12-1996

SimWorx: An ADA Distributed Simulation Application Framework Supporting HLA and DIS

Earl C. Pilloud

Recommended Citation

Pilloud, Earl C., "SimWorx: An ADA Distributed Simulation Application Framework Supporting HLA and DIS" (1996). *Theses and Dissertations*. 5879.
https://scholar.afit.edu/etd/5879

SIMWORX:
AN ADA 95 DISTRIBUTED SIMULATION APPLICATION FRAMEWORK SUPPORTING HLA AND DIS

THESIS

Earl Conrad Pilloud, Captain, USAF

AFIT/GCS/ENG/96D-23

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/96D-23

SIMWORX:
AN ADA 95 DISTRIBUTED SIMULATION APPLICATION FRAMEWORK SUPPORTING HLA AND DIS

THESIS

Earl Conrad Pilloud, Captain, USAF

AFIT/GCS/ENG/96D-23

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

AFIT/GCS/ENG/96D-23

SIMWORX:
AN ADA 95 DISTRIBUTED SIMULATION
APPLICATION FRAMEWORK
SUPPORTING HLA AND DIS

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Systems

Earl Conrad Pilloud, B.S.
Captain, USAF

December 1996

Approved for public release, distribution unlimited

Acknowledgments

Despite the fact that we live in a world of automation and convenience, few people can truthfully say they've accomplished something solely through their own powers. This thesis effort is no exception--it could not exist without the time, effort, and heart of numerous mentors, associates, and friends.

First, I thank God for giving me opportunities, strength, and hope. I'm sure the person who wrote the poem Footprints In The Sand was a graduate student at one time or another.

I want to thank Major (Dr.) Mark Kanko for selecting me to be on his research team and giving me the privilege of putting my "book learning" to work on something "cool". He gave my creativity free reign, yet kept me on track lest I strayed. He was a great mentor and made the positive difference in my AFIT academic experience.

I also thank the members of my committee for their valuable ideas and guidance. Dr. Thomas Hartrum gave me a firm foundation in Object-Oriented Software Engineering and injected me with a sense of formalism and rigor. I could always rely on Major (Dr.) Keith Shomper to provide keen insights...and new requirements ;^).

I also owe a debt to current and previous student researchers. Steve Sheasby provided insight, algorithms, and answers to perplexing DIS questions. Dave Hightower always had answers to my UNIX questions.

Finally, I especially want to thank my family for their support during this 18 month adventure. My wife, and best friend, Recy "held down the fort" and always had a smile, encouraging word, or technical answer when I needed it. Heidi, Mireille, and Colin continued to be great kids despite busy times and a grouchy dad. Thank you all, I couldn't have done it without you.

Conrad Pilloud

Table of Contents

List of Figures

Abstract

This research consisted of the analysis, design, and implementation of a reusable application framework for distributed simulation which is compliant with both the DoD High Level Architecture (HLA) for Modeling and Simulation and the Distributed Interactive Simulation (DIS) standards. The goal was to create an Ada-based system for experimentation in distributed simulation. A subsidiary goal was to integrate the system with an existing Air Force Institute of Technology (AFIT) application framework for virtual simulations, Easy_Sim.

The application framework was designed using object-oriented techniques to enable experimenters to customize it via inheritance extension. The application framework, named SimWorx, consists of two sections: an HLA Federate skeleton, and a surrogate HLA Run-Time Infrastructure (RTI) which has an HLA "front-end" and a DIS "back end" to provide DIS compatibility. The SimWorx framework was successfully integrated with Easy_Sim to provide an Ada-based joint simulation system for distributed virtual simulations.

This Page Intentionally Left Blank

1. Introduction

1.1 Background

The computer age has completely changed the face of modern warfare. Striking change is evident in the area of computer simulation. Computers enable us to simulate warfare, saving both lives and money. Although the Department of Defense (DoD) uses simulation in numerous applications, recent advances in computer networking have spawned the field of distributed simulation. Distributed simulation is the linking of aircraft, tank, and other combat simulations from diverse locations so the participants can interact with each other [OTA95]. Distributed simulation enables high-level military commanders to conduct wargames economically, using simulated forces in conjunction with real soldiers, sailors, and airmen, without requiring the participants to travel from their home bases.

Although numerous standards exist for distributed simulations in the DoD, two are the most important: Distributed Interactive Simulation (DIS), and the DoD High Level Architecture for Modeling and Simulation (HLA). DIS is currently the dominant standard for linking simulations [DIS94]. However, DIS has some shortcomings. To transcend those shortcomings, the DoD recently began the design of the HLA as the new standard for distributed simulation [DMSO95]. The HLA is intended to be used in all future DoD distributed simulation efforts.

For all its benefits, the computer age has also adversely affected the DoD. Computer technology is complex. While computer hardware has become faster and cheaper, computer software has grown larger, more complex, and more expensive. As part of a wide-ranging response to these trends, the DoD adopted the Ada programming language as its standard. Ada is a rich language with features that promote good software engineering practices. Ada's usage is not limited to the DoD; it is also widely used in the commercial world for applications ranging from control processing in steel mills to processing geophysical data in the petro-chemical industry [GSAM94].

The Air Force Institute of Technology (AFIT) is heavily involved with research in the field of modeling and simulation. The AFIT Graphics Lab is a pioneer in inexpensive virtual environments for distributed simulation of combat. Lab projects include:

The Virtual Cockpit: An inexpensive, F-15E man-in-the-loop simulator [Eri93, Ger93, Dia94].

The Synthetic Battle Bridge: A tool to show commanders a "bird's eye" view of a battle taking place in a synthetic environment [Sol93, Wil93, Kes94, Roh94].

The Red Flag Debriefing Tool: A tool to view, manipulate, and manage data for Red Flag exercise de-briefings [Gard93, For94].

Although successful, none of these efforts employed the Ada language. Since practical Ada experience is vital to the establishment of a culture of Ada usage by choice rather than mandate, two previous thesis efforts addressed Ada's use in the Graphics Lab. In 1994, Captain Jordan Kayloe developed a software architecture for virtual combat simulations using Silicon Graphics, Inc. workstations [Kay94]. Captain Shawn Hannan followed that effort by re-engineering Kayloe's work to achieve platform independence [Han95]. Despite their success, these projects have not produced widespread employment of Ada in the lab. This result is attributed to the lack of a distributed simulation interface for the Ada virtual simulation architecture.

This problem could be solved by adapting the lab's current DIS standard interface [She92] to Ada. Adaptation would permit Ada-based simulations to inter-operate with the other lab projects as well as the majority of current, important DoD simulations. However, this is a short-sighted approach for two reasons. First, the current DIS interface is highly dependent upon the Silicon Graphics operating system; it would require re-design to support multiple computer platforms. More importantly, by the end of 1997, the DoD High Level Architecture (HLA) will supersede DIS, since HLA compatibility is mandated by the Defense Modeling and Simulation Office (DMSO).

Since mandatory HLA compatibility is on the horizon, creating an Ada-interface to the HLA is a superior solution. Unfortunately, as this is written, the HLA is still in a prototyping stage; it is not yet ready to be used on a widespread basis. Although the HLA standard was promulgated on 15 Aug 96, initial HLA products simply aren't available.

1.2 Problem

The AFIT Graphics Lab lacked an Ada-based distributed simulation system which supported both DIS and HLA. This research effort investigated the creation, documentation, and use of a distributed simulation system which supports both DIS and HLA standards.

1.3 Summary of Current Knowledge

1.3.1 Distributed Simulations and Distributed Simulation Standards

1.3.1.1 Distributed Simulation

Distributed simulation is one of the principle research areas in the AFIT Electrical and Computer Engineering department. The Congressional Office of Technology Assessment [OTA95] defines distributed simulation as the linking of aircraft, tank, and other combat simulations from diverse locations so simulation participants can interact with each other.

The military has used stand-alone simulators for training purposes since the late 1920s. In the early 1980s, the U.S. Army pioneered the use of distributed simulation to connect physically distant combat simulators (fighting vehicles) together into a virtual battlespace. This training surpassed the bounds of training with a single fighting vehicle to include interactions between many vehicles. Distributed simulation is less expensive and much more flexible than practice with real weapons. The benefits of distributed simulation of combat have not been ignored by the Air Force or Navy; all three services are actively pursuing the use of models and simulations. The DoD Modeling and Simulation Office (DMSO) is an umbrella organization which coordinates the activities of the services.

Distributed simulations are classified into three notional categories: constructive, virtual, and live. A constructive simulation simulates the behavior of military forces. For example, a constructive simulation could simulate the existence and behavior of an entire squadron of aircraft via a computer software model that calculates aircraft and pilot actions. A virtual simulation places a human operator (soldier, sailor, airman) in a virtual environment which represents the battlespace. Traditional flight simulators and task trainers are virtual simulators. A live simulation uses real combat units, vehicles, and people to simulate warfare. The Air Force Red Flag wargame is an example of a live simulation.

1.3.1.2 Distributed Interactive Simulation (DIS)

Distributed Interactive Simulation (DIS) is a set of complementary standards for distributed simulations promulgated by the DIS community [DIS94]. The community includes representatives from the DoD, as well as academic and commercial organizations.

The DIS standards define physical communication protocols and an ever-growing set of logical communication protocols (protocol data unit standards). Work is proceeding to develop other standards for environmental and data representations. Although DIS has been successful in defining simulation interoperability standards, the standards-forming process is too unwieldy to quickly adapt to changing requirements and the immense variety of modeling and simulation needs within the large DIS community.

1.3.1.3 The DoD High Level Architecture for Modeling and Simulation (HLA)

In order to transcend some of the limitations of the DIS approach to simulation, the DoD began the High Level Architecture for Modeling and Simulation (HLA) initiative in 1994 [DMSO95]. The HLA's purpose is to create a DoD-wide common technical framework for modeling and simulation [DoD95]. The principle goal of this technical framework is to increase the reusability and interoperability of distributed simulations. The HLA approaches simulation from an object-oriented perspective, representing individual simulations and federations of inter-working simulations with object models. The HLA effort is currently in a prototyping effort. The initial HLA standard (v 1.0) was released in August 1996. Already, the DMSO has mandated that all future DoD models and simulations will be "HLA compliant". DMSO is defining compliance definitions and standards.

1.3.2 Software Architecture

Software architecture has numerous definitions. In what has become the seminal paper in the field, Garlan and Shaw [Gar93] define software architecture as a view of a software system in terms of its overall structure. This is a higher level, more abstract view of software than the traditional view of software as algorithms and data structures. The principal advantage of using a software architecture approach is that it allows one to focus on the "big picture" of a system rather than getting lost in all of the details. Focusing on architectural issues can bring order and discipline to the creation of very large systems.

1.3.3 Object-Oriented Application Frameworks for Design Reuse

Booch [94] defines a framework as a group of object-oriented components that provides a set of services for a specific domain. Application frameworks are pre-fabricated, reusable, software design skeletons for creating new software systems in a specific problem area. The main structure and pieces of the new system are provided in the framework. Frameworks make software engineers more productive since they aren't inventing the entire software system from scratch. They simply fill in the pieces which are necessary for the specific problem they are solving--adding flesh to the skeleton.

1.3.4 Design Patterns

A design pattern is recurring element of software design [Gam95]. The main idea behind design patterns is applying specific, proven designs to recurring problems in software design. A design pattern consists of a name for the pattern, the general problem that the design pattern solves, the general form of the solution to the problem, and the consequences of using the pattern to solve the problem. Besides providing "canned" ideas for solving problems, design patterns are useful for documenting the overall design of a system [Joh92], since they provide software engineers a common vocabulary for design solutions.

1.3.5 Previous AFIT Efforts

1.3.5.1 ObjectSim 1.0

Captain Mark Snyder's [Sny93] ObjectSim thesis resulted in an object-oriented application framework written in C++ for building DIS-capable virtual simulations. Snyder derived the framework's functionality from several AFIT virtual simulations by finding and implementing their common functions and behavior (each project was implemented within the ObjectSim framework; their designs evolved together).

ObjectSim was written in C++ since the existing Ada standard at that time did not support object-oriented programming. ObjectSim encapsulated many of the details of programming using Silicon Graphics' Performer^{TM} library. (Performer^{TM} provides routines optimized for rendering 3D images in real-time on Silicon Graphics proprietary graphics hardware). Further, it provided a DIS interface by embedding the functionality of Captain Steve Sheasby's Object Manager [She92].

Snyder showed a framework could provide large productivity gains for the graphics lab projects [Sny93,79].

1.3.5.2 Easy_Sim

The main goal of Captain Jordan Kayloe's [Kay94] Easy_Sim architecture was to show that an ObjectSim-like framework could be successfully implemented in Ada 95 (then Ada 9X). To achieve his goal, Kayloe developed an abstract architecture based on ObjectSim (but more coherent and consistent) and created both C++ and Ada 95 frameworks from it. He then built an application in each language to compare the performance of the two frameworks.

By measuring the screen refresh rate for identical virtual simulations, Kayloe showed there was no performance difference between the Ada95 application and the C++ application.

1.3.5.3 ObjectSim 3.0

In 1995, Captain Shawn Hannan [Han95] developed ObjectSim 3.0, a redesigned Easy_Sim. Hannan's goal was to remove Easy_Sim's dependence on the Silicon Graphics platform. Hannan produced mixed results. Although he was able to create an abstraction of the Performer^{TM} library, time limitations prevented him from implementing the abstraction on other workstations.

1.3.5.4 Limitations of previous efforts

Despite their success, all of the previous efforts were limited to virtual simulations running solely on Silicon Graphics workstations. The main theme of the ObjectSim and Easy_Sim application frameworks was to reduce the effort needed to create virtual simulations by hiding the details of the sophisticated Silicon Graphics system, rather than to provide a set of generalized tools or solutions to solve problems in distributed simulation. This is evident in their class structures. Out of ObjectSim's seven classes, four were devoted to graphics management functions, while only one had a subclass for network access. Easy_Sim was further limited since it lacked a distributed simulation interface.

1.4 Assumptions

The main assumption underlying this research effort was that the HLA Run Time Infrastructure would not be available for use.

1.5 Scope

The main focus of this thesis effort was the creation of a distributed simulation system which supports both DIS and HLA. This system is entitled SimWorx. SimWorx is more than a simple distributed simulation interface; it is an application framework, a pre-fabricated software design skeleton for creating distributed simulations in Ada 95. Other software engineers can extend SimWorx's pre-existing design and hierarchy of classes to create new Ada 95 distributed simulations. SimWorx enables software engineers to focus on the details of their simulations; they can ignore the details of interacting with DIS or the HLA. SimWorx initially supports only soft-real-time simulations; it currently does not support the full HLA standard which also supports event-driven simulations.

Application framework interaction was a corollary thesis issue. Application framework interaction occurs when a software engineer uses more than one application framework to build an application. Recent literature does not provide details on possible problems in this area. The SimWorx framework was integrated with the Easy_Sim framework to create an Ada 95-based distributed virtual simulation. The primary difficulties encountered were linking related objects and establishing a unified thread of control.

1.6 Standards

SimWorx was designed to interface with the HLA Run Time Infrastructure as defined by the High Level Architecture Interface Specification [DMSO96a]. SimWorx also supports the core subset of DIS standard 2.0.4. SimWorx is implemented in Ada 95, as defined by ISO standard 8652. The SimWorx source code was written following guidelines in the Software Productivity Consortium's Ada 95 Quality and Style Guide [SPC95].

1.7 Approach/Methodology

SimWorx was built using an iterative development approach. The approach was similar to Snyder's [Sny93], in that SimWorx was "grown" from three separate simulation applications. Through each iteration, the SimWorx design evolved to a higher level of generality and functionality.

The first iteration of SimWorx development was the design and implementation of an application to print the current state of DIS entities on the DIS network. This application was used to define SimWorx's input interface and entity dead-reckoning capabilities.

The second iteration of SimWorx development was the design and implementation of a simple constructive simulation to simulate movement of a "gaggle" of DIS entities. This application was used to define SimWorx's output interface.

The third iteration of SimWorx development was integration of SimWorx with the Easy_Sim application framework. The resulting application is a simple virtual flight simulator. Easy_Sim provides the graphical interface for the application; SimWorx provides the distributed simulation behavior.

1.8 Materials and Equipment

SimWorx development depended solely upon the computers of the AFIT graphics lab: Silicon Graphics workstations. SimWorx was implemented on the Silicon Graphics platform using GNAT (GNU Ada Translator for Ada 95) version 3.07.

Rational Software Corporation's Rose product was used to create, represent, and manipulate the SimWorx analysis and design documents (see Appendices B-E).

1.9 Document Overview

Chapter 2 provides thesis background information on distributed simulation, software engineering methods and concepts, and the Easy_Sim application framework for visual simulations. Chapter 3 describes the thesis methodology. Chapter 4 describes the analysis, design, and implementation of the SimWorx application framework. Finally, Chapter 5 provides overall conclusions and offers recommendations for further study.

2. Summary of Current Knowledge

2.1 Overview

This background information is divided into six sections. The first two sections are meant for readers who are unfamiliar with the Distributed Interactive Simulation (DIS) and High Level Architecture (HLA) distributed simulation standards. The following three sections shift focus from the research problem subject area to the arena of software engineering methods and concepts. Section 2.4 opens with a discussion of Software Architecture. Section 2.5 continues with Object Oriented Application Frameworks. Section 2.6 provides an overview of Software Design Patterns. Finally, the last section introduces the reader to Easy_Sim, an Air Force Institute of Technology application framework for virtual combat simulations, created in 1994 by Captain Jordan Kayloe [Kay94].

2.2 Distributed Interactive Simulation (DIS)

As mentioned in the Introduction, the Air Force Institute of Technology (AFIT) is involved in distributed simulation research. However, none of AFIT's research is conducted with the use of the Ada programming language. This thesis, however, applies the Ada language to create an environment for experimenting with distributed simulation (including DIS).

Distributed Interactive Simulation (DIS) is a set of complementary standards for distributed simulations. Although DIS is used by both the commercial and educational sectors, it is primarily a Department of Defense (DoD) standard. In fact, DIS is the predominant distributed simulation standard in use within the DoD.

A DIS simulation is an aggregation of independent simulators (called simulation applications) which interact with each other via a computer network. The independent simulators need not be collocated--they could be across the room, across town, or across the nation. The simulation applications communicate using a standardized set of communication protocols. Each simulation application may control one or many simulation participants (called entities). Simulation participants interact within a common, virtual environment. Figure 2-1 shows a notional diagram of a DIS simulation.

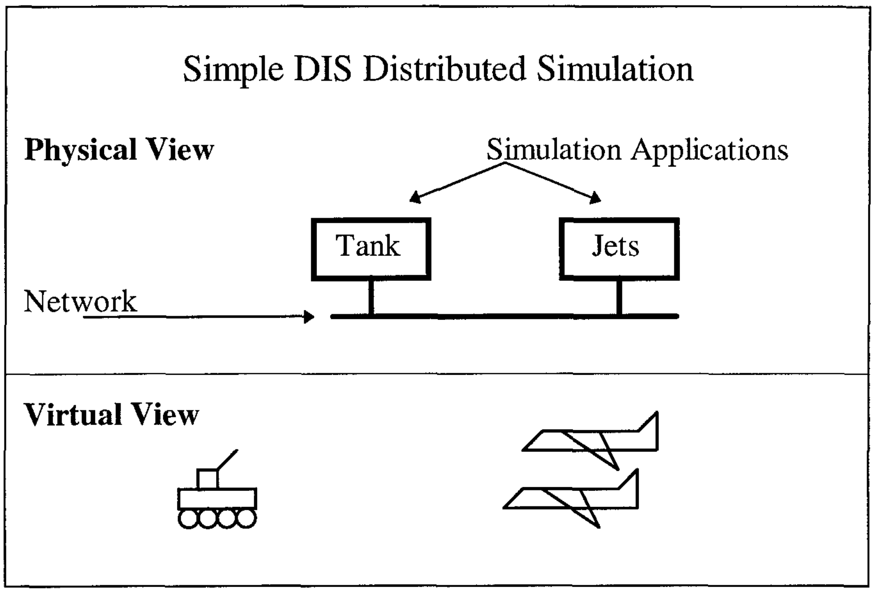


Figure 2-1. Simple DIS Distributed Simulation

2.2.1 Origin and Uses of DIS

DIS is the successor of the Defense Advance Research Projects Agency (DARPA) Distributed Simulator Networking Program (SIMNET) [She92]. The SIMNET program began in 1983 with the goal of developing a nation-wide network of human-in-the-loop (HITL) combat simulators for the U.S. Army. Over 250 simulators were built, with separate simulators for the M1 Abrams Tank, the M2 Bradley Fighting Vehicle, and Combat Helicopters. The primary purpose of SIMNET was improved combat crew training, teaching group-oriented concepts beyond simple vehicle operation.

Like SIMNET, DIS is primarily used for military training purposes. However, the scope of DIS has expanded to include other activities, from commercial entertainment to government evaluation of technical systems [DIS94, 11].

2.2.2 DIS Principles

The DIS standards are based on a foundation of several design principles [DIS94].

Object/Event Architecture. Participants in a DIS-based simulation are represented as objects in a simulated, virtual environment. The Object/Event Architecture principle states that all objects keep each other informed of their movements and the events they cause by broadcasting packets of information called Protocol Data Units (PDUs). As a corollary of this principle, information about non-changing objects in the virtual environment (terrain, structures, etc.) is assumed to be known to all of the participating simulations. Hence, there are no PDUs defined for static simulation objects.

Autonomy of Simulation Nodes. A DIS simulation is really an aggregation of separate simulations, each on a separate node of a computer network. This principle states that each node broadcasts information about objects and events without calculating which other nodes might be interested in that information. It is the responsibility of the nodes receiving the information to determine whether the information is meaningful to them. The autonomy principle allows simulations to join and quit the DIS simulation without affecting the other simulations.

Transmission of "Ground Truth" Information. Each node broadcasts the absolute truth about the objects it is representing. It is the responsibility of nodes receiving information to realistically portray the "Fog of War". This principle allows nodes to represent the events of the simulation in a way which is most appropriate for each of them.

Transmission of State Change Information Only. Nodes transmit only events and changes to the objects they represent. This principle minimizes unnecessary transmission of information.

Dead Reckoning Algorithms to Extrapolate State Information Between Updates. Nodes maintain simplified models of the state of objects belonging to other nodes of the simulation by "dead reckoning", or extrapolating the information about the object. Nodes refer to these models in between the object state updates from the other nodes. This principle reduces the amount of information transmitted between nodes, since they can broadcast state changes at a reduced rate without affecting the accuracy of the simulated objects.

Simulation Time Constraints. The DIS standards are primarily meant to support human-in-the-loop simulations. Since humans can distinguish time differences of approximately 100 milliseconds, interactions between nodes must take place in less than 100 milliseconds to ensure simulation fidelity.

2.2.3 The DIS Protocol Standard

The DIS Protocol Standard is the best-defined DIS standard. The latest version of the standard has been published by the Institute for Simulation and Training as DIS standard 2.0.4 [DIS94b]. It has also been published by the Institute of Electrical and Electronic Engineers (IEEE), as standard 1278.1, entitled "Standard for Distributed Interactive Simulation--Application Protocols."

The protocol standard captures the DIS principles in an explicit set of requirements. The document precisely defines the DIS coordinate system, dead reckoning algorithms, and functional requirements. The bulk of the standard is the definition of the Protocol Data Units (PDUs) defined for broadcasting information between nodes of a DIS simulation.

The DIS PDUs fall into seven main categories: Entity Information, Weapons Fire, Logistics Support, Collisions, Simulation Management, Distributed Emission Regeneration, and Radio Communications.

2.2.4 Advantages and Disadvantages of the DIS approach

Sheasby [92] discusses some of the advantages to DIS-based distributed simulation.

Scalability. A nearly unlimited number of simulators can participate in a DIS simulation. The limitations result from external factors such as the number of simulations available and the ability of the communication system to handle the load, rather than internal factors such as the size or ability of a single simulator.

Flexibility. DIS-based simulation is flexible. Participating simulations can be added, removed, or changed during the execution of the aggregate DIS exercise.

Heterogeneity. Participating simulations may represent a variety of heterogeneous simulation entities. One simulator may represent an Army tank crew, while another may represent an entire squadron of aircraft.

Non-locality. Physical distance is relatively unimportant. Participating simulations may be in neighboring rooms or in different countries.

Despite these powerful advantages, DIS based simulation has some disadvantages. The first three of the following disadvantages are problem areas for all distributed simulations; the remainder are DIS-specific problems.

Link Failure. Communication links between the participating nodes may fail, causing unpredictable simulation results.

Bandwidth. Bandwidth is synonymous with communication link capacity. An increase in the number of simulation nodes causes an even greater increase in the amount of information transmitted between participating nodes. At some point, the communication infrastructure could be overwhelmed, again causing unpredictable simulation results.

Latency. Latency is the information delay between simulation nodes. Latency is caused by processing delays at nodes and electrical signal propagation delay. Long delays in information transmission may cause information to be obsolete by the time it reaches the sender.

Homogeneity. All participating simulations must transmit roughly the same types of information as defined by the standard. This limits DIS applicability to a specific arena, i.e., simulations where entities move about the earth and have simple, limited actions with each other. For example, DIS would probably not be a good choice to represent a virtual surgery simulation since the distance between a surgeon and a patient is extremely small in the DIS coordinate system and subject to numeric representation errors. The DIS standard, however, would be appropriate for simulating long-haul trucking operations since trucks range about a wide geographic area and have relatively simple interactions: load, transfer, unload, etceteras.

Unwieldy Standards Process. DIS standard changes take much longer to enact than the underlying technology changes which drive the standards, since the standards are changed by a community-review/voting process. It takes more than a year for a DIS standard to be adopted by the IEEE. The long lead time coupled with the homogeneity problem mentioned above can promote non-standard implementations since organizations can not afford to wait for a standard change to implement a function they need.

Major Keith Shomper [Sho95] notes the following DIS liabilities:

Pre-exercise Coordination. The flexibility advantage noted above is not as simple or easy as it sounds. Setting up a DIS simulation takes an enormous amount of pre-exercise coordination. Although the DIS communication standard denotes the types of information which may be transmitted between participants, there are still many components of a DIS exercise which are undefined or exercise specific. Take for example the environment; the standard does not denote terrain or weather effects which nevertheless must be coordinated between participating simulations.

Fidelity. Despite the use of information interchange standards which prescribe standard numerical interchange units, DIS-based simulations suffer fidelity problems stemming from simulation-internal computer resource differences. That is, despite the fact that two different computers are both using 64 bits to represent the position of simulation entities, differences in computer system implementation can affect the results of calculations made with the numbers. The net result is one simulation which believes a tree is between two entities whereas another simulation using the same data may believe the tree is slightly out of the entities' line of sight.

2.2.5 A DIS Simulation Example

The following example illustrates some of the DIS principles stated above. It is merely a "toy" example--it is by no means representative of the scope or breadth of typical DIS simulations.

2.2.5.1 Scenario

The scenario for this example is an engagement between a USAF F-15E air-superiority fighter human-in-the-loop simulator and two Iraqi Mig-29 fighters modeled by a constructive simulator. In this simulation, the F-15E pilot's mission is to eliminate the Mig-29 threat over the airspace of southern Iraq.

2.2.5.2 Physical Environment

The physical environment consists of a DIS-capable F-15E flight simulator and a constructive aircraft combat simulator such as EADSIM[1]. The DIS-capable flight simulator is a multi-million dollar training equipment item, procured as part of the F-15 aircraft acquisition program. The constructive simulation runs on a commercial-off-the-shelf (COTS) computer workstation with no special hardware. Each simulator runs autonomously. The simulators need not be co-located.

2.2.5.3 Virtual Environment

The virtual environment is the airspace over southern Iraq. Both simulators must be manually configured with models of the terrain of southern Iraq. This occurs via two separate, distinct configuration operations, since configuration methods are not standardized. Further, the models must be synchronized so that environment features have the same position and representation.

Entities (the F-15E and Mig-29s) are positioned in the environment using the DIS standard, earth-centered, earth-fixed coordinate system. The simulators may be required to convert between coordinate system representations if the DIS coordinate system is not the native system of the simulator.

2.2.5.4 General

The pilot flies the flight simulator. As he flies about the virtual environment displayed by the flight simulator, the flight simulator periodically broadcasts state information to all other simulators in the simulator network--in this case there is only one recipient, the constructive

[1] EADSIM is the Extended Air Defense Simulation, produced for the U.S. Army Space and Strategic Defense Command.

simulator (this is an example of the Object/Event Architecture principle of DIS). The state information is packaged in a DIS Entity State PDU. The Entity State PDU has constant data values indicating the pilot's plane is a USAF, friendly-force, air platform (fighter, F-15E). The PDU has varying data values giving the position, velocity, orientation, and appearance (damaged, undamaged, smoking, etc.) of the F-15E. State changes are broadcast every time the aircraft's position changes by more than 100 meters or when its appearance changes.

The constructive simulator receives all PDUs broadcast on the network (in this case, they are only sent from one simulation, the flight simulator). It models the flight of the F-15E, updating the model in two ways. First, it updates the state from each new Entity State PDU received from the network. Second, state updates between PDU receptions are accomplished by dead-reckoning, i.e., the F-15E's position and orientation are extrapolated from its last known position and orientation. The constructive simulator's primary purpose, however, is to model *and control* the flight and behavior of the Mig-29s. As it models the behavior of the Migs, the constructive simulator broadcasts their changing state information to the network.

2.2.5.5 Action

1. Pilot flies plane.

    - Flight Simulator displays view of airspace over southern Iraq. As plane "flies", Entity State PDUs are broadcast to the network. The flight simulator models the Mig-29s based upon information contained in received Entity State PDUs.
    - Constructive simulator models the Mig-29s. As the Migs "fly", Entity State PDUs are broadcast to the network. The constructive simulator models the F-15E based upon information contained in received Entity State PDUs.
    - Note: Both simulators always know the exact, true position of all simulation participants.

2. F-15E's radar detects opposing planes.
    - The flight simulator determines the Mig-29s are within the detection range of its radar system. This could be done via a sophisticated radar model which models electromagnetic properties or via an extremely simplistic geometry-based model.
3. Pilot fires missile at enemy planes.
    - The flight simulator models the flight of the air-to-air missile. First, it broadcasts a Fire PDU to declare the F-15E's attack upon the first Mig-29. Then, it broadcasts Entity State PDUs for both the F-15E and the missile.
    - The constructive simulator models the position and orientation of the missile, based upon received Entity State PDUs. Using information from the Fire PDU, the constructive simulator may cause the Mig-29 to conduct evasive maneuvers.
4. Missile hits one Mig-29, destroying it.
    - The flight simulator determines the missile has hit the Mig-29, based upon the Mig's position in the flight simulator's model. The flight simulator broadcasts a Detonation PDU which describes the detonation of the missile.
    - The constructive simulator receives the Detonation PDU and determines the damage inflicted upon the Mig-29. It models the descent of the destroyed aircraft, broadcasting Entity State PDUs with a "destroyed" appearance indicator.
    - The flight simulator examines the appearance indicator in received Entity State PDUs, detects the "destroyed" appearance, and displays the crashing Mig-29.
5. Second Mig-29 withdraws.
    - Based upon the destruction of the first Mig, the constructive simulator decides to withdraw the second aircraft from the combat area. It models the flight of Mig and broadcasts corresponding Entity State PDUs.

2.2.6 Summary

DIS is the predominant standard for distributed simulation in the DoD. However, it is mainly used for human-in-the-loop simulators and thus is not widely applicable to other types of simulations.

2.3 High Level Architecture (HLA) for Modeling and Simulation

Although DIS is the predominant DoD standard for distributed simulation, it is not the newest. This research effort also addressed the latest distributed simulation standard: The DoD High Level Architecture (HLA) for Modeling and Simulation.

2.3.1 Origin of HLA

Faced with a plethora of simulation systems which didn't work together, the Defense Modeling and Simulation Office (DMSO) initiated the High Level Architecture (HLA) for Modeling and Simulation in 1994 [DoD95]. The purpose of the HLA is to increase simulation interoperability and reuse by providing a common technical framework for all modeling and simulation efforts within the DoD.

As of this writing, the HLA is in a prototyping stage. Its only use within the DoD is within the six "proto-federations" which comprise the prototyping effort. A "proto-federation" is a prototype HLA federation (see terminology below) created to validate a specific modeling and simulation domain. For example, DoD components engaged in operational training are members of the Training proto-federation; DoD components who use simulations in support of acquisition are members of the Engineering proto-federation. The purpose of the prototyping effort is to evaluate the effectiveness of a common technical framework for all DoD modeling and simulation efforts.

The HLA Version 1.0 standard was released on 15 August, 1996 for DoD wide use on all modeling and simulation systems. An experimental version of the accompanying HLA Run

Time Infrastructure (see terminology below), version F.0, will be available on a limited basis in December, 1996.

The DoD Modeling and Simulation Master Plan mandates HLA usage on all DoD modeling and simulation programs [DoD95]. All DoD programs will be reviewed by the second quarter of 1997 to establish HLA compliance milestones.

As with many efforts sponsored by the Department of Defense, the HLA has its own unique vocabulary. Unfortunately, knowledge of the vocabulary is necessary to discuss the HLA in a meaningful way.

2.3.2 HLA Terminology

Federation: A set of independent simulations, interacting with each other for a specific purpose. Military crew training and weapon system evaluation are examples of two different, independent federations since they have different purposes.

Federation Execution: The act of executing the federation of simulations.

Federate: An individual simulation which is a member of some federation(s).

Simulation: A model of the real world.

Object: An entity that exists in an HLA simulation. Sample objects: soldier, F-15 jet, surgeon, bouncing ball.

Attribute: A feature of an entity. For example, hair color, eye color, height, and weight are all attributes (to name a few) of a person object.

Interaction: An interaction between two or more objects within a federation. Sample interactions: soldier *shoots at* F-15 jet, surgeon *throws* ball.

Simulation Object Model (SOM): A formalized description of a simulation object. The Simulation Object Model describes the attributes and interactions of an entity that are important to other objects in a simulation.

Federation Object Model (FOM): A formalized description of all the objects, interactions, environment features, and other details of a Federation. Consider it a "contract" between the federates of a Federation, which defines, standardizes, and limits the scope of their actions.

Federation Required Execution Details (FRED): Additional execution-specific information necessary to instantiate the execution of a federation. This information is derived from the FOM.

HLA Run Time Infrastructure (RTI): A standardized software system which transfers information between the simulation federates and provides the federates with certain common services. Although it could be considered the "Network Interface" for a federate, the term "Intelligent Delivery Service" is more fitting, since the RTI has a higher order of functionality than a simple network interface. Logically, the RTI is a single software system. It physically exists, however, in each separate computing node of the distributed system (i.e., a piece of the RTI is running on each individual workstation or platform of the simulation). Thus, although each federate interfaces with a component of the RTI on its system, the RTI itself is really the collection of all of the federates' associated RTI components, working together in concert.

2.3.3 Advantages and Disadvantages

As of this writing, there are no reports available describing the results of any HLA prototype testing. Therefore, the items listed below are based upon HLA concepts rather than experimental results.

The High Level Architecture has three main advantages over other approaches to distributed simulation .

Interoperability: Interoperability is one of the two principle goals of the HLA. Although the DoD has many simulation systems, most do not work together. For example, the

Joint Modeling and Simulation System (JMASS) was designed to support engineering analysis of weapons systems. It does not interoperate with the Joint Simulation System (JSIMS) which was designed for joint war-fighting staff training. Thus, the two systems can't be combined to analyze new strategies which employ new, simulated weapons systems.

The HLA solves this problem by specifying a standard with broader applicability than DIS. For example, the DIS standard specifies distinct PDU types for distinct interactions between entities. If a simulation requires some form of interaction which is not provided by the standard, it must either operate without that interaction or else lose compliance with the DIS standard. In contrast, the HLA simply defines a standard for the *format* of an interaction. Federations are free to invent their own interactions which suit their needs. For example, a medical federation could invent various interactions to simulate surgery: wash, cut, scrape, eviscerate, suture, etc. All medical federates would be interoperable since they comply with the definitions of the medical federation.

Reusability: Reusability is the other principle goal of the HLA. All weapons systems acquired by the DoD have associated simulators for analysis and personnel training. Much of the functionality of these systems is the same, yet almost all of them are built from scratch with unique components. In this era of reduced defense budgets, the DoD can no longer afford such duplication.

The HLA enables simulation reuse through the use of Simulation Object Models (SOMs). As defense contractors build new simulators, they'll be able to reuse pre-fabricated components, each of which is described by a SOM. That is, instead of "re-inventing the wheel," they'll be able to "copy the wheel" from a library of simulation components.

Structured, Standardized Pre-Exercise Coordination. The HLA offers more than standards for simulation description and execution (product standards). The HLA also defines a process standard for developing Federations. The Federation Execution Development Process defines development stages from definition of federation objectives to federation execution. By using a standard process, organizations can communicate and define their simulation requirements in a structured, disciplined way which reduces uncertainty and coordination difficulties.

The initial vision and implementation of the High Level Architecture opens the door to a world of truly reusable and interoperable simulations throughout the DoD. There are, however, some disadvantages to the HLA approach to modeling and simulation.

All Things to All People. The HLA is meant to satisfy all of the modeling and simulation needs of the DoD. However, the overriding goals of the HLA are interoperability and reusability. Other criteria such as cost and performance (speed) are of lesser importance.

In some modeling and simulation applications, performance is critical. The Air Force tests aircraft electronic warfare (EW) systems via a number of methods. One method is to place an aircraft in an anechoic (echoless) chamber to test the aircraft's actual EW equipment against simulated electronic threats. Since the chamber is larger than a football field, coordinating the electronic signals of the simulation equipment is extremely complicated and very time critical. Critical timing constraints are measured in millionths of a second. Here, the HLA RTI simply will not be able to perform fast enough to achieve the simulation goals of the test program. It is unrealistic to expect all DoD modeling and simulation programs to employ the HLA.

Semantic Stovepipes[2]. The HLA will only provide limited, rather than universal interoperability. As an explanation, note the difference between *syntax* and *semantics*. Syntax describes how parts are put together; English syntax describes how words are combined into

[2] A "stovepipe" is a system which isn't interoperable with other systems. This term is commonly used in the military to refer to weapons systems which were independently acquired and sustained without regard to interoperability.

phrases and sentences. Semantics describes the *meaning* of the parts. The HLA defines only the syntax for describing and composing simulations and their components. It does not describe the semantics of a simulation; that responsibility lies within the FOMs and SOMs.

Although limiting the HLA standard to simulation syntax overcomes the DIS homogeneity problem, it doesn't offer true 100% simulation interoperability. Simulations simply can't and don't universally share semantics. For example, "suturing a wound" has no meaning in the context of an aircraft electronic warfare simulation. The net effect of this problem will be the creation of "Semantic Stovepipes" within the DoD modeling and simulation community. DoD simulations will not be universally interoperable. They will, however, be more interoperable than they are today.

Novelty. The HLA's greatest disadvantage may be that it is new and relatively untried. Sophisticated systems are rarely "right" the first time.

2.3.4 An HLA Simulation Example

The following example uses the same scenario as the DIS example above to illustrate the character of an HLA based simulation.

2.3.4.1 Scenario

This scenario is the same as the scenario in section 2.2.5.1.

2.3.4.2 Physical Environment

The physical environment is also nearly identical to the example in section 2.2.5.2. However, this example stretches reality by assuming the flight simulator and constructive simulators are compatible with the HLA. Although HLA usage is mandated by DoD, it will be several years before all defense acquisition programs are HLA compliant. The flight and constructive simulators are members of the AFIT Thesis Toy Example (ATTE) Federation, an HLA federation dedicated solely for expository purposes.

In addition to federates (simulators), the physical environment includes the HLA Run Time Infrastructure (RTI) to manage information transfer between the federates.

2.3.4.3 Virtual Environment

The virtual environment is the airspace over southern Iraq. In the case of the HLA, the details of the environment are captured in the ATTE Federation Object Model (FOM) and Federation Required Execution Details (FRED). Environmental features have the same representation and position since they come from the common FRED (this still does not completely eliminate fidelity problems since floating point arithmetic may vary between physical environments). Common data formats and standardized, mostly-automated initialization procedures speed and simplify the configuration of the federation execution.

2.3.4.4 General

The pilot flies the flight simulator. As he flies about the virtual environment displayed by the flight simulator, the flight simulator provides the RTI with F-15E attribute updates at a pre-determined rate specified by the flight simulator's Simulation Object Model (SOM). Attribute updates aren't packaged in data packets--instead, the updates occur as software program calls between the flight simulator and the RTI. The HLA does not require dead-reckoning of entity attributes.

The constructive simulator only receives attribute updates concerning the F-15E when the RTI deems appropriate, based upon discovery criteria supplied by the constructive simulator at the beginning of the simulation. For example, if the Mig-29s radar systems can only detect threats within a 50 Km. radius, the RTI will only declare the existence of the F-15E to the constructive simulator when the F-15E is within 50 Km. of a Mig-29. The constructive simulator federate models the behavior of the Mig-29s, providing state updates to the RTI at a rate specified by the FOM.

All possible interactions between simulated objects within the federation are described in the Federation Object Model. The interactions listed below are specific to the ATTE federation and may not necessarily be used by any other HLA simulations.

2.3.4.5 Action

1. Pilot flies plane.
   - The flight simulator displays view of airspace over southern Iraq. As plane "flies", the flight simulator federate provides attribute updates via software procedure calls to the RTI. The flight simulator will not model the Mig-29s until the RTI notifies it of their existence (based upon F-15E discovery criteria provided to the RTI at simulation run-time).
   - The constructive simulator models the Mig-29s. As the Migs "fly", the constructive simulator federate provides attribute updates via software procedure calls to the RTI. The constructive simulator does not model the F-15E until the RTI notifies it of its existence.
2. F-15E's radar detects opposing planes.
   - The RTI determines the Mig-29s are within the detection range of the F-15E's radar system based upon discovery criteria specified by the F-15E earlier in the simulation. This criteria is specified by the F-15E model and like in the DIS example above, could be calculated via a sophisticated radar model which models electromagnetic properties or via an extremely simplistic geometry-based model. The RTI notifies the flight simulator of the Migs existence via software procedure calls.
3. Pilot fires missile at enemy planes.

