaponStatus**

Because a tank has real operations as defined before, the developer should the relevant implementation into these files.

For example, the following C++ code shows that a tank will view the position of the other tanks in the method *updateValueFromRTI*():

```
// to decide which attribute should be updated
if ( attrHandle == Position::Getpos_xHandle() )
    {// to update Pos_x
      double pos_x;
      theAttributes.getValue( i, (char*)&pos_x, valueLength );
      cout << "new pos_x value from proxy: " << pos_x << endl;
    }
// same as above goes to update Pos_y
```

- **Class HwFederateAmbassador**

In the file HwFederateAmbassador.cpp file, the function *reflectAttributeValues*() automatically supports any subclass of the class ObjectClassBase. The developer can reuse this function. For the federate Tank, it defines a void pointer array as void * g_EntityInstances[ ]. This array stores the local proxy object instances for all the HLA object class no matter which class it belongs to. It makes the simulation software more scalable.

The following is the C++ code of the function *reflectAttributeValues*() :

```
// Lookup for each local proxy object instance
if( g_EntityInstances[i] != 0 ) {
        ObjectClassBase* pos= (ObjectClassBase*) g_EntityInstances[i];
if( theObject == pos->getInstanceHandle() ) {
// Update the attributes values
pos->updateValueFromRTI(theAttributes);
        }
    }
```

In the function *discoverObjectInstance*(), the developers may implement necessary function to identify the class Country and class Position.

The following is the C++ code of the function *discoverObjectInstance*():

```
// if the registering object belongs to class Position,
//create the local proxy of class Position to store;
if ( theObjectClass == Position::GetPositionHandle() )
        Position*pPosition = new Position(theObject);
        // store this object instance in a pointer array;
        g_EntityInstances[i] = pPosition ;

// same as above if the registering object belongs to class Country
```

Similarly, the developer may implement the function to identify the class Communication and class WeaponStatus. Because the class WeaponStatus is a transient data object, the receiving data can be stored in a temporary variable in the method *receiveInteraction*().

The following is the C++ code of this method:

```
// if sending data from the class WeaponStatus
//store this interaction in a temporary variable;
if ( paramHandle == WeaponStatus ::GetFireLevelHandle() )
{
        char msg[ 1024 ];
        theParameters.getValue( i, (char*)msg, valueLength );
}
```

```
// same as above if sending data from the class Communication
```

3) Simulation execution program

This program defines the relevant functions to operate the array void * g_EntityInstances[ ]as:

```
void addEntity(void * oEntity);
void deleteEntity(void * oEntity);
```

The following is the C++ code of function *addEntity*():

```
void addEntity(void * oEntity) {
  for(int i=0; i < MAX_ENTITY_NUM; i++) {
    if( g_EntityInstances[i] == 0) {
             g_EntityInstances[i] = oEntity;
             break;
    }
  }
}
```

There is the similar C++ code in the function *deleteEntity*().

This program also added the necessary code to allow the class Position and class WeaponStatus to join, resign and delete the federation. In the simulation execution environment, a federate Tank joins the existing federation firstly. It has a given name which is set by the attribute of class Country. It can move by updating “pos_x” and “pos_y” value according to user control. If another tank registers and joins the same federation, the previous one can discover its position and decide to fire at it by sending class WeaponStatus and its parameter FireLevel or say “Hello World” by sending class Communication and its parameter Message.

Chapter 8. Conclusion and future work

8.1. Conclusion

The main research work described in this thesis can be concluded into two parts: the basic federate design framework and its supporting tool.

1) Designing a basic federate design framework

Currently, there are two main problems related to federate development.

- ➢ The federate implementation is a tedious work because the HLA APIs are huge and monolithic.
- ➢ There are many open problems for the design of the federates.

This project proposes a basic design framework to solve both the problems. This framework provides the architectural guideline for the designers to build the federate software. It supports reuse of the existing federate structure and code in a given federation.

The following are the framework components and their services:

- ➢ The user defined object class and the user defined interaction class contain the representations of HLA sharable elements and their handles in accordance with the HLA object model. These classes also encapsulate some basic RTI APIs and their implementation.
- ➢ The class ObjectClassBase provides the dynamic class loading function to reflect the remote object attributes updating.