- The flight simulator models the flight of the air-to-air missile. First, it declares the existence of the fired missile object to the RTI (via procedure call, in fact all interactions with the RTI are via procedure call). Next, it notifies the RTI of a Missile Fire interaction between the F-15E and the Mig-29. Then, it delivers attribute updates to the RTI for both the F-15E and the missile.
- The RTI does not notify the constructive simulator about the existence of the missile until the missile's attributes fit the constructive simulation's discovery criteria. Upon "discovery", the RTI instructs the constructive simulation to instantiate a missile object. The RTI then provides attribute updates (reflections of attribute information provided by the flight simulator) for the missile object. Using information from the attribute updates, the constructive simulator may cause the Mig-29 to conduct evasive maneuvers.

4. Missile hits one Mig-29, destroying it.
   - The flight simulator determines the missile has hit the Mig-29, based upon the Mig's position in the flight simulator's model. The flight simulator delivers a "missile hit" interaction to the RTI.
   - The constructive simulator receives the "missile hit" interaction from the RTI and determines the damage inflicted upon the Mig-29. It provides a "destroyed aircraft" interaction to the RTI indicating the aircraft's destruction. It models the descent of the destroyed aircraft and provides attribute updates to the RTI until it impacts the earth.
   - The flight simulator receives the "destroyed aircraft" interaction from the RTI and displays the destroyed aircraft.
5. Second Mig-29 withdraws.

- Based upon the destruction of the first Mig, the constructive simulator decides to withdraw the second aircraft from the combat area. It models the flight of Mig and provides attribute updates to the RTI for the remainder of the simulation.

2.3.5 Summary

The High Level Architecture (HLA) for Modeling and Simulation is meant to unify the DoD's modeling and simulation efforts. It is designed to promote interoperability between historically mismatched simulations and reusability of common simulation components.

2.4 Software Architecture

Knowledge of software architecture concepts is important to analyze and discuss specific computer systems' organization and design. This section discusses the relatively new field of software architecture. The following sections introduce concepts from the software architecture literature and explain their relevance to this research effort.

2.4.1 Software Architecture Foundation

In what is viewed as the seminal paper in the field of software architecture, Garlan and Shaw [Gar93] describe software architecture as *a level of software design*. The software architecture level of design exists at a higher level of abstraction than the traditional design level which is concerned with algorithms and data structures. The authors suggest software architecture is the next wave of design abstraction, following High Order Programming Languages (HOLs) and Abstract Data Types (ADTs).

Software architecture encompasses numerous gross design issues: structural issues such as the overall organization and control scheme in a computer system, communication issues between the gross pieces of the system, functional responsibility issues for gross pieces of the system, system performance scaling issues, and issues of optimum design selection from alternative designs.

Garlan and Shaw provide an informal vocabulary for discussing, reasoning about, and comparing software architectures. Software architectures are made of *components* and *connectors*. Components are the computational parts of the system; the gross pieces of the system which perform work. Connectors describe how the components interact with each other. Figure 2-2 shows the components and connectors of a trivial system.

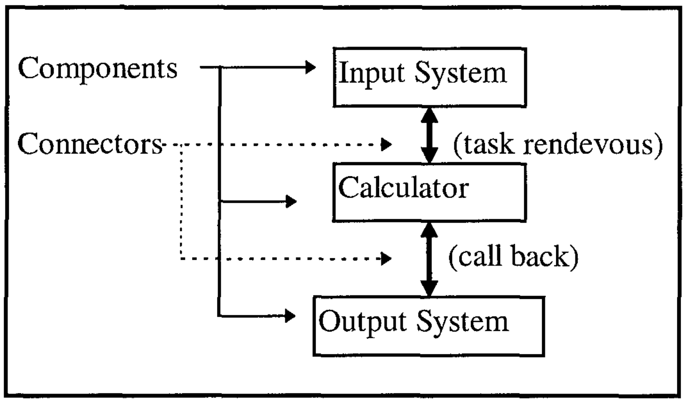


Figure 2-2 - Components and Connectors

Garlan and Shaw present several architectural styles to show the variety of available styles and tradeoffs in choosing one style over another.

Pipes and Filters. In the Pipe and Filter style, each component of the architecture has an input data stream and an output data stream (pipes). Components read the input stream and do some sort of transformation on it to create the output stream, hence the term *filter*. Filters are independent from each other, they don't share internal states or data, or even know the identity of other filters in the architecture.

Pipe and Filter systems support software reuse via reusable filters. They also allow the overall system to be viewed as a simple collection of parts whose behavior is simply the composition of the individual filters. However, pipe and filter architectures are not very suited to interactive applications and may suffer poor performance since the system can only go as fast as its slowest component.

Object-Oriented Architectures. Object-oriented architectures are composed of a collection of objects (components) which interact with each other through function and procedure calls (connectors). Each object has its own identity and internal state.

Object-oriented architectures are easy to modify and maintain. They are usually easy to understand since the objects directly relate to objects of the real world: telephones, pressure gauges, ledger entries, etc. However, object-oriented architectures have the disadvantage that each object needs to know the identity of every object it must interact with. Contrast this with the pipe and filter architecture, in which the filters didn't care who they were connected to.

Event-Based, Implicit Invocation. Event-Based, Implicit Invocation architectures add the concept of *implicit invocation* to the object-oriented style. Here, objects announce events (a form of connector) to the system; the system then invokes objects which have previously "registered" to receive the events.

Implicit Invocation architectures are easy to extend since it's a simple matter to add a component to the system by registering it for events it is interested in. The main disadvantage of implicit invocation is components no longer directly control the actions within the system since they have no control over which objects register for their events.

Layered Systems. A layered system is organized into hierarchical layers (components). Each layer provides services to the layer above it while using the services of the layer below. The connectors of a layered system are the protocols which define how the layers interact. Communication protocols like the International Standards Organization (ISO) Open Systems Interconnect (OSI) seven layer model are prime examples of layered systems.

Layered systems allow designers to create systems in abstract layers; each more abstract than the layer below. Abstract layers simplify system development. Further, abstract layers are flexible like filters in that they can be replaced by identical layers as long as the protocols match.

Unfortunately, not all systems can be defined in a layered style. Layered styles may also suffer performance penalties associated with protocol transformations.

Repositories. Repository architectures have two kinds of components. The first is a central data element which represents the current state of the system. The second kind represents other processing components which independently interact with the data element. The connectors of the system are the interactions between the processing components and the central data element. This style is commonly called the *blackboard architecture* since it resembles a blackboard shared by collaborating students.

Repository architectures are common in artificial intelligence systems. The latest knowledge about the problem at hand resides in the central data store; the various processing components work independently on the knowledge, growing and manipulating it to a solution.

Other Architectural Styles. There are other architectural styles which Garlan and Shaw describe in less detail such as Domain-Specific software architectures and State Transition systems. All of these styles are "pure" architectural styles. The authors emphasize that most real systems have elements of several architectural styles; these are called *heterogeneous architectures.* Typically, systems composed in one architectural style will have components internally organized in completely different styles. For example, the filters of the UNIX operating system may be implemented internally as object-oriented systems.

2.4.2 Visualizing Software Architecture

Frederick Brooks [Bro87, 12] states *Invisibility* is an essential element of software; since software is not part of the physical world, it is inherently difficult to visualize.

Phillipe Kruchten addresses the software invisibility problem through a model which represents software system architecture as a collection of views [Kru95]. Each view describes the system from a different perspective. Together, the views constitute the description of a system's architecture.

Logical View. The logical view of an architecture describes its functions, the services the system provides to its end users. For an object-oriented style, Kruchten uses *class diagrams* to show the object classes of the architecture and their relationships. Designers use the logical view to organize classes and relationships, and to communicate the architectural design to the system's users.

Process View. The process view of an architecture describes how the architecture satisfies the non-functional requirements of a system such as performance and system availability. It shows the independent tasks and processes of the system and the logical communication links between them.

Development View. The development view shows the organization of an architecture's software modules. It helps software engineers partition the implementation workload of the system into team-sized pieces. It also supports configuration management of the architecture's pieces.

Physical View. The physical view shows the mapping of logical processes from the process view to the actual hardware they run on. There can be more than one physical view for a given process view, since the architecture may be ported to more than one different computer hardware environment. The physical view enables software engineers to plan the tactical-level organization of the architecture to meet the non-functional requirements of the system.

Scenarios. Scenarios are a fifth view which show specific interactions and operations of the architecture. Kruchten states scenarios are a form of Ivar Jacobson's [Jac92] Use Cases; a

Use Case is a form of software requirement which describes the interactions between actors, or users of a system, and the components of a system. Scenarios are used to show how the architectural pieces work together. They provide a view of the dynamics of the architecture.

The views are not fully independent since elements of one view are connected to elements of other views by following certain design rules and heuristics. Thus, Kruchten's view model not only provides architecture visibility, it also provides a method for defining and refining an architecture through design rules.

2.4.3 Importance to Research

Software architecture concepts provide software engineers a vocabulary and a logical framework to think and reason about software systems at a higher level of abstraction than traditional design methods which focus on algorithms and data structures. This research relied on these tools as a means to design, compare, and document software systems for distributed simulation.

2.5 Object Oriented Application Frameworks

Since software is almost purely the result of intellectual labor, it is labor intensive, and thus costly. Object-oriented application frameworks are a means of increasing software engineering productivity (and reducing cost) through a specific form of software reuse. This section introduces application framework concepts and discusses issues related to framework construction and use. It concludes by showing how application frameworks apply to this thesis research effort.

2.5.1 Definition

An application framework is a reusable design for a family of related applications. In an object-oriented context, it is implemented as a set of abstract classes whose instances form the skeleton of the design [Joh88].

Frameworks are beneficial since they achieve software reuse at a higher level of design than the simple use of software component libraries. An application framework enables software engineers to reuse a common design for various similar applications. The Hotdraw application framework [Bec94] demonstrates the power of design-level reuse. Hotdraw provides the skeletal design of a graphical drawing program; software engineers can use the Hotdraw framework to create a variety of different drawing applications for different problem domains such as music, architecture, civil engineering, etc.

Design reuse provides corollary benefits such as portability and rapid prototyping [Cam93]. Design reuse supports portability since the design is represented as a skeleton of abstract classes. Those classes can be extended with concrete subclasses to support different computing platforms. For example, a general output class could have separate concrete subclasses for the UNIX and Windows operating systems. Design reuse supports rapid prototyping since much of the design and resulting code are captured in the framework, thus the time necessary to design and code new applications is dramatically reduced.

2.5.2 Framework Issues

Issues related to frameworks can be categorized in two groups: issues related to framework analysis, design and construction for software engineers who create frameworks, and issues related to framework use for client software engineers who use frameworks.

2.5.2.1 Framework Construction Issues

Designing and constructing application frameworks is different from designing individual applications since a framework is meant to be a general design which applies to many different software applications. Crucial design tradeoffs must be made which will affect all of the resulting applications. One important tradeoff noted by Coplien and Schmidt [Cop95] is the determination of which components in a framework are meant to be variable and which are meant to be stable. Insufficient variation makes it hard for users to customize the framework components. In contrast, insufficient stability makes it hard to understand and use the framework's behavior.

Deborah Adair [Ada95] presents a process for developing frameworks and offers several guidelines. The guidelines take into account the unique aspects of development mentioned above.

- **Derive frameworks from existing problems and solutions.** Adair suggests frameworks which are not derived from existing problems may be too general and hard to use.
- **Develop small, focused frameworks.** Smaller frameworks are easier to learn and use effectively.
- **Build frameworks using an iterative process.** Variation and stability tradeoffs may not be apparent following the first iteration of the design. Practice makes perfect; further iterations will refine the framework.
- **Treat frameworks as products.** Frameworks are often created in support of other projects within an organization. In order for frameworks to be used, understood, and maintained, Adair emphasizes they must be clearly documented.

2.5.2.2 Framework Use Issues

Application frameworks improve productivity each time they are reused, thus framework usability is vital to productivity.

Since application frameworks are reusable designs, Johnson [Joh92] states they are more abstract than most software, and therefore, more difficult to document. Frameworks are usually designed by expert designers in a particular problem domain and then used by non-experts. Thus, effective framework usage documentation is necessary to capitalize on the expert's investment. Johnson provides a method of documenting frameworks which uses software design patterns (see section 2.6). Non-experts use frameworks to solve typical problems; therefore, since design patterns describe and solve problems, they are particularly suited to documenting frameworks.

Multiple application frameworks can be used in concert to solve problems. The Choices framework [Cam93] is an application framework for an operating system which is composed of interacting subframeworks, each of which solves specific problems. The highest abstraction level of Choices provides consistency constraints between the subframeworks.

It is important to note the Choices subframeworks were designed to work in concert with each other within the constraints of the system. Little has been written, however, about marrying distinct application frameworks to solve a problem than spans the bounds of two problem domains. This may be an area for future study.

2.5.3 Importance to Research

Creation of an object-oriented application framework for distributed simulation which supports both DIS and HLA was the primary goal of this research.

2.6 Design Patterns

Few things in the field of software engineering are as valuable as a great software designer. Brooks [Bro87, 18] argues great software designers are an order of magnitude more productive than ordinary ones. Since there are a small number of really great designers, the software engineering community needs a means of capturing their ideas and knowledge so others can mimic and learn from their efforts. Software Design Patterns promote the capture of great ideas.

2.6.1 Definition

A software design pattern describes a software design problem which occurs over and over, along with a generic description of the solution to the problem which can be implemented in many different ways [Gam95, 2]. Gamma et al list four essential elements of a software design pattern:

- **Pattern Name.** The pattern name is a one or two word label or *handle* for the pattern. Pattern names add to a software engineer's vocabulary, enabling discussion of designs at a higher level of abstraction. Pattern names are descriptive; some common pattern names are Decorator, Prototype, Chain of Responsibility, or Active Object.
- **Problem.** The problem describes the actual problem at hand, i.e., when the pattern should be used.
- **Solution.** The solution describes an abstract, generic solution to the problem. It describes the components of the solution and their relationships.
- **Consequences.** The consequences are the results of using the pattern. Different solutions come with a price. Consequences are essentially engineering tradeoffs, secondary considerations which affect the choice of the solution. For example, "Implementing solution X generally will result in a increase of timing overhead."

Schmidt [Sch95, 70] makes the distinction between *tactical* and *strategic* design patterns. Tactical patterns solve problems at a relatively small scope within a software system. For example, the GOF[3] *Decorator* pattern [Gam95, 175], which provides a way to dynamically attach additional responsibilities to an object, causes only a small impact on a given software architecture. In contrast, strategic patterns have an extensive impact on the software architecture of systems.

At a strategic level, the High Level Architecture (HLA) for Modeling and Simulation is an example of the *Observer* pattern [Gam95, 293]. The Observer pattern defines a one-to-many dependency relation between objects so when one object changes state, all dependent objects are notified. In an HLA-based simulation, simulation objects declare their interest in other objects to the HLA Run Time Infrastructure which manages dependency information. As simulation objects change state, the HLA automatically provides updated information to dependent objects. This is a strategic level pattern because the observer behavior is one of the cornerstones of the HLA.

2.6.2 Design Pattern Application

Design patterns can be used in several complementary ways.

2.6.2.1 Engineering Body of Knowledge

One factor that dubiously distinguishes software engineering from other engineering disciplines is the lack of a standardized body of professional knowledge. One reason for this is software is not rooted in the physical world; software is an indistinct abstraction of the physical

[3] GOF stands for "Gang of Four", and is commonly used in the literature and on the Web to refer to the four authors of the seminal work Design Patterns: Elements of Reusable Object-Oriented Software: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

world, unbound from physics or biology. By its very nature, software is difficult to define and catalog.

Design patterns provide a subset of a software engineering body of knowledge. Important software design ideas can be captured as patterns. Software engineering students can learn design ideas from great designers via patterns [Sch95, 71]. Practicing software engineers can refer to patterns for solutions in specific situations.

2.6.2.2 Documentation

Design patterns can also be used to document specific software systems. When a software engineer employs a vocabulary of patterns, then specifications can be written which promote understanding of the system at a higher level of abstraction than algorithms and data structures.

Schmidt [Sch95, 71] states, "Pattern descriptions explicitly record engineering tradeoffs and design alternatives." This explicit record is vitally important for documenting software design since it captures *why* the system exists as it does; traditional forms of system documentation often fail to capture why the design satisfies the requirements in a certain way.

As stated in section 2.5.2.2, Johnson [Joh92] suggests an application framework can be sufficiently documented for its users (the application programmers) using only patterns. Although this appears sound, he does not offer any empirical evidence to validate his claim.

2.6.2.3 Reuse

Software reuse involves reusing previously constructed software artifacts (requirements, specifications, designs, code, tests, etc.) on new software efforts. Historically, software reuse has occurred at lower, less powerful levels of abstraction (code reuse) since other forms of reuse are technically difficult. However, since patterns describe software design solutions at a higher level of abstraction than simple algorithms and data structures, they enable design reuse. Schmidt

[Sch95, 69], states that in a multi-platform communication software environment, design patterns are often the only way to leverage design experience since specific designs, algorithms, and code may not be portable between operating systems.

2.6.3 Importance to Research

Design patterns played multiple roles in this research effort:

- Design patterns were used to understand and reason about simulation architectures and the behavior of their gross components.
- Design patterns were used as a source of engineering design knowledge during the construction of the SimWorx application framework.

2.7 Easy_Sim, an Application Framework for Visual Simulation

This section describes Easy_Sim, an Ada 95 application framework for virtual combat simulations. Easy_Sim is the product of Captain Jordan Kayloe's thesis work on architectures for visual simulation [Kay94].

2.7.1 Introduction to Easy_Sim

Easy_Sim is an application framework designed to improve client developer productivity in the realm of virtual combat simulations. Virtual combat simulations are essentially a time sequence of graphical entity actions as seen from some specific viewpoint. Easy_Sim hides from the client developer much of the details of setting up and managing a Performer application running on the Silicon Graphics Inc. computer hardware. Performer is high-level, high-performance 3-D graphical software library.

An Easy_Sim simulation is generally composed of several entities (aircraft, tanks, trucks, etc.) which move about and interact with each other in a 3-D graphical space. The view from some vantage point in the space is drawn on the computer screen 20-30 times each second, giving the human viewer a sense of smooth motion about the virtual space.

Figure 2-3 below shows the structure of the Easy_Sim framework (see Appendix A for a key to reading Rumbaugh diagrams). Client developers create new virtual simulation applications by defining new child classes which override the default behavior of the framework.

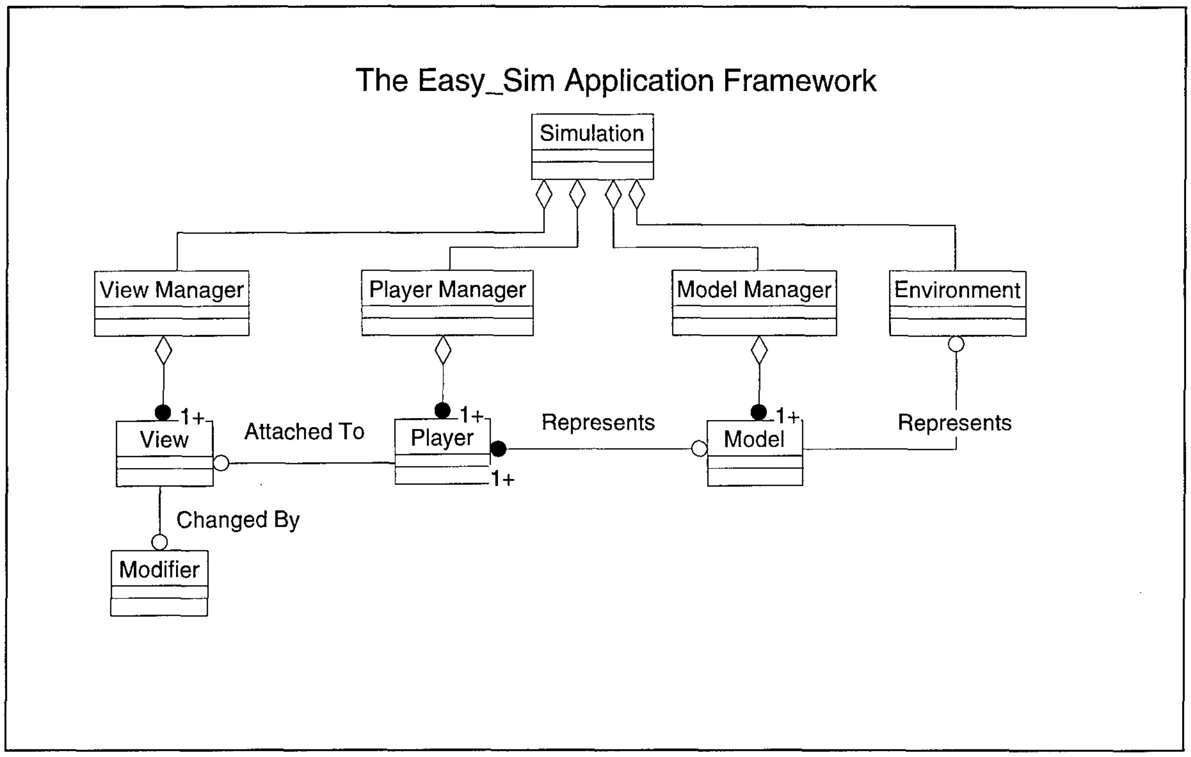


Figure 2-3. The Easy_Sim Application Framework

2.7.2 Easy_Sim Classes

This section briefly describes each of the Easy_Sim classes.

- **Simulation**: The simulation class is "the glue that holds the other pieces of the application together [Kay94, 76]." The overall control of the Easy_Sim framework takes place here. It is responsible for allocating an initializing the other Easy_Sim classes, for controlling the sequencing of their actions, and for drawing the virtual scene. Client developers devise a specialized subclass of Simulation to be the main driver of their simulations.

- **View**: The View class represents the vantage point of the virtual simulation.
- **View_Manager**: The View Manager class simply manages (contains) multiple Views.
- **Player**: The Player class represents a participant in a simulation (an aircraft, tank, submarine, etc.). Thus, subclasses of Player model the behavior of various kinds of combat participants. Position and orientation in the virtual space are the principle attributes of the Player class.
- **Player_Manager**: The Player Manager class manages (contains) all of the players of the simulation.
- **Model**: The Model class represents the graphical geometry of a player or element of the environment (trees, mountains, buildings, etc.)
- **Model_Manager**: The Model Manager class manages (contains) all of the models used during an execution of the Simulation.
- **Environment**: The Environment class represents the environment (terrain and lighting effects) of the virtual simulation.
- **Modifier**: The Modifier class represents an abstract means to modify (i.e., manipulate) a View during a simulation. Concrete modifiers include the keyboard and the mouse. Other possibilities include joysticks, head-mounted displays, etc.

2.7.3 Importance to Research

This research investigated joining the Easy_Sim application framework and the SimWorx application framework to create an HLA-compliant, DIS-capable virtual combat simulation.

2.8 Summary

This chapter attempted to provide detailed background information covering the issues of this thesis. The disjoint nature of the chapter's organization is by design, since DIS, HLA, and various software engineering concepts aren't easily reconciled within a simple conceptual framework (no pun intended). The following chapter describes the methodology used to synthesize these disparate concepts to create the SimWorx Application Framework for Distributed Simulation, the main product outcome, or deliverable, of this thesis.

This Page Intentionally Left Blank

3. Thesis Methodology

This chapter discusses the methodology used to create the SimWorx application framework. The method used stems from a synthesis of ideas from Booch [Boo94], Rumbaugh [Rum91], [Rum94], Jacobsen [Jac92], Adair [Ada95], and others. Overall, the approach was iterative phases of object-oriented analysis/design followed by implementation in Ada95. The framework's overall architecture is represented using Kruchten's 4+1 view model [Kru95].

The following sections explain the importance of iteration in application framework development and outline the analysis and design steps used to create the SimWorx application framework.

3.1 Iteration

In the past 10 years, software methodologists have stressed the importance of iterative software development. With software, it's difficult to get a product right the first time. This difficulty stems from the very nature of reification, that is, making abstract ideas concrete. As one "fleshes out" an abstract idea, disturbing details are discovered which challenge the original intellectual underpinnings of the abstraction. The vexing details are often accommodated in a manner which has the least overall impact to the system's current state. The result is a complicated, ever-growing nest of conflicting conceptual modifications patched on top of the system's original, pure (but possibly weak) conceptual foundation. However, if one takes an iterative approach, those vexing details can be discovered earlier and accommodated before too much has been built upon a weak foundation.

Kruchten [Kru95, 49] describes an iterative approach to software architecture employing working prototypes to validate assumptions and decisions. This approach not only mitigates risk, but provides developers with experience to apply to later iterations of the project. Adair [Ada95] states an iterative approach to building application frameworks provides recurring opportunities to find ways to make client programmers' tasks easier. Early versions of a framework can be application-specific, and thus easier for the framework developer to understand, and implement. Later iterations of the framework can focus on generalizing a specific implementation to an entire class of related applications.

Given the advantages of an iterative approach, the SimWorx development effort consisted of three development iterations (further iterations are possible, but were not in the scope of this effort). The first iteration resulted in a framework which could receive information from other distributed simulations, but could not transmit to them. The second iteration added the capability to transmit information. The third iteration combined the SimWorx framework with AFIT's Easy_Sim real-time graphics simulation application framework. Each of these main iterations consisted of small micro-iterations--conceiving, building, and testing classes and their interactions with other classes--growing the framework within each main iteration.

3.2 Analysis

Rumbaugh describes analysis as a process which produces a precise, understandable, correct model of the real-world [Rum91, 148]. Rumbaugh's technique uses three different representations to describe three aspects of that model. The *object model* describes the static relationship between elements of the model. The *dynamic model* describes the sequence of interactions within and between the elements of the model. The *functional model* describes the data transformations of the model. The three representations, taken as a whole, provide the precise, understandable, correct model that is the goal of analysis. This analysis model serves as the Logical view of Kruchten's 4+1 view model [Kru95, 43]

Analyzing the real world is difficult. Although Rumbaugh provides steps and guidelines for building the analysis model, this research augmented Rumbaugh's basic analysis method with the use of a kind of scenario known as a *use case.* A use case describes a sequence of possible interactions between a system and its user. Each way of using the system is a use case.

Employing use cases offers several advantages [Rum94]. Use cases focus the system analysis on the user's needs, promoting system validity. Use cases can also be used to partition the system into separate threads of control. Use cases are valuable throughout the software lifecycle, not just the analysis and design steps--they can be used to partition the system's validation tasks into separate independent validation sequences.

Although use cases by their very nature are user-centric, that does not preclude them from use on systems without external users or with little human interaction. In cases such as these, the use cases can be drawn from the viewpoint of significant system objects.

A use case approach helped focus the analysis of the SimWorx application framework, drawing an understanding of the problem domain from an unfocused collection of discrete requirements to a comprehensible set of problem domain behavioral sequences.

The following analysis steps were used in the construction of SimWorx.

Analysis Steps:

1. Enumerate a set of use cases (simply name them). Identify a core subset of the use cases and describe them in more detail.
2. Identify objects and classes of objects. These are objects from the HLA Run Time Infrastructure (RTI) documentation and significant objects from use cases.
3. Identify attributes of objects/classes. These define the character and nature of different instances of the classes.
4. Identify associations between objects. These are drawn from the static relationships between the objects and the dynamic relationships identified by the use cases.
5. Identify sequencing of interactions between objects. These are drawn from the use cases.

The SimWorx analysis documents (appendices B and D) resulted from the above steps. SimWorx use cases are presented as Message Trace Diagrams (see Appendix A for a key to reading Message Trace Diagrams). Note that most SimWorx object and class attributes were not discovered during analysis, since many of them don't exist at a high level of abstraction. Thus, there are only a few in the SimWorx analysis documents.

Analysis for the SimWorx framework took on a slightly different flavor from the analysis examples typically provided by methodologists (e.g., ATM machines, hydroponics experiment stations, robot controllers, etc.) due to the inherent abstract nature of application frameworks. Objects in the problem domain are themselves software objects (objects, interactions, Protocol Data Units, etc.), rather than concrete physical objects (cruise missiles, traffic lights, robots, etc.). This made the indistinct dividing-line between analysis and design become even more nebulous.

3.3 Design

Software design is the process of transforming an analysis model of the problem domain into a concrete model in the solution domain. Rumbaugh divides software design into two phases: System Design and Object Design. System Design corresponds closely to a software architecture level of design. Object Design is concerned with the design of the individual objects of the software system. This research design approach differs from Rumbaugh in that some of the details of the object design are relevant to the overall architecture of the system, thus a certain amount of Object Design is necessary before the overall System Design can gel.

Design Steps (not necessarily ordered):

- Reify use cases to concrete scenarios to discover important design details. These are the scenarios in Kruchten's 4+1 view model.

- Augment problem domain objects with solution domain objects. Concrete detail will reveal additional objects which are part of the solution domain which are necessary to perform the abstract interactions visible in the analysis model. This results in extensions to the 4+1 Logical view.
- Find operations on classes from analysis models, i.e., determine the classes' and objects' methods.
- Look for design patterns to apply to problems, or reform problems to fit appropriate patterns.
- Structure objects and classes according to architectural considerations. Kruchten suggests characterizing the classes by autonomy, persistence, subordination, and distribution [Kru95, 47]. Gomaa [Gom93, 187], provides guidelines for allocating classes to independent concurrent tasks. This will result in high level changes in the 4+1 Logical view and definition of the 4+1 Process view.
- Iterate through different implementation and packaging approaches within the implementation language. Ada 95 often provides several different ways of implementing a solution. The best may not be evident without actually trying to implement several. This step defines the 4+1 Development view.
- Look for opportunities for reuse. Reuse can be applied with design patterns (mentioned above) or through more traditional *ad hoc reuse*, reusing other people's source code.

The SimWorx design documents (appendices C and E) resulted from the above steps. SimWorx scenarios are presented as Message Trace Diagrams, just like the use cases--scenarios simply have more detail.

This chapter has discussed the research methodology used to create the SimWorx application framework. The next chapter describes the resulting framework, discussing analysis, design, and implementation decisions.

This Page Intentionally Left Blank

4. Application Framework Analysis, Design, and Implementation

This chapter describes the results of applying the methodology described in Chapter 3 to creating the SimWorx application framework for distributed simulation. The first section describes the product goals of this thesis. The second section shows how the product goals drove an overarching software architecture for the SimWorx framework. The remaining sections describe the analysis, design, and implementation decisions of the three development iterations.

4.1 SimWorx Application Framework Product Goals

As discussed in previous chapters, this research applied the Ada language to create an environment for experimenting with distributed simulation. The main goal of this thesis effort was the creation of an application framework which supports both DIS and HLA. That main goal implied three distinct subgoals: support for DIS, support for HLA, and improved client developer productivity in implementing distributed simulations.

4.1.1 Support for DIS

Support for DIS in the AFIT Graphics Lab meant interoperability with legacy lab applications to provide greater research opportunities. Legacy applications principally employed the DIS 2.0.3 communication standard with lab-specific extensions, such as the AFIT Environment Protocol Data Unit (PDU) used to convey changing environmental information between simulations. Principle lab uses have traditionally been combat simulations--there has been no need in the past for DIS capabilities in Logistics Support, Simulation Management, Distributed Emission Regeneration, and Radio Communications. Therefore, for this research, it was appropriate for the initial SimWorx application framework to focus on combat simulation, as long as extended DIS capabilities were not precluded from future versions of the framework.

4.1.2 Support for HLA

Support for HLA in the AFIT Graphics Lab meant providing services for simulations defined via an HLA object model. Unfortunately, since the actual High Level Architecture was not available at the time of this research, SimWorx HLA support could not be limited solely to the federate component of an HLA simulation. Instead, the SimWorx framework implementation had to support both the Federate and a surrogate for the HLA Run Time Infrastructure (RTI). Although this approach sounds imposing, it actually provided a means of ensuring DIS compatibility by hiding much of the DIS domain details inside the surrogate RTI.

"Support for HLA" evolved to the design goal of being as HLA-like as possible to minimize integration woes with the government furnished RTI when it becomes available and when DIS compatibility will no longer be required at AFIT. Similarity with the HLA Interface specification (if not equivalence) was important to avoid problems with architectural mismatch [Gar95].

4.1.3 Improved Client developer Productivity

As mentioned in previous chapters, the main point of an application framework is to improve client developer productivity by providing a pre-fabricated skeletal design of a software application. In light of this, reducing client developer workload was an important goal of the SimWorx framework. Client developers should be able to simply define a Simulation Object Model [DMSO96b], for their simulation, and then implement those classes and their corresponding interactions. Interaction with the HLA RTI should be nearly transparent, allowing the client developer to focus on his/her specific problem at hand.

Measuring or proving productivity improvements was beyond the scope of this thesis effort. Other research efforts have documented the productivity improvements of using frameworks. Snyder found that Graphics Lab researchers using his application framework were able to concentrate on the domain of their applications rather than becoming consumed with implementation details [Sny93, 79].

4.2 Deriving an Overall SimWorx Software Architecture from Product Goals

This section discusses how the overall SimWorx software architecture was derived from the product goals presented above.

4.2.1 Product Goals Drive Initial Analysis Model

During analysis, software engineers are often faced with a daunting, almost infinite set of possibilities to explore. In the case of this research, however, these possibilities were constrained by the three overall product goals for the SimWorx framework.

In the lifetime of the SimWorx framework, HLA support will outlive the requirement for DIS support since DIS will be superseded by HLA within the DoD. Thus, the DIS support goal is subordinate and analysis could focus on satisfying HLA constraints first.

DIS Support is achieved by performing dead-reckoning and supporting the Entity State, Fire, Detonation, and Collision PDUs. Later revisions of the SimWorx framework may support other PDUs.

The client developer productivity goal leads to a design which automatically handles simulation "house-keeping" tasks, such as initialization, storing and retrieving state information, and other general simulation services. Further, the design is flexible to provide a variety of ways for client developers to produce applications. Specifically, the framework is *inheritance focused* [Tal94, 4] in order to allow client developers latitude in specializing existing classes through inheritance and overriding member functions.

Since the HLA support goal is the most important, analysis began by focusing on the HLA-defined simulation architecture.

4.2.2 HLA Driven Architecture

The HLA RTI defines an overarching architecture for participating federate simulations. Each simulation presents a standardized interface to the RTI as specified by the Federate Ambassador[4] Interface Definition Language (IDL) specification [DMSO96a, 111]. The RTI's interface is specified in a corresponding RTI Ambassador IDL specification [DMSO96a, 96]. These specifications coerce an initial "big picture" architecture for the SimWorx application framework (see Figure 4-1 below).

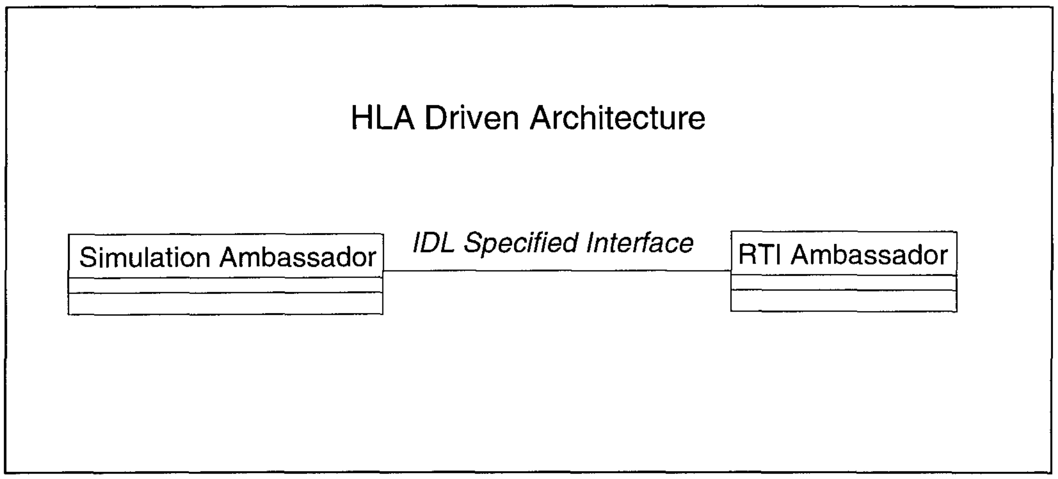


Figure 4-1. HLA Driven Architecture

Although each interface specification lists more than 20 "methods" or functions each ambassador performs, this effort only implemented the minimal subset necessary for a simulation to participate in a DIS network as a DIS simulation--full implementation of all methods was beyond the scope of this effort. Thus, the initial HLA functionality is limited to the following

[4] The term *Ambassador* is an HLA term for a kind of interface software component. Just as a diplomatic ambassador represents a foreign country, an *Ambassador* represents a local application programming interface for a system component which may actually be physically located somewhere else (another computer, another state, etc.). Thus, the client can make software procedure calls to the Ambassador, rather than relying on some specific message passing interface, since the Ambassador "abstracts-out" the details of the actual interaction.

methods, or operations (broken down by HLA Interface Specification [DMSO96a] function categories):

1. Federation Management
    - None (although these functions are vital for future SimWorx implementations)
2. Declaration Management
    - None (beyond scope of DIS)
3. Object Management
    - Request ID
    - Register Object
    - Discover Object
    - Delete Object
    - Remove Object
    - Update Attribute Values
    - Reflect Attribute Values
    - Send Interaction
    - Receive Interaction
4. Ownership Management
    - None (although beyond the scope of DIS, this is another area vital to future SimWorx development since it is a core HLA compliance area)
5. Time Management
    - Request Federation Time
6. Data Distribution Management
    - None (beyond scope of DIS)

The methods above enable a SimWorx HLA federate to create objects which interact with each other and with the objects of other federates. The federate operates in real-time, rather than in an event-based-time environment. All objects of the federation are known to all federates since no HLA Declaration Management methods are used. Objects can not abrogate responsibility for modeling their attributes since no HLA Ownership Management functions are used.

4.2.3 DIS as a base application within HLA

AFIT-required DIS functionality is specified in HLA terms by using the following mapping from DIS to HLA concepts:

1. DIS Entities are equivalent to HLA objects. The reception of an entity state PDU is equivalent to the HLA RTI *Discover Object* method if it is the first PDU received for a given entity. Subsequent entity state updates are equivalent to the HLA RTI *Reflect Attribute Values* method.
2. The DIS Collision, Fire and Detonation PDUs are all forms of HLA interaction.

Given this mapping between DIS and HLA, required DIS functionality can be represented within the Federate Ambassador as HLA objects and interactions. DIS transmission, reception, and dead-reckoning requirements can be handled by a DIS version of the HLA RTI (i.e., the surrogate RTI mentioned above). See Figure 4-2 below for details (Appendix A provides a key to reading Rumbaugh diagrams).

Further, the DIS Entity Objects mentioned above form the basis for a DIS-based HLA federation with object classes based upon different DIS entities, and interaction classes based upon "interaction type" PDUs. In a sense, DIS functionality is itself a base client application in the SimWorx framework. That is, it forms a layer of application software common to all DIS Federates founded on the "core level" of HLA functionality. When the AFIT Graphics Lab abandons DIS, the DIS portions of the SimWorx framework can be discarded, leaving the core SimWorx skeleton remaining.

Thus, the resulting overall software architecture for the SimWorx application framework is an example of what Garlan & Shaw [Gar93] refer to as a heterogeneous software architecture since it is both object-oriented and layered.

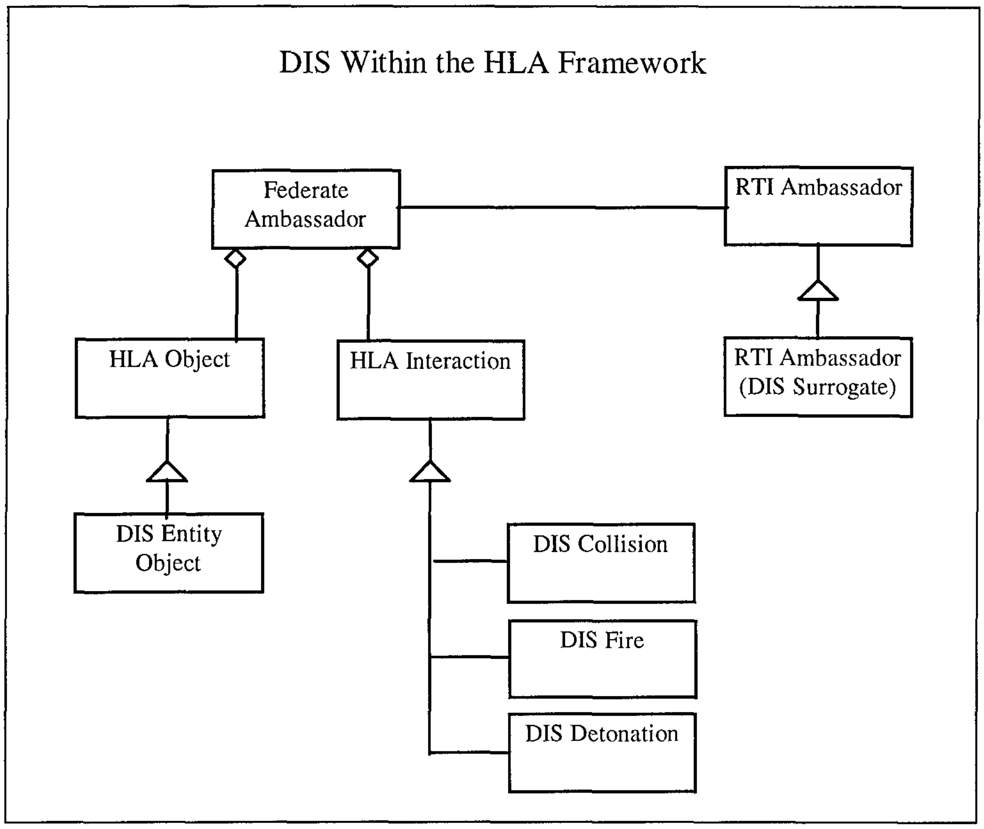


Figure 4-2. DIS Within the HLA Framework

4.3 Analysis and Design Decisions by Iteration.

The following sections present the evolution of the SimWorx framework, created following the steps outlined in Chapter 3. Analysis and design products are attached in Appendices B through E. Iteration One focused on the capability to receive information from other distributed simulations.