- The class DefaultFederateAmbassador provides the default implementation for each abstract callback method. The users can define its subclass to do its own work.

2) Implementing a supporting tool

The federate design framework is useful only if it is supported by a set of tools. The UML diagrams are not sufficient to support the proposed framework because the user does not know how to use it. Thus, a CASE tool, ArgoUML, is extended to support the framework. It includes:

- UML stereotype extensions support HLA specific classes' representations. These new stereotypes differentiate the HLA specific class and ordinary class in the federate framework.
- A default class structure which supports the federate framework is added in the ArgoUML.
- Cognitive help system support to use the federate framework. It will produce the ToDoItems to help designers to make relevant decisions.
- C++ template code generation and source customization editor tool is provided for the future implementation work.

The supporting tool, ArgoUML, results in a significant reduction of the designing work.

It can also automatically generate source code skeleton for the federate software. This will improve the productivity of the development process.

8.2. Future work

This project has started to be a foundation for future work of the complete federate framework. Some extensions of the research work may be considered.

1) Consider a more flexible framework

The proposed framework is binding with FOM. Reuse of a federate under this framework usually depends on the correctness of the upfront FOM specification. A more flexible framework can automatically publish, subscribe, send, and receive sharable elements in accordance to SOM. It can achieve more reusability than the proposed framework. But some conditions should be considered in this possible work:

- Such framework becomes more complex. The federate software also should involve more codes for a federate. If there are too many HLA object classes and HLA interaction classes in FOM, the system performance will become a bottleneck under this framework and it increases the code maintenance effort. Thus, the tradeoff between the development productivity versus the HLA application requirement and simulation execution environment factors should be considered by the simulation users and developers.
- In some simulation application scenario, if there is no need for rebuilding FOM or a new FOM is compatible, such flexible framework will not improve the productivity of development and the software reusability than proposed framework.
- This flexible framework is not useful without a set of tools to support automatic mechanism. As ArgoUML is only for software design phase and it is not very

stable at the current stage, it is not suitable to be extended to support more complex functions. So the developers need to consider implementing the new supporting tools.

2) Consider more complete software framework

Due to the time constraints, the proposed federate framework only covers the basic necessary functionalities of the federate. In some situations, the complex federate software involves the optional RTI services such as Data Distribution Management (DDM), Ownership Management, and Time Management. The proposed federate design framework can be extended to the other aspects of the federate software implementation. Moreover, these services are also related the federation management. Thus, the more complete software framework, called “Federation framework” can cover all these services. The proposed basic design framework is the first step of this future work.

Appendix A: An introduction to UML notation

A.1 UML diagrams

UML diagram	Notation
Class Diagram	A class diagram is a collection of static declarative model elements, such as classes, interfaces, and their relationships, connected as a graph to each other and to their contents.
Use Case Diagram	A use case diagram is a graph of actors, a set of use cases, possibly some interfaces, and the relationships between these elements. The relationships are associations between the actors and the use cases, generalizations between the actors, and generalizations, extend, and includes among the use cases.
Sequence Diagram	A sequence diagram has two dimensions: the vertical dimension represents time, and the horizontal dimension represents different instances. Normally time proceeds down the page. (The dimensions may be reversed, if desired.) Usually only time sequences are important, but in real-time applications the time axis could be an actual metric. There is no significance to the horizontal ordering of the instances.
Collaboration Diagram	A collaboration diagram shows a graph of either Instances linked to each other, or ClassifierRoles and AssociationRoles; it may also include the communication stated by an Interaction or InteractionInstanceSet.
Statechart Diagram	A statechart diagram is a graph that represents a state machine. States and various other types of vertices (pseudostates) in the state machine graph are rendered by appropriate state and pseudostate symbols, while transitions are generally rendered by directed arcs that interconnect them.
Activity Diagram	An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states.
Component Diagram	A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships.
Deployment Diagram	A deployment diagram is a graph of nodes connected by communication associations. Nodes may contain component instances. This indicates that the component runs or executes on the node. Components may contain instances of classifiers, which indicates that the instance resides on the component.