4.3.1 Iteration One Analysis

Based upon the overall SimWorx architecture model presented in the section above, Iteration One analysis began with use cases (see Appendix B). Use cases were defined by imagining the actions a surrogate DIS RTI would undertake to read, manage, and convert DIS Protocol Data Units (PDUs) into HLA information to pass to a DIS federate. Four HLA

Interface Specification methods define the interface from the RTI to the federate for passing that information: Discover Object [DMSO96a, 44], Reflect Attribute Values [DMSO96a, 46], Receive Interaction [DMSO96a, 49], and Delete Object [DMSO96a, 50]. These methods formed the core of the use cases.

Classes and associations between them were drawn from the use cases. Note the similarity between Figure 4-2 above and the SimWorx Analysis Architecture Figure 4-3 below. The main structure of the SimWorx framework is defined by the HLA Interface Specification as outlined above. The analysis architecture is neatly split into two main components: The RTI and the Federate.

The DIS Surrogate RTI performs DIS simulation functions specified by the DIS Standard, i.e., receiving PDUs, dead reckoning entity states between PDU updates, and forwarding state information to the Federate as it is received or generated.

The Federate consists of HLA_Objects which represent the DIS entities of the simulation, and HLA_Interactions which represent the possible interactions between the entities. Since Iteration One was only concerned with input from the DIS network, HLA_Objects had no intrinsic behavior at this point in SimWorx development--their behavior was solely determined by the other simulations (federates) on the network.

Federate actions (Object Interaction, Use Interaction) in Iteration One use cases are rather weak, but that didn't really affect the outcome of the design since those cases weren't really applicable to DIS network input because no simulating is actually being done by the Federate (in retrospect, these two use cases probably should have been omitted from Iteration One, but then again, that's part of the benefit of making software iteratively, one can learn from mistakes without having to pay a dear price).

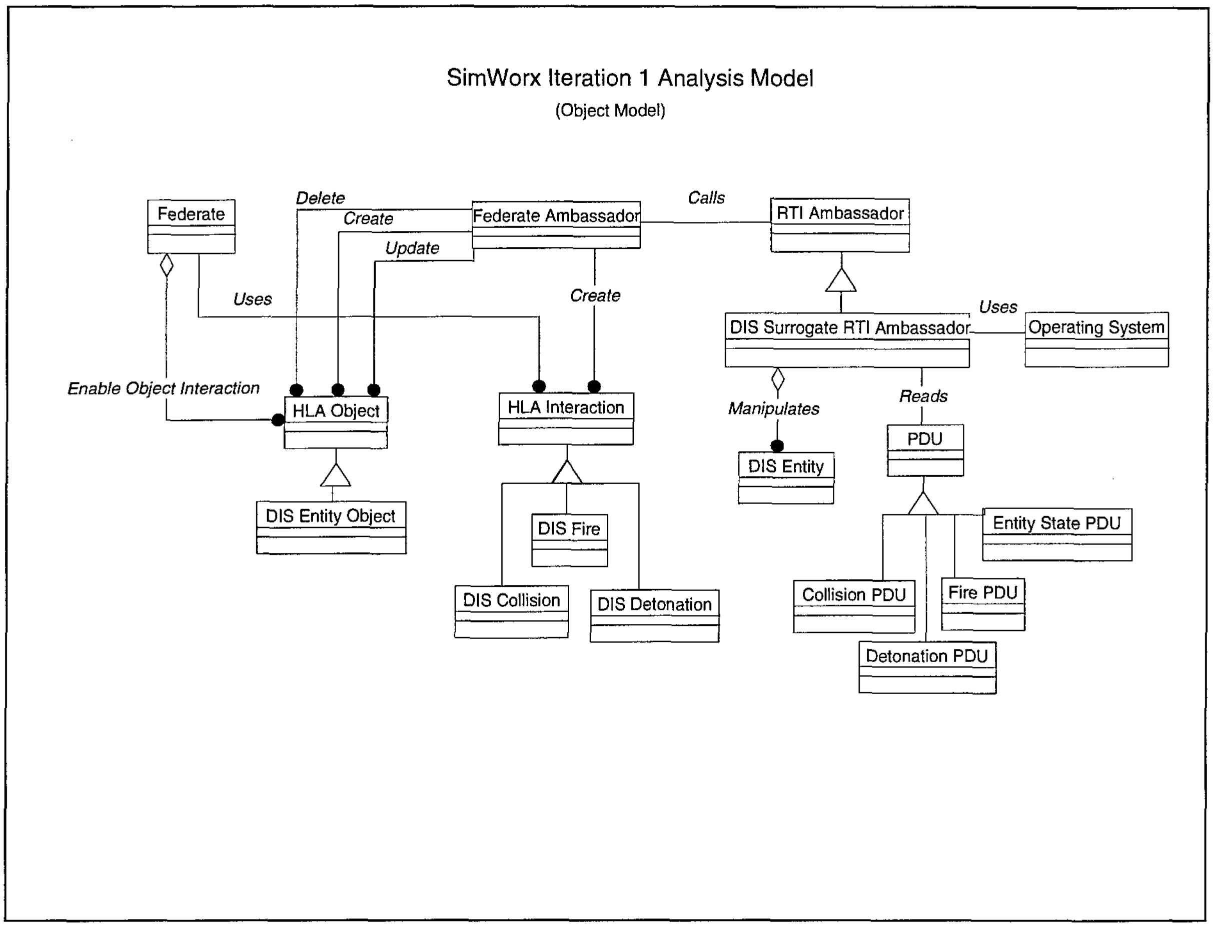


Figure 4-3. SimWorx Analysis Architecture

4.3.2 Iteration One Design

Iteration One design evolved from the Iteration One analysis model via the design process steps outlined in Chapter 3. Use cases were refined into concrete scenarios to find solution domain details missing in the analysis model. The following paragraphs discuss resulting design concurrency issues and solution domain classes. Note how the Iteration One design is much more concrete than the analysis level architecture--this is evident by comparing the analysis diagram with the design diagrams in figures 4-4, 4-5, and 4-6 below.

The design reflects the division of labor suggested by the Analysis Architecture (see Figure 4-3)--it is split into two main pieces, the Federate Category[5], and the RTI Category, with a third piece, the Support Category, playing a supporting role. Figure 4-4 shows the Federate Category--it is responsible for the federate-like behavior of SimWorx. The Federate Category's sole dependence upon the RTI Category comes from method calls to the Abstract_RTI_Ambassador (see right side of figure). Figure 4-5 depicts the RTI Category. It is responsible for the RTI-like (ergo DIS-like) behavior of SimWorx. Figure 4-6 shows two support classes identified during design. These classes are used by both the Federate and RTI categories.

4.3.2.1 Scenarios

Use of scenarios was instrumental in discovering the details of the interactions between SimWorx classes identified during analysis. Further, scenarios were key in discovering additional solution domain classes (see below).

At the design level, the principle class structuring issue [Kru95, 47] was resolving the mutual dependence identified in the scenarios between the Federate and RTI Ambassadors. Although mutual dependence is more of an implementation issue (Ada does not support circular "with" relationships among packages) the solution to the problem affects the design level.

[5] *Category* is a Booch term [Boo94] for a mutually cohesive collection of classes. Grouping classes into categories supports loose coupling and high cohesion in object-oriented systems.

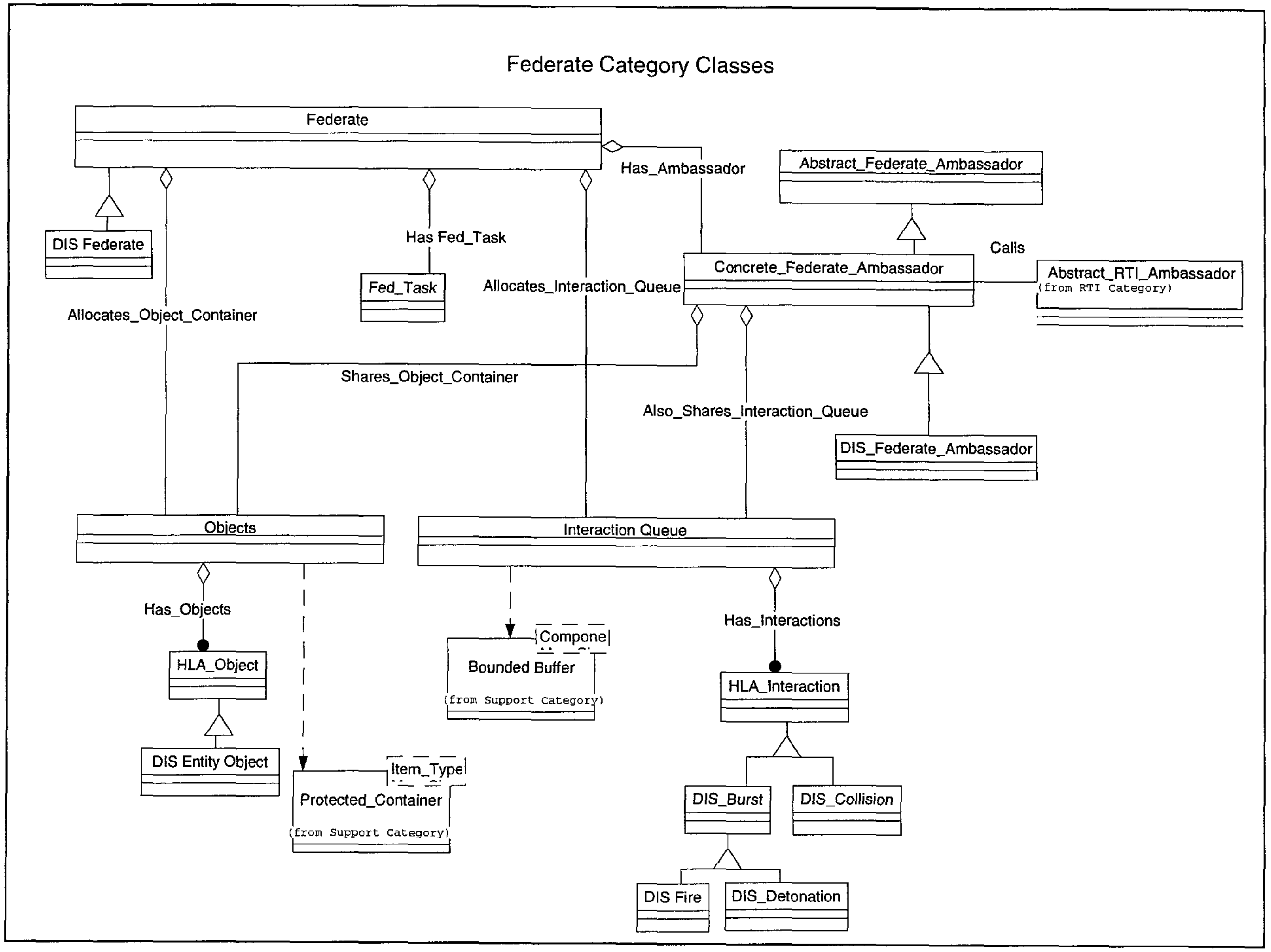


Figure 4-4. Federate Category Classes

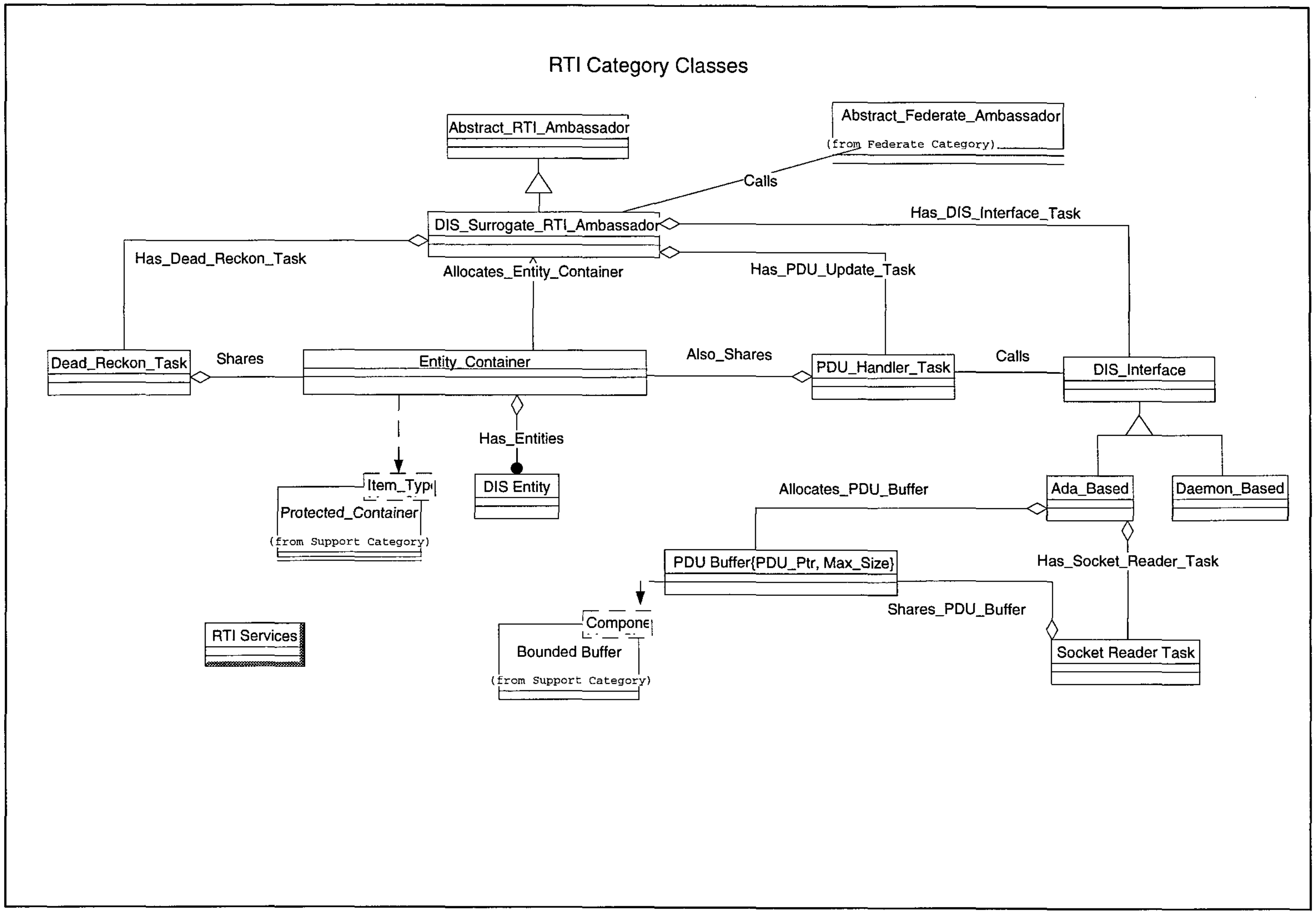
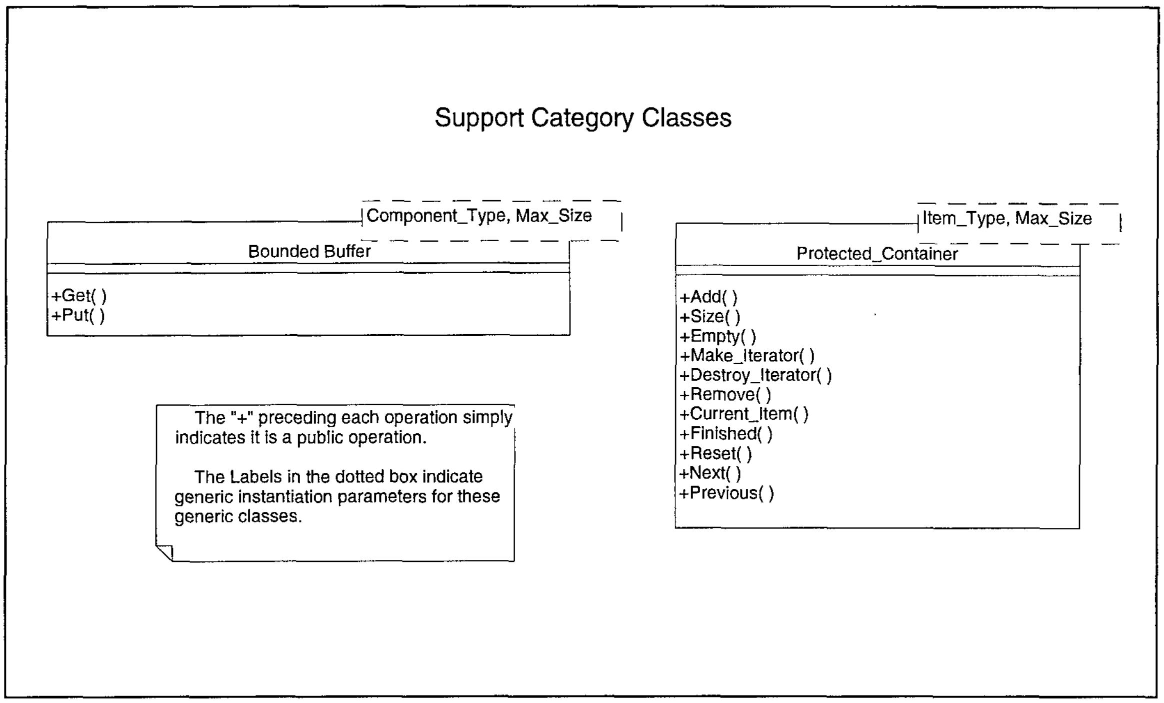


Figure 4-5. RTI Category Classes



The "+" preceding each operation simply indicates it is a public operation.

The Labels in the dotted box indicate generic instantiation parameters for these generic classes.

Figure 4-6. Support Category Classes

Figure 4-7 below shows the structure of the solution to the mutual dependency problem. The idea is to make the RTI and Federate Ambassadors dependent upon an abstract parent class. Cohen provided this solution using dummy parent types [Coh96, 563]. SimWorx however, does not use dummy parent types. The Abstract_Federate_Ambassador and Abstract_RTI_Ambassador classes serve as the external interface specification for the methods specified by the HLA Interface Specification. That is, unlike the DIS_Surrogate_RTI_Ambassador and the Concrete_Federate_Ambassador which have SimWorx specific methods and behavior, the abstract parents declare only the behavior specified by the HLA Specification.

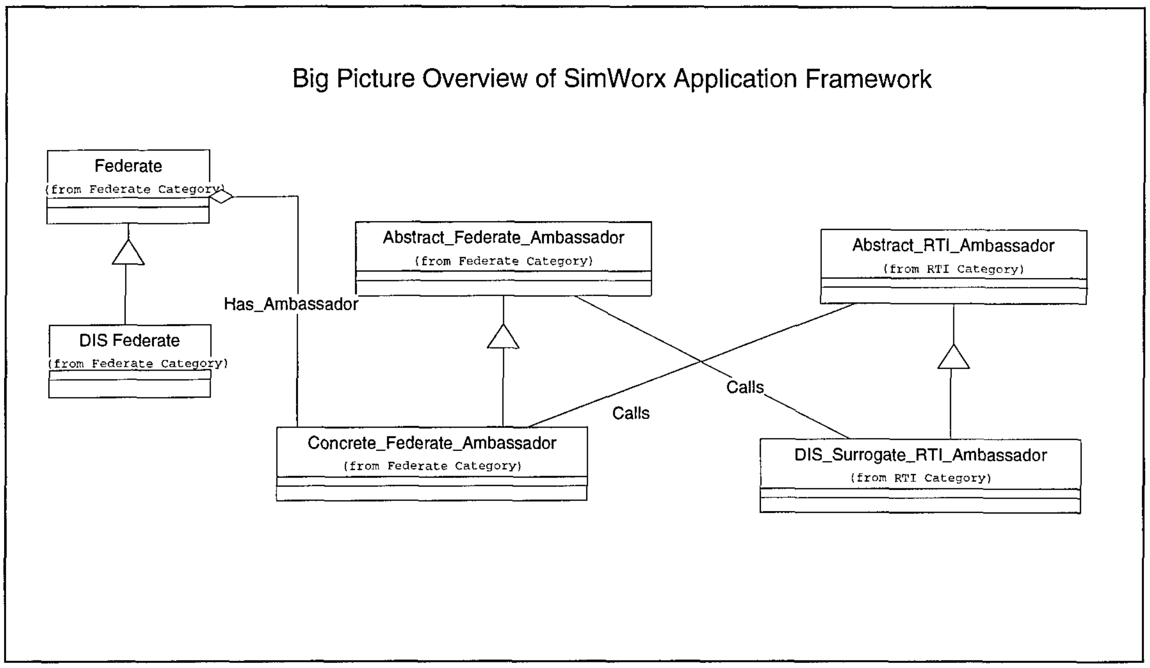


Figure 4-7. Big Picture Overview of SimWorx Application Framework (Depicts Solution to Mutual Dependency Problem)

The only other class structuring issue was creating the container classes to keep track of multiple object instances. Thus, following the aforementioned class structuring issues, major design issues encountered were related to handling the concurrency of class operations.

4.3.2.2 Concurrency Issues

The DIS Standard defines dead reckoning performance requirements which must be met regardless of a simulation's function or purpose [DIS94b, 24]. These requirements point to a concurrent implementation where federate simulation and dead reckoning are separate threads of control, allowing the federate thread to be free of the timing constraints of the dead reckoning thread.

Opening the door to a concurrent solution, however, imposes other design constraints. For example, dead reckoning is a periodic task which must occur at some rate which ensures fidelity of the modeled entity. Conversely, PDU Reception is asynchronous. Thus according to

Gomaa's task structuring criteria [Gom93, 188], dead reckoning and PDU handling should be separate tasks.

Another example of design complication due to concurrency is the need to protect objects from concurrent access. In SimWorx, this was handled via various Ada 95 constructs--details follow in section 4.3.3 below.

4.3.2.3 Solution Domain Classes

The following paragraphs discuss some of the interesting new classes discovered during design. A description of all of the SimWorx classes from Iteration One is located in SimWorx Iteration One Design Document (Appendix C).

Two generic classes were defined during design: Protected_Container, and Bounded_Buffer. Identifying container classes is relatively simple: each one-to-many association in the analysis model (e.g. "Federate Uses HLA_Interaction") indicates an opportunity to contain the "many" class (i.e., HLA_Interaction in previous example) [Rum93, 246]. The resulting containers are the Objects container of HLA_Objects, the Entity_Container of DIS_Entities, and as a special case, the Interaction_Queue of HLA_Interactions (see below).

Initial development micro-iterations attempted to employ ad-hoc reuse by utilizing Mr. David Weller's Ada 95 Booch Component's Queue class [Wel95, 207-213] as a container since it supported broad container operations beyond the typical enqueue and dequeue operations. These attempts failed for two reasons. First, Mr. Weller's software didn't work (problems existed in compiling and linking generic child packages). Second, Weller's queue could not work correctly in a concurrent environment.

In a concurrent environment, containers must guarantee atomic access to their contents to ensure the correctness of resulting computations. Thus, the SimWorx Protected_Container class is implemented as a generic Ada Protected Object. It isn't really a bona fide class since Ada Protected Objects are not inheritable.

Containers can be made more flexible through the use of *iterators* [Gam95, 257]. An *iterator* is a key or token or "cursor" used by clients to keep track of the "current element" during a traversal of a container (assuming the container is ordered somehow, i.e., it is not a set or a bag). Iterators for a container can be external or internal. With an external iterator, the client controls the manipulation of the "current element" pointer. With an internal iterator, the "current element" pointer is maintained by the container. The SimWorx Protected_Container provides external iterators to allow clients flexibility in how they access the elements of the container. For example, one client may have several traversals pending simultaneously via multiple iterators associated with the same container.

A concurrent container with multiple external iterators must employ some mechanism to ensure container additions and removals don't affect clients who are iterating through the container. A *robust iterator* [Gam95, 261], [Kof93] ensures that iterations and removals don't interfere with a client's traversal of the container. The SimWorx Protected_Container provides robust iterators by keeping track of the state of each external iterator, and adjusting it, when necessary, to account for added or deleted container elements.

In contrast to the Protected_Container, the generic Bounded_Buffer is a textbook example of the use of Ada Protected Objects. In fact, it was essentially copied from Burns and Wellings' Concurrency in Ada [Bur95, 218]. Essentially, it enables some supplier task to "Put" items in the buffer, and some consumer task to "Get" items from the buffer. It's instantiated as the Interaction_Queue and as the PDU_Buffer of the Ada_Based DIS Interface.

The DIS Interface (see right side of Figure 4-5) evolved from the Operating System interface of the analysis architecture. It is an abstract DIS Interface for reading and writing PDUs. This abstract interface has two concrete child classes. The first is an Ada-Based interface which reads and writes PDUs directly to a socket. The second (the Daemon interface) relies on an AFIT Graphics Lab standard PDU interface, written in the C programming language by Mr. Bruce Clay of Science Applications International Corporation. The Ada_Based interface was devised first since 100% development in Ada was a secondary goal of the research effort. The Daemon interface was added later as a means of ensuring future compatibility with AFIT DIS network upgrades. In the future, if the hardware or software implementation of the AFIT DIS network is changed (due to ATM, IP Multicast, etc.), those changes will be insulated from AFIT applications since the public Daemon interface will not be changed.

The RTI Services class represents two important Surrogate RTI services discovered during implementation: Time Services and ID Services. Time services were centralized in one package to reap the benefits of information hiding.

ID services were deemed necessary for two reasons. First, they are necessary for keeping track of IDs. During testing, it was discovered that remote DIS Entities could be represented within SimWorx with more than one HLA ID (violating the HLA standard [DMSO96a, 40]). This result occurred whenever an Entity State PDU for a given entity was not received within the “DIS Heartbeat” time threshold (by DIS regulation, simulations discontinue modeling absent DIS Entities). Subsequently, when the entity “reappeared” sometime later, SimWorx would issue a new (different) HLA ID. To overcome this problem, the ID services map DIS IDs to HLA IDs and vice versa. A given DIS ID and its corresponding HLA ID will correspond for the lifetime of a given federation execution. The second reason for defining centralized ID services was due to anticipation of the RTI Request_ID method [DMSO96a, 40] in Iteration Two.

Some analysis classes were *removed* at the design level of development. At the SimWorx design level, PDUs are no longer considered bona fide classes because they have no intrinsic behavior. PDUs simply contain data values for transmission across the DIS network--

their definition is static, defined by the DIS standard. Instead of encapsulating each PDU as a class, and thus forcing clients to access data via public method calls, PDUs were defined as simple record types--clients have direct access to their data. Although this sounds like poor software engineering practice, it is simply an example of a typical performance and labor versus clarity and maintainability design trade-off software engineers face. PDUs are defined by the DIS standard. Their representation is unlikely to change; thus, there is little value in encapsulating their representation to insulate clients from change.

4.3.3 Iteration One Implementation

The SimWorx Iteration One design was implemented in Ada 95. Generally, each class is implemented as an Ada 95 Tagged Type and encapsulated in its own Ada package using the ROMAN-9X technique [Cer93] (SimWorx Ada packages are depicted in the SimWorx Architecture Development Views, Appendix E, Figures 28, 29, and 30). Additionally, at the implementation level, there are several packages or services which don't exist at the design level: "types" packages (DIS.Types, SW.Types, and RTI.Types), Support.Semaphores, and Support.RT_Parms.

The types packages exist to centralize certain domain type information. The DIS.Types package encapsulates type information for items pertaining to the DIS standard. The SW.Types package provides global type standards for consistency throughout the application framework, insulating the framework from future types changes. The RTI.Types package encapsulates RTI-specific types.

The Support.Semaphores package is simply an Ada Protected type playing the role of a semaphore. It is used by HLA_Object (see below).

The Support.RT_Parms package (a.k.a. Run-Time Parameters) provides a means of specifying integer constant values at run-time. It is the first package elaborated when SimWorx is initialized. It reads a file of (String, Integer) pairs upon initialization. Following that, clients can pass a string to the RT_Parms.Get function and receive the string's corresponding integer. This capability is used throughout the SimWorx framework to specify values which would otherwise be specified via compile-time constants, like container sizes, etc.

Using a protected container is not enough to control concurrent access to HLA_Objects and DIS_Entities in the SimWorx framework since the containers are implemented as containers of *references* or pointers[6], and there is no guarantee that two different clients wouldn't attempt to access the same object at the same time. SimWorx uses two different approaches to solve this problem.

The first approach is via Ada 95's protected type. The Ada 95 protected type is a natural choice for guaranteeing atomic access to resources (like HLA_Objects or DIS_Entities). Unfortunately, it can't be used to represent a class since protected types don't support inheritance. Thus, HLA_Objects can not be implemented via protected types. Conversely, DIS_Entity is implemented as a protected type since there is no need to specialize DIS Entities through inheritance--their representation is based upon the DIS standard.

The second approach to solve the concurrent access problem is to guarantee atomic access via semaphores. This was the approach taken for HLA_Objects, since they are tagged types. Each HLA_Object has an associated semaphore to control access to it. The semaphore is implemented with an Ada 95 Protected type (see above) and is a private attribute of the HLA_Object. Client subprograms call any method of the HLA_Object using the following protocol.

[6] The Objects container contains references to HLA_Objects since different subclasses of HLA_Object may be present in the container at the same time--it would be impractical/nearly impossible to contain varying size subclasses in the container. The Entity_Container contains references to DIS_Entities for performance reasons--each DIS_Entity encapsulates too many bytes to be casually allocating, copying, and deallocating them with every container operation.

```
HLA_Object.Lock(MyObject);           -- Wait for exclusive
                                     --   access.
HLA_Object.Do_Something(MyObject);  -- Call some method.
HLA_Object.Unlock(MyObject);         -- Release exclusive access
```

It would be better if the client developer did not have the burden of explicitly manipulating the semaphore (lock). However, an approach which encapsulates the calls to the semaphore inside the Do_Something method can't be employed without debilitating side-effects. Burns and Wellings [Bur95, 318-326] describe several possible solutions to the problem, none of which provide the functionality required by the SimWorx framework.

Just as SimWorx uses several approaches to control concurrent access, it also employs two different task packaging approaches. The primary approach is to encapsulate each task in its own package. This approach is used for most tasks in the system, since it fits into the same conceptual framework as the package per class idea.

A secondary approach in which the task's definition is encapsulated in some other class's package is used for two different reasons. First, the DIS Interface classes encapsulate the tasks they employ in order to reinforce the intellectual concept of DIS Interface unity. That is, the DIS Interface classes are meant to be thought of as independent units, with no dependence on the other parts of the SimWorx framework.

Finally, the Fed_Task is encapsulated within the Federate package since it is mutually dependent with the Federate tagged type. That is, since Ada does not allow circular "with" relationships between packages, the Fed_Task type and the Federate tagged type must be declared in the same package (or else employ some kind of abstract parent type as noted in section 4.3.2.1 above). The Fed_Task is dependent upon the Federate since it calls the Federate's Client_Initialize, Client_Loop_Body, and Client_Finalize methods. The Federate is dependent upon the Fed_Task because the Fed_Task is one of its attributes.

4.3.4 Iteration One Application: The DIS Printer.

Each SimWorx development iteration has a resulting SimWorx application. The Iteration One application is the DIS Printer, which simply continuously prints to standard output the state of all of the DIS Entity Objects of a DIS simulation. It was created by subclassing the Federate class to iterate through the Objects container and invoke the Show method on each DIS Entity Object in the container (see Figure 4-8 below). It overrides most of the Federate's methods. The Federate_Ambassador_Factory_Method[7] method is overridden to allocate the DIS Federate Ambassador, (a specialized Federate Ambassador with knowledge of DIS Entity Obect classes). The Client_Loop_Body method is called periodically by the Fed_Task to "Show" each of the HLA_Objects in the Objects container. The DIS_Entity_Object Show method prints the current state of the object to standard output.

4.3.5 Iteration Two Analysis

Iteration Two builds upon the Iteration One framework by adding the capability to send information to the DIS network. Iteration Two analysis also began with use cases (see Appendix D). Use cases were developed around six methods defined in the HLA Interface Specification: Request_ID [DMSO96a, 40], Register_Object [DMSO96a, 41], Update_Attribute_Values [DMSO96a, 42], Send_Interaction [DMSO96a, 47], Delete_Object [DMSO96a, 50], Request_Federation_Time [DMSO96a, 72].

At the analysis level, no new classes were identified for Iteration Two. However, most classes gained additional responsibilities.

[7] The Federate_Ambassador_Factory_Method is a *Factory Method.* That is, it is a method which is a kind of "factory" for producing (allocating) Federate_Ambassadors. See [Gam95].

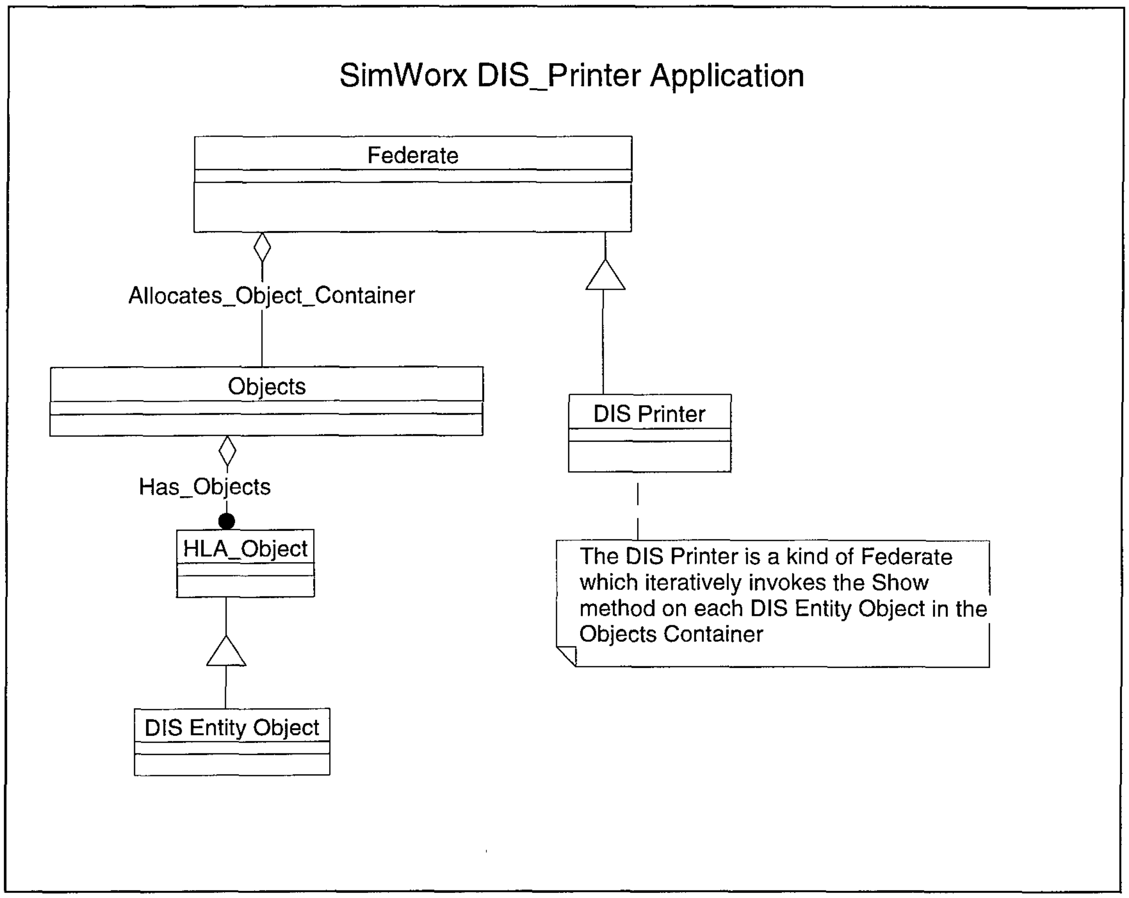

SimWorx DIS_Printer Application
Federate
Allocates_Object_Container
Objects
DIS Printer
Has_Objects
HLA_Object
DIS Entity Object
The DIS Printer is a kind of Federate which iteratively invokes the Show method on each DIS Entity Object in the Objects Container

Figure 4-8. SimWorx DIS_Printer Application

4.3.6 Iteration Two Design

Iteration Two design began by extending the Iteration Two use cases into concrete scenarios. Updates were made to both the Federate and RTI class categories (see Figure 4-9 and Figure 4-10 below).

For the Federate side of SimWorx, the policy of employing the Federate Ambassador as a service provider (from the product goal of improving developer productivity) takes shape in Iteration Two. The Concrete Federate Ambassador adds methods for creating and deleting HLA Objects, sending HLA_Interactions to the RTI, and functions for returning both the current federation time and a reference to the Objects container of shared HLA_Objects. More importantly, the Concrete Federate Ambassador orchestrates the automatic transmission of

HLA_Object state changes for Federate-owned objects to the RTI. (This is only a small beginning for the service provider policy, future versions of SimWorx should support Federate saves and restores via methods in the Concrete Federate Ambassador).

State change updates are handled in the following manner. The Federate Ambassador's Scan_Task periodically scans the Objects container, querying each HLA_Object to see if it has changed recently. If a change is detected, the Scan_Task builds a Attribute_Handle_Value_Pair_Set with the modified attributes and sends it to the RTI via the Attribute_Update method.

The Concrete Federate Ambassador's other concurrent task exists to manage deletion of objects in the Objects container. It was made a separate task following Gomaa's task structuring criteria [Gom93, 188], since deletion is an aperiodic task, unlike object scanning. It was important to establish one task which deletes HLA_Objects in order to avoid deadlock. If two separate tasks attempted to remove the same object from the Protected_Container, they would both wind up in endless loops waiting for the other to move its iterator off of the object in question.

The Scan_Task requires two pieces of information to fulfill its responsibility. First, it must be able to discern if an object has changed state. Second, it must only transmit information to the RTI for objects owned by the Federate (the other objects are actually modeled in other federates of the federation). Both capabilities are handled by adding attributes to HLA_Object.

Whenever an HLA_Object modifies its internal state, it sets its Changed flag to True. The Scan_Task queries each object's Changed method and only sends information to the RTI when Changed returns True.

Each HLA_Object also has a Locally_Owned flag to indicate if it is owned by the local Federate. The Scan_Task ignores objects which aren't locally owned. Future versions of SimWorx which support more advanced HLA ownership management functions will rely on the

Locally_Owned flag more heavily in order to support remote attribute ownership [DMSO96a, 59-70].

Design Patterns led to one of the most exciting SimWorx ideas: the view of an HLA_Interaction as an instance of the Command [Gam95, 233] design pattern. Viewing HLA_Interaction as a command opened up new vistas in creative thinking about how HLA_Objects can interact with each other.

The initial object interaction idea was to have locally-owned HLA_Objects directly call the methods of other objects to interact with each other, e.g., an aircraft fires on a tank by calling the tank's Handle_Fire method. Objects could interact indirectly with other objects via an HLA_Interaction, but that was only possible for non-Federate-owned objects. HLA_Interactions would be handled only by a method within the Federate (the Federate would examine the HLA_Interaction and directly call the appropriate HLA_Object methods). This scheme was bad for several reasons. First, it forced each HLA_Object to have knowledge of the objects it interacted with (mutual dependency problems highly likely). Second, it required the Federate to have knowledge of every kind of HLA_Object in order to convert HLA_Interactions into specific HLA_Object method calls.

Applying the Command pattern to HLA_Interactions side-steps the problems above. All inter-object communication is done via HLA_Interations. All HLA_Objects override a base method to handle interactions--Handle_Interaction. Thus, when an aircraft wants to fire at a tank, it builds an HLA Fire Interaction, and dispatches it to the tank's Handle_Interaction method, without having to "with" the tank package.

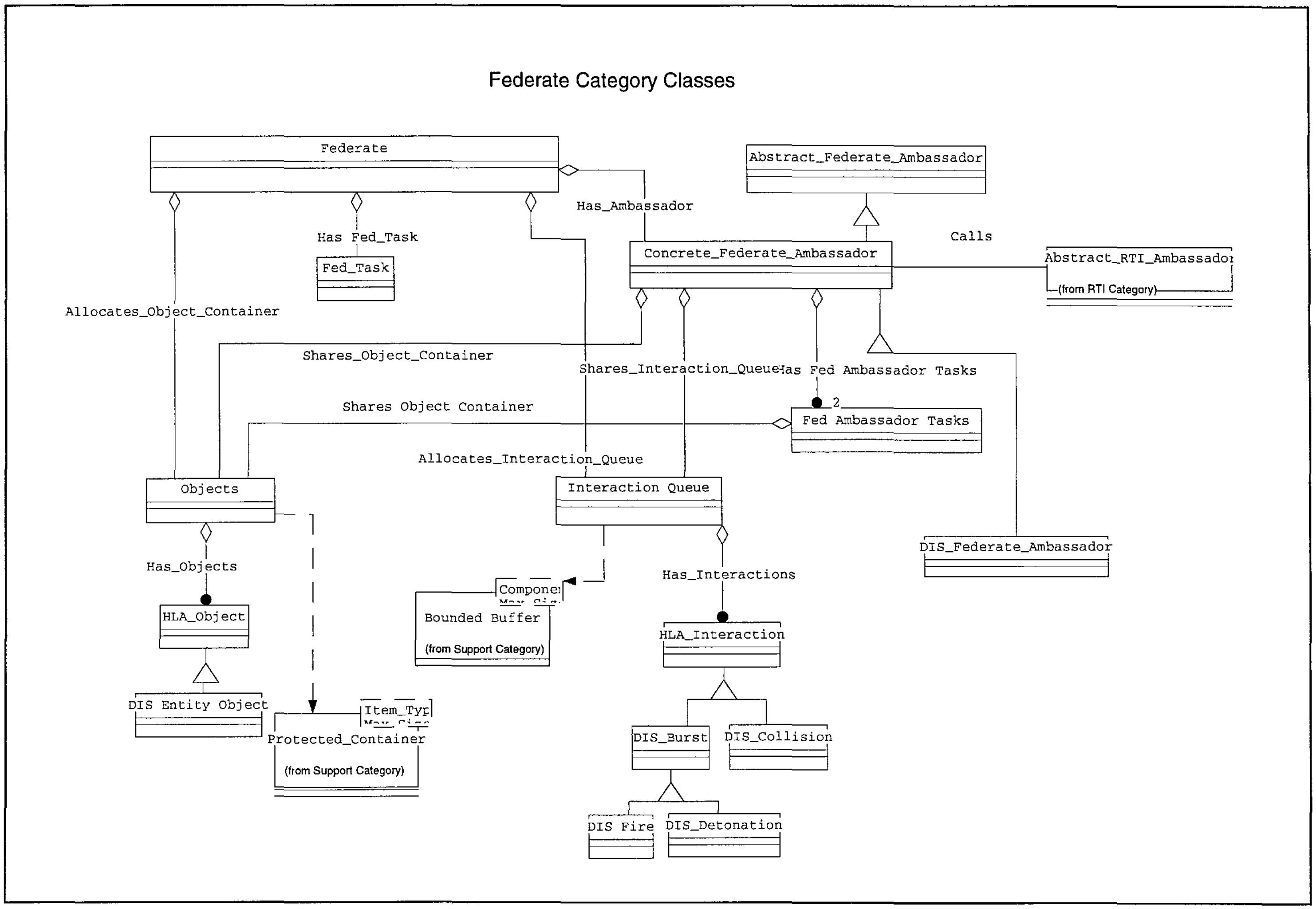


Figure 4-9. Federate Category Classes (Iteration Two)

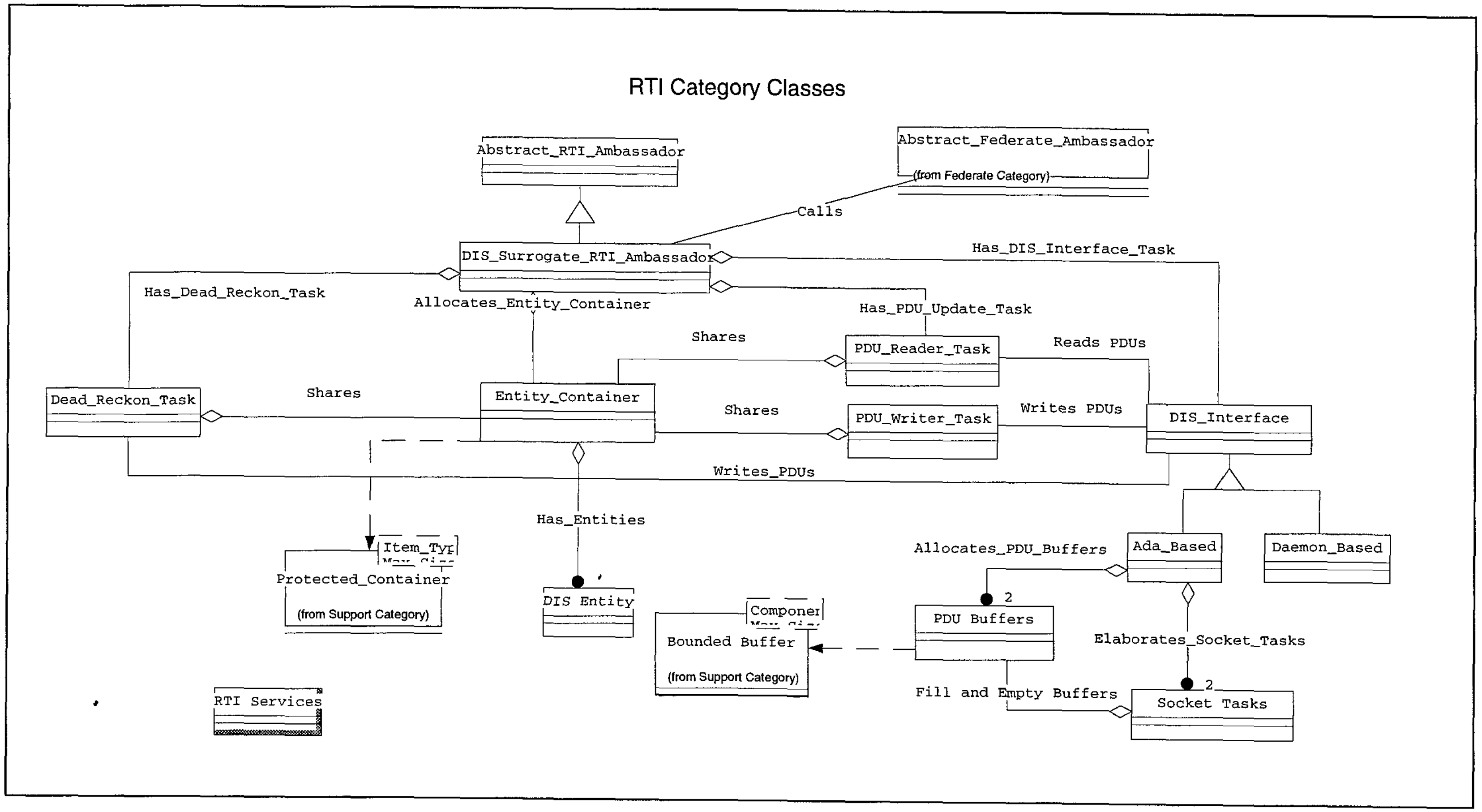


Figure 4-10. RTI Category Classes (Iteration Two)

The RTI also underwent changes for Iteration Two. The additional responsibility to broadcast PDUs to the network arose from two places. First, according to the DIS Standard [DIS94b, 23] a simulation must maintain a dead reckoned model of each entity it is simulating. Thus, the Dead_Reckon_Task must also dead reckon each locally owned entity and broadcast an Entity State PDU whenever the entity's appearance changes or its position or orientation change more than some threshold value. The Dead_Reckon_Task and DIS_Entity protected object were modified to handle these additional responsibilities.

Second, Entity State PDUs must be issued with each state change propagated from the Federate (regardless of when the last dead reckoned PDU was broadcast for the entity[8]). Since this is an aperiodic activity, a separate task (the PDU_Writer_Task) was created following Gomaa's task structuring criteria.

4.3.7 Iteration Two Implementation

There were no new implementation issues encountered in Iteration Two, since all of the interesting problems were solved in Iteration One.

4.3.8 Iteration Two Application: The SimWorx Gaggle

The application created for Iteration Two is the SimWorx Gaggle. The Gaggle is an extremely simplistic Computer Generated Forces (CGF) simulation. It creates some number of DIS Entity Objects (user defined) and moves them in a straight line at some user specified rate.

[8] Note that this may violate the DIS protocol standard since the RTI could issue "extra" PDUs even though a position or orientation threshold has not been exceeded. The violation stems from the fact that a DIS simulation is not supposed to issue a PDU unless its dead-reckoning model exceeds its high fidelity model by some threshold. The author considers this a minor problem which is unavoidable since the surrogate RTI has no idea what attributes are being updated during an attribute update operation (unless it explicitly looks at every attribute--a bad idea for performance reasons).

It was created by subclassing the Federate and DIS_Entity_Object classes (see Figure 4-11 below).

The SimWorx Gaggle Federate subclass creates and initializes the DIS_Entity_Objects within the Client_Initialize method. The Client_Loop_Body method was overridden to call each DIS_Entity_Object's Action method to cause it to move.

The Aircraft and B-52 subclasses were created to add movement behavior to the standard SimWorx DIS_Entity_Object class. Each time the B-52 Action method is called, the B-52 calculates a new position based upon its current position, its velocity vector, and the time since the last Action method call.

Users customize the SimWorx Gaggle via the SimWorx Run-Time Parameters file. They can specify the number and class of entities, their initial position, orientation, and velocity. The Gaggle is a useful tool for generating distributed simulation test data.

4.3.9 Iteration Three Analysis

Iteration Three represents the integration of SimWorx and Easy_Sim (see Chapter 2) to create a DIS-capable, HLA-based virtual combat simulation.

Iteration Three did not actually begin with an analysis phase for two reasons. First, there was no existing analysis model available for Easy_Sim with a level of abstraction corresponding to the SimWorx analysis model . Second, and more importantly, detailed understanding of how the frameworks functioned was necessary to determine how to integrate them--a detailed understanding which transcended the level of detail available in an analysis model. Thus, Iteration Three development began with design.

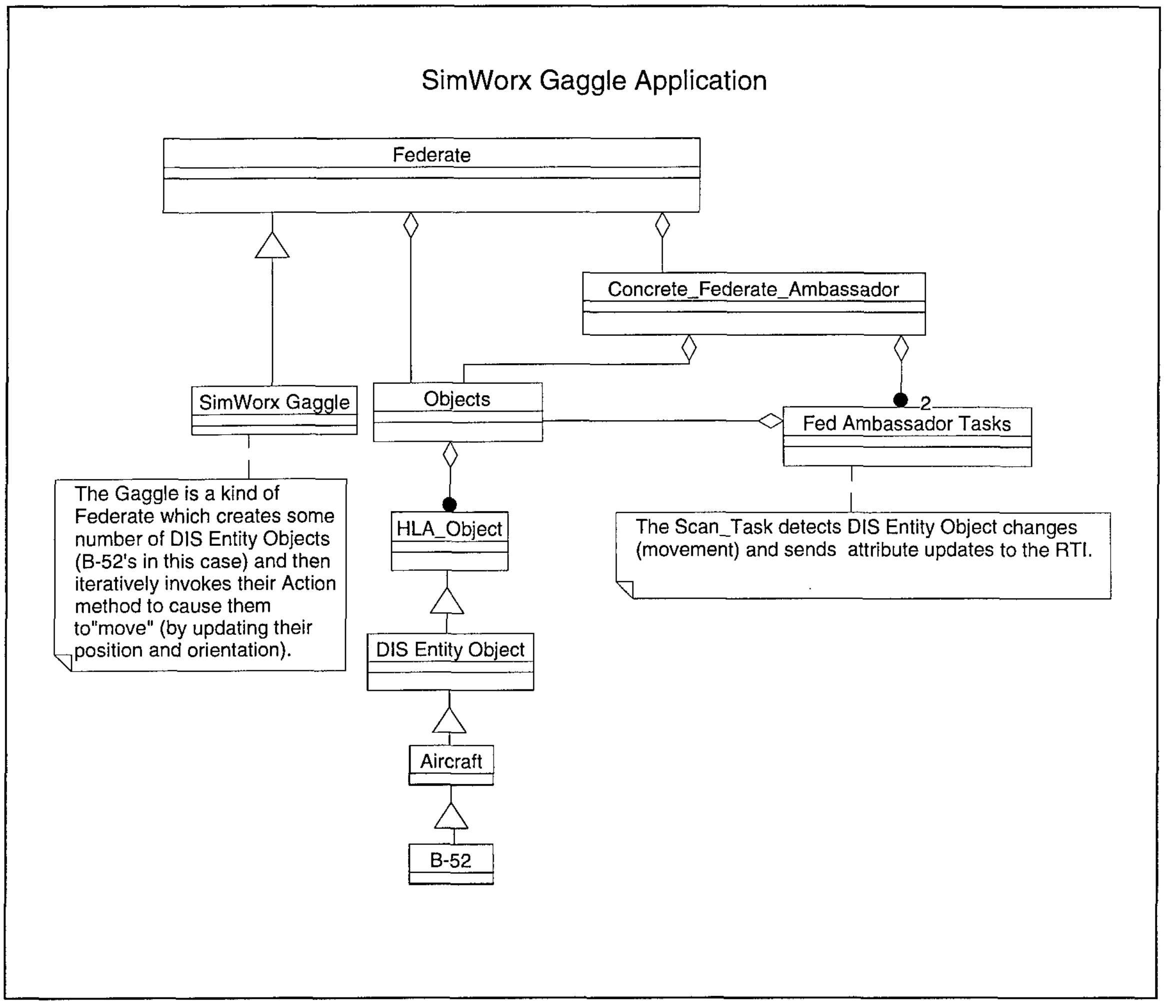

SimWorx Gaggle Application
Federate
Concrete_Federate_Ambassador
SimWorx Gaggle
Objects
2
Fed Ambassador Tasks
HLA_Object
DIS Entity Object
Aircraft
B-52
The Gaggle is a kind of Federate which creates some number of DIS Entity Objects (B-52's in this case) and then iteratively invokes their Action method to cause them to"move" (by updating their position and orientation).
The Scan_Task detects DIS Entity Object changes (movement) and sends attribute updates to the RTI.

Figure 4-11. SimWorx Gaggle Application

4.3.10 Iteration Three Design

Easy_Sim and SimWorx are very similar. Both are object-oriented, soft-real-time application frameworks for simulation written in Ada 95. Yet each has its own threads of control and distinct object classes. Integration of the two frameworks into a joint simulation system requires linking classes together and establishing a unified thread of control.

Which classes should be linked together? Examining the class specifications shows the Easy_Sim Simulation class and the SimWorx Federate class play similar roles: both perform initialization and overall control of a simulation. Similarly, the Easy_Sim Player and SimWorx HLA_Object both represent individual entities during a simulation.

It would seem straight-forward to combine these pairs into new classes which share the attributes of the parent classes. However, Ada 95 does not strictly support dual-inheritance and a joint Simulation-Federate does not behave exactly the same as the union of its parents' behavior. Instead, a better approach is to use *delegation* [Rum91, 244], where an object selectively invokes the methods of a related class. Rumbaugh regards delegation as a safer means of implementing joint behavior than dual inheritance since it prevents unwanted, meaningless operations from being inherited and prevents unwanted side-effect behavior. Delegation is achieved by giving one of the classes a reference to the other class via aggregation (containment). See Figure 4-12 below for a pictorial representation of delegation.

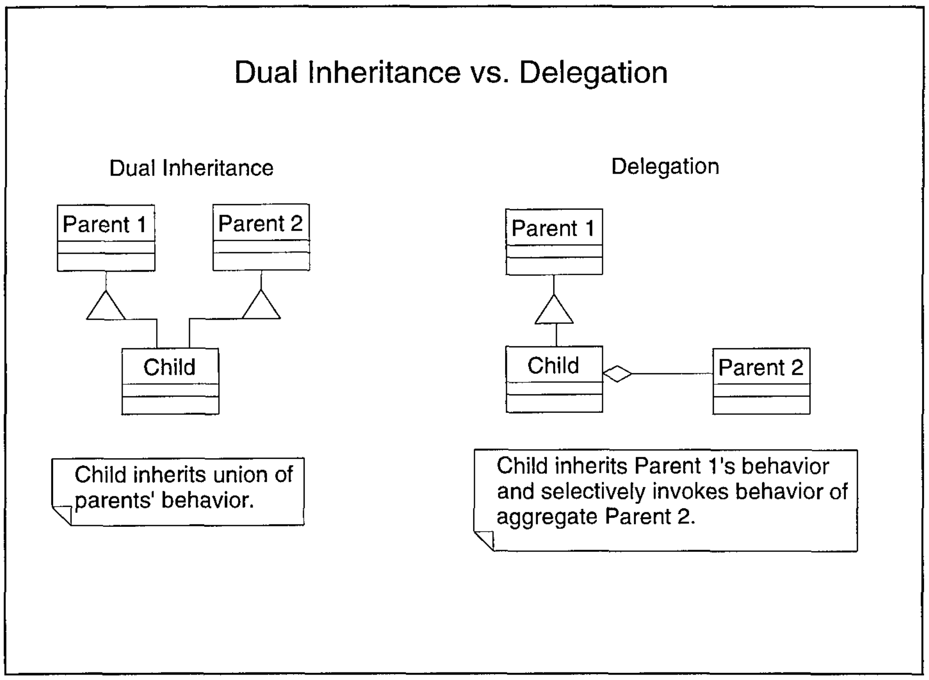

Dual Inheritance vs. Delegation
Dual Inheritance
Parent 1
Parent 2
Child
Child inherits union of parents' behavior.
Delegation
Parent 1
Child
Parent 2
Child inherits Parent 1's behavior and selectively invokes behavior of aggregate Parent 2.

Figure 4-12. Dual Inheritance vs. Delegation

Figure 4-13 depicts the structure of the joint SimWorx-Easy_Sim simulation system. Note how the Joint Simulation delegates Federate behavior to an associated Joint Federate class. Similarly, the Joint Player class delegates HLA Object behavior to an associated HLA_Object.

Simply linking objects does not complete the integration of these application frameworks. Steps must be taken to ensure a meaningful integrated thread of control. Garlan et al, describing architectural mismatch problems note, "One of our most serious problems was due to the assumptions made about what part of the software held the main thread of control" [Gar95, 22].

The Easy_Sim thread of control lies principally in the Simulation class. Following initialization of an Easy_Sim application, the Simulation.Render method loops repeatedly, updating the players, models, and views of the simulation and then rendering the updated scene.

SimWorx has multiple threads of control since it is a concurrent system. Most threads are dedicated to the task of propagating state information from the Objects container to the network or updating state information in the Objects container from the network. The Fed_Task is the only thread concerned with "application level" manipulation of HLA_Objects. That is, all behavior changes of Federate owned HLA_Objects are handled by the Fed_Task.

The joint SimWorx-Easy_Sim simulation system avoids conflicting thread of control problems by eliminating the Fed_Task thread of control in favor of the Easy_Sim.Simulation thread of control. Essentially, locally owned Joint Player objects are updated with each frame of the rendering loop. With each update, the Joint Player updates its corresponding HLA_Object, ensuring their states are in agreement. Periodically, the Joint Federate Ambassador's Scan_Task propagates the updated attribute values to the RTI.

The following paragraphs describe the classes of the Joint SimWorx-Easy_Sim Simulation System:

- **Abstract Joint Simulation**: The Abstract Joint Simulation exists to resolve the circular dependency between the Joint Simulation and Joint Federate Ambassador. The Joint Simulation is dependent upon the Joint Federate Ambassador since the ambassador is contained within the Joint Federate. The Joint Federate Ambassador is dependent upon the Joint Simulation since it must call the Add_Player method whenever the RTI invokes Discover_Object. The mutual dependency solution is the same as was used between the RTI and Federate ambassadors.
- **Joint Simulation**: The Joint Simulation class is a specialization of the Easy_Sim Simulation class. It delegates the SimWorx distributed simulation behavior to its associated Joint Federate. It adds new methods to add and delete players in response to Discover_Object and Remove_Object invocations. It also adds a Add_Owned_Player method to add locally owned joint players to the simulation and a parameterized Player_Factory method to allocate and initialize a new player based upon an object class parameter.
- **Joint Player**: The Joint Player class is an Easy_Sim Player class with an associated HLA_Object which it delegates distributed behavior to. It overrides the Easy_Sim Player.Update method to ensure it's position and orientation agree with the associated HLA_Object. That is, if it is a locally owned player, then it copies its position and orientation to the HLA_Object. If it is not locally owned, then it copies its position and orientation from the HLA_Object. The Joint Player also includes a SimWorx_Setup method to perform initialization related to the associated HLA_Object.
- **Joint Federate**: The Joint Federate is a specialized SimWorx Federate class. It overrides the Federate Initialize method so the Fed_Task is not allocated or initialized. It adds a Configure method to allow the Joint Simulation to pass its identity to the Federate Ambassador (necessary for the ambassador to invoke the Add_Player and Delete_Player methods.

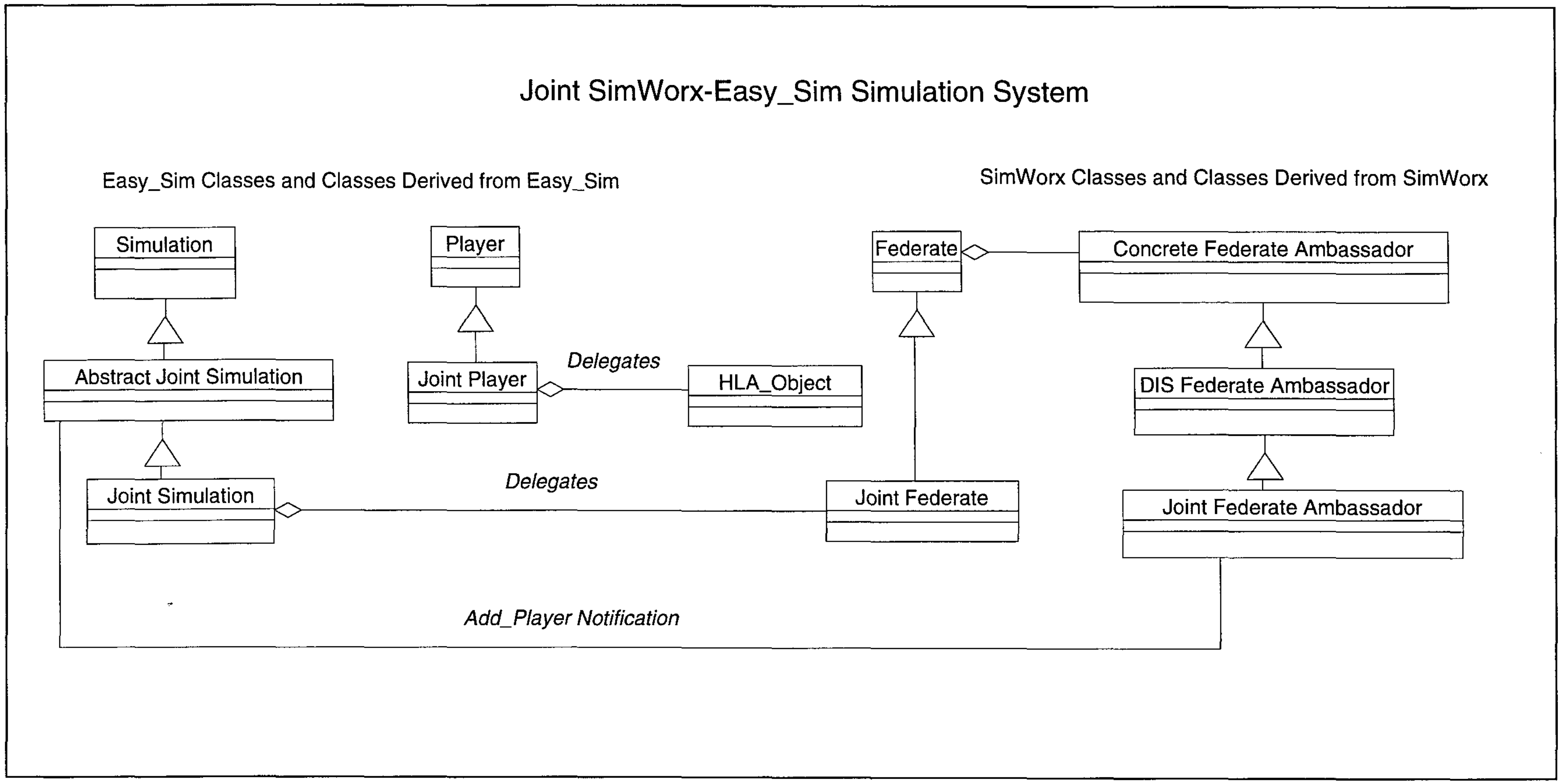

Joint SimWorx-Easy_Sim Simulation System
Easy_Sim Classes and Classes Derived from Easy_Sim
SimWorx Classes and Classes Derived from SimWorx
Simulation
Player
Federate
Concrete Federate Ambassador
Abstract Joint Simulation
Joint Player
Delegates
HLA_Object
DIS Federate Ambassador
Joint Simulation
Delegates
Joint Federate
Joint Federate Ambassador
Add_Player Notification

Figure 4-13. Joint SimWorx-Easy_Sim Simulation System

- **Joint Federate Ambassador**: The Joint Federate Ambassador is a specialized SimWorx Federate Ambassador. It calls the Joint Simulation Add_Player and Delete_Player methods in response to the RTI Discover_Object and Delete_Object calls.

4.3.11 Iteration Three Application

The application created for Iteration Three is the Ada Virtual Cockpit, a virtual simulation which enables a user to "fly" a B-52H Bomber about a virtual environment, controlling the plane via the keyboard. Figure 4-14 shows the class extensions to the Joint SimWorx-Easy_Sim Simulation System necessary to create the Ada Virtual cockpit application.

Description of Ada Virtual Cockpit classes:

- **AVC_Sim**: The Ada Virtual Cockpit simulation class is a specialized Joint Simulation class. It extends its parent's Initialize method to set up a virtual environment with a Joint Player B-52 aircraft.
- **Aircraft**: The Aircraft class is a specialized Joint Player class which adds the "flyable" behavior. Flyability is achieved by changing the position of the aircraft based upon keyboard input from the Easy_Sim Standard Input Modifier class.
- **B-52**: The B-52 class is a specialized SimWorx DIS Entity Object class which contains DIS Entity type information corresponding to a U.S. Air Force B-52H aircraft. The entity type information comes from the DIS Enumeration Document, which provides unique data values for thousands of possible DIS Entity types.

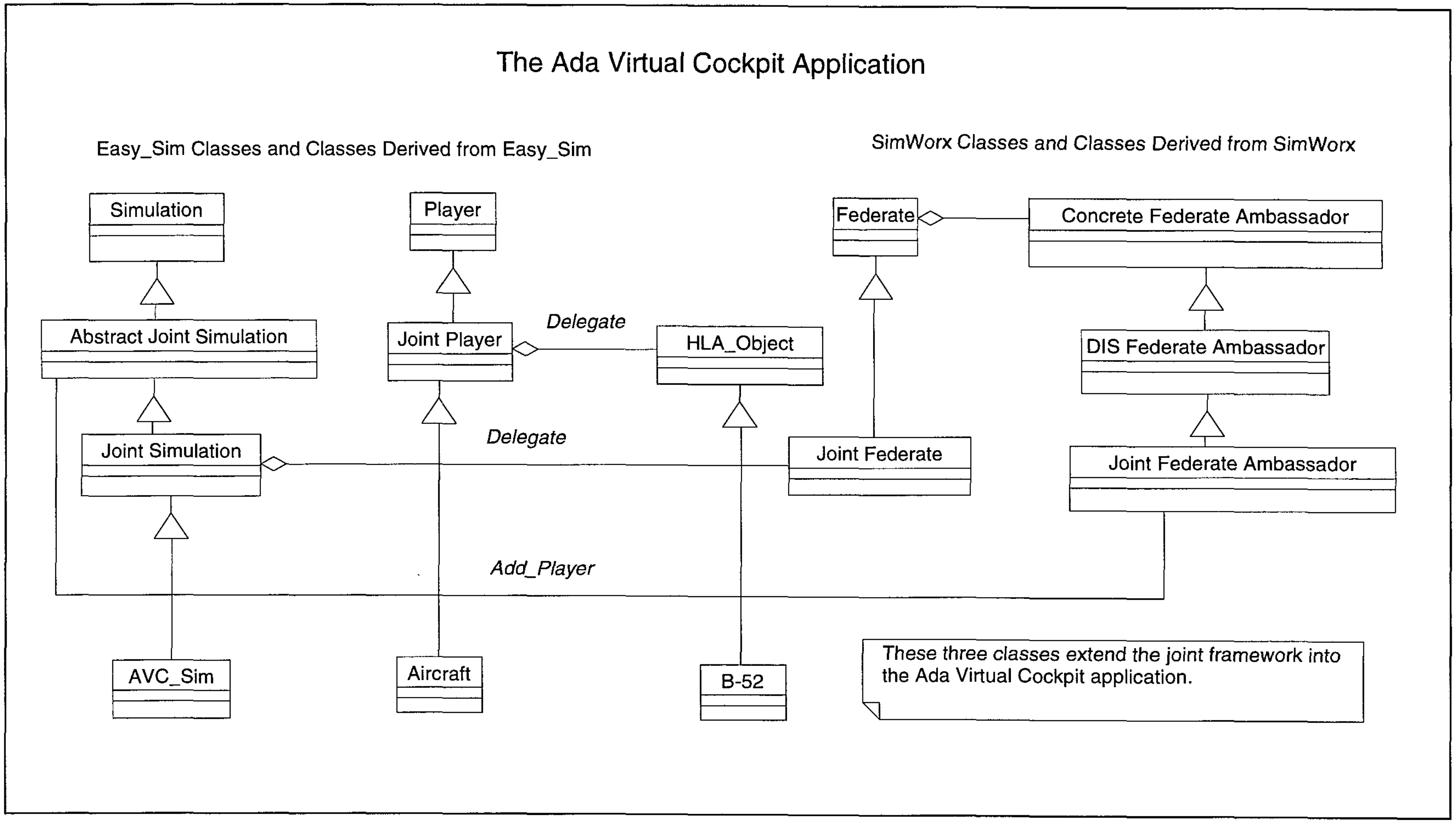

The Ada Virtual Cockpit Application
Easy_Sim Classes and Classes Derived from Easy_Sim
SimWorx Classes and Classes Derived from SimWorx
Simulation
Player
Federate
Concrete Federate Ambassador
Abstract Joint Simulation
Joint Player
Delegate
HLA_Object
DIS Federate Ambassador
Joint Simulation
Delegate
Joint Federate
Joint Federate Ambassador
Add_Player
AVC_Sim
Aircraft
B-52
These three classes extend the joint framework into the Ada Virtual Cockpit application.

Figure 4-14. The Ada Virtual Cockpit Application

4.3.12 SimWorx Design Patterns

SimWorx employs several design patterns [Gam95]:

- **Factory Method** [Gam95, 107]: The Concrete_Federate_Ambassador class employs a parameterized factory method to produce instances of HLA_Objects with subclass IDs corresponding to the method's Object_Class_Handle parameter. The Federate class provides factory methods so federate subclasses can allocate the specific class of RTI Ambassador and Federate Ambassador the Federate child class will use at run-time. (The factory methods are called from the Federate.Initialize method via a form of the Template Method Pattern--see below).
- **Iterator** [Gam95, 257]: The Protected_Container class supports multiple concurrent transactions via external iterators.
- **Command** [Gam95, 233]: The HLA_Interaction class is a form of the command pattern. HLA Interactions encapsulate the details of an interaction between two (or more) HLA_Objects. All HLA_Objects inherit a Handle_Interaction method which enables them to respond to interactions initiated by others.
- **Template Method** [Gam95, 325]: The way the Fed_Task calls the Client_Initialize, Client_Loop_Body, and Client_Finalize methods is a form of the Template Method. Clients override the "Client_" methods to customize the behavior of the Fed_Task.

This chapter discussed the development of the SimWorx Application Framework for Distributed Simulation Supporting HLA and DIS. The following final chapter draws conclusions about the thesis effort and presents recommendations for future research efforts.

5. Conclusions and Recommendations

This chapter concludes this thesis by presenting some conclusions and recommendations. The first section covers product conclusions. The second section discusses results from integrating the Easy_Sim and SimWorx application frameworks. The third section presents recommendations for future research. Finally, the last section concludes this thesis document with a few final remarks.

5.1 Product Conclusions

The SimWorx Application Framework for Distributed Simulation achieved two of its three product goals.

Although it did not start out as the main goal, the desire for SimWorx to support the DoD High Level Architecture for Modeling and Simulation eventually became the overriding product goal. The HLA Interface specification drove the overall SimWorx architecture. Although SimWorx did not implement the full HLA standard as specified by the HLA Interface Specification [DMSO96a], the functionality that was implemented does conform to the standard.

AFIT students and faculty can define HLA-conforming object models for simulations and implement the models as SimWorx derived classes, thus creating HLA-compliant simulations.

The second SimWorx product goal was support for DIS. SimWorx achieves DIS compliance by supporting the most common DIS PDUs and the DIS dead reckoning rules. DIS support was verified by running the SimWorx DIS Printer application on a DIS network while a

ModSAF[9] application simulated the flight of an aircraft. The DIS Printer successfully showed the current state of the simulated aircraft.

Although SimWorx does not support the entire DIS 2.0.4 standard, it provides the functionality necessary to experiment with distributed simulations and interoperate with AFIT Graphics Lab legacy simulation applications.

The third SimWorx product goal was improvement of client developer productivity. Since productivity measurement was outside the scope of this thesis, it can not be conclusively shown that this product goal was met.

SimWorx provides a ready-made foundation for HLA-compliant experiments in distributed simulation. By using SimWorx, AFIT students and faculty can perform valuable research and also increase their exposure to the Ada programming language.

5.2 Integrating Application Frameworks

It was not particularly difficult to integrate the SimWorx and Easy_Sim application frameworks. Three factors affect the ease of this task.

First, SimWorx and Easy_Sim support very similar problem domains: entities (objects) interacting with each other. Integration would have been more difficult if the problem domains were dissimilar since class linkages between the frameworks would have been more difficult to resolve.

Second, the ease of integration reflects the flexibility of the frameworks' solution domains. Application frameworks are meant to be extended! Extension was a key principle in joining Easy_Sim and SimWorx.

[9] ModSAF stands for Modular Semi-Automated Forces. It is a constructive simulation which models both ground and air forces. It was created by Loral Advanced Distributed Simulation group for U.S. Army Simulation, Training, and Instrumentation Command.

Finally, it's well known that object-oriented systems are easier to maintain and modify than structured or other systems. Since SimWorx and Easy_Sim are both object-oriented application frameworks, both profited from the advantages of the OO approach.

Chapter 1 stated few results were available in the literature concerning application framework integration. This researcher concludes that little has been written because there's nothing special or exciting to write about. Integrating application frameworks from similar domains is relatively easy.

The largest obstacle encountered during framework integration was determining the thread of control for Easy_Sim, i.e., determining what events had to happen, in what order, to get the framework to behave as expected. The reason for this difficulty can be attributed to lack of user documentation for Easy_Sim. There were no examples or tutorials available for Easy_Sim other than three scantily documented sample programs.

Researchers have documented that frameworks are hard to learn. Effective documentation is critical. The SimWorx analysis and design documents and user's guide should ensure users can learn SimWorx quickly.

SimWorx's success in distributed simulation, coupled with Easy_Sim's proven performance [Kay94, 124] removes any barriers to Ada's usage in the AFIT Graphics Lab. The joint Easy_Sim-SimWorx simulation system provides an effective Ada based simulation environment which should provide increased Ada exposure to AFIT students and faculty.

5.3 Recommendations for Further Study

The following recommendations are divided into two categories: recommendations for future study employing the SimWorx application framework, and recommendations for SimWorx product improvements.

5.3.1 Suggested Uses for the SimWorx Application Framework

Productivity Improvement Measurement: As mentioned above, measurement of productivity improvements provided by the SimWorx application framework were out of the scope of this thesis. Future work should be done to document productivity improvement in distributed simulation development from using the SimWorx framework.

Experimental RTIs: Development of the DIS Surrogate RTI showed that it is not conceptually difficult to create alternate RTIs (as opposed to the standard RTI, RTI F.0). Alternate RTIs could be built for a variety of reasons. A "light weight" RTI could be built which supplied only the minimal services necessary for its federates' domain. For example a "tiny DIS" RTI could be built which did not employ dead reckoning--it would be appropriate for aircraft combat modeling since aircraft generally emit state changes at a frequency greater than the dead reckoning frequency. A "high performance" RTI could be built to take advantage of recent improvements in computer networking such as FDDI and ATM (RTI F.0 is based upon TCP/IP).

SimWorx as a Tool for Knowledge-Based Software Engineering (KBSE): Because SimWorx provides services for object-oriented simulation, it could be the run-time component in a KBSE simulation system. Developers could specify the federate's object model using some formal language. A KBSE software system could transform the object model into appropriate SimWorx child classes. It could then compile and execute the resulting simulation system.

SimWorx as a Testbed for Experiments in Framework Documentation and Usage: Although research has shown application frameworks are difficult to learn, little has been done to study effective ways of teaching/documenting/learning application frameworks. SimWorx could be used as an effective tool for such studies.

5.3.2 Recommended SimWorx Product Improvements

Full HLA Compatibility: SimWorx implements the minimal subset of HLA functions necessary to be DIS capable (compare the methods of the Abstract Federate and RTI Ambassadors with the HLA Interface Specification [DMSO96a]). Future versions of SimWorx must implement the entire HLA Interface Specification (at least from the perspective of the Federate Ambassador) in order to be truly HLA compliant.

Support for Aggregate Objects: Currently SimWorx does not support aggregate objects. Aggregate objects are those which contain other objects. For instance, a cruise missile HLA object might contain a guidance system HLA object and a propulsion system HLA Object. Aggregate Objects could be implemented in SimWorx by applying the Composite Structural Design Pattern [Gam95, 163] (see Figure 5-1 below) The Composite Design Pattern illustrates a means of defining objects (the *Composite* in Figure 5-1) which are composed of other objects, all of which are derived from some common parent object (the *Component* in Figure 5-1).. The Protected_Object container would have to be redesigned as a tree of Composite objects, rather than the simple array of its current implementation.

Support for Coordinate System Transformations: Currently, the DIS Surrogate RTI manipulates all coordinates in the standard DIS coordinate system. Application developers must supply their own coordinate transformations if they must use a different coordinate system. Snyder demonstrated virtual simulation rendering speedups by performing dead reckoning using the coordinate system of the graphics application rather than the DIS coordinate system [Sny93, 89]. Coordinates were transformed from DIS to graphics only when new Entity State PDUs were received. Future versions of SimWorx could provide customizable, built-in coordinate transformation services, so client developers could specify where the transformations would occur in the pipeline of coordinate information flow.

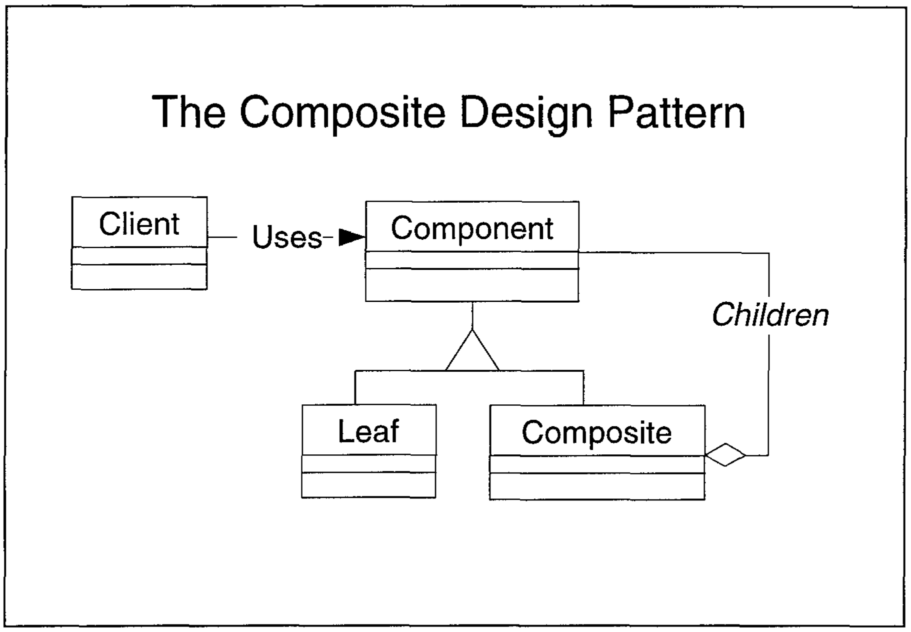


Figure 5-1. The Composite Design Pattern [Gam95, 163]

5.4 Final Remarks

It is likely that computer modeling and simulation will have a tremendous impact on U.S. Air Force mission accomplishment. The High Level Architecture for Modeling and Simulation will provide a reusable and interoperable foundation for DoD simulation efforts into the next century. It is this author's hope that the SimWorx application framework will support AFIT educational and research goals for simulation into the next century.

Bibliography

[Ada95] Adair, Deborah, "Building Object-Oriented Frameworks, Part 2," AIXpert; May 1995.

[Bec94] Beck and Johnson, "Patterns Generate Architectures," ECOOP '94 Conference Proceedings, ACM Press, New York; 1994.

[Boo94] Booch, Grady, Object-Oriented Analysis and Design with Applications, Benjamin-Cummings, Inc., Redwood City, California; 1994.

[Bro87] Brooks, Frederick, "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer; April 1987.

[Bur95] Burns and Wellings, Concurrency in Ada, Cambridge University Press, Cambridge, United Kingdom; 1995.

[Cam93] Campbell, Islam, Raila, and Madany, "Designing and Implementing Choices: an Object-Oriented System in C++," Communications of the ACM; September 1993.

[Cer93] Cernosek, Gary, "ROMAN-9X: A Technique for Representing Object Models in Ada 9X Notation," Tri-Ada '93 Conference Proceedings, ACM Press, New York; 1993.

[Coh96] Cohen, Norman, Ada as a Second Language, McGraw-Hill, Inc., New York; 1996.

[Cop95] Coplien and Schmidt, editors, Pattern Languages of Program Design, Addison-Wesley, Inc., Reading, Mass.; 1995.

[Dia94] Diaz, Milton, "The Photo Realistic AFIT Virtual Cockpit," MS Thesis AFIT/GCS/ENG/94D-02, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[DIS94] The DIS Vision, A Map to the Future of Distributed Simulation, version 1, DIS Steering Committee, University of Central Florida/Institute for Modeling and Simulation, Orlando Florida; 1994.

[DIS94b] Standard for Distributed Interactive Simulation--Application Protocols, University of Central Florida/Institute for Modeling and Simulation, Orlando Florida; 1994.

[DMSO95] Primary Definition of the DoD High Level Architecture for Modeling and Simulation, Briefing Slides, U.S. Department of Defense, Defense Modeling and Simulation Office; May 31, 1995 (Available on-line: www.dmso.mil/dmso/wrkgrps/amg/amgbriefs).

[DMSO96a] Department of Defense High Level Architecture Interface Specification Version 1.0, Simulation Standard, U.S. Department of Defense, Defense Modeling and Simulation Office; August 15, 1996 (Available on-line: www.dmso.mil /dmso/projects/hla).

[DMSO96b] Department of Defense High Level Architecture Object Model Template Version 1.0, Simulation Standard, U.S. Department of Defense, Defense

Modeling and Simulation Office; August 15, 1996 (Available on-line: www.dmso.mil /dmso/projects/hla).

[DoD95] Department of Defense Modeling and Simulation (M&S) Master Plan, Under Secretary of Defense for Acquisition and Technology, DoD 5000.59-P, Defense Technical Information Center, Cameron Station, Alexandria, VA, 22304-6145; 1995.

[Eri93] Erichsen, Matthew, "Weapon System Sensor Integration for a DIS v2.0.3 Compatible Virtual Cockpit," MS Thesis AFIT/GCS/ENG/93D-07, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1993.

[For94] Fortner, Jonathan, "Distributed Interactive Simulation Virtual Cassette Recorder: A Datalogger with Variable Speed Replay," MS Thesis AFIT/GCS/ENG/94D-10, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Gam95] Gamma, Helm, Johnson, and Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Inc., Reading, Mass.; 1995.

[Gar93] Garlan and Shaw, "An Introduction to Software Architecture," Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Co., Singapore; 1993.