Table A- 1: UML diagrams

A.2 Class diagram

A class diagram describes the static overview of the systems through showing the classes and their architecture. It may also contain classes, interfaces, packages, and relationships.

UML defines some relationships including association, generation, dependence, composition and aggregation in the class diagram. In this project, two types of relationships are used in the class diagram: association, and generalization. The other relationships can be found in [BOOC99].

Association -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes. Figure A-1 shows that class Order is associated to class Customer.

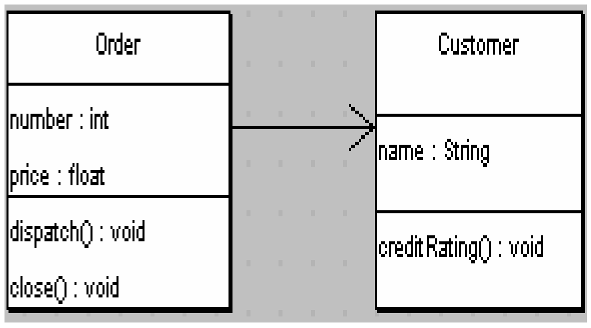


Figure A- 1: An example association relationship

Generalization -- an inheritance link indicating one class is a superclass of the other classes. A generalization has a triangle pointing to the base class. All base class attributes and operations are also part of the subclass.

Figure A-2 shows that class DataInputStream is generalized from the class InputStream.

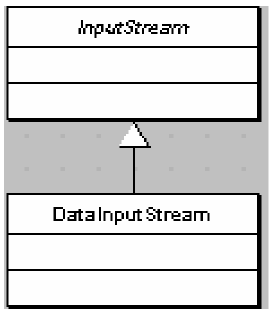


Figure A- 2: An example generalization relationship

A.3 Sequence diagram

A sequence diagram is an interaction diagram that details how operations are carried out. It describes how objects collaborate through an exchange of messages. Sequence diagrams are organized according to time. The objects involved in the operation are listed from left to right according to when they take part in the message sequence. A single sequence diagram often represents the flow of events for a single use case. Figure A-3 shows an example of a sequence diagram for using ATM system.

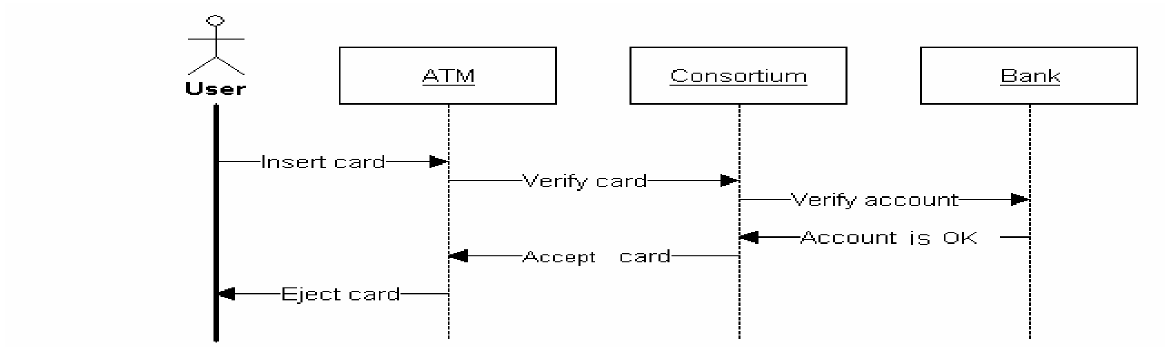


Figure A- 3: A sequence diagram of using ATM

The details about sequence diagram and how to create sequence diagrams can be found in [BOOC99] [SINA02].

Appendix B: A cognitive CASE tool: ArgoUML

B.1 ArgoUML's graphic user interface

1) Argo Menu

The Argo Menu is located at the top of the ArgoUML. The menu consists of File, Edit, View, Arrange, Create, Generate, Critique, and Help items. It provides all the command and control mechanisms for the users. For example, from the "Create Diagram" item, designer can create UML diagram and put necessary components for the design purpose.

2) Argo Navigator pane

The Navigator Pane is located on the upper left part of the window. It lists the contents of the diagrams and objects of the model that the users are selected. The designer can view the structure of the diagram through the predefined tree like perspective. ArgoUML provides multiple explorer perspectives. The designer can set the choice by ***choice menu*** at the top of the explorer. Each perspective shows a hierarchical view of the design. For example, the Diagram-centric perspective shows the design structure according to the type of diagrams like class diagram, use case diagram etc.

3) Argo Editing pane

The Editing Pane, which is the user work field, is located on the upper right area of the window. This is where all the design diagrams are edited. The designer can use drag-drop action or quick-links to create new objects in the diagrams.

The editing pane has a tool bar at the top and it provides main shortcut keys for the editing pane. The toolbar at the top of the editing pane provides the main functions of the pane.

The tools can be divided into four categories.

- ***Layout tools:*** They are used to assist in laying out artifacts on the diagram.
- ***Annotation tool:*** It is used to annotate artifacts on the diagram.
- ***Drawing tool:*** They are used to add general graphic artifacts to diagrams..
- ***Diagram specific tools:*** They are used to add UML artifacts specific to a particular diagram type to the diagram like Use case diagram, Class diagram, sequence diagram, state diagram, collaboration diagram, activity diagram, and development diagram.

4) Argo Details pane

The Details Pane is located on the lower right part of the window. It describes the details of various contents of the components in the diagrams. It contains several tabs to update the selected target. These tables are *ToDoItem* tab, *Properties* tab, *Documentation* tab, *Style table*, *Source* tab, *Constrains* tab, *Tagged Value* tab and *Checklist* tab.

- ***ToDo Item tab:*** It describes the selected ToDoItem in the "To Do" Pane. It presents the design problem and possible solution in the short paragraph. For some problems, corrective wizard can be available to lead designer to fix the problem through the “Next” and “Back” button.
- ***Properties tab:*** It describes properties of the selected UML model element. For example, when some class element is selected, the contents of this tab include Name, Stereotype, Namespace, Modifier, attributes, operations and relationship with other model elements. The contents of this tab vary much based on the different types of the selected elements.

- ***Documentation tab:*** It allows the designer to input the basic document for the selected element. The document includes Author, Version and other related information.
- ***Style tab:*** It can be used to configure the selected target style. Designer can enter the selected target background color type, line type and shadow type.
- ***Source tab:*** It previews the selected class, interface or package Java source skeleton codes which will be generate in the future. For the selected class or interface, it presents the attributes, operations and associations of this class. For the selected package, it presents the contents of the classes or interfaces which are included in this package. If there is no any class or interface in this package, only the package name and empty body left.
- ***Constraints tab:*** It allows the designer to enter the OCL constraints on the selected target so that it can have additional meaning. Syntax assistant function is provided for this tab to check the OCL syntax and save the constraints.
- ***Tagged Value tab:*** It allows the designer to enter the tagged value on the selected element. Tagged values are value pairs to store and system will not interpret it.

B.2 Overview of packages in ArgoUML

Here is a brief explanation of the main packages in the ArgoUML.

- **org.argouml**

This package contains all the classes for ArgoUML. The HLA extensions will be created as a child-package of the ArgoUML package.

- **org.argouml.application**

This package provides general classes and interfaces that are fundamental to ArgoUML and other ArgoUML modules.

- **org.argouml.kernel**

This package contains the core class of Argo: the Project class. ArgoUML uses this class to manage the project.

- **org.argouml.pattern**

This package contains Critics which deal with patterns. Currently this includes the critics for recognizing whether a class violates the Singleton pattern, and one Critic to check whether a user should consider using the Singleton pattern for a class.

- **org.argouml.language**

This package defines representation of a model fragment and converts model into textual representations. Currently, ArgoUML only supports the generation of Java source code from UML class diagram.

- **org.argouml.persistence**

This package is used for setting up MySQL. It contains the functionality to load a model from a MySQL database and write a model into a mysql database.

- **org.argouml.uml**

This package includes the classes that relates to the UML notation in ArgoUML including UML different types of Diagrams, connection between the chart and the model, generation of code and the reverse engineering and Panels of property, allowing user to control the elements of the model as well as the various elements which are posted there.