[Gar95] Garlan, Allen, and Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," IEEE Software; November 1995.

[Gard93] Gardner, Michael, "A Distributed Interactive Simulation Based Remote Debriefing Tool for Red Flag Missions," MS Thesis AFIT/GCS/ENG/93D-09, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1993.

[Ger93] Gerhard, William, "Weapon System Integration for the AFIT Virtual Cockpit," MS Thesis AFIT/GCS/ENG/93D-10, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1993.

[Gom93] Gomaa, Hassan, Software Design Methods for Concurrent and Real-Time Systems, Addison Wesley, Inc., Reading, Massachusetts; 1993.

[GSAM94] Guidelines for Successful Acquisition and Management of Software Intensive Systems, Department of the Air Force, Software Technology Support Center, Hill AFB, Utah; September 1994.

[Han95] Hannan, Shawn, "ObjectSim 3.0: A Software Architecture for the Development of Portable Visual Simulation Applications," MS Thesis AFIT/GCS/ENG/95D-05, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1995.

[Jac92] Jacobson, Christerson, Jonsson, and Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Inc., Reading, Mass.; 1992.

[Joh88] Johnson and Foote, "Designing Reusable Classes," Journal of Object Oriented Programming; June 1988.

[Joh92] Johnson, Ralph, "Documenting Frameworks Using Patterns," OOPSLA '92 Conference Proceedings, ACM Press, New York; 1992.

[Kay94] Kayloe, Jordan, "Easy_Sim: A Visual Simulation System Software Architecture with an Ada 9X Application Framework," MS Thesis AFIT/GCS/ENG/94D-11, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Kes94] Kestermann, Jim, "Immersing the User in a Virtual Environment: The AFIT Information Pod Design and Implementation" MS Thesis AFIT/GCS/ENG/94D-13, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Kof93] Kofler, Thomas, "Robust Iterators in ET++," Structured Programming; March 1993.

[Kru95] Kruchten, Phillipe, "The 4+1 View Model of Architecture," IEEE Software; November 1995.

[Kun93] Kuntz, Andrea, "A Virtual Environment for Satellite Modeling and Orbital Analysis in a Distributed Interactive Simulation," MS Thesis AFIT/GCS/ENG/93D-14, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1993.

[OTA95] Distributed Interactive Simulation of Combat, U.S. Congress, Office of Technology Assessment, OTA-BP-ISS-151, U.S. Government Printing Office, Washington DC.; September 1995.

[Roh94] Rohrer, J., "Design and Implementation of Tools to Increase User Control and Knowledge Elicitation in a Virtual Battlespace," MS Thesis AFIT/GCS/ENG/94D-20, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Rum91] Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Inc., New York, New York; 1991.

[Rum94] Rumbaugh, James, "Getting Started: Using Use Cases to Capture Requirements," Journal of Object-Oriented Programming; September 1994.

[Sch95] Schmidt, Douglas, "Using Design Patterns to Develop Reusable Object-Oriented Communication Software," Communications of the ACM; October 1995.

[Sho95] Shomper, Keith, Interview, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio; October 1995.

[She92] Sheasby, Steven, "Management of Simnet and DIS Entities in Synthetic Environments," MS Thesis AFIT/GCS/ENG/92D-16, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1992.

[Sny93] Snyder, Mark, "ObjectSim: A Reusable Object-Oriented DIS Visual Simulation," MS Thesis AFIT/GCS/ENG/93D-20, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1993.

[Sol93] Soltz, Brian, "Graphical Tools for Situational Awareness Assistance for Large Synthetic Battle Fields," MS Thesis AFIT/GCS/ENG/93D-21, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1993.

[SPC95] Ada 95 Quality and Style Guide, Software Productivity Consortium, 1995. (Available on-line: http://sw-eng.falls-church.va.us/adaic/docs/style-guide/95style/html/cover.html).

[Tal94] “Building Object-Oriented Frameworks,” Taligent, Inc., Cupertino, California; 1994.

[Van94] Vanderburgh, John, “Space Modeler: An Expanded, Distributed, Virtual Environment for Space Visualization,” MS Thesis AFIT/GCS/ENG/94D-23, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Wel95] Weller, David, “The Ada 95 Booch Components,” Tri-Ada ‘95 Tutorial Proceedings, ACM Press, New York; 1994.

[Wil93] Wilson, Kirk, “Synthetic Battle Bridge: Information Visualization and User Interface Design Applications in a Large Virtual Reality Environment,” MS Thesis AFIT/GCS/ENG/93D-26, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1993.

Vita

Captain Earl Conrad Pilloud [redacted] [redacted]. He spent his formative years in central Minnesota where he graduated from Little Falls Community High School in 1982. Following a year of studying Mechanical Engineering at South Dakota State University, he joined the United States Air Force as an enlisted computer programmer.

His first assignment was at Offutt AFB, Nebraska, as a maintenance programmer on the Stored Programming Element of the SAC Automated Command Control System, a first-generation computer programmed in a proprietary assembly language with punched cards. In 1985, he was selected Headquarters Strategic Information Systems Division Airman of the Year.

In 1987 he began attending Texas A&M University via the Airmens' Education and Commissioning Program. After two years, he graduated Magna Cum Laude with a Bachelor of Science Degree in Computer Science. Following college, he attended Air Force Officer Training School, where he was commissioned into the Air Force as a Distinguished Graduate on 22 Dec 1989.

His first three assignments as an officer were all associated with the Global Positioning System, first as a software maintenance analyst and section team leader, second as chief of software requirements for the 2nd Space Operations Squadron, and finally as Chief of Global Positioning Control Segment Requirements, Air Force Space Command.

In May, 1995, he entered the School of Engineering, Air Force Institute of Technology. He graduated on 17 Dec 1996, earning a Master of Science in Computer Systems, with specialization in Software Engineering.

Following AFIT, he was assigned to the Software Management Division of the Air Force Communications Agency, Scott AFB, Illinois.

[redacted] [redacted]

This Page Intentionally Left Blank

Appendix A. Object Modeling Technique Notation

This appendix provides a key to reading Object Modeling Technique (OMT) Diagrams. It is by no means complete. Greater detail can be found in [Rum91].

The diagrams below were drawn with the demonstration version of Rational Rose C++, available from http://www.rational.com.

Class Notation. In OMT, a class is represented by a box. Class boxes typically have three sections. The top-most box is the class name.

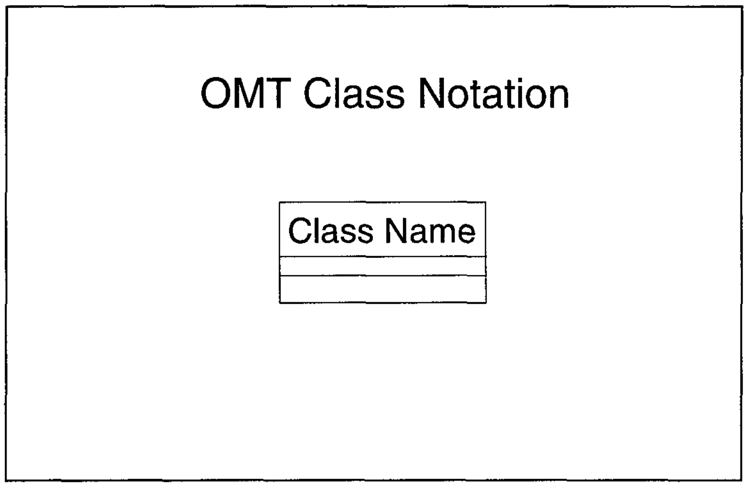


Figure 1. OMT Class Notation

Association. In OMT, an association is represented by a line between two classes (or three for the rare ternary associations). The line usually has a label which describes the association.

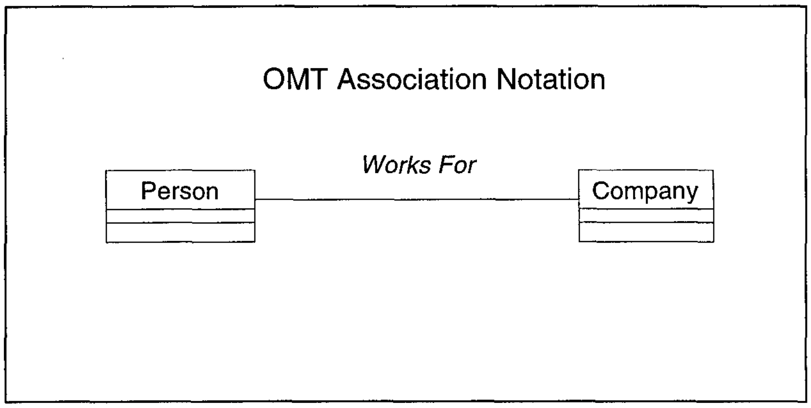


Figure 2. OMT Association Notation

Aggregation. In OMT, aggregation is the "has a" association between two classes. OMT represents aggregation by placing a diamond next to the class which "has" the other class as a component.

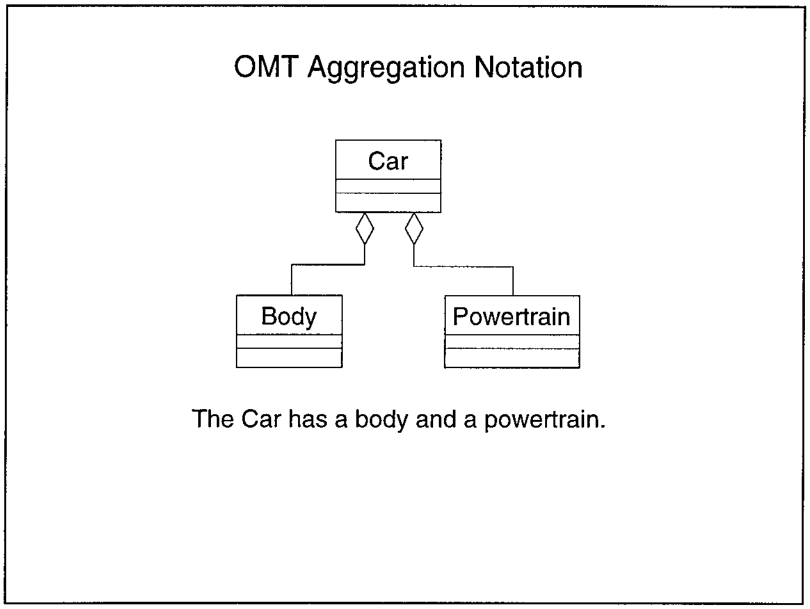


Figure 3. OMT Aggregation Notation

Generalization. In OMT, generalization is the "is a" inheritance association between two classes. OMT represents generalization with a triangle; the top of the triangle points to the parent (more general) class in the association.

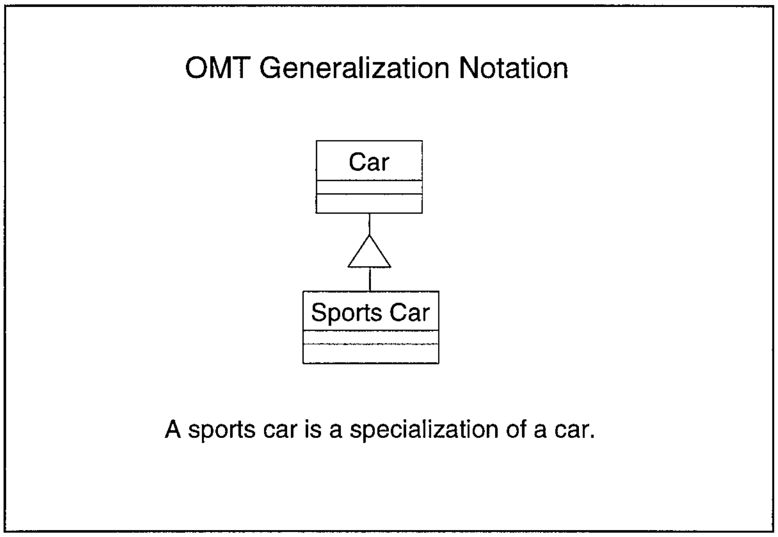

OMT Generalization Notation
Car
Sports Car
A sports car is a specialization of a car.

Figure 4. OMT Generalization Notation

Quantification Decorators. OMT notation uses empty circles, spots, and numbers to quantify the associations between classes.

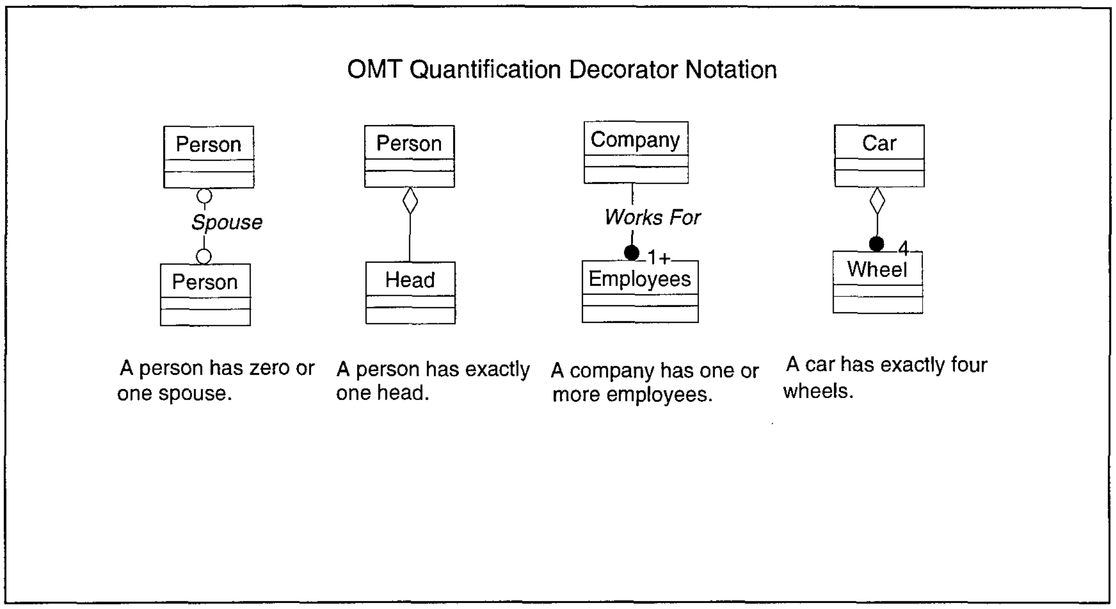

OMT Quantification Decorator Notation
Person
Spouse
Person
Person
Head
Company
Works For
1+
Employees
Car
4
Wheel
A person has zero or one spouse.
A person has exactly one head.
A company has one or more employees.
A car has exactly four wheels.

Figure 5. OMT Quantification Decorator Notation

Generic Instantiation. In Rational's version of OMT notation, generic classes are represented with a specialized class box. The box has an extra dashed line section which shows the generic instantiation parameters. Instantiation of a generic class into an instantiated version of that class is shown via a dashed arrow from the instantiated class to the generic class.

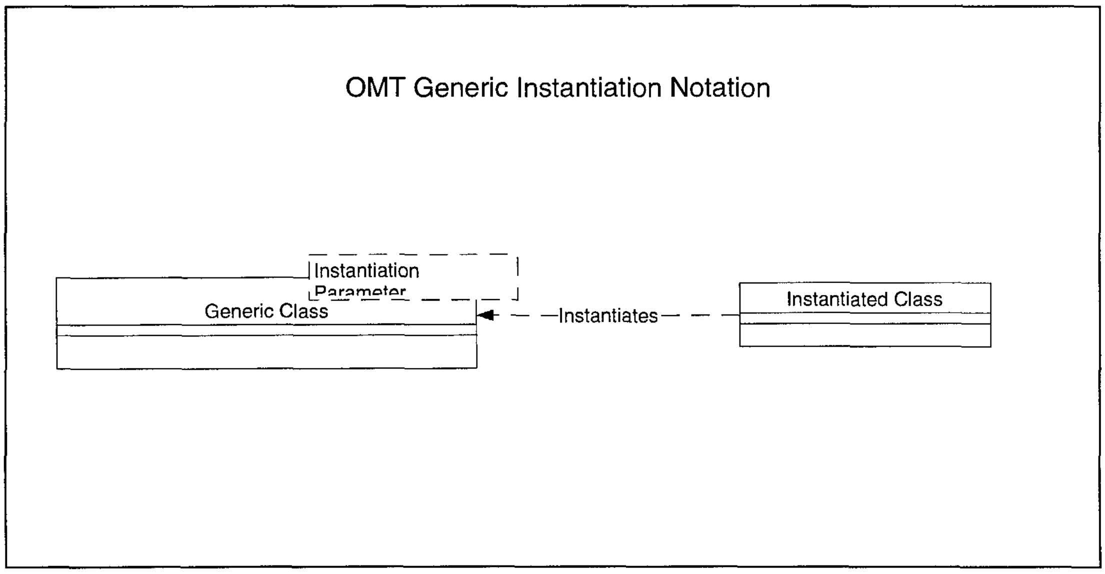


Figure 6. OMT Generic Instantiation Notation

Message Trace Diagrams. Message Trace Diagrams show a time ordered sequence of transactions between objects of a system. The transactions could be abstract messages between the objects or concrete subprogram invocations. The transactions are numbered in time order.

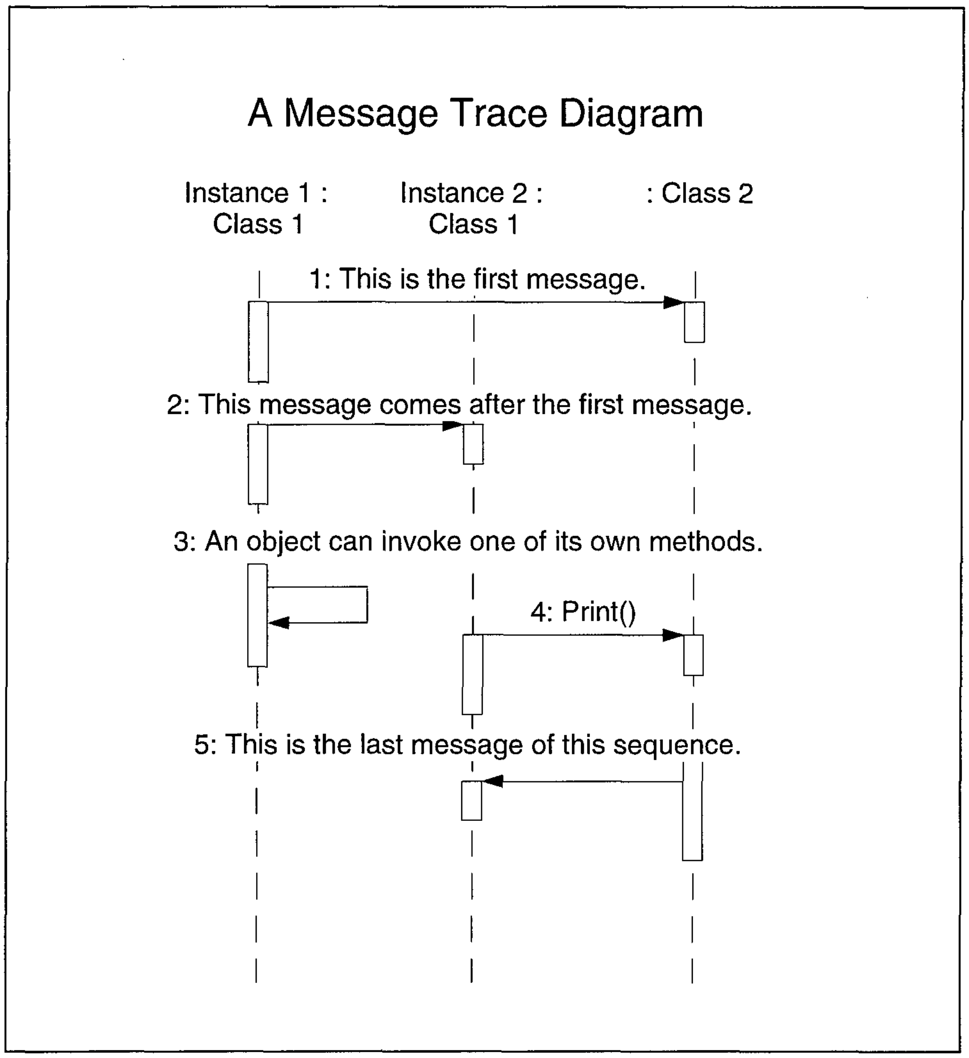

A Message Trace Diagram
Instance 1 : Class 1
Instance 2 : Class 1
: Class 2
1: This is the first message.
2: This message comes after the first message.
3: An object can invoke one of its own methods.
4: Print()
5: This is the last message of this sequence.

Figure 7. A Message Trace Diagram

Appendix B: SimWorx Iteration One Analysis Model

SimWorx: Analysis Model

Iteration One Analysis Model

Software Architectures in Ada 95
AFIT Software Engineering Group
Air Force Institute of Technology
Wright-Patterson AFB, OH 45434

This Page Intentionally Left Blank

Table Of Contents

Thesis Appendix Note: Since this stand-alone analysis document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute B-5 for 5, B-11 for 11, etc.

List of Figures

Thesis Appendix Note: Since this stand-alone analysis document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute B-5 for 5, B-11 for 11, etc.

Introduction

This document reflects the first iteration of software analysis of the SimWorx application framework for distributed simulation.

The SimWorx framework enables application programmers (a.k.a. client programmers) to quickly and easily generate distributed simulation applications conforming to the DoD High Level Architecture (HLA) for Modeling and Simulation. In addition, the SimWorx framework provides a Surrogate HLA Run-Time Infrastructure (RTI) which performs Distributed Interactive Simulation (DIS) version 2.0.4 network input and output.

Initial versions of the SimWorx framework do not support either the full HLA nor full DIS standards. Initial versions supply the minimal functionality necessary for simple objects to interact for typical DIS interactions (entity movement and collision, weapons fire and detonation).

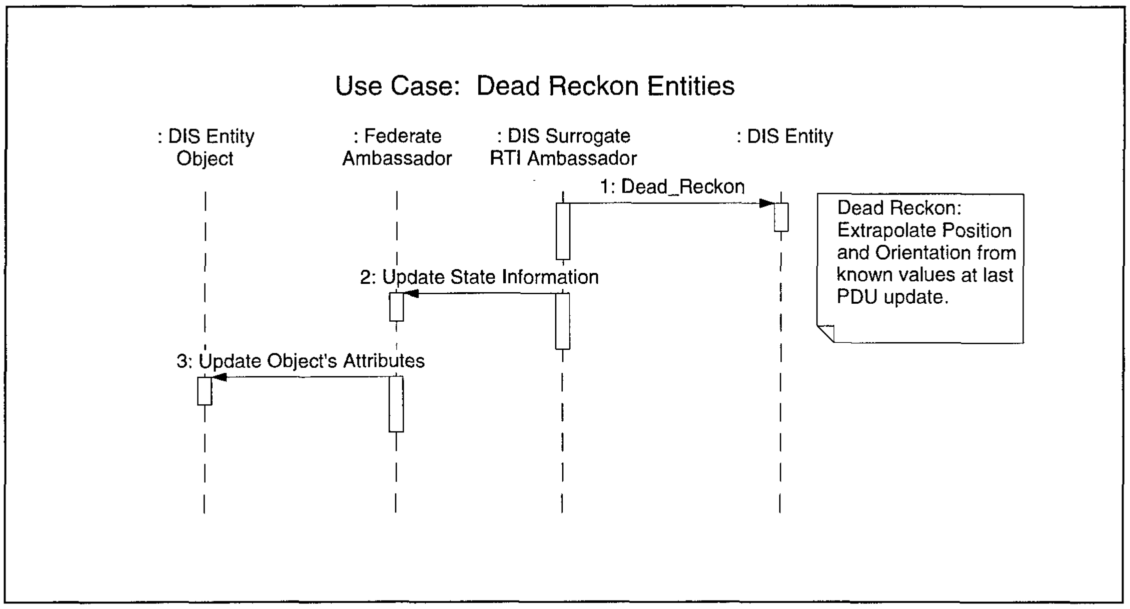
This first iteration of the SimWorx framework concentrates on receiving input from the DIS network. The application framework is divided into two main categories: Federate components with the behavior of an HLA Federate, and RTI components with the behavior of the HLA RTI.

DIS Protocol Data Units (PDUs) are received from the network by the Surrogate RTI Ambassador[1]. DIS Entities are modeled and their states are dead-reckoned between network updates. DIS Entity information is forwarded by the RTI Ambassador to the Federate Ambassador via the standard HLA ambassador interface, so the federate components may be reused with various sorts of RTIs--see the Federate Ambassador class for details.

[1] The term *Ambassador* is an HLA term for a kind of interface software component. Just as a diplomatic ambassador represents a foreign country, an *Ambassador* presents a local application programming interface for a system component which may actually be physically located somewhere else (another computer, another state, etc.). Thus, the client can make software procedure calls to the Ambassador, rather than relying on some message passing interface, since the Ambassador "abstracts-out" the details of the actual interaction.

Use Cases

The following Use Cases describe the first iteration functionality of the SimWorx Application Framework. The Use Cases are described with Message Trace Diagrams. Each numbered item is a message from a sending class to a receiving class. The messages are ordered by time: later numbered messages are later in time than earlier numbered messages.


Use Case: Dead Reckon Entities
: DIS Entity Object
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: DIS Entity
1: Dead_Reckon
2: Update State Information
3: Update Object's Attributes

Dead Reckon: Extrapolate Position and Orientation from known values at last PDU update.

Figure 1. Use Case: Dead Reckon Entities

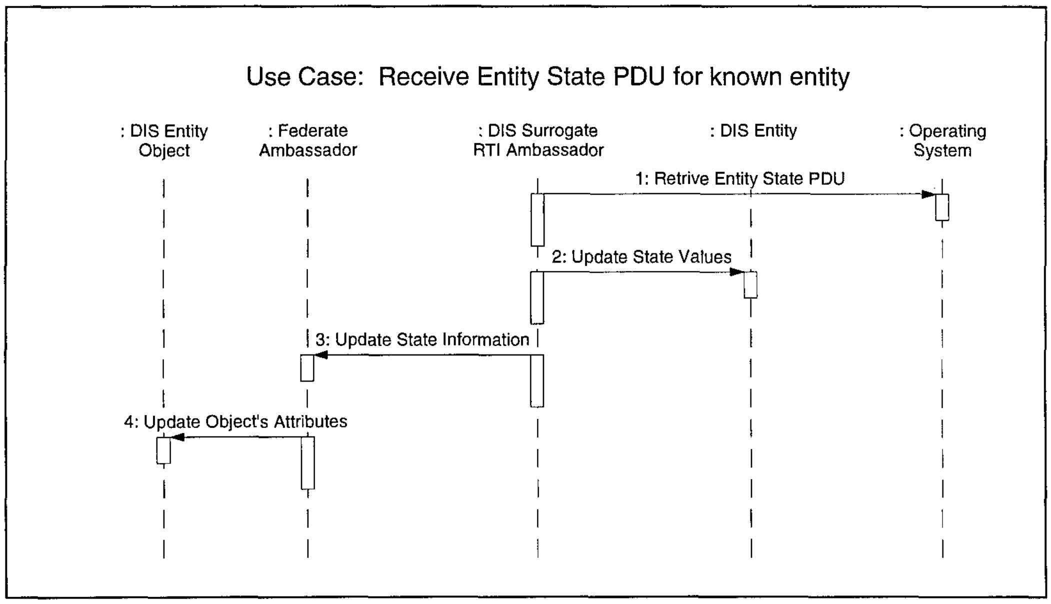

Use Case: Receive Entity State PDU for known entity
: DIS Entity Object
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: DIS Entity
: Operating System
1: Retrive Entity State PDU
2: Update State Values
3: Update State Information
4: Update Object's Attributes

Figure 2. Use Case: Receive Entity State PDU for Known Entity

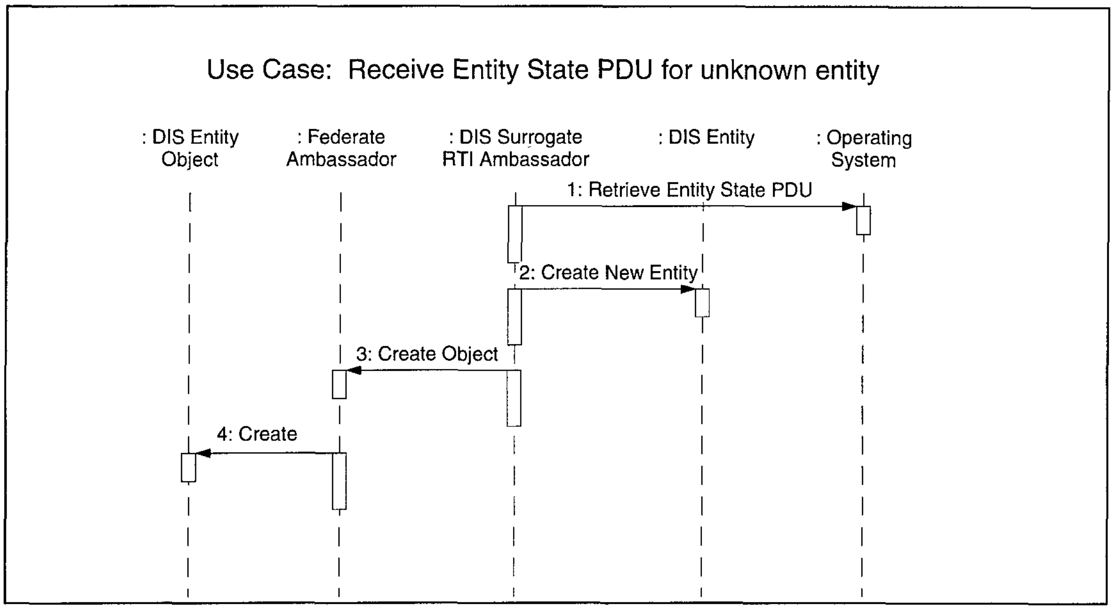


Figure 3. Receive Entity State PDU for Unknown Entity

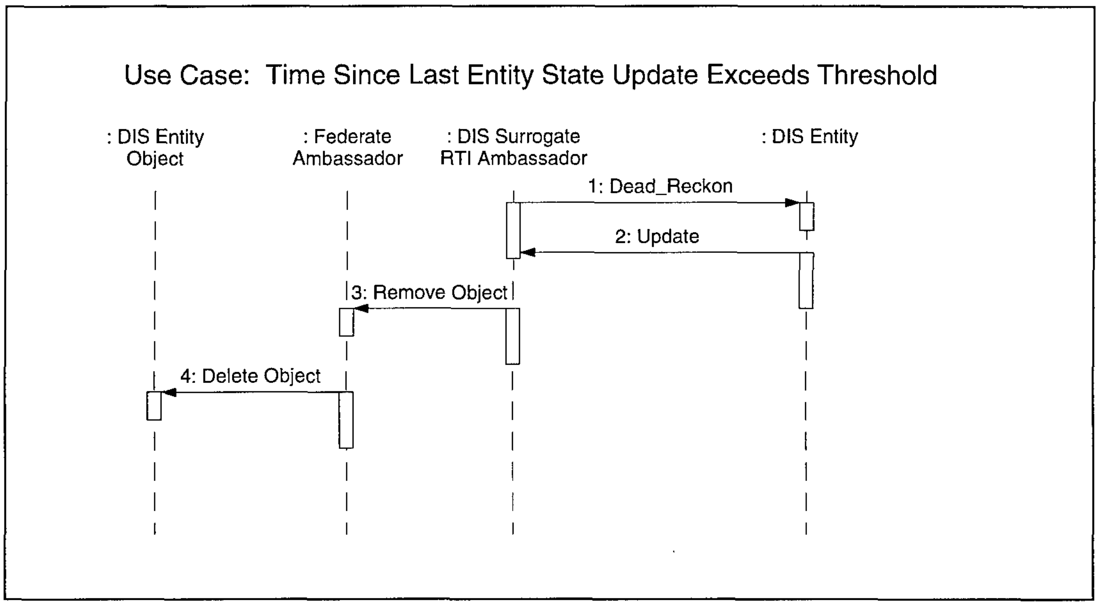


Figure 4. Use Case: Time Since Last Entity State Update Exceeds Threshold

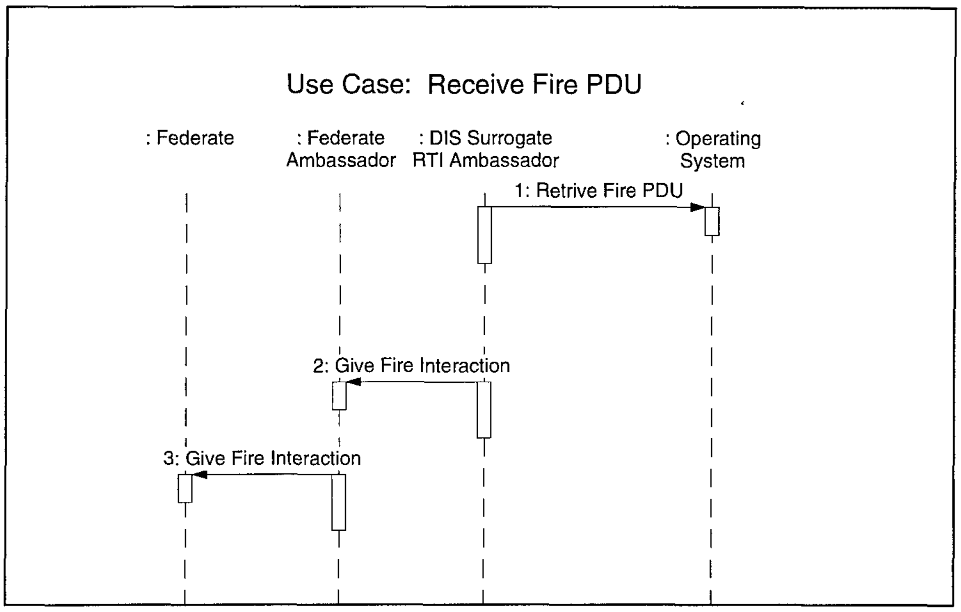


Figure 5. Use Case: Receive Fire PDU

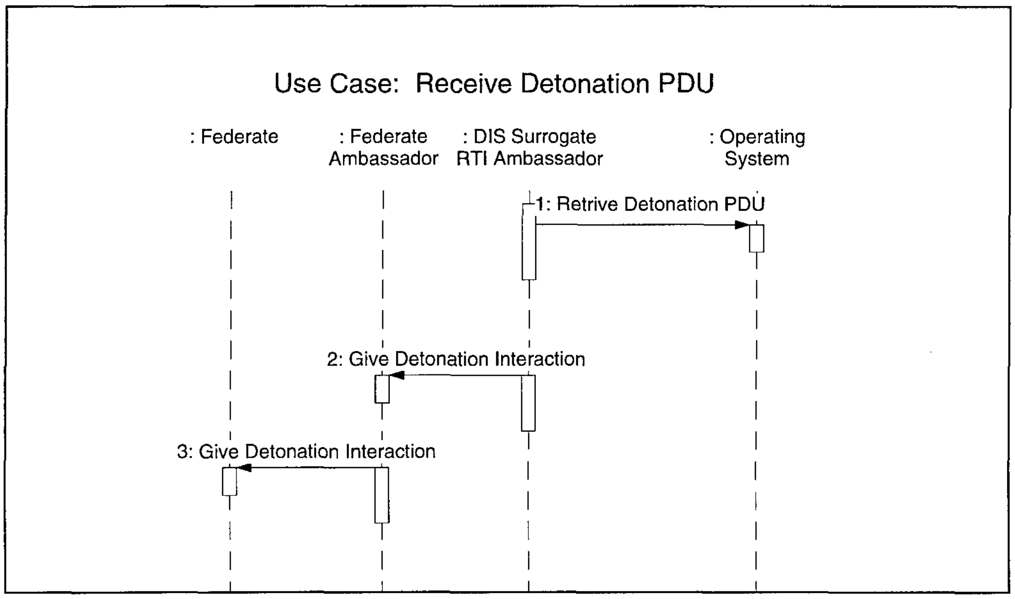

Use Case: Receive Detonation PDU
: Federate
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: Operating System
1: Retrive Detonation PDU
2: Give Detonation Interaction
3: Give Detonation Interaction

Figure 6. Use Case: Receive Detonation PDU

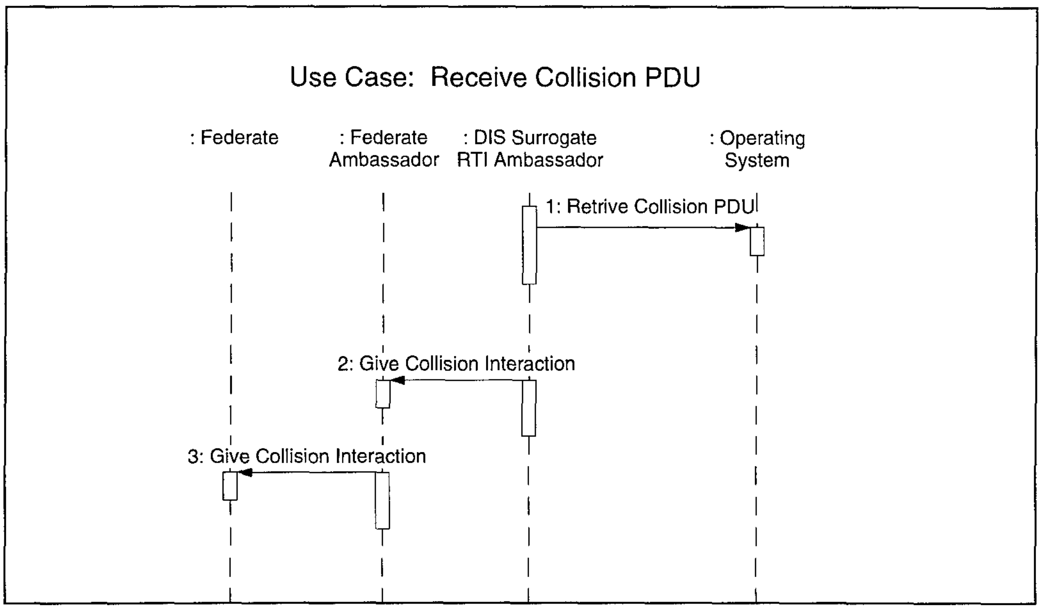

Use Case: Receive Collision PDU
: Federate
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: Operating System
1: Retrive Collision PDU
2: Give Collision Interaction
3: Give Collision Interaction

Figure 7. Use Case: Receive Collision PDU

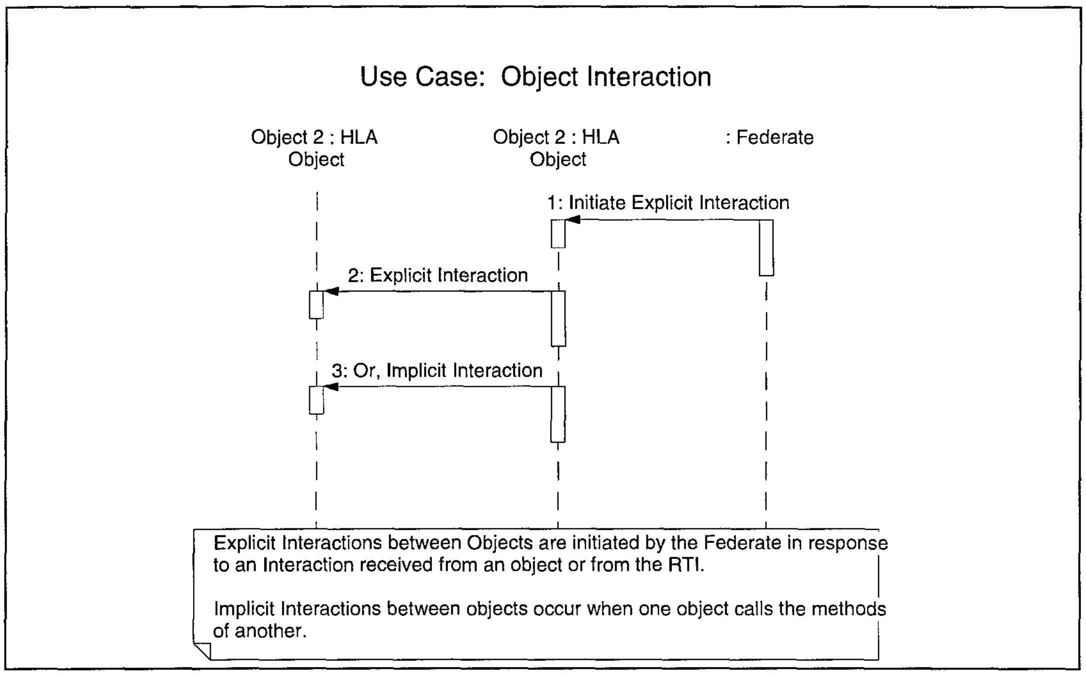
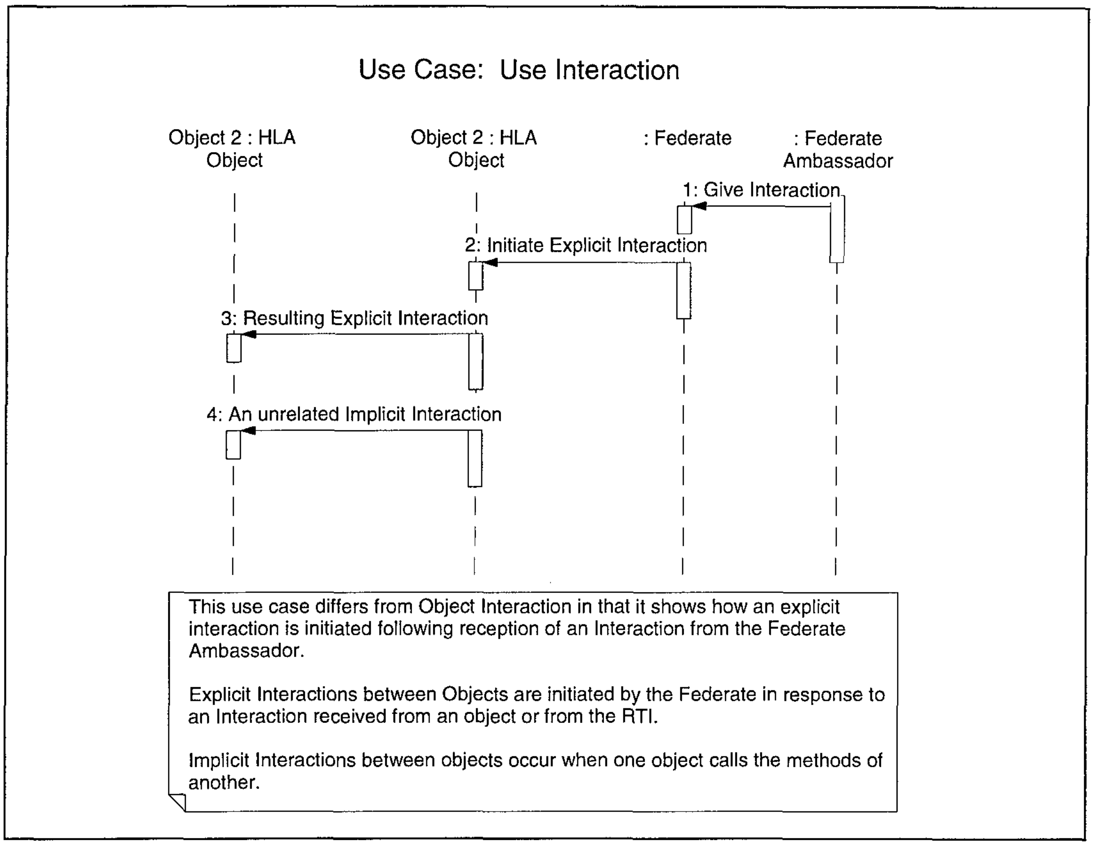

Use Case: Object Interaction
Object 2 : HLA Object
Object 2 : HLA Object
: Federate
1: Initiate Explicit Interaction
2: Explicit Interaction
3: Or, Implicit Interaction
Explicit Interactions between Objects are initiated by the Federate in response to an Interaction received from an object or from the RTI.
Implicit Interactions between objects occur when one object calls the methods of another.