- **org.argouml.i18n**

This package contains resource bundle that provides strings for UML related critiques and check lists.

- **org.argouml.util**

This package contains utilities to provide operating system independence and extensions to logging packages `log4j` and `java.util.logging`.

- **org.argouml.images**

This package contains the icons which are used in the ArguUML.

- **org.argouml.ocl**

This package defines the methods to support OCL notation. It makes it possible support language OCL of description of constraints.

- **org.argouml.swingext**

This package contains a collection of utility methods for Swing Actions and Dimensions including ArrowButton, ArrowIcon, BorderSplitPane and layout design components.

- **org.argouml.xml**

This package contains parsers to load and write file in graphical representation: PGML (Precision Graphics Markup Language).

ArgoUML can directly load the files argo or xmi but a format of file was created: zargo.

Appendix C: HLA Terminology

A ***federation*** is the combination of a particular FOM, a particular set of interoperating simulations, and the RTI services.

A ***federate*** is one simulation that operates in a federation.

A ***Simulation*** is a synonym of Federate.

A ***federation execution*** is a session of a federation executing.

The ***RTI*** is supporting software that allows the federates to communicate and cooperate with each other.

The ***Object Model Template (OMT)*** is a system for documenting objects in the world

The ***Federation Object Model (FOM)*** is a common object model for exchanging data among simulations.

The ***Simulation Object Model (SOM)*** describes the federate Objects and Interactions. All the SOMs of all Federates together constitute the Federation Object Model (FOM).

An ***Object*** is instance of a class and handled by the RTI.

An ***Interaction*** is collection of data sent out at one time through the RTI to other federates.

An ***Attribute*** is a set of data belonging to an instance of a class of type ***Object***. The datum can be of any type including integer, float, double and string and have defined cardinality.

A ***parameter*** is a set of data belonging to an instance of a class of type ***Interaction***. The datum can be of any type including integer, float, double and string and have defined cardinality.

Reference

[ARGO03] *ArgoUML tool*, 2003, website: http://argouml.tigris.org

[BANK91] Banker, R.D. and Kauffman, R.J. *Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study*, MIS Quarterly, (15:3), September 1991, pp. 375-401.

[BAUD96] Baudoin, Claude & Hollowell, Glenn. *Realizing the Object-Oriented Lifecycle,* Upper Saddle River, NJ: Prentice Hall, 1996.

[BECK89] K. Beck and W. Cunningham. *A Laboratory for Teaching Object-Oriented Thinking*, OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications, 1989, pp. 1- 6.

[BELL97] David Bellin and Susan Suchman Simone. *The CRC Card Book*, Addison Wesley Professional, 1997.

[BOEH96] B.E. Boehm and W. Scacchi. *Simulation and Modeling for Software Acquisition (SAMSA),* Final Report, Center for Software Engineering, University of Southern California, Los Angeles, CA, http://sunset.usc.edu/SAMSA/samcover.html, March 1996.

[BOLO98] Boloix, G.; Robillard, P.N. *CASE tool learnability in a software engineering course*, Education, IEEE Transactions on Volume: 41 Issue: 3 , Aug. 1998, pp. 185 -193.

[BOOC99] Booch G, Rumbaugh J, and Jacobson I. *The Unified Modelling Language User Guide*, Addison-Wesley, 1999.

[BRAT87] Bratley, P., B. L. Fox, and L. E. Schrage. *A Guide to Simulation, Second Edition*, Springer-Verlag, 1987.

[CASE03] *CASE tool index,* 2003, website: http://www.cs.queensu.ca/Software-Engineering/tools.html.

[COAD90] Coad, P., Yourdon, E. *Object Oriented Analysis*, Prentice Hall, 1990.

[COST94] Costain, G. *A comparison of CASE-based O-O methodologies: Coad/Yourdon OOA and Booch OOD,* Software Education Conference, Proceedings, 22-25 Nov. 1994, pp.120-127.

[COX998] Cox, K. *A Framework-based Approach to HLA Federate Development,* Simulation Interoperability Workshop, September 14-18, 1998.

[DAHM97] Judith S. Sahmann, Richard M. Fujimoto and Richard M.Weatherly. *The Department of Defense High Level Architecture*, Proceedings of the 1997 Winter Simulation Conference, 1997, pp. 142-149.

[DANE96] Daneva, M and R. Terzieva. *Assessing the Potentials of CASE-Tools in Software Process Improvement: A Benchmarking Study*. In Proceedings of the Fourth International Symposium on Assessment of Software Tools, Toronto, Ontario, Canada, 22-24 May 1996.

[DMS98A] Defense Modeling and Simulation Office. *High Level Architecture Rules*, Version 1.3, 5 February 1998.

[DMS98B] Defense Modeling and Simulation Office. *High Level Architecture Interface Specification*, Version 1.3, 5 February 1998.

[DMS98C] Defense Modeling and Simulation Office. *High Level Architecture Object Model Template*, Version 1.3, 5 February 1998.

[DMSO97] Defense Modeling and Simulation Office. *Test Procedures For High Level Architecture Object Model Template*, Version 1.1, 1997.

[DMSO00] Defense Modeling and Simulation Office. *High Level Architecture Run-Time Infrastructure RTI 1.3-Next Generation Programmer's Guide Version 4*, 2001,

[DOD994] DoD Directive 5000.59. *DoD Modeling and Simulation (M&S) Management,* January 4, 1994.

[DOUG99] Douglas, L. *Learning object-oriented software design at a distance*, Frontiers in Education Conference, FIE '99. 29th Annual, 1999.

[DSOU99] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML, the Catalysis Approach*, AddisonWesley, 1999.

[FISH95] P. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds,* Prentice-Hall, 1995.

[G52A03] *An Introduction to Object Oriented Methodolgy (OOM),* 2003*,* website: http://www.itsd.gov.hk/itsd/english/itgov/download/g52a.pdf

[GAMM95] *Erich* Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Massachusetts, 1995.

[GRAP03] *Graph Editing Framework*, 2003, website: http://gef.tigris.org

[GUIN87] Guindon, R., Krasner, H., and Curtis, W. *Breakdown and processes during early activities of software design by professionals,* In: Olson, G. M. and

Sheppard S., eds. Empirical Studies of Programmers: Second Workshop, Norwood, NJ: Ablex Publishing Corporation. 1987, pp. 65-82.

[HEND92]Henderson-Sellers, B. *A Book of Object-Oriented Knowledge*, Prentice Hall Inc, 1992.

[HERZ94] Herzwurm, G., A.Hierholzer, M.Kung. *The Appropriateness of the Conventional and Objectoriented CASE-tools to Construct a Quality Management Systems According to ISO 9000,* Journal of Information Management, No.3, 1994, pp.72-76.

[IEEE89] IEEE Std 610.3-1989, *IEEE standard glossary of modeling and simulation terminology***,** 15 May 1989.

[IEEE96] IEEE Std 1348-1995, *IEEE recommended practice for the adoption of Computer-Aided Software Engineering (CASE) tools***,** 10 April 1996.

[JOHN88] Ralph Johnson and Brian Foote. *Designing Reusable Classes,* Journal of Object-Oriented Programming, SIGS, 1988, pp. 22-35.

[JUNI03] JUnit. *Testing Resources for Extreme Programming ,* 2003*,* website: http://www.junit.org

[KEIT02] Keith-Magee, K., Parr, S. *Visualising Distributed Simulation Design and deployment*, Proceedings of the Interservice/Industry, Simulation and Education Conference (IITSEC) 2002, Paper ID 258, 2002.

[KUHL99] Kuhl, F., Dahmann, J. and Weatherly, R. *Creating computer simulation systems: an introduction to the high level architecture,* Prentice Hall PTR, 1999.