Figure 8. Use Case: Object Interaction


Use Case: Use Interaction
Object 2 : HLA Object
Object 2 : HLA Object
: Federate
: Federate Ambassador
1: Give Interaction
2: Initiate Explicit Interaction
3: Resulting Explicit Interaction
4: An unrelated Implicit Interaction

This use case differs from Object Interaction in that it shows how an explicit interaction is initiated following reception of an Interaction from the Federate Ambassador.

Explicit Interactions between Objects are initiated by the Federate in response to an Interaction received from an object or from the RTI.

Implicit Interactions between objects occur when one object calls the methods of another.

Figure 9. Use Case Use Interaction

Analysis Architecture

The following diagram shows the primary iteration analysis architecture in terms of a Rumbaugh Object Model Diagram.

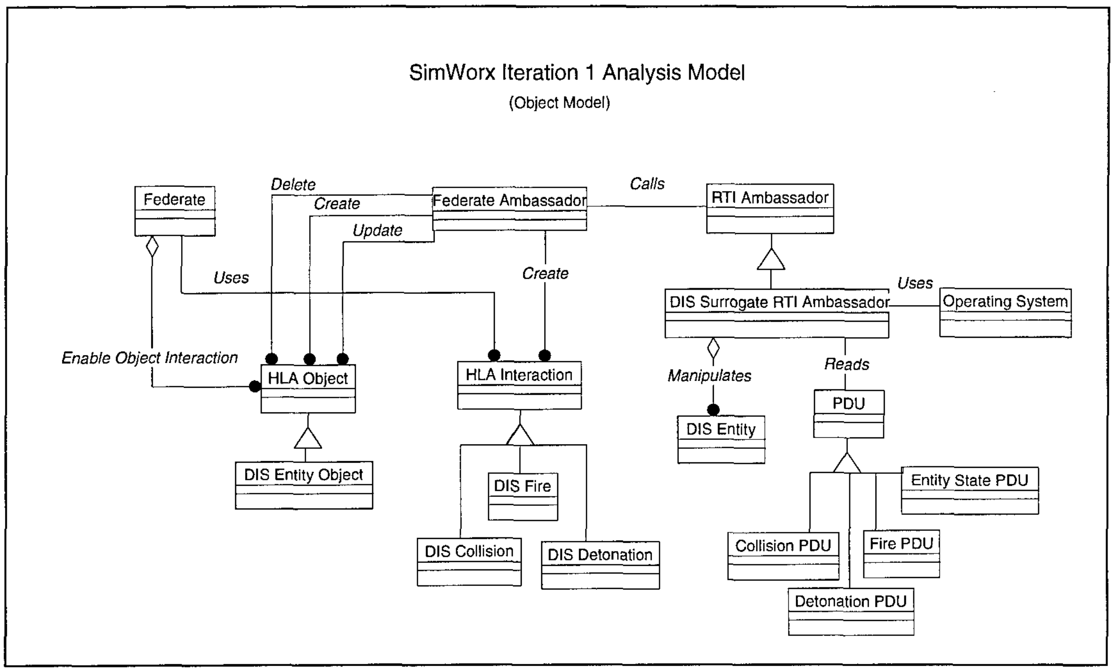


Figure 10. SimWorx Analysis Architecture

Classes

Class name:

Collision PDU

Documentation:

This class represents the Collision PDU as defined by DIS standard 2.0.4 paragraph 5.4.3.2

Superclasses:

PDU

Associations:

<none>

Attributes:

Issuing Entity ID :
Colliding Entity ID :
Collision Event ID :
Collision Type :
Collision Velocity :
Mass :
Location Offset :

Operations:

<none>

Class name:

Detonation PDU

Documentation:

This class represents the Detonation PDU as defined by DIS standard 2.0.4 paragraph 5.4.4.2

Superclasses:

PDU

Associations:

<none>

Attributes:

Firing Entity ID :
Target Entity ID :
Munition ID :
Fire Event ID :
Fire Velocity :
Location In World :
Burst Descriptor :
Location Offset :
Detonation Result :
Articulation Parameter Count :
Articulation Parameters :

Operations:

<none>

Class name:

DIS Collision

Documentation:

This class represents a DIS Collision interaction between two DIS Entity Objects.

Superclasses:

HLA Interaction

Associations:

<none>

Attributes:

Collision_Event_ID :
Collision_Type :
Collision_Velocity :
Mass :
Location_Offset :

Operations:

<none>

Class name:

DIS Detonation

Documentation:

This class represents a DIS Detonation interaction between two DIS Entity Objects.

Superclasses:

HLA Interaction

Associations:

<none>

Attributes:

Munition_ID :
Fire_Event_ID :
Location :
Burst_Descriptor :
Fire_Velocity :
Location_Offset :
Result :

Operations:

<none>

Class name:

DIS Entity

Documentation:

This class represents an active DIS Entity of a DIS Simulation.

Superclasses:

<none>

Associations:

Association Manipulates

Attributes:

ID :
Force_ID :
Entity_Type :
Marking :
Dead_Reckoning_Parms :
Appearance :
Position_At_Update :
Orientation_At_Update :
Linear_Velocity :
Linear_Acceleration :
Angular_Velocity :
Location :
Orientation :

Operations:

<none>

Class name:

DIS Entity Object

Documentation:

This class is a DIS Entity which is an HLA Object. It is a "ready-made" HLA object for use with DIS-based simulations. Client programmers will subclass this object to create specific DIS entities such as F-15E aircraft and M-1A tanks.

Superclasses:

HLA Object

Associations:

<none>

Attributes:

Entity_ID : Entity Identifier Record
Force_ID :
Entity_Type :
Marking :
Appearance :
Position :
Orientation :

Operations:

<none>

Class name:

DIS Fire

Documentation:

This class represents a DIS Fire interaction between two DIS Entity Objects.

Superclasses:

HLA Interaction

Associations:

<none>

Attributes:

Munition_ID :
Fire_Event_ID :
Location :
Burst_Descriptor :
Fire_Velocity :
Fire_Range :
Fire_Mission_Index :

Operations:

<none>

Class name:

DIS Surrogate RTI Ambassador

Documentation:

This class is a form of RTI Ambassador which presents an HLA face to HLA simulations, but presents a DIS face to the network.

Superclasses:

RTI Ambassador

Associations:

Association Reads
Association Uses
Association Manipulates

Attributes:

<none>

Operations:

<none>

Class name:

Entity State PDU

Documentation:

This class represents the Entity State PDU as defined by DIS standard 2.0.4 paragraph 5.4.3.1

Superclasses:

PDU

Associations:

<none>

Attributes:

Entity_ID :
Force_ID :
Articulation_Parameter_Count :
Entity_Type :
Alternate_Entity_Type :
Entity_Linear_Velocity :
Entity_Location :
Entity_Orientation :
Entity_Appearance :
Dead_Reckoning_Parameters :
Entity_Marking :
Entity_Capabilities :
Articulation_Parameters :

Operations:

<none>

Class name:

Federate

Documentation:

This abstract class represents the client programmer's simulation application. The client programmer will subclass this class to handle the chores of her simulation. For example, a flight simulator subclass would model flight.

The Federate class enables the objects it has to interact with each other.

When it receives an interaction, it explicitly invokes the methods of the objects involved to carry out the interaction.

Superclasses:

<none>

Associations:

Association Uses
Association Enable Object Interaction

Attributes:

<none>

Operations:

<none>

Class name:

Federate Ambassador

Documentation:

This class represents the Federate's interface to the RTI. Rather than simply being an interface wrapper, in the SimWorx framework, the Federate Ambassador is responsible for performing the "mundane" tasks required by all HLA simulations: simulation pauses, simulation saves, simulation restores, etc.

Note that the SimWorx framework initially only supports four of the eighteen methods specified for the Federate Ambassador in the HLA Interface Spec. These methods are the minimum necessary to operate DIS simulations under the HLA (no initial need for advanced Declaration, Object, Ownership, or Time Management functions).

Superclasses:

<none>

Associations:

Association Create
Association Delete
Association Update
Association Create
Association Calls

Attributes:

<none>

Operations:

Discover_Object( )

From section 4.4 of the HLA Interface Spec v1.0

This service informs the federate that the RTI has discovered an object.

Remove_Object( )

From section 4.9 of the HLA Interface Spec v1.0

This method instructs the Federate Ambassador to remove the specified object since the object has been deleted from the federation execution.

Reflect_Attribute_Values(Object_ID : Object_ID, Attribute_Value_List : Name_Value_Pair_Set, The_Time : Federation_Time)

From section 4.5 of the HLA Interface Spec v1.0

Provides the federate with new values for a discovered attribute. This service, coupled with the Update Attribute Values Service, forms the primary data exchange mechanism supported by the RTI.

Receive_Interaction( )

From section 4.7 of the HLA Interface Spec v1.0

Provides information about an action taken by one federation object potentially towards another object. Provides federate with information about an interaction between two objects.

Class name:

Fire PDU

Documentation:

This class represents the Fire PDU as defined by DIS standard 2.0.4 paragraph 5.4.4.1

Superclasses:

PDU

Associations:

<none>

Attributes:

Firing Entity ID :
Target Entity ID :
Munition ID :
Fire Event ID :
Fire Mission Index :
Location In World :
Burst Descriptor :
Fire Velocity :
Fire Range :

Operations:

<none>

Class name:

HLA Interaction

Documentation:

This class represents an abstract HLA Interaction. Framework client programmers will subclass this class for each interaction class specified by their simulation's SOM.

Superclasses:

<none>

Associations:

Association Uses
Association Create

Attributes:

Class : Interaction_Class = 0
The interaction class this interaction instance belongs to.

ID : Interaction_ID = 0
The unique interaction ID of this interaction occurrence.

Initiator : Object_ID = 0
This is the object ID of the object which initiates the interaction.

Fed_Time : Federation_Time
The time the interaction is effective.

Operations:

<none>

Class name:

HLA Object

Documentation:

This class represents an HLA object. Objects interact with each other to carry out the duties of the simulation. Objects may be instantiated and controlled locally by this simulation and also instantiated on the behalf of some other federate. Object interactions may be implicit via method calls to other objects, or explicit via an Interaction invoked by the simulation.

Superclasses:

<none>

Associations:

Association Create
Association Delete
Association Update
Association Enable Object Interaction

Attributes:

Class : Object_Class = 0
The object class this object belongs to.

ID : Object_ID = 0
The ID of this object. It is unique within the federation execution.

Operations:

Create(Object_ID : Object_ID, Object_Class : Object_Class)
Initialize an Object

Class name:

Operating System

Documentation:

This is not so much a class as a notion of the DIS Surrogate RTI Ambassador's dependence upon the operating system to read PDUs.

Superclasses:

<none>

Associations:

Association Uses

Attributes:

<none>

Operations:

<none>

Class name:

PDU

Documentation:

This class represents the common components of all categories of DIS PDU.

Superclasses:

<none>

Associations:

Association Reads

Attributes:

PDU_Header : PDU Header Record
The PDU Header Record is defined by DIS 2.0.4 standard paragraph 5.3.24

Operations:

<none>

Class name:

RTI Ambassador

Documentation:

This class represents the RTI's interface to the Federate.

Provides an abstract interface (specified by RTI IDL specifications) for different kinds of RTIs. SimWorx initially implements a DIS surrogate RTI.

Superclasses:

<none>

Associations:

Association Calls

Attributes:

<none>

Operations:

<none>

Index

Thesis Appendix Note: Since this stand-alone analysis document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute B-5 for 5, B-11 for 11, etc.

Appendix C: SimWorx Iteration One Design Model

SimWorx: Design Model

Iteration One Design Model

Software Architectures in Ada 95
AFIT Software Engineering Group
Air Force Institute of Technology
Wright-Patterson AFB, OH 45434

This Page Intentionally Left Blank

Table Of Contents

Thesis Appendix Note: Since this stand-alone design document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute C-5 for 5, C-11 for 11, etc.

List of Figures

Thesis Appendix Note: Since this stand-alone design document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute C-5 for 5, C-11 for 11, etc.

Introduction

This document reflects the first iteration of software design for the SimWorx application framework for distributed simulation.

The SimWorx framework enables application programmers (a.k.a. client programmers) to quickly and easily generate distributed simulation applications conforming to the DoD High Level Architecture (HLA) for Modeling and Simulation. In addition, the SimWorx framework provides a Surrogate HLA Run-Time Infrastructure (RTI) which performs Distributed Interactive Simulation (DIS) version 2.0.4 network input and output.

Initial versions of the SimWorx framework do not support either the full HLA nor full DIS standards. Initial versions supply the minimal functionality necessary for simple objects to interact for typical DIS interactions (entity movement and collision, weapons fire and detonation).

This first iteration of the SimWorx framework concentrates on receiving input from the DIS network. The application framework is divided into two main categories: Federate components with the behavior of an HLA Federate, and RTI components with the behavior of the HLA RTI.

DIS Protocol Data Units (PDUs) are received from the network by the Surrogate RTI Ambassador[1]. DIS Entities are modeled and their states are dead-reckoned between network updates. DIS Entity information is forwarded by the RTI Ambassador to the Federate Ambassador via the standard HLA ambassador interface, so the federate components may be reused with various sorts of RTIs--see the Federate Ambassador class for details.

[1] The term *Ambassador* is an HLA term for a kind of interface software component. Just as a diplomatic ambassador represents a foreign country, an *Ambassador* presents a local application programming interface for a system component which may actually be physically located somewhere else (another computer, another state, etc.). Thus, the client can make software procedure calls to the Ambassador, rather than relying on some message passing interface, since the Ambassador "abstracts-out" the details of the actual interaction.

Design Architecture

The following diagrams show the design architecture for the Iteration One analysis architecture in terms of a Rational Rose Class Category Diagram and Rumbaugh Object Model Diagrams.

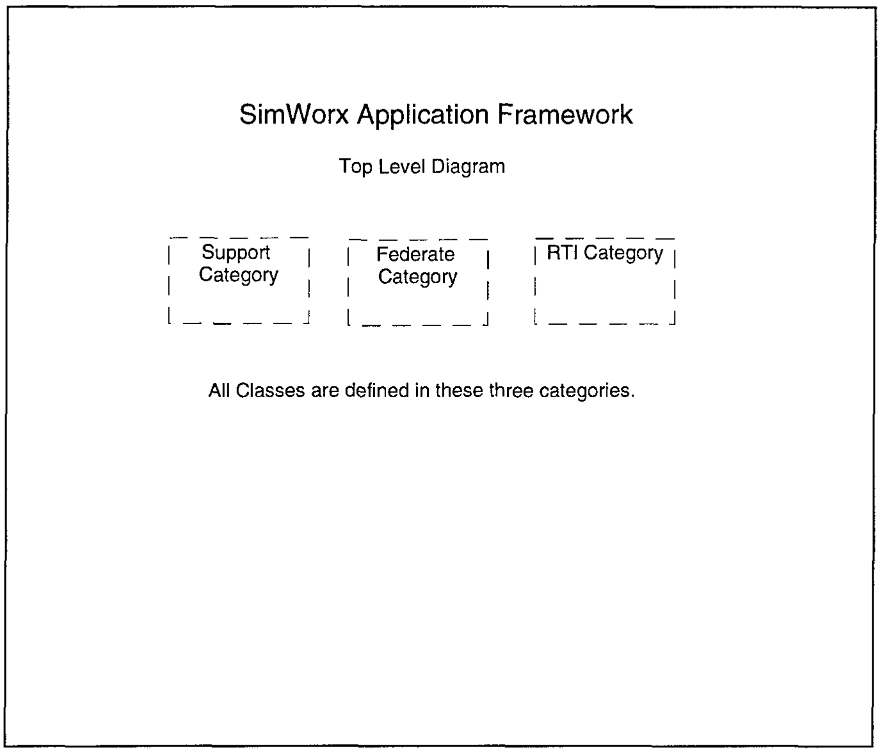

SimWorx Application Framework
Top Level Diagram
Support Category
Federate Category
RTI Category
All Classes are defined in these three categories.

Figure 1. SimWorx Application Framework

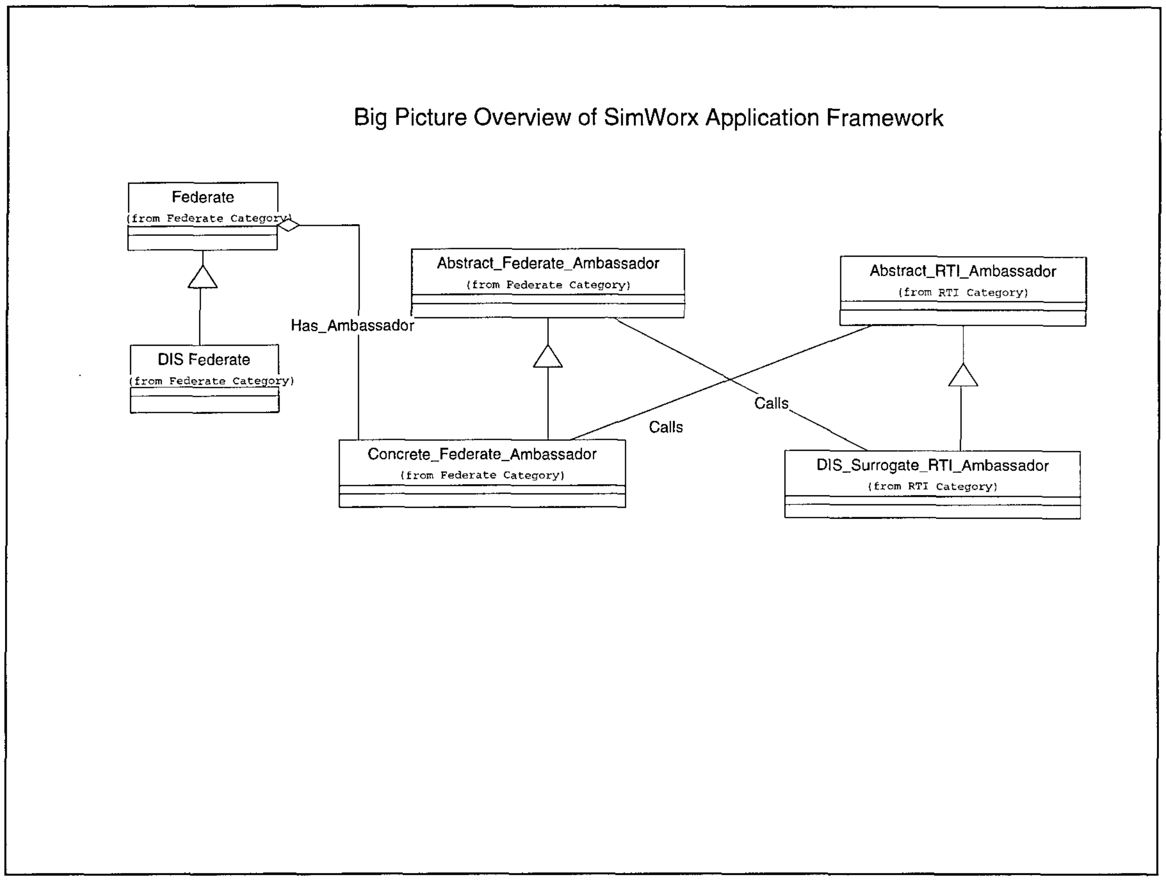


Figure 2. Big Picture Overview of SimWorx Application Framework

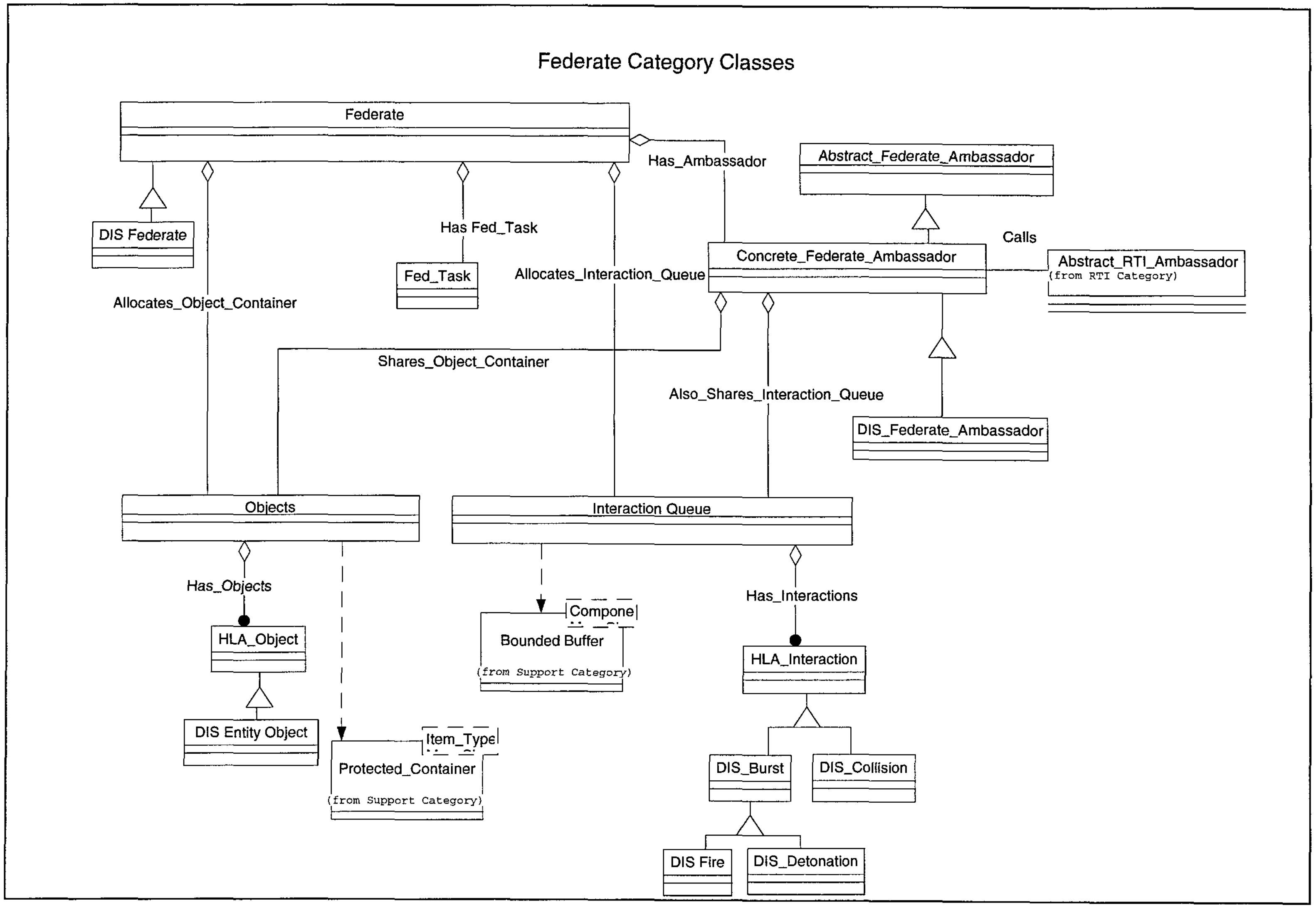


Figure 3. Federate Category Classes

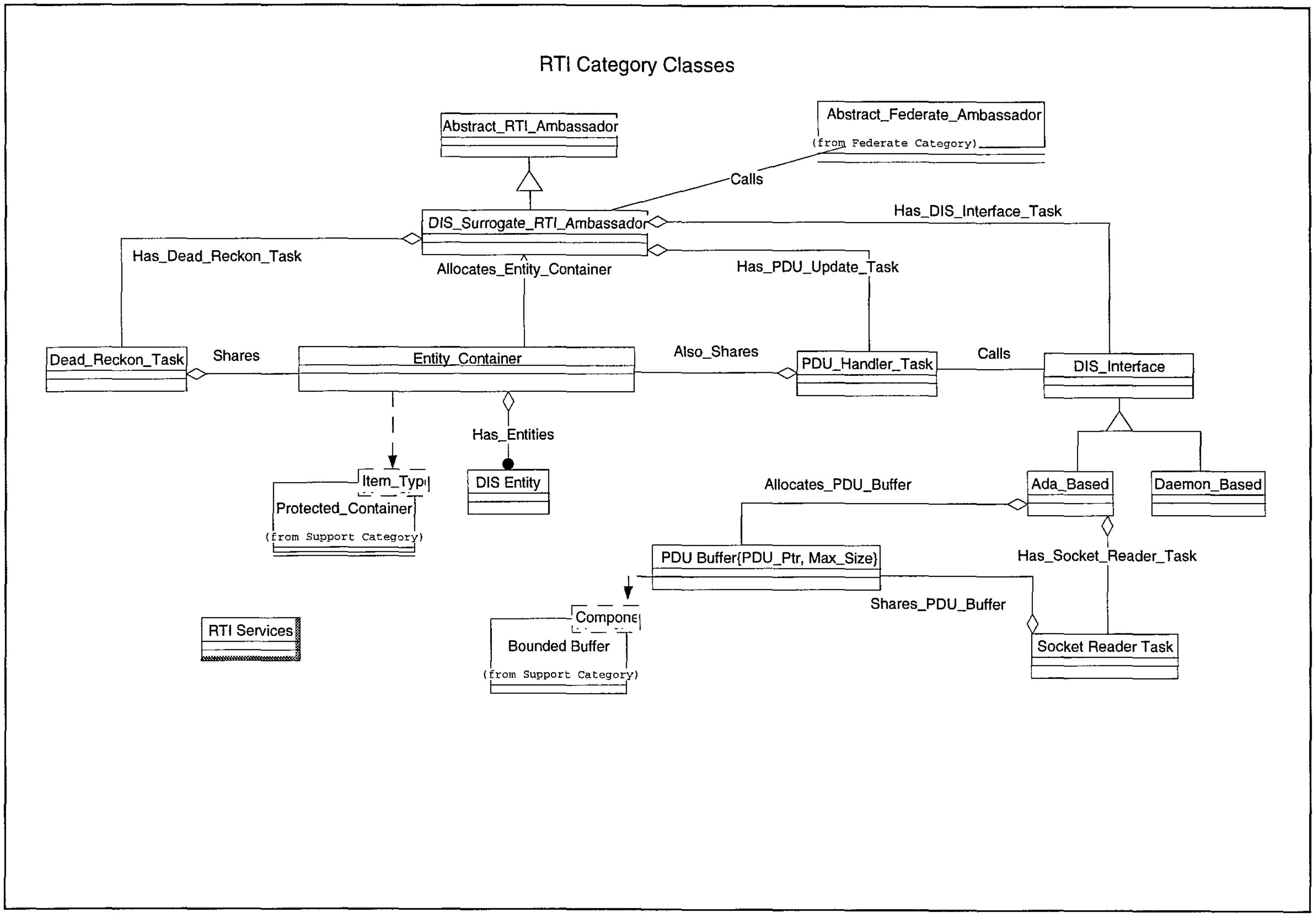
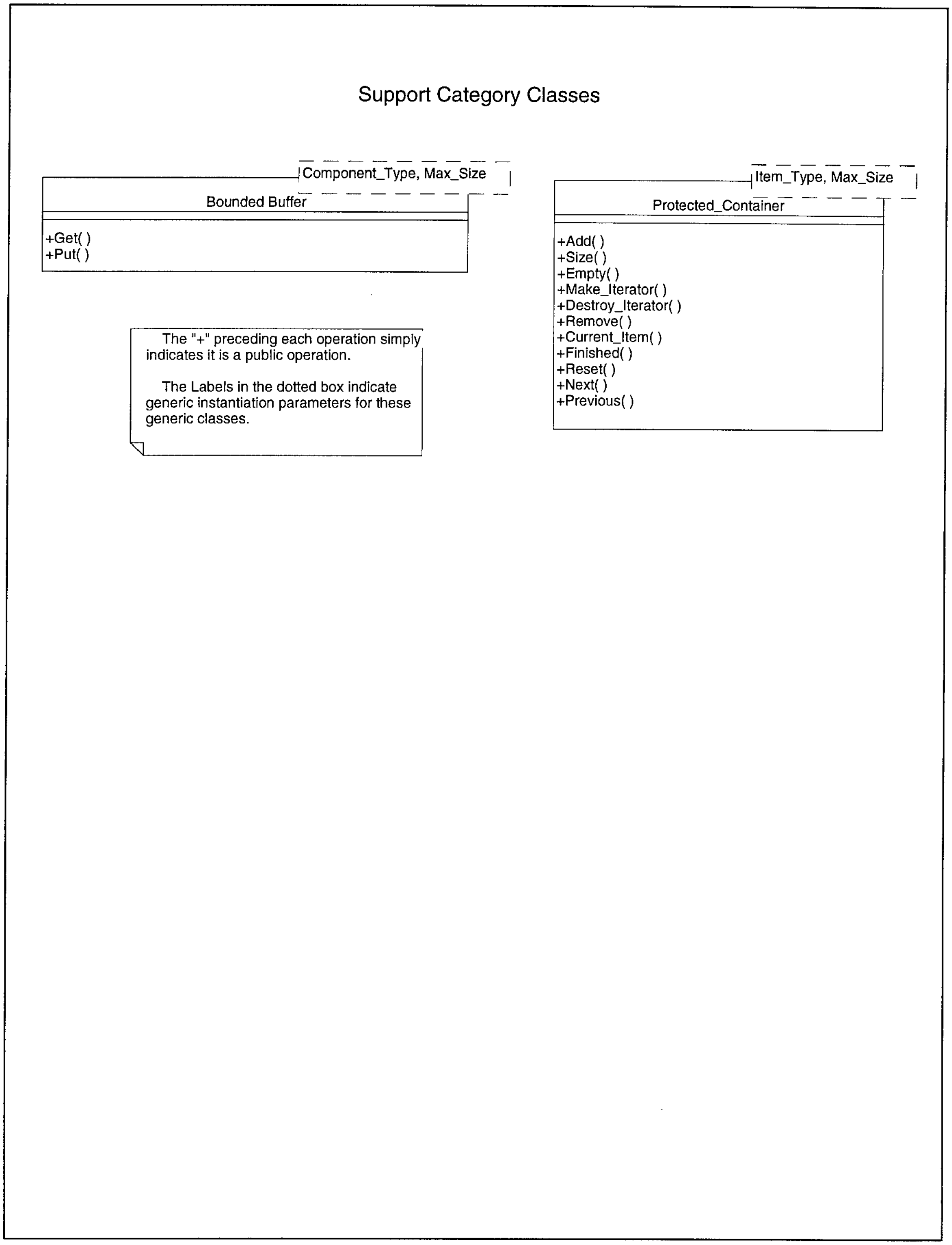


Figure 4. RTI Category Classes



The "+" preceding each operation simply indicates it is a public operation.

The Labels in the dotted box indicate generic instantiation parameters for these generic classes.

Figure 5. Support Category Classes

Scenarios

The following scenarios describe the first iteration functionality of the SimWorx Application Framework. They expand upon the use cases of the analysis model, and are described with Message Trace Diagrams. Each numbered item is a message from a sending class to a receiving class. The messages are ordered by time: later numbered messages are later in time than earlier numbered messages.

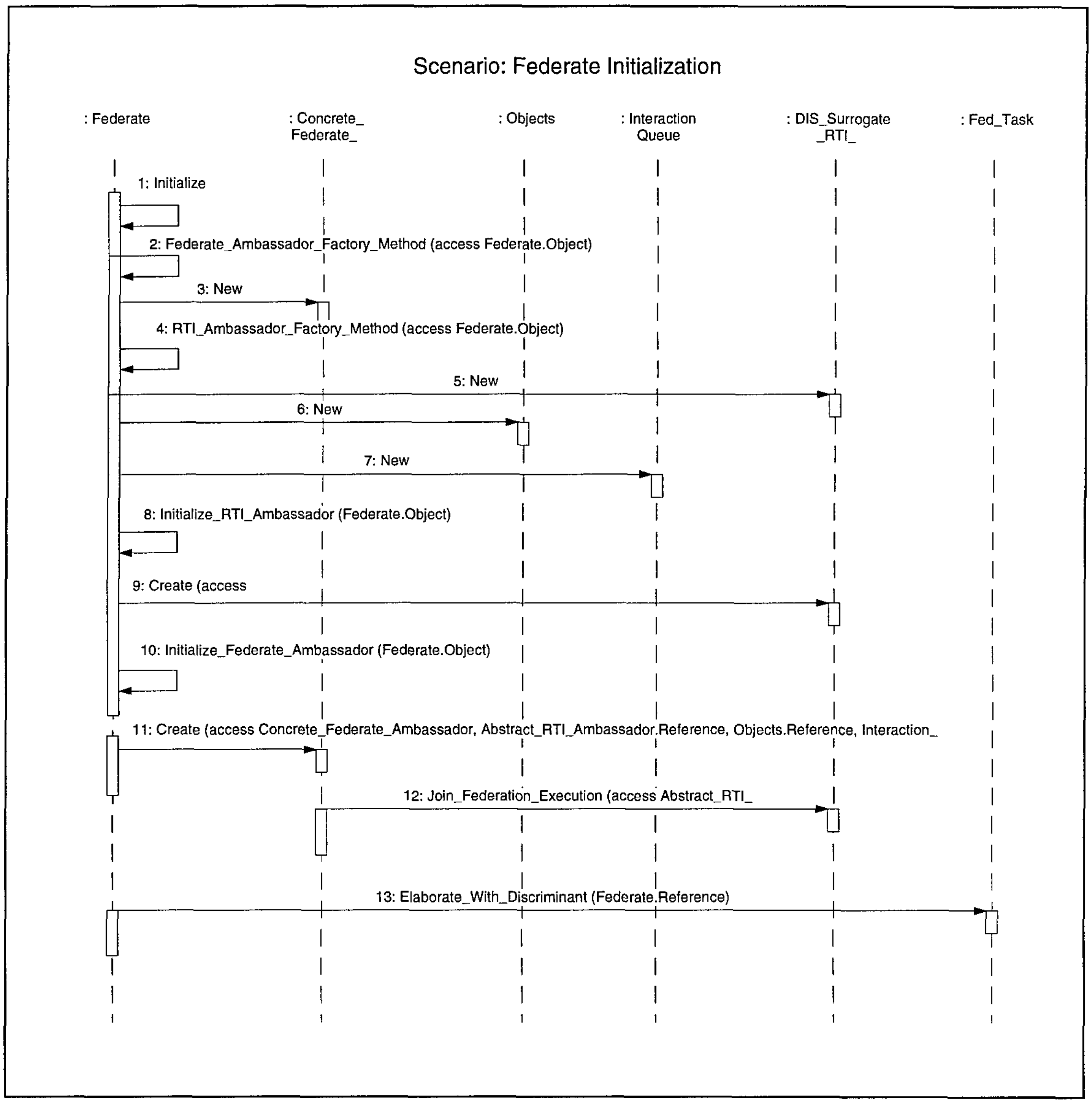

Scenario: Federate Initialization
: Federate
: Concrete_ Federate_
: Objects
: Interaction Queue
: DIS_Surrogate _RTI_
: Fed_Task
1: Initialize
2: Federate_Ambassador_Factory_Method (access Federate.Object)
3: New
4: RTI_Ambassador_Factory_Method (access Federate.Object)
5: New
6: New
7: New
8: Initialize_RTI_Ambassador (Federate.Object)
9: Create (access
10: Initialize_Federate_Ambassador (Federate.Object)
11: Create (access Concrete_Federate_Ambassador, Abstract_RTI_Ambassador.Reference, Objects.Reference, Interaction_
12: Join_Federation_Execution (access Abstract_RTI_
13: Elaborate_With_Discriminant (Federate.Reference)

Figure 6. Scenario: Federate Initialization

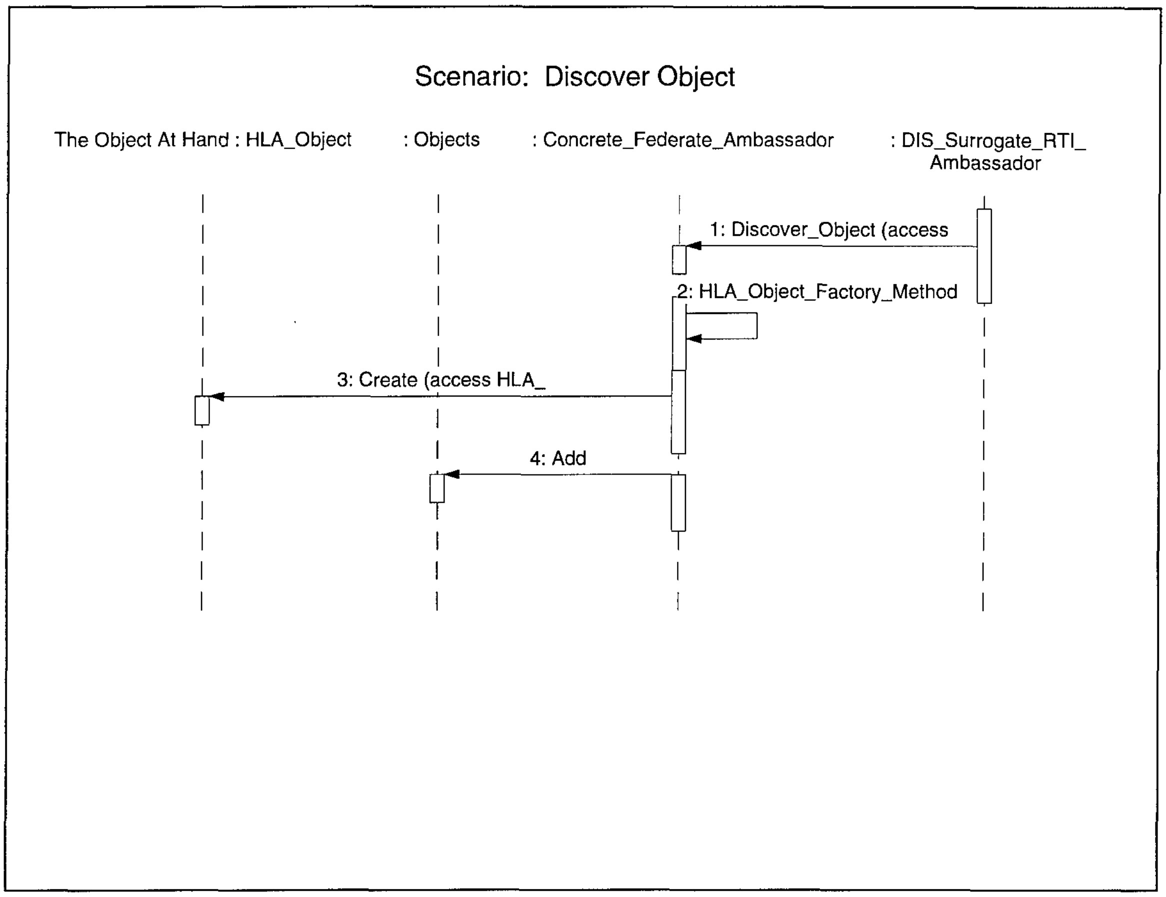

Scenario: Discover Object
The Object At Hand : HLA_Object
: Objects
: Concrete_Federate_Ambassador
: DIS_Surrogate_RTI_ Ambassador
1: Discover_Object (access
2: HLA_Object_Factory_Method
3: Create (access HLA_
4: Add

Figure 7. Scenario: Discover Object

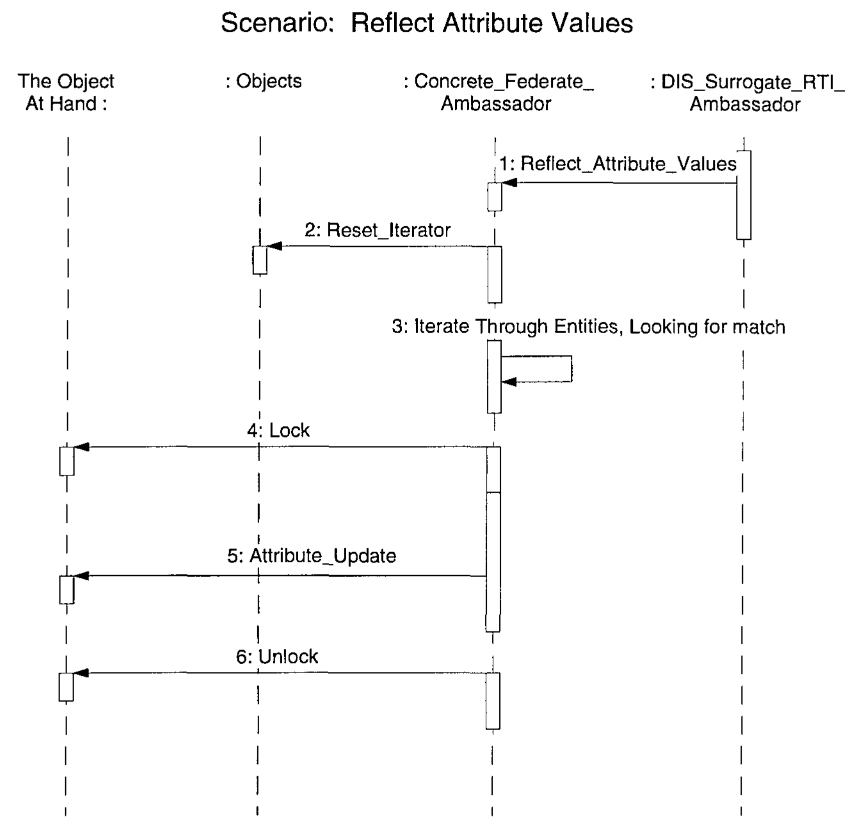

Scenario: Reflect Attribute Values
The Object At Hand :
: Objects
: Concrete_Federate_ Ambassador
: DIS_Surrogate_RTI_ Ambassador
1: Reflect_Attribute_Values
2: Reset_Iterator
3: Iterate Through Entities, Looking for match
4: Lock
5: Attribute_Update
6: Unlock

Figure 8. Scenario: Reflect Attribute Values

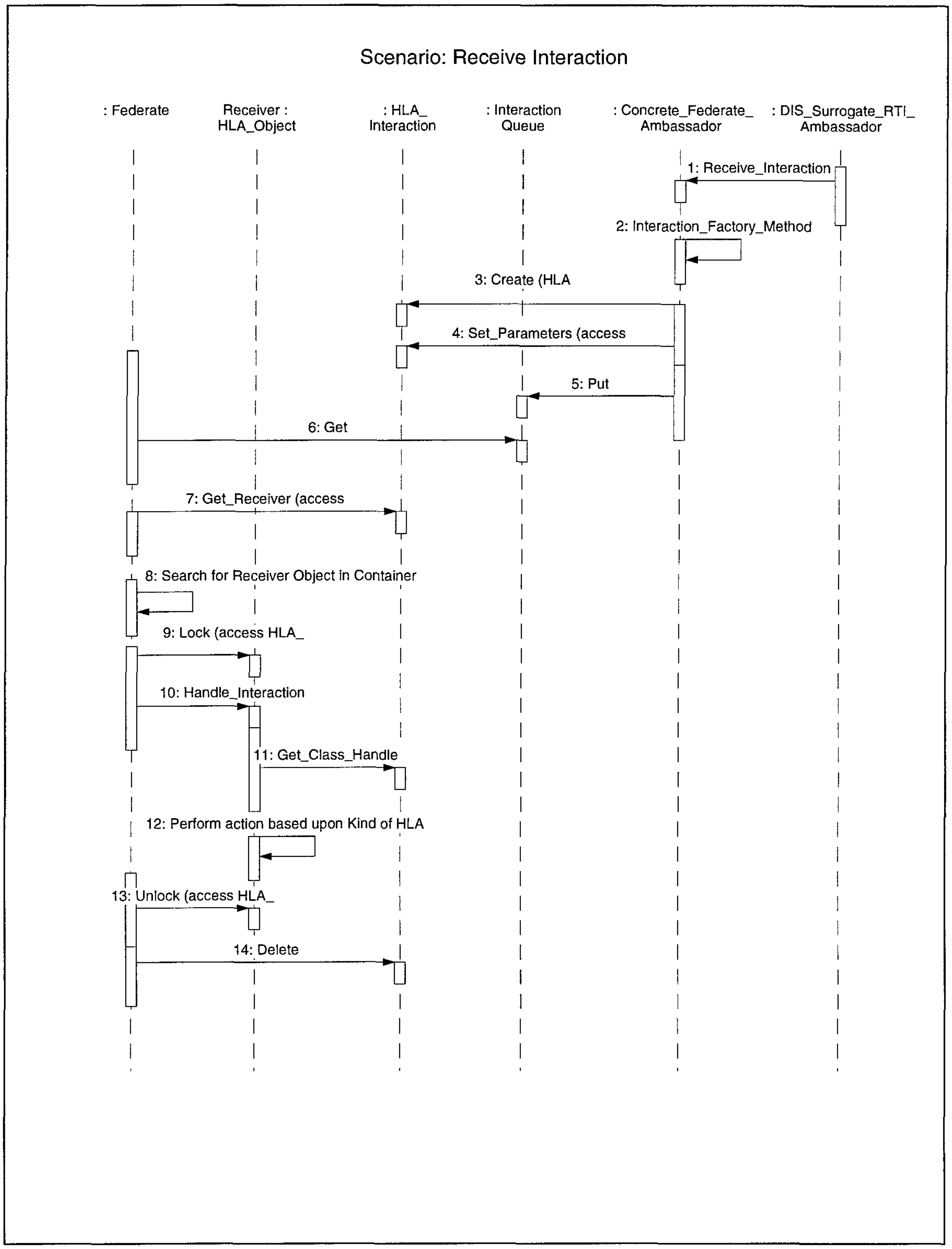


Figure 9. Scenario Receive Interaction

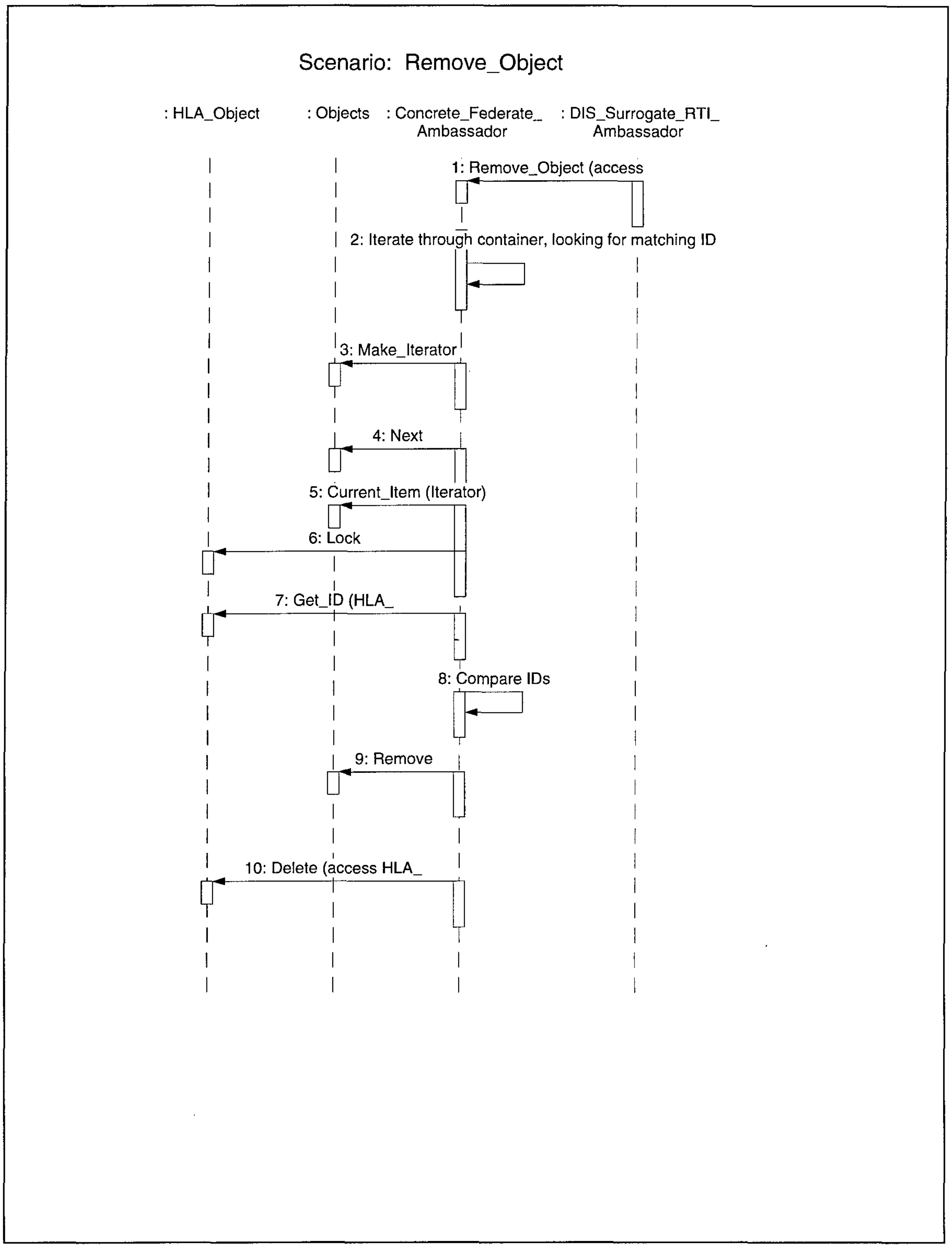


Figure 10. Scenario: Remove Object

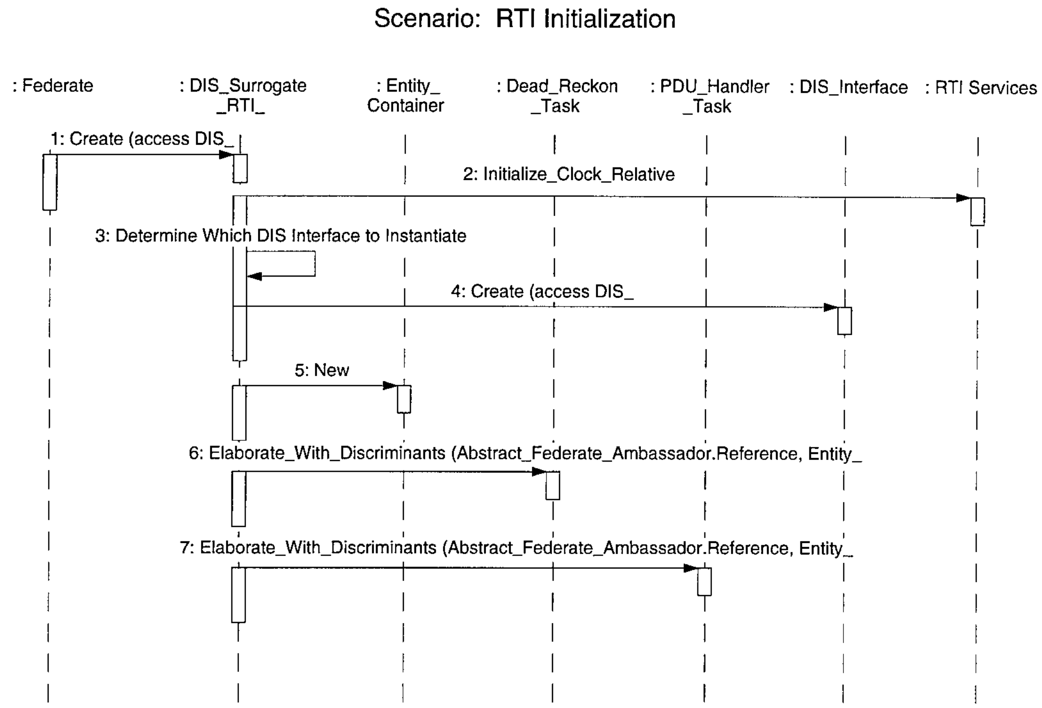

Scenario: RTI Initialization
: Federate
: DIS_Surrogate _RTI_
: Entity_ Container
: Dead_Reckon _Task
: PDU_Handler _Task
: DIS_Interface
: RTI Services
1: Create (access DIS_
2: Initialize_Clock_Relative
3: Determine Which DIS Interface to Instantiate
4: Create (access DIS_
5: New
6: Elaborate_With_Discriminants (Abstract_Federate_Ambassador.Reference, Entity_
7: Elaborate_With_Discriminants (Abstract_Federate_Ambassador.Reference, Entity_

Figure 11. Scenario: RTI Initialization

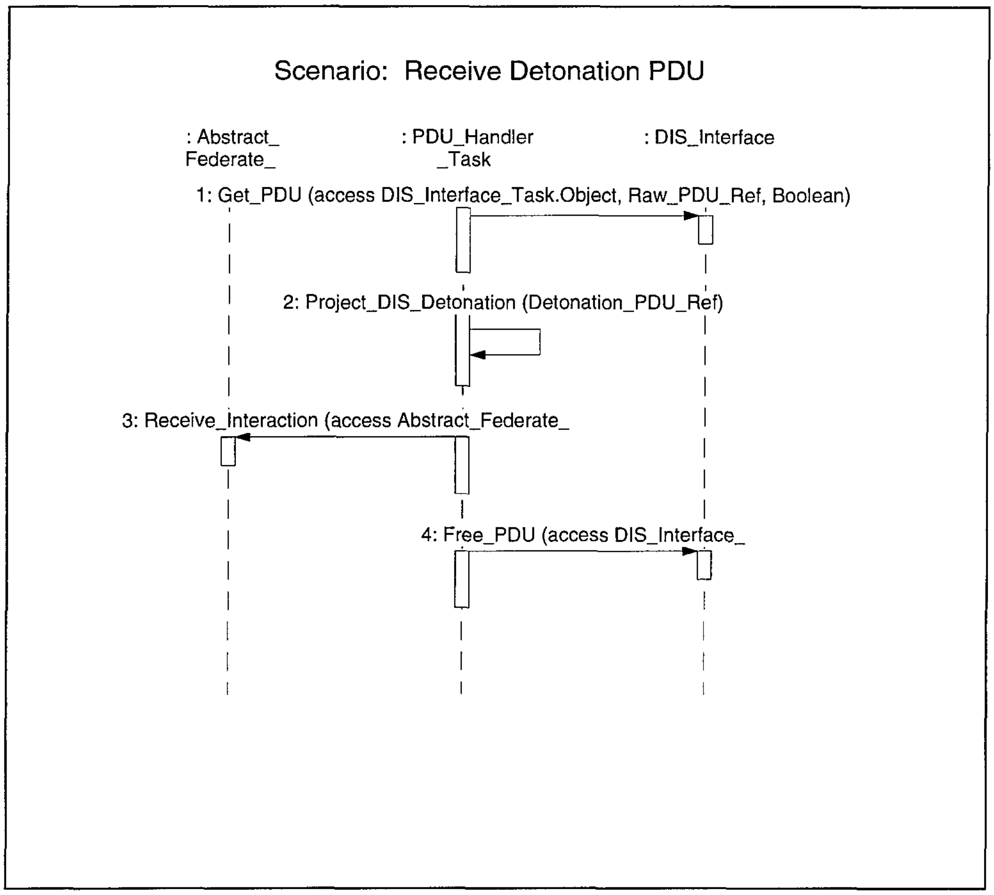

Scenario: Receive Detonation PDU
: Abstract_ Federate_
: PDU_Handler _Task
: DIS_Interface
1: Get_PDU (access DIS_Interface_Task.Object, Raw_PDU_Ref, Boolean)
2: Project_DIS_Detonation (Detonation_PDU_Ref)
3: Receive_Interaction (access Abstract_Federate_
4: Free_PDU (access DIS_Interface_

Figure 12. Scenario: Receive Detonation PDU

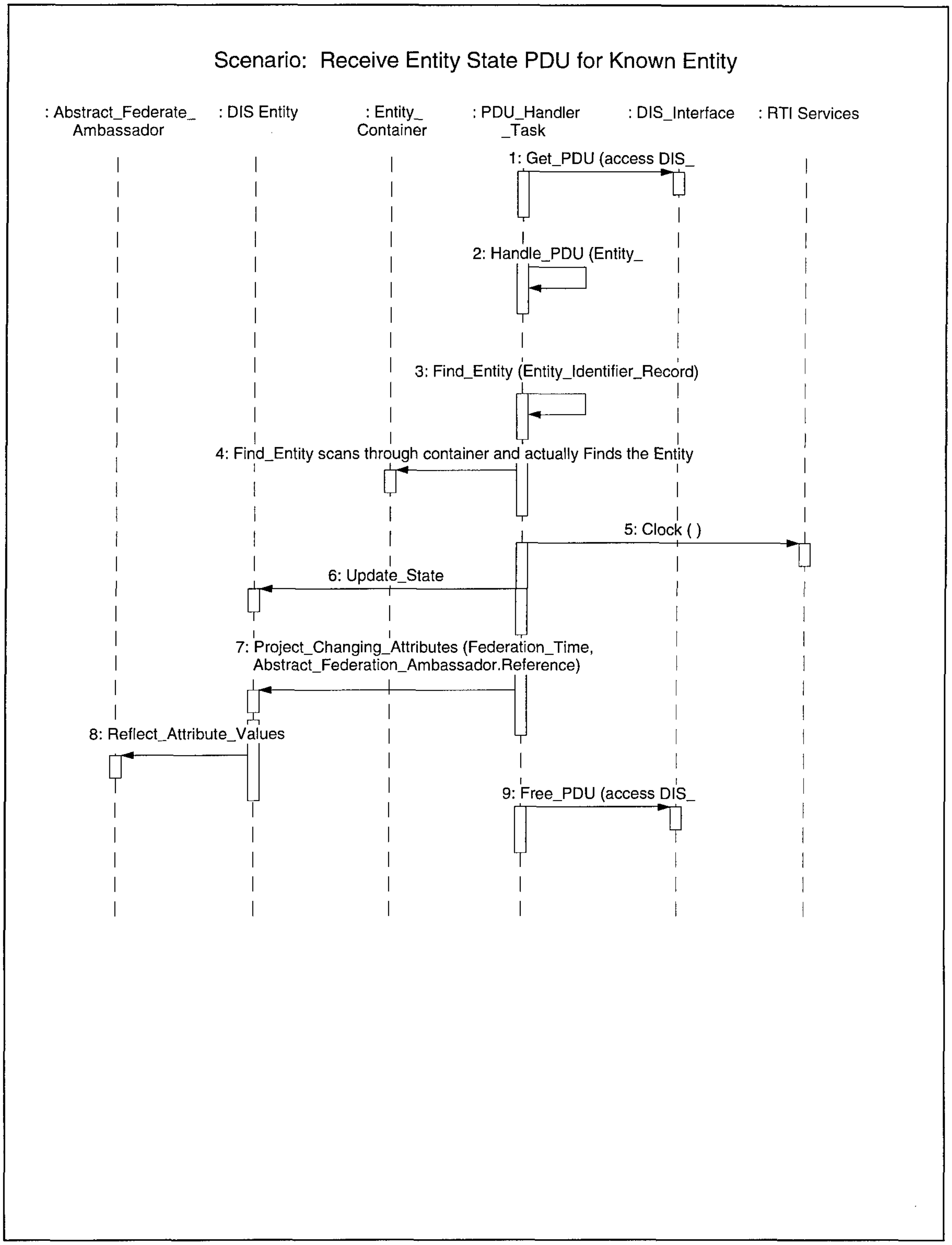

Scenario: Receive Entity State PDU for Known Entity
: Abstract_Federate_ Ambassador
: DIS Entity
: Entity_ Container
: PDU_Handler _Task
: DIS_Interface
: RTI Services
1: Get_PDU (access DIS_
2: Handle_PDU (Entity_
3: Find_Entity (Entity_Identifier_Record)
4: Find_Entity scans through container and actually Finds the Entity
5: Clock ( )
6: Update_State
7: Project_Changing_Attributes (Federation_Time, Abstract_Federation_Ambassador.Reference)
8: Reflect_Attribute_Values
9: Free_PDU (access DIS_

Figure 13. Scenario: Receive Entity State PDU for Known Entity

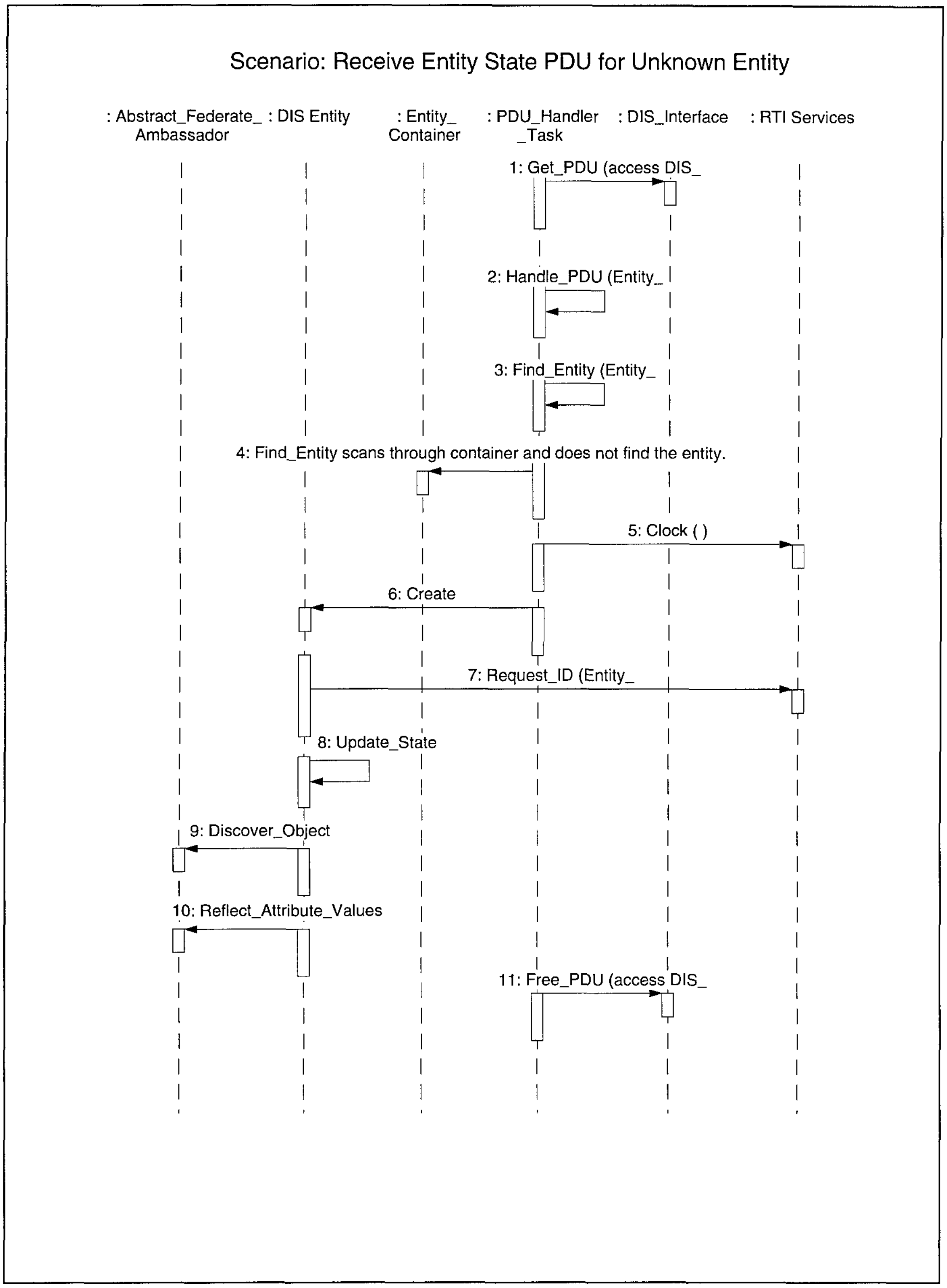


Figure 14. Scenario: Receive Entity State PDU for Unknown Entity

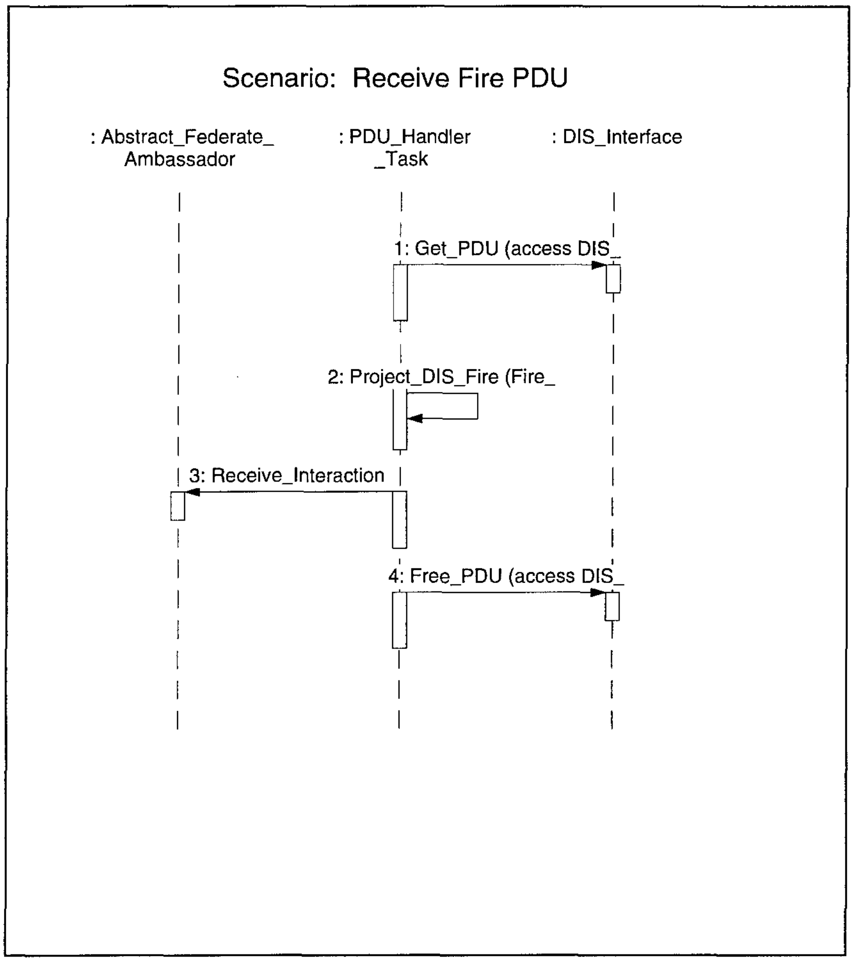


Figure 15. Scenario: Receive Fire PDU

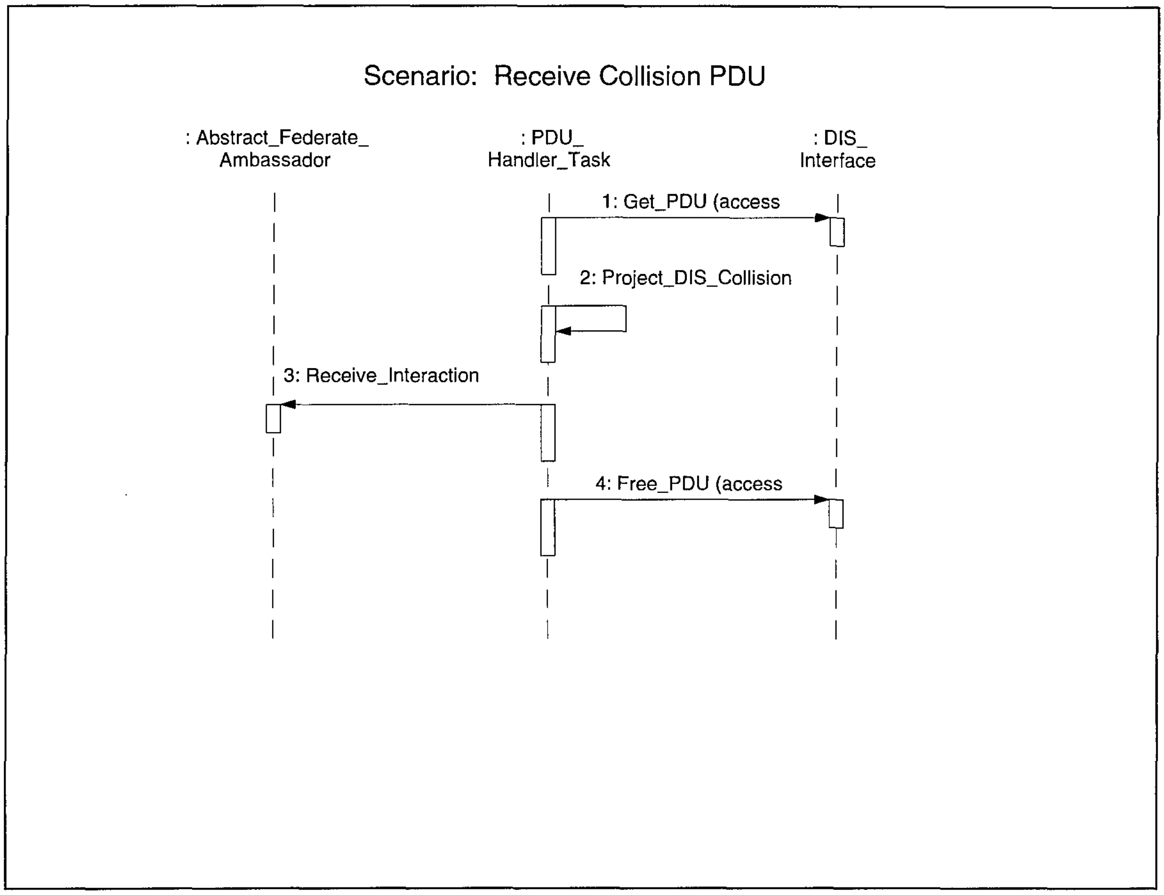


Figure 16. Scenario: Receive Collision PDU

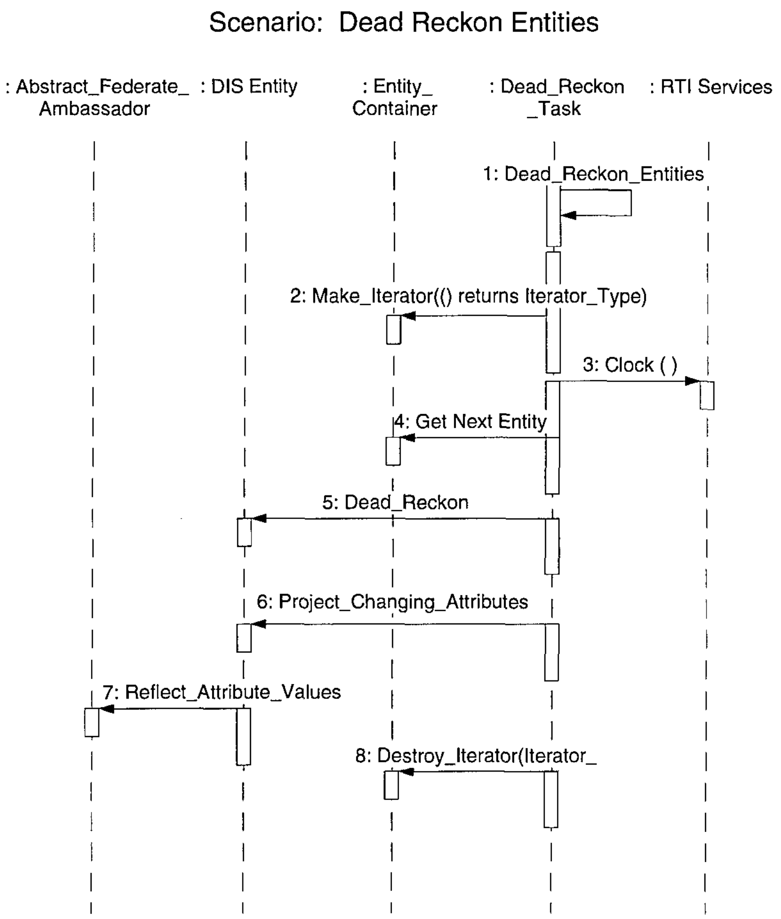

Scenario: Dead Reckon Entities
: Abstract_Federate_ Ambassador
: DIS Entity
: Entity_ Container
: Dead_Reckon _Task
: RTI Services
1: Dead_Reckon_Entities
2: Make_Iterator(() returns Iterator_Type)
3: Clock ( )
4: Get Next Entity
5: Dead_Reckon
6: Project_Changing_Attributes
7: Reflect_Attribute_Values
8: Destroy_Iterator(Iterator_

Figure 17. Scenario: Dead Reckon Entities

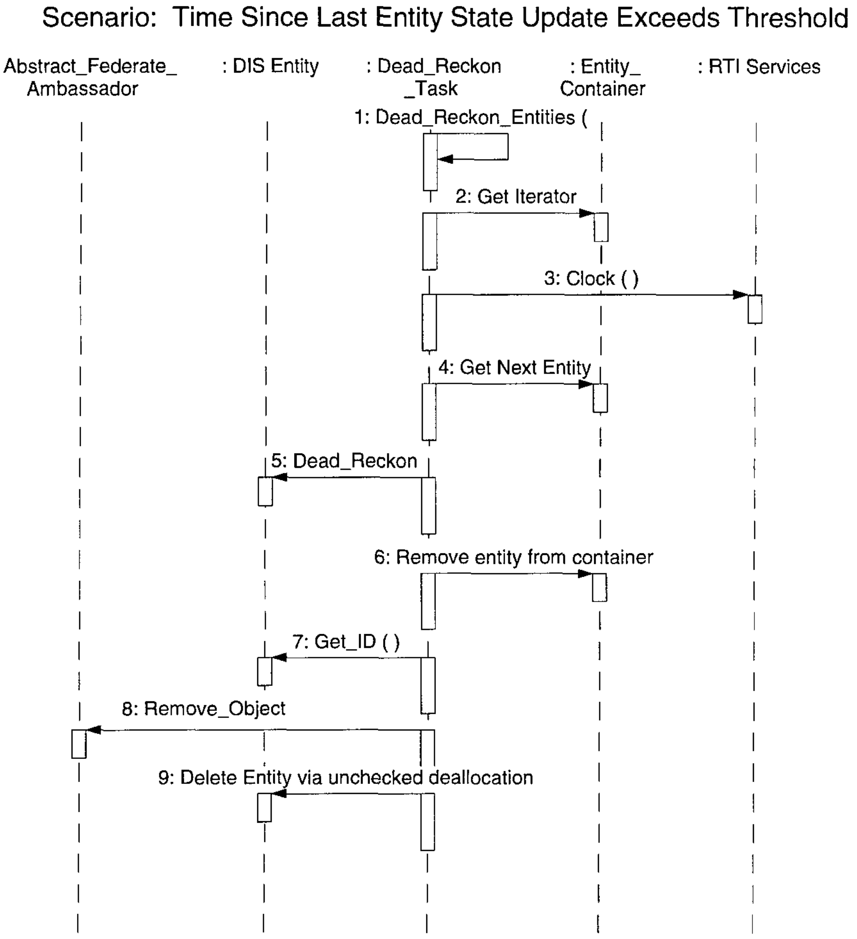

Scenario: Time Since Last Entity State Update Exceeds Threshold
: Abstract_Federate_ Ambassador
: DIS Entity
: Dead_Reckon _Task
: Entity_ Container
: RTI Services
1: Dead_Reckon_Entities (
2: Get Iterator
3: Clock ( )
4: Get Next Entity
5: Dead_Reckon
6: Remove entity from container
7: Get_ID ( )
8: Remove_Object
9: Delete Entity via unchecked deallocation

Figure 18. Scenario: Time Since Last Entity State Update Exceeds Threshold

Classes

This section of the document describes each class of the SimWorx Application Framework.

Class name:

Abstract_Federate_Ambassador

Documentation:

This class represents the Simulation's interface to the RTI. Rather than simply being an interface wrapper, in the SimWorx framework, the Federate Ambassador is responsible for performing the "mundane" tasks required by all HLA simulations: simulation pauses, simulation saves, simulation restores, etc.

This class is abstract since Ada does not allow "circular withs". In order for the Federate Ambassador and RTI Ambassador to call each other, each needs to be defined with no knowledge of the other. Given that, the concrete Child Class of Sim Ambassador will reference the operations of the Abstract RTI Ambassador and the concrete Child Class of the RTI Ambassador will reference the operations of the Abstract Federate Ambassador.

Note that the SimWorx framework initially only supports only four of the eighteen methods specified for Federate Ambassadors in the HLA Interface Spec. These methods are the minimal necessary to operate DIS simulations under HLA (no need for advanced Declaration, Object, Ownership, or Time Management functions). The other methods should be implemented at some future date.

Superclasses:

<none>

Associations:

Association Calls

Attributes:

<none>

Operations:

Discover_Object(Instance : access Abstract_Federate_Ambassador.Object, The_Object : Object_ID, The_Object_Class : Object_Class_Handle, The_Time : Federation_Time, The_Tag : User_Supplied_Tag, The_Handle : Event_Retraction_Handle)

From section 4.4 of the HLA Interface Spec v1.0
This service informs the federate that the RTI has discovered an object.
Raises Exceptions:
Could_Not_Discover
Object_Class_Not_Known
Invalid_Federation_Time
Federate_Internal_Error

Reflect_Attribute_Values(Instance : access Abstract_Federate_Ambassador.Object, The_Object : Object_ID, The_Attributes : Attribute_Handle_Value_Pair_Set, The_Time : Federation_Time, The_Tag : User_Supplied_Tag, The_Handle : Event_Retraction_Handle)
From section 4.5 of the HLA Interface Spec v1.0
Provides the federate with new values for a discovered attribute. This service, coupled with the Update Attribute Values Service, forms the primary data exchange mechanism supported by the RTI.
Possible Exceptions:
Object_Not_Known
Attribute_Not_Known
Invalid_Federation_Time
Federate_Internal_Error

Receive_Interaction(Instance : access Abstract_Federate_Ambassador.Object, The_Interaction : Interaction_Class_Handle, The_Parameters : Parameter_Handle_Value_Pair_Set, The_Time : Federation_Time, The_Tag : User_Supplied_Tag, The_Handle : Event_Retraction_Handle)
From section 4.7 of the HLA Interface spec v1.0
Provides information about an action taken by one federation object potentially towards another object.
Possible Exceptions:
Interaction_Class_Not_Known
Interaction_Parameter_Not_Known
Invalid_Federation_Time
Federate_Internal_Error

Remove_Object(Instance : access Abstract_Federate_Ambassador.Object, The_Object : Object_ID, The_Reason : Object_Removal_Reason, The_Time : Federation_Time, The_Tag : User_Supplied_Tag, The_Handle : Event_Retraction_Handle)
From section 4.9 of the HLA Interface Spec v1.0
This method instructs the Simulation Ambassador to remove the object specified by Object_ID since the object has been deleted from the federation execution.
Possible Exceptions:
Object_Not_Known
Invalid_Federation_Time
Federate_Internal_Error

HLA_Object_Factory_Method(Instance : access Abstract_Federate_Ambassador.Object, Object_Class : Object_Class_Handle, Object_ID : Object_ID) : Object.Reference

This method is a parameterized factory method which returns a tagged reference to an initialized HLA_Object of the class corresponding to the Object_Class parameter. This method is meant to be overridden by a child class.
Client programmers override this method with one which has knowledge of the client-programmer written classes.
For more information, see the book
"Design Patterns: Elements of Reusable Object-Oriented Software", ISBN 0-201-63361-2, on page 107.

HLA_Interaction_Factory_Method(Instance : access Abstract_Federate_Ambassador.Object, Interaction_Class : Interaction_Class_Handle) : Interaction.Reference

This method does for HLA Interactions what the Object Factory Method does for HLA Objects. This method is meant to be overridden by a child class.

Finalize(Instance : access Abstract_Federate_Ambassador.Object)

Provides an abstract place-holder for a finalization method to "clean up" the instantiated Federate_Ambassador upon finalization.

Class name:

Abstract_RTI_Ambassador

Documentation:

This class represents the RTI's interface to the Simulation.

Provides an abstract interface (specified by RTI IDL specifications) for different kinds of RTIs. SimWorx initially uses a DIS surrogate RTI, since AFIT does not have an HLA RTI.

See comments about Ada & "circular withs" under class Abstract_Federate_Ambassador.

Note that for iteration one, the SimWorx framework only supports one of the forty-six methods specified for the RTI Ambassador in the HLA Interface Spec since it only supports input from the DIS network (no need for advanced Declaration, Object, Ownership, or Time Management functions). The other methods should be implemented at some future date.

Superclasses:

<none>

Associations:

Association Calls

Attributes:

<none>

Operations:

Join_Federation_Execution(Instance : access Abstract_RTI_Ambassador.Object, Your_Name : Federate_Name, Execution_Name : Federation_Execution_Name) : Federate_Handle

From Section 2.3 of the HLA Interface Spec v1.0.

This service affiliates the federate with the named federation execution. Calling this method indicates a federate's intention to participate in the federation.

Possible Exceptions:
Federate_Already_Execution_Member
Federation_Execution_Does_Not_Exist
Concurrent_Access_Attempted
RTI_Internal_Error
Unimplemented_Service

Finalize(Instance : access Abstract_RTI_Ambassador.Object)

Clients call this method prior to deallocating the RTI Ambassador. It is responsible for deallocating and cleaning up internal objects.

Class name:

Ada_Based

Documentation:

This class provides a native Ada network interface for reading and writing DIS PDUs (as opposed to using Bruce Clay's C-language Daemons).
Since this is a polymorphic child class of the DIS_Interface, it implements the same methods defined by that class.

This class provides protected access to Getting PDUs by calling entries in the protected PDU Buffer.

Superclasses:

DIS_Interface

Associations:

Association Allocates_PDU_Buffer
Association Has_Socket_Reader_Task

Attributes:

<none>

Operations:

<none>

Class name:

Bounded Buffer

Documentation:

This class is a generic bounded buffer. It contains up to Max_Size items of Component_Type. It is essentially copied from Burns & Wellings "Concurrency in Ada", Cambridge, ISBN 0-521-41471.

The buffer provides buffering between a producer and a consumer. It's implemented as a protected object containing a simple array of size Max_Size.

Superclasses:

<none>

Associations:

<none>

Attributes:

Store : Buffer_Array
The array of items.

First : an integer type
The index of the first valid item in the array.

Last : an integer type
The index of the last valid item in the array.

Operations:

Get(Item : Item_Type)
Put(Item : Item_Type)

Class name:

Concrete_Federate_Ambassador

Documentation:

This class is the concrete Federate Ambassador, created to resolve the Ada "circular with" problem (see Abstract Federate Ambassador class for more details).

This class overrides all of the methods of the Abstract_Federate_Ambassador, and adds the Create initialization method.