[MARC03] Marcus Eduardo Markiewicz, Carlos J.P. Lucena. *Object oriented framework development*, ACM Crossroads 2003, website: http://www.acm.org/crossroads/xrds7-4/frameworks.html

[MARK03] Marcus Eduardo Markiewicz and Carlos J.P. Lucena. *Object Oriented Framework Development,* 2003, website: http://www.acm.org/crossroads/xrds7-4/frameworks.html

[MART96] Martin, James and James J. Odell. *Object-Oriented Methods: A Foundation*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[MODE03] *Model resources*, 2003, website: http://members.aol.com/lpang10473/sim.htm

[NOVO03] *Novosoft UML API*, 2003, website: http://nsuml.sourceforge.net

[ODEL98] James J. Odell. *Advanced object-oriented analysis and design using UML*, Cambridge University Press, 1998, pp. 229-232.

[OMAN90] P. W. Oman. *CASE Analysis and Design Tools*, IEEE Software, May 1990, pp. 37-43.

[OMG003] *OMG Unified Modeling Language Specification*, Version 1.5, March 2003. Object Management Group, Inc., Framingham, Mass., 2003, website: http://www.omg.org

[PATE96] Patel, J.N.; Jamieson, L.H. *An object-oriented framework for the Cloner software prototyping environment*, SIGNALS, Systems and Computers, 1996. Conference Record of the Thirtieth Asilomar Conference on, 3-6 Nov. 1996, pp. 1354 -1358.

[RADE02] Radeski, A., Parr, S., Keith-Magee, R., and Wharington, J. *Component-Based development Extensions to HLA,* Proceedings of the 2002 Spring Simulation Interoperability Workshop (SISO Spring 2002). Paper ID 02S-SIW-046, March 2002.

[ROBB97] Robbins, J. E., Hilbert, D. M., and Redmiles, D. F. *Argo: a design environment for evolving software architectures,* In Proceedings of the 1997 International Conference on Software Engineering, Boston, MA, USA, 17-23 May 1997, pp. 600-601.

[ROBB98] Robbins, J. E., Medvidovic, N., Redmiles, D. F., and Rosenblum, D. S. *Integrating architecture description languages with a standard design method*, In Proceedings of the 1998 International Conference on Software Engineering, Kyoto, Japan. 19-25 April 1998. pp. 209-18.

[RUMB91] Rumbaugh, J., Blaha, M., Premerlani, W., Frederick, E., and Lorenson, W. *Object- Orzented, Modeling and Design*, Prentice Hall, 1991.

[SCHM97] Douglas C. Schmidt, *Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software*,' Handbook of Programming Languages, Volume I, edited by Peter Salus, MacMillan Computer Publishing, 1997.

[SINA02] Sinan Si Alhir. *Guide to applying the UML,* Springer New York, 2002.

[SISO03] SISO/SCS Panel. *Discussion, Priorities for M&S Standards,* Spring Simulation Interoperability Workshop, Orlando, Florida, March 2003.

[SRID96] Sridhar, M.A.; Paranjpe, P. *An object-oriented framework for embedded WWW application*, Emerging Technologies and Applications in Communications, Proceedings, First Annual Conference on, 7-10 May 1996, pp 97 -100.

[STAC95] Stacy, W. and MacMillian, J. *Cognitive Bias in Software Engineering,* Communications of the ACM, June 1995. pp. 57-63.

[TOLK02] Andreas Tolk. *Avoiding Another Green Elephant – A Proposal for the Next Generation HLA based on the Model Driven Architecture*, Paper 02F-SIW-004, Proceedings of the Fall Simulation Interoperability Workshop, Best Paper SIW F02, Orlando, Florida, September 2002.

[TOLK03] Andreas Tolk, James A. Muguira. *The Levels of Conceptual Interoperability Model (LCIM),* Fall Simulation Interoperability Workshop 2003, Paper 03F-SIW-007, Orlando, Florida, September 2003.

[WALD96] Waldspurger, C.A.; Weihl, W.E. *An object-oriented framework for modular resource management*, Object-Orientation in Operation Systems, 1996., Proceedings of the Fifth International Workshop on , 27-28 Oct. 1996, pp 138 -143.

[WILK95] Nancy M. Wilkinson. *Using CRC Cards: An Informal Approach to Object-Oriented Development*, Cambridge University Press, Paperback, Published April 1995.

[ZEIG00] Zeigler, B. P., T. G. Kim, and H. Praehofer. Theory of Modeling and Simulation, New York, NY, Academic Press, 2000.