The abstract Factory methods are overridden to raise exceptions if they are called, since the Concrete_Federate_Ambassador is not meant to be used as-is, it's meant to be overridden.

Superclasses:

Abstract_Federate_Ambassador

Associations:

Association Shares_Object_Container
Association Also_Shares_Interaction_Queue
Association Has_Ambassador
Association Calls

Attributes:

RTI_Amb : Abstract_RTI_Ambassador.Reference

This is an Ada classwide type which refers to the Federate_Ambassador's associated RTI. It's necessary to be able to dispatch method calls to the RTI.

The_Handle : Federate_Handle

This handle represents the ID of the Federate (and thus, it's associated Federate_Ambassador) during a federation execution. The Federate Ambassador receives the handle from the RTI via the Join_Federation_Execution method.

Operations:

Create(Instance : access Concrete_Federate_Ambassador, RTI : Abstract_RTI_Ambassador.Reference, Objects : Objects.Reference, Interaction_Queue : Interaction_Queue.Reference, Success : Boolean)

This method initializes the Concrete Federate Ambassador following allocation. It sets it's instance variables to the values passed from the Federate since the Federate is responsible for allocating and initializing the values. Then, it invokes the Join_Federation_Execution method to get the Federate's Handle for this execution.

Class name:

Daemon_Based

Documentation:

This class is the C-based DIS network interface. Is
interfaces with Bruce Clay's (SAIC Corp.) PDU Daemons.
See comments in the DIS_Interface class.
This class provides protected access to Getting PDUs
by wrapping the calls to dsi_user within an Ada
protected object..

Superclasses:

DIS_Interface

Associations:

<none>

Attributes:

<none>

Operations:

<none>

Class name:

Dead_Reckon_Task

Documentation:

This task repeatedly dead-reckons the entities in the container--it "wakes up" periodically (as determined by the Dead-Reckon iteration rate parameter in the Run Time parameters file) and scans through the container, dead-reckoning each entity. As each entity is dead-reckoned, it projects it's values to the Federate. Stale entities are removed from the container.

The exception Program_Error may be raised by the protected container or by a protected DIS Entity if the object is deleted while this task is waiting to use it. Thus, Program_Errors are simply handled in the container loop iteration by doing nothing.

Superclasses:

<none>

Associations:

Association Has_Dead_Reckon_Task
Association Shares

Attributes:

<none>

Operations:

Elaborate_With_Discriminants(Fed_Ambassador : Abstract_Federate_Ambassador.Reference, Entity_Container : Entity_Container.Reference)

This method represents elaborating the task with discriminants.

Dead_Reckon_Entities( )

This method dead-reckons each entity in the Entity_Container. It is called repeatedly by the Dead_Reckon_Task.

```
For each entity:
  Dead-Reckon the entity
  if it's stale,
    remove it from the container,
    tell the Federate to delete the corresponding
object,
    delete the entity.
  if it's not stale,
```

then tell the entity to project the updated values to the Federate.

Kill( )

Kill the Dead_Reckon_Task.

Class name:

DIS Entity

Documentation:

This class represents an active DIS Entity of a DIS Simulation. It is implemented as an Ada protected object so concurrent operations are performed atomically.

Superclasses:

<none>

Associations:

Association Has_Entities

Attributes:

Fed : Federate_Ambassador.Reference = null
Each DIS Entity requires a reference to the Federate Ambassador so it can make the project HLA Object attributes for DIS Entity Objects to the Federate.

ID : Entity_Identifier_Record
This is the Entity's DIS ID as defined by DIS 2.0.4 paragraph 5.3.14.

Force_ID : Nat8
The Entity's Force ID as defined by DIS 2.0.4 standard paragraph 5.3.21.

Entity_Type : Entity_Type_Record
DIS Entity type as defined by DIS Standard paragraph 5.3.16.

Marking : Entity_Marking_Record
DIS Entity Marking as defined by DIS standard 2.0.4 paragraph 5.3.15.

Dead_Reckoning_Parms : Dead_Reckoning_Parameters_Record
DIS DR Parameters as defined by DIS standard 2.0.4 paragraph 5.4.3.1.11.

Appearance : Nat32
The DIS Entity's appearance as of the last PDU received, as defined by DIS Standard paragraph 5.3.12.

Position_At_Update : World_Coordinates_Record

The Entity's coordinates from the last PDU received. As defined by DIS Standard 2.0.4 paragraph 5.3.33.

Orientation_At_Update : Euler_Angles_Type
The Euler Angles (psi, theta, phi) defining the DIS Entity's orientation as of the last PDU received. As defined by DIS standard paragraph 5.3.1.

Linear_Velocity : Linear_Velocity_Record
The Linear Velocity of the DIS Entity as of the last PDU update. As defined by DIS standard paragraph 5.3.32.

Linear_Acceleration : Linear_Acceleration_Record
The entity's linear acceleration at the last PDU update, as defined by DIS Standard 2.0.4 paragraph 5.3.32.

Angular_Velocity : Angular_Velocity_Record
The DIS Entity's angular velocity as of the last PDU update. As defined by DIS standard paragraph 5.3.2.

Location : World_Coordinates_Record
The Entity's current coordinates. As defined by DIS Standard 2.0.4 paragraph 5.3.33.

Orientation : Euler_Angles_Type
The Euler Angles (psi, theta, phi) defining the DIS Entity's current orientation. As defined by DIS standard paragraph 5.3.1.

WW_ID : Object_ID = 0
This is the Entity's HLA Object ID, provided by the RTI.

Last_PDU_Stamp : DIS_Time_Stamp
The DIS Time Stamp from the PDU Header Record. This is the time the PDU was issued.

WB_Matrix : Flt32_Matrix
This matrix is the World Coordinates to Body Coordinates orientation matrix from section B3.6.1 of the DIS Standard.
It's updated with each new PDU received for entities employing Dead Reckoning models using orientation: RPW, RVW, FPB, RPB, RVP, FVB (see section B3.6.1).

Roll_Squared : Flt32
The entity's roll, squared. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Pitch_Squared : Flt32

The entity's pitch, squared. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Yaw_Squared : Flt32

The entity's yaw, squared. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Roll_Pitch : Flt32

The entity's roll * pitch. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Roll_Yaw : Flt32

The entity's roll * yaw. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Pitch_Yaw : Flt32

The entity's pitch * yaw. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Omega_Magnitude_Squared : Flt32

This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update. See paragraph B3.6.1.1

Omega_Magnitude : Flt32

This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update. See paragraph B3.6.1.1

Last_PDU_Received : Federation_Time

The time the last PDU was received by the DIS_Surrogate_RTI_Ambassador.

DR_Matrix : Flt32_Matrix

The [DR] Matrix from section B3.6.1 of the DIS standard. It's updated with each new PDU and with each dead-reckoning cycle.

New_WB_Matrix : Flt32_Matrix

The [R]w->B matrix from section B3.6.1 of the DIS standard. It's updated with each new PDU and with each dead-reckoning cycle.

Operations:

Set_Fed(Federate : Federate_Ambassador.Reference)

Set the Entity's Federate Ambassador reference used to dispatch calls to the Federate Ambassador.

Set_ID(New_ID : Object_ID)
Set the Entity's HLA ID

Get_ID( ) : Object_ID
Get the Entity's HLA object ID.

Get_DIS_ID( ) : Entity_Identifier_Record
Get the Entities DIS ID (different from it's HLA Object ID).

Update_State(Current_Time : Federation_Time, Fresh_PDU : Entity_State_PDU_Ref)
Update the entity's state from the fresh PDU.

Dead_Reckon(Current_Time : Federation_Time, Is_Stale : Boolean)
Dead_Reckon the Entity based upon it's dead-reckoning parameters. If the max time between updates has been exceeded, then return Is_Stale = True to indicate the entity has become stale.

Print( )
Print the current values of the Entity's attributes on standard output.

Project_All_Attributes(Current_Time : Federation_Time, Federate : Abstract_Federate_Ambassador.Reference)
Send the current attribute values to the Federate via the Reflect_Attribute_Values method.

Project_Changing_Attributes(Current_Time : Federation_Time, Fed_Ambassador : Abstract_Federation_Ambassador.Reference)
Send the attributes which change during each dead-reckoning cycle to the Federate via the Reflect_Attribute_Values method.

Propagate_Discovered_Object(Current_Time : Federation_Time, Fed_Ambassador : Abstract_Federate_Ambassador.Reference)
Tell the Federate about this new entity. Contains both Discover_Object and Reflect_Attribute_Values method calls.

Class name:

DIS Entity Object

Documentation:

This class is a DIS Entity which is an HLA Object. It is a "ready-made" HLA object for use with DIS-based simulations. Client programmers will subclass this object to create specific DIS entities such as F-15E aircraft and M-1A tanks.

Superclasses:

HLA_Object

Associations:

<none>

Attributes:

Entity_ID : Entity_Identifier_Record
This is the DIS Entity ID. Reference DIS Standard 2.0.4 paragraph 5.3.14

Force_ID : Nat8
This is the DIS Force ID, as specified by DIS Standard 2.0.4 paragraph 5.3.21

Entity_Type : Entity_Type_Record
DIS Entity type as defined by DIS Standard paragraph 5.3.16.

Marking : Entity_Marking_Record
DIS Entity Marking as defined by DIS Standard 2.0.4 paragraph 5.3.15.

Appearance : Nat32
DIS Appearance as defined by DIS Standard 2.0.4 paragraph 5.3.12.

Position : World_Coordinates_Record
The DIS Entitiy's coordinates as defined by DIS Standard 2.0.4 paragraph 5.3.33.

Orientation : Euler_Angles_Type
The Euler Angles (psi, theta, phi) defining the DIS Entitiy's orientation as specified by DIS Standard 2.0.4 paragraph 5.3.1.

Operations:

Set_Attribute_X(Instance : DIS_Entity_Object.Reference, New_Value : Attribute Value)

DIS Entity Object provides a Set operation for each of it's attributes.

Get_Attribute_X(Instance : DIS_Entity_Object.Reference) : Attribute Value

DIS Entity Object provides a Get operation for each of it's attributes.

Class name:

DIS Federate

Documentation:

This class is the DIS-specific version of the Federate. It's a simple DIS Federate which just periodically prints information about the HLA_Objects and HLA_Interactions of the federation.

It overrides the Federate_Ambassador_Factory_Method to allocate a DIS-specific Federate_Ambassador which has knowledge of the DIS Objects and DIS Interactions. It also overrides the Initialize_Federate_Ambassador method for initialization.

It also overrides the RTI_Ambassador_Factory_Method to allocate the DIS_Surrogate_RTI_Ambassador. Also overrides Initialize_RTI_Ambassador.

It overrides Client_Initialize with a null operation since there is nothing special to do here.

It overrides Client_Loop_Body to periodcally do two things. First, it loops through all of the HLA_Objects in the Objects container, invoking the Show method on each. Second, it pulls each queued HLA_Interaction off of the Interaction_Queue and invokes Show on it.

It overrides Client_Finalize with a null operation since nothing special needs to be done to clean up.

Superclasses:

Federate

Associations:

<none>

Attributes:

<none>

Operations:

<none>

Class name:

DIS Fire

Documentation:

This class represents a DIS Fire interaction between two DIS Entity Objects.

It overrides the Set_Parameters, Make_Parameters, Show, and Delete methods of its parent.

Superclasses:

DIS_Burst

Associations:

<none>

Attributes:

Fire_Range : Flt32
The range of the Fire action.

Fire_Mission_Index : Nat32
The mission index of the fire action.

Operations:

Create(Instance : access DIS_Fire.Object, Class : Interaction_Class_Handle, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag, Event_Retraction_Handle : Event_Retraction_Handle, Initiator : Object_ID, Receiver : Object_ID, Munition_ID : Entity_Identifier_Record, Fire_Event_ID : Event_ID_Record, Location : World_Coordinates_Record, Burst_Descriptor : Burst_Descriptor_Record, Fire_Velocity : Linear_Velocity_Record, Fire_Range : Flt32, Fire_Mission_Index : Nat32)
Initialize a DIS Fire interaction following allocation.

Set_Fire_Range(Instance : access DIS_Fire.Object, New_Value : Flt32)
Set the DIS_Fire interaction's fire range.

Get_Fire_Range(Instance : access DIS_Fire.Object) : Flt32
Get the DIS_Fire Interaction's fire range.

Set_Fire_Mission_Index(Instance : access DIS_Fire.Object, New_Value : Nat32)
Set the DIS_Fire Interaction's mission index.

Get_Fire_Mission_Index(Instance : access DIS_Fire.Object) : Nat32
Get the DIS_Fire Interaction's mission index.

Class name:

DIS_Burst

Documentation:

This class results from factoring the common elements out of the DIS Fire and DIS Detonation subclasses.

It overrides the Set_Parameters, Make_Parameters, Show, and Delete methods of its parent.

Superclasses:

HLA_Interaction

Associations:

<none>

Attributes:

Initiator : Object_ID
The DIS Entity doing the firing.

Receiver : Object_ID
The DIS Entity being fired upon.

Munition_ID : Entity_Identifier_Record
An Entity_Identifier_Record which refers to the munition being fired (if it is a "tracked" munition, record is zeros otherwise.)

Fire_Event_ID : Event_ID_Record
The DIS event id of the fire interaction.

Location : World_Coordinates_Record
The location from which the munition was fired.

Burst_Descriptor : Burst_Descriptor_Record
Interaction's burst descriptor.

Fire_Velocity : Linear_Velocity_Record
The initial velocity of the munition.

Operations:

Create(Instance : access DIS_Burst.Object, Class : Interaction_Class_Handle, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag, Event_Retraction_Handle : Event_Retraction_Handle, Initiator : Object_ID, Munition_ID : Entity_Identifier_Record, Fire_Event_ID : Event_ID_Record, Location : World_Coordinates_Record, Burst_Descriptor : Burst_Descriptor_Record, Fire_Velocity : Linear_Velocity_Record)

Initialize a DIS_Burst object following allocation.

Set_Munition_ID(Instance : access DIS_Burst.Object, New_Value : Entity_Identifier_Record)
Set the DIS Burst's munition id.

Get_Munition_ID(Instance : access DIS_Burst.Object) : Entity_Identifier_Record
Get the DIS_Burst's munition id.

Set_Fire_Event_ID(Instance : access DIS_Burst.Object, New_Value : Event_ID_Record)
Set the DIS_Burst's fire event id.

Get_Fire_Event_ID(Instance : access DIS_Burst.Object) : Event_ID_Record
Get the DIS_Burst's fire event id.

Set_Location(Instance : access DIS_Burst.Object, New_Value : World_Coordinates_Record)
Set the DIS_Burst's location.

Get_Location(Instance : access DIS_Burst.Object) : World_Coordinates_Record
Get the DIS_Burst's location.

Set_Burst_Descriptor(Instance : access DIS_Burst.Object, New_Value : Burst_Descriptor_Record)
Set the DIS_Burst's descriptor.

Get_Burst_Descriptor(Instance : access DIS_Burst.Object) : Burst_Descriptor_Record
Get the DIS_Burst's descriptor.

Set_Fire_Velocity(Instance : access DIS_Burst.Object, New_Value : Linear_Velocity_Record)
Set the DIS_Burst's initial velocity.

Get_Fire_Velocity(Instance : access DIS_Burst.Object) : Linear_Velocity_Record
Get the DIS_Burst's initial velocity.

Class name:

DIS_Collision

Documentation:

This class represents a DIS Collision interaction between two DIS Entity Objects. The DIS Collision is described in paragraph 4.4.5 of the DIS Standard.

It overrides the Set_Parameters, Make_Parameters, Show, and Delete methods of its parent.

Superclasses:

HLA_Interaction

Associations:

<none>

Attributes:

Initiator : Object_ID
The DIS Entity issuing the collision PDU...the collider.

Receiver : Object_ID
The "collidee" DIS Entity.

Collision_Event_ID : Event_ID_Record
The DIS Collision Event ID

Collision_Type : Collision_Types
The Collision type of the collision.

Collision_Velocity : Linear_Velocity_Record
The velocity vector of the issuing entity (the initiator).

Mass : Flt32
The mass of the issuing entity.

Location_Offset : Entity_Coordinate_Record
The location of impact in entity coordinates of the entity with which the issueing entity collided.

Operations:

Set_Initiator(Instance : access DIS_Collision.Object, New_Value : Object_ID)
Set the Collision Interaction's Initator.

Create(Instance : access DIS_Collision.Object, Class : Interaction_Class_Handle, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag,

Event_Retraction_Handle : Event_Retraction_Handle, Initiator : Object_ID, Receiver : Object_ID, Collision_Event_ID : Event_ID_Record, Collision_Type : Collision_Types, Collision_Velocity : Linear_Velocity_Record, Mass : Flt32, Location_Offset : Entity_Coordinate_Record)

Initialize a Collision Interaction following allocation.

Get_Initiator(Instance : access DIS_Collision.Object) : Object_ID

Get the Collision Interaction's initiator.

Set_Receiver(Instance : access DIS_Collision.Object, New_Value : Object_ID)

Set the Collision Interaction's receiver.

Get_Receiver(Instance : access DIS_Collision.Object) : Object_ID

Get the Collision Interaction's receiver.

Set_Collision_Event_ID(Instance : access DIS_Collision.Object, New_Value : Event_ID_Record)

Set the Collision Interaction's Event_ID.

Get_Collision_Event_ID(Instance : access DIS_Collision.Object) : Event_ID_Record

Get the Collision Interaction's event id.

Set_Collision_Type(Instance : access DIS_Collision.Object, New_Value : Collision_Types)

Set the Collision Interaction's collision type.

Get_Collision_Type(Instance : access DIS_Collision.Object) : Collision_Types

Get the Collision Interaction's Collision Type

Set_Collision_Velocity(Instance : access DIS_Collision.Object, New_Value : Linear_Velocity_Record)

Set the Collision Interaction's collision velocity.

Get_Collision_Velocity(Instance : access DIS_Collision.Object) : Linear_Velocity_Record

Get the Collision Interaction's collision velocity.

Set_Mass(Instance : access DIS_Collision.Object, New_Value : Flt32)

Set the Collision Interaction's mass.

Get_Mass(Instance : access DIS_Collision.Object) : Flt32

Get the Collision Interaction's mass.

Set_Location_Offset(Instance : access DIS_Collision.Object, New_Value : Entity_Coordinate_Record)

Set the Collision Interaction's location offset.

Get_Location_Offset(Instance : access DIS_Collision.Object) : Entity_Coordinate_Record

Get the Collision Interaction's location offset.

Class name:

DIS_Detonation

Documentation:

This class represents a DIS Detonation interaction between two DIS Entity Objects.

It overrides the Set_Parameters, Make_Parameters, Show, and Delete methods of its parent.

Superclasses:

DIS_Burst

Associations:

<none>

Attributes:

Location_Offset : Entity_Coordinate_Record
The location of the detonation relative to the receiving entity's origin.

Result : Nat8
The result of the detonation.

Operations:

Create(Instance : access DIS_Detonation.Object, Class : Interaction_Class_Handle, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag, Event_Retraction_Handle : Event_Retraction_Handle, Initiator : Object_ID, Receiver : Object_ID, Munition_ID : Entity_Identifier_Record, Fire_Event_ID : Event_ID_Record, Location : World_Coordinates_Record, Burst_Descriptor : Burst_Descriptor_Record, Fire_Velocity : Linear_Velocity_Record, Location_Offset : Entity_Coordinate_Record, Result : Nat8)
Initialize a DIS_Detonation interaction following allocation.

Set_Location_Offset(Instance : access DIS_Detonation.Object, New_Value : Entity_Coordinate_Record)
Set the DIS_Detonation Interaction's location offset.

Get_Location_Offset(Instance : access DIS_Detonation.Object) : Entity_Coordinate_Record
Get the DIS_Detonation Interaction's location offset.

Set_Result(Instance : access DIS_Detonation.Object, New_Value : Nat8)
Set the DIS_Detonation Interaction's result.

Get_Result(Instance : access DIS_Detonation.Object) : Nat8
Get the DIS_Detonation Interaction's result.

Class name:

DIS_Federate_Ambassador

Documentation:

The DIS Federate ambassador is a Concrete Federate Ambassador with specific knowledge about the DIS Object and Interaction types.
It overrides the Object and Interaction Factory methods of the Concrete_Federate_Ambassador class.

Superclasses:

Concrete_Federate_Ambassador

Associations:

<none>

Attributes:

<none>

Operations:

<none>

Class name:

DIS_Interface

Documentation:

This class represents the abstract DIS interface. It is a protected object (well, sort of... as a minimum, it's children must be implemented via protected objects or tasks to enable protected concurrent access) which encapsulates an abstact interface to Get or Put PDUs from/to the DIS network.

The abstract interface component has two child classes. The Daemon_Based interface uses Bruce Clay's PDU Daemons, as defined by the interface in dsi_user.h (dsi_user.ads). The daemons are themselves two sprocs: one for reading PDUs and one for writing PDUs. The interface provided by dsi_user.h is an AFIT Graphics Lab standard, used by various applications. Future DIS network enhancements, such as ATM, will be built using the same interface, allowing applications such as SimWorx to interface without internal changes.

The Ada_Based interface relies on socket reading and writing tasks to directly read and write to a BSD socket. It was the original SimWorx DIS interface, built before the SimWorx requirement to use dsi_user.h existed.

Either interface may be used at run-time, based upon run-time parameters in the ww_parms.dat file.

Superclasses:

<none>

Associations:

Association Calls
Association Has_DIS_Interface_Task

Attributes:

<none>

Operations:

Create(Instance : access DIS_Interface.Object, Success : Boolean)

This method initializes the interface after the client allocates it via the new operator. Returns True if successful.

Get_PDU(Instance : access DIS_Interface.Object, PDU_Reference : Raw_PDU_Ref, PDU_Available : Boolean)

This method returns a reference to the next available PDU. If no PDU is immediately available, PDU_Available is False. When the client is finished with the PDU, he calls Free_PDU to indicate he's finished.

Free_PDU(Instance : access DIS_Interface.Object)
This method free's the last PDU provided by the interface.

Finalize(Instance : access DIS_Interface.Object)
Clients call this method prior to deallocating the DIS_Interface_Task

Class name:

DIS_Surrogate_RTI_Ambassador

Documentation:

This class is a form of RTI Ambassador which presents an HLA face to HLA simulations, but presents a DIS face to the network. It implements all of the methods of the Abstract RTI Ambassador, and adds a Create method for class-specific initialization.

A client class allocates the DIS_Surrogate_RTI_Ambassador and then calls the Create method. Following the initializing call to Create, the ambassador is ready to accept the method calls inherited from the Abstract_RTI_Ambassador.

Further, following initialization, the DIS_Surrogate_RTI_Ambassador is able to generate calls to the Federate's interface based upon actions within the DIS network.

Superclasses:

Abstract_RTI_Ambassador

Associations:

Association Allocates_Entity_Container
Association Has_Dead_Reckon_Task
Association Has_PDU_Update_Task
Association Calls
Association Has_DIS_Interface_Task

Attributes:

Fed_Amb : Abstract_Federate_Ambassador.Reference = null

This is an Ada classwide type which refers to the RTI_Ambassador's associated Federate_Ambassador. It's necessary to enable dispatching calls to the Federate Ambasssador.

Operations:

Create(Instance : access DIS_Surrogate_RTI_Ambassador.Object, Fed_Amb : Abstract_Federate_Ambassador.Reference)

This method initializes the DIS_Surrogate_RTI_Ambassador following allocation by the new operator. It allocates and initializes the Entity Container, the PDU_Update_Task, the Dead_Reckon_Task, and the DIS_Interface_Task.

Remember, in Ada, as soon as each task is elaborated, it starts to run concurrently.

Class name:

Entity_Container

Documentation:

This class is an instance of the generic Protected_Container. It contains references to DIS_Entities. The container lets the two tasks maintain multiple iterators into it.

Superclasses:

<none>

Associations:

Association Allocates_Entity_Container
Association Shares
Association Also_Shares
Association Has_Entities

Attributes:

<none>

Operations:

<none>

Class name:

Fed_Task

Documentation:

This task realizes the overall scheme behind the Federate class. It loops forever, at some periodic rate, calling a client specified routine over and over until it is killed. Most simulation applications have a "main loop"--The Fed_Task is the SimWorx default main loop. If client simulations don't have a "main loop", they can override the Federate.Initialize method, (simply leaving the Fed_Task reference null) and do some other form of overall simulation control.

The Fed_Task is created by the Federate.Initialize method. Upon initialization, it calls the Federate.Client_Initialize subprogram. Following that, it loops forever (or until the Kill entry is called).

Upon acceptance of the Kill entry, the Fed_Task calls Federate.Client_Finalize to perform client code wrap-up duties.

This task guarantees the concurrency of the Federate's operations. It's implemented as a task to collaborate with the tasks of the RTI Ambassador.

Superclasses:

<none>

Associations:

Association Has Fed_Task

Attributes:

Fed : Federate.Reference

This attribute is an Ada classwide type which refers to the Fed_Task's associated Federate--necessary to allow Fed_Task to call the dispatching methods of the Federate: Client_Initialize, Client_Loop_Body, etc.

Operations:

Kill( )

The Kill entry causes the task to exit after it calls the Client_Finalize method.

Elaborate_With_Discriminant(Federate : Federate.Reference)

This method represents elaborating the Fed_Task with a reference to it's associated Federation object.

Class name:

Federate

Documentation:

This abstract class represents the client programmer's simulation application. The client programmer will subclass this class to handle the chores of her simulation. For example, a flight simulator subclass would model flight.

The Federate class enables the objects it has to interact with each other.

When it receives an interaction, it explicitly invokes the methods of the objects it owns to handle the interaction.

The Federate Class allocates and initializes the RTI, the Object container, the Interaction_Queue, the Fed_Task, and some version of the Concrete Federate Ambassador (specified by the client-overridden Federate_Ambassador_Factory_Method).

Superclasses:

<none>

Associations:

Association Has Fed_Task

Association Has_Ambassador
Association Allocates_Object_Container
Association Allocates_Interaction_Queue

Attributes:

<none>

Operations:

Initialize(Instance : Federate.Object)

This subprogram initializes the Federate object following allocation by the new operator. It is automatically called since the Federate is a child class of Ada.Finalization.Limited_Controlled.

Finalize(Instance : Federate.Object)

This method overrides the Ada Controlled type Finalize method. It "cleans up" the Federate object when it is deallocated.

Client_Initialize(Instance : access Federate.Object)

At task initialization, the Sim_Task calls this client supplied initialization subprogram to perform initialization duties for the client.

This subprogram is meant to be overridden.

Client_Loop_Body(Instance : access Federate.Object)

For each iteration of it's main loop, Sim_Task calls this client supplied subprogram to do a "frame's worth" of simulation work. Note that this implies the SimWorx framework has a main loop which repeats some client-specified action over and over. If you don't like that, then over-ride the Initialize subprogram to define a different behavior for a simulation!

Client_Finalize(Instance : access Federate.Object)

When the Sim_Task is terminated, it calls this subprogram (overridden by the client programmer, of course) to perform clean-up duties within the client simulation.

Federate_Ambassador_Factory_Method(Instance : Federate.Object) : Abstract_Federate_Ambassador.Reference

This method is called by the Initialize method to determine the actual instance of Federate Ambassador the Federate will be using. Client programmers override this method in their child Federate class, specifying what kind of Federate Ambassador to use. It is an example of the Factory Method pattern from the book Design Patterns: Elements of Reusable Object-Oriented Software. ISBN 0-201-63361-2

RTI_Ambassador_Factory_Method(Instance : Federate.Object)

This method is called by the Initialize method to determine the actual instance of RTI Ambassador the Federate will be using. Client programmers override this method in their child Federate class, specifying what kind of RTI Ambassador to use.

Initialize_Federate_Ambassador(Instance : Federate.Object)

Called by the Initialize method to initialize the Federate_Ambassador created by the factory method. Initialization can't be embedded in the factory method since both Ambassadors must be allocated before either is initialized. Meant to be overridden by client programmers.

Initialize_RTI_Ambassador(Instance : Federate.Object)

Called by Initialize method to initialize the RTI Ambassador. Same comment about initilization applies to the RTI Ambassador as noted above in

Initialize_Federate_Ambassador. Meant to be overridden by client programmers.

Class name:

HLA_Interaction

Documentation:

This class represents an abstract HLA Interaction. Framework client programmers will subclass this class for each interaction class specified by their simulation's SOM.

Interactions are a form of the Command Behavioral Pattern. See pate 233 of "Design Patterns: Elements of Reusable Object-Oriented Software", ISBN 0-201-63361-2. Commands are used to "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."

However, unlike the Design Pattern's Command, an interaction is not responsible for executing itself--that would require it to have knowledge of different HLA_Objects (aaaa, the circular with thing again!!), instead, we'll make the HLA_Objects have knowledge of the different interactions they can be subjected to. HLA_Objects are then responsible for converting an explicit HLA_Interaction into the implicit effects (method calls) the interaction has on them.

Superclasses:

<none>

Associations:

Association Has_Interactions

Attributes:

Class_Handle : Interaction_Class_Handle = 0

The interaction class this interaction instance belongs to. The handle is unique for each kind of HLA_Interaction.

Fed_Time : Federation_Time

The time the interaction is effective.

User_Tag : User_Supplied_Tag

A user supplied tag--implemented as a pointer to string in C.

Event_Retraction_Handle : Event_Retraction_Handle

An event retraction handle. This is probably for use in "undoing" or "recalling" an event during event driven sims. Not currently used in SimWorx since DIS and SimWorx support real-time sims.

Initiator : Object_ID = Null_Object_ID

The initiating HLA object of the HLA Interaction. The value for this attribute is updated via the Set_Parameters method since the HLA Interface Spec didn't specifiy this as an explicit method parameter.

Receiver : Object_ID = Null_Object_ID

The receiving HLA object of the HLA Interaction (not that every HLA_Interaction will necessarily have a specific receiver). The value for this attribute is updated via the Set_Parameters method since the HLA Interface Spec didn't specifiy this as an explicit method parameter.

Operations:

Create(Instance : HLA_Interaction.Reference, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag, Event_Retraction_Handle : Event_Retraction_Handle)

Initialize an interaction following allocation.

Set_Parameters(Instance : access HLA_Interaction.Object, Parameters : Parameter_Handle_Value_Pair_Set)

Set the parameters of the Interaction by copying the parameters from the Parameter_Handle_Value_Pair_Set.

Make_Parameters(Instance : access HLA_Interaction.Object, Parameters : Parameter_Handle_Value_Pair_Set)

Build a Parameter_Handle_Value_Pair_Set from the parameters (attributes) of this interaction. Allows a client class to use Set_Attribute calls to initialize an HLA Interaction, then call this method to create the pair set for transmission to the RTI (or Federate).

Get_Class_Handle(Instance : access HLA_Interaction.Object) : Interaction_Class_Handle

This is a dispatching method meant to be overridden by each subclass. It returns the Interaction_Class_Handle for this kind of interaction.

Note that there is no corresponding Set_Class_Handle method since each kind of interaction has a constant Class_Handle, set upon instantiation.

Delete(Instance : access HLA_Interaction.Object)

This method safely deallocates the storage for this interaction. It is meant to be overriden by every child interaction class defined by client programmers.

Set_Time(Instance : access HLA_Interaction.Object, New_Value : Federation_Time)

Set the Interaction's Federation Time attribute. I believe this is the time the Interaction is effective.

Get_Time(Instance : access HLA_Interaction.Object) : Federation_Time

Get the HLA Interaction's time.

Set_Initiator(Instance : access HLA_Interaction.Object, New_Value : Object_ID)

Set the HLA Interaction's initiator.

Get_Initiator(Instance : access HLA_Interaction.Object) : Object_ID

Get the HLA Interaction's initiator.

Set_Receiver(Instance : access HLA_Interaction.Object, New_Value : Object_ID)

Set the Interaction's receiver. Note that HLA_Interactions do not necessarily have a receiver, but many child classes of the HLA_Interaction do. This method is meant to be overridden.

Get_Receiver(Instance : access HLA_Interaction.Object) : Object_ID

This method returns the Interaction's receiver. If the interaction does not have a specific receiver, then the method returns Null_Object_ID. Note that HLA_Interactions do not necessarily have a receiver, but many child classes of the HLA_Interaction do. This method is meant to be overridden.

Show(Instance : access HLA_Interaction.Object)

Show the interaction (print it, although printing could be overridden for graphical applications).

Class name:

HLA_Object

Documentation:

This class represents an HLA object. Objects interact with each other to carry out the duties of the simulation. Objects may be instantiated and controlled locally by this simulation and also instantiated on the behalf of some other federate.

Object interactions may be implicit via method calls to other objects, or explicit via an Interaction invoked by the simulation. The method Handle_Interaction is meant to be overridden for each kind of HLA_Object in order to handle the different kinds of interactions each kind of Object can be subjected to.

Superclasses:

<none>

Associations:

Association Has_Objects

Attributes:

Class : Object_Class_Handle = 0

The object class this object belongs to. Each child class has a unique constant value for this attribute.

ID : Object_ID = Null_Object_ID

The ID of this object. It is unique within the federation execution.

Time_Stamp : Federation_Time

Time which something was last done to the HLA_Object.

Operations:

Create(Instance : access HLA_Object.Object, Object_ID : Object_ID, Object_Class : Object_Class_Handle)

Initialize an Object following allocation.

Delete(Instance : access HLA_Object.Object)

This method safely de-allocates the storage for this object. It is meant to be overridden by every child class defined by client programmers.

Attribute_Update(Instance : access HLA_Object.Object, Time_Stamp : Federation_Time, Attribute_Value_List : Attribute_Handle_Value_Pair_Set)

This method copies the attributes from the Attribute_Handle_Value_Pair_Set to the Object's corresponding attributes. It is meant to be overridden by every child class defined by client programmers.

Handle_Interaction(Instance : access HLA_Object.Object, Interaction : HLA_Interaction.Reference)

This method handles an interaction. That is, it takes an explicit HLA_Interaction and performs the necessary actions to react to it (usually by calling other methods, depending upon the kind of interaction).

Lock(Instance : access HLA_Object.Object)

Lock the Object for exclusive access. All clients must Lock the HLA_Object prior to executing a method which could change the HLA_Object's state.
The protocol is...

```
 Lock
 Do_Some_Method
 Unlock
```

NOTE: Children of the HLA_Object do not call the Lock method since that will lead to deadlock.

Unlock(Instance : access HLA_Object.Object)

The Unlock operation. See Lock method for more details.

Set_ID(Instance : HLA_Object.Reference, Object_ID : Object_ID)

Set the Object's Object_ID.

Get_ID(Instance : HLA_Object.Reference) : Object_ID

Return the Object's ID.

Get_Class(Instance : access HLA_Object.Object) : Object_Class_Handle

Return the Object's Class. Note that there is no corresponding Set_Class method since each subclass will have its own constant value.

Show(Instance : access HLA_Object.Object)

Show the Object. Meant to be overridden for each child class. For example, a simple data logger may simply print the attribute values. A graphical program could draw the object.

Class name:

Interaction Queue

Documentation:

This class is an instance of the generic Bounded Buffer. It buffers interactions received by the Concrete_Federate_Ambassador so they can be used by the Federate when he gets around to them.

Superclasses:

<none>

Associations:

Association Also_Shares_Interaction_Queue
Association Has_Interactions
Association Allocates_Interaction_Queue

Attributes:

<none>

Operations:

<none>

Class name:

Objects

Documentation:

This class is an instance of the generic Protected_Container. It manages concurrent access to the HLA_Objects it contains (actually, it contains References to Objects) from the Fed_Task and the tasks of the RTI_Ambassador (via the Concrete Federate Ambassador).

Since it is a generic, it has all of the operations and type definitions of the Protected_Container class.

Superclasses:

<none>

Associations:

Association Shares_Object_Container
Association Has_Objects
Association Allocates_Object_Container

Attributes:

<none>

Operations:

<none>

Class name:

PDU Buffer

Documentation:

The PDU Buffer is an instantiation of the generic Bounded_Buffer. It's meant to handle the slack between consuming PDUs (done by the Ada_Based interface) and producint PDUs (done by the Socket_Reader_Task).

Note that this means the Ada Based interface, if it can't keep up with PDUs on the network, will "lose" the most recent PDUs on the network since the Socket_Reader task will not be able to put them in the "full" bounded buffer.

This is in contrast to the Daemon based implementation which will overwrite waiting PDUs with newer ones as they are received from the network.

Superclasses:

<none>

Associations:

Association Allocates_PDU_Buffer
Association Shares_PDU_Buffer

Attributes:

<none>

Operations:

<none>

Class name:

PDU_Handler_Task

Documentation:

This task object repeatedly reads PDUs from the DIS_Interface, performing different actions depending upon the type of PDU read.

Entity State PDU: If this is for an unrecognized entity, the task creates a new DIS Entity and puts it in the Entity Container, and updates the entity's state based upon data in the PDU. If this is for a recognized entity, the task simply updates the entity's state from the PDU.

Fire PDU: Propagates a DIS Fire Interaction to the Federate with data from the PDU.

Detonation PDU: Propagates a DIS Detonation Interaction to the Federate with data from the PDU.

Collision PDU: Propagates a DIS Collision Interaction to the Federate with data from the PDU.

The exception Program_Error may be raised by the protected container or by a protected DIS Entity if the object is deleted while this task is waiting to use it. Thus, Program_Errors are simply handled in the task loop iteration by doing nothing.

Superclasses:

<none>

Associations:

Association Has_PDU_Update_Task
Association Also_Shares
Association Calls

Attributes:

<none>

Operations:

Elaborate_With_Discriminants(Fed_Ambassador : Abstract_Federate_Ambassador.Reference, Entity_Container_Handle : Entity_Container.Reference, DIS_Interface : DIS_Interface.Reference)

This method represents elaborating the task with the above discriminants.

The Federate Ambassador Reference is necessary to allow the task to call the methods of the Federate Ambassador.

The DIS_Interface reference is the necessary handle to the DIS Interface for making method calls.

Handle_PDU(Entity_State_Ptr : Entity_State_PDU_Ref)
This method handles each PDU received. See the main documentation section above for details.

Project_DIS_Collision(Collision_Ptr : Collision_PDU_Ref)
This method sets up for, and then calls the Federate Receive_Interaction method for the DIS Collision interaction. It builds the Parameter_Value_Handle_Pair_Set (from values in the Collision PDU).

Project_DIS_Fire(Fire_Ptr : Fire_PDU_Ref)
This method sets up for, and then calls the Federate Receive_Interaction method for the DIS Fire interaction. It builds the Parameter_Value_Handle_Pair_Set (from values in the Fire PDU).

Project_DIS_Detonation(Detonation_Ptr : Detonation_PDU_Ref)
This method sets up for, and then calls the Federate Receive_Interaction method for the DIS Detonation interaction. It builds the Parameter_Value_Handle_Pair_Set (from values in the Detonation PDU).

Find_Entity(Entity_ID : Entity_Identifier_Record) : DIS_Entity.Reference
This method scans through the Entity_Container, looking for an entity with a matching DIS Entity ID. If found, it returns a reference to the Entity. If not found, it returns null.

Kill( )
Kill the PDU_Handler_Task.

Class name:

Protected_Container

Documentation:

This is the generic Protected_Container Class. It is implemented as a protected type.

This container allows several different iterations to be performed concurrently via "Iterators". An iterator is an external cursor, or marker, marking a client's current position within the container.

For more information on iterators, see the Behavioral Design Pattern "Iterator" on page 257 of "Design Patterns: Elements of Reusable Object-Oriented Software", ISBN0-201-63361-2

Note that the only tricky bit here is that there could be a dead-lock condition when two separate clients try to remove an item at the same time (that their iterator's are both pointed at). This shouldn't be a problem in the SimWorx framework since only one client will have the responsiblity to remove items from the container. (Thomas Kofler provides a solution to this problem in the Paper "Robust Iterators in ET++", in Structured Programming, 14:62-85, March 1993, however his approach requires use of an atomic Visit method which removes much of the flexiblity of the external iterator).

Superclasses:

<none>

Associations:

<none>

Attributes:

Iterator_Table : Iterator_Table_Type
This table stores the current values of the iterators registered with the container.

Operations:

Add(Item : Item_Type)
Add an Item to the Container

Size( ) : Natural
Returns the current size of the container.

Empty( ) : Boolean

Returns True if the container is empty.

Make_Iterator( ) : Iterator_Type

This method returns an iterator (or external cursor) for use in iterating through the items in the container.

Destroy_Iterator(Iterator : Iterator_Type)

Destroy the iterator. Following this call, the iterator is no longer valid.

Remove(Iterator : Iterator_Type, Success : Boolean)

Remove the item pointed to by the iterator from the container. Returns Success = False if the item couldn't be removed (happens when some other iterator is currently pointing to that item--try again later).

Note: Waiting on Sucess could lead to dead-lock if more than one client is responsible for removing items. (But no one would ever create such an irresponsible, dangerous design, would they!?.)

Current_Item(Iterator : Iterator_Type) : Item_Type

Returns a copy of the current item in the container.

Finished(Iterator : Iterator_Type) : Boolean

Returns True if iterator has exhausted the list (in either the forward or reverse direction.)

Reset(Iterator : Iterator_Type)

Reset the iterator to the beginning of the container.

Next(Iterator : Iterator_Type)

Advances the Iterator to the next item in the container. When you're at the end of the container, and you call Next, the iterator's position will no longer be valid, and Finished will return True. (So test Finished before you do anything with Current_Item).

Previous(Iterator : Iterator_Type)

Move the iterator to the previous item in the container. When you're at the beginning of the container, and you call Previous, the iterator's position will no longer be valid, and Finished will return True. (So test Finished before you do anything with current item).

Class name:

RTI Services

Documentation:

This class represents several RTI services (see attributes below). These are encapsulated here mainly because of the 30 class limit to the demonstration version of the Rational Rose tool used to capture the design. Otherwise, they'd be represented as separate classes.

Superclasses:

<none>

Associations:

<none>

Attributes:

Clock : Protected_Clock
This clock represents the ada protected object which controls concurrent access to the Clock.

ID_Server : Protected_ID_Server
This is the Ada protected object which controls concurrent access to the ID Server

Operations:

Clock( ) : Federation_Time
Returns the current Federation_Time in seconds. Federation Time may be relative, or absolute (wall clock time). If the time is relative, the value returned is relative to the initialization time of the RTI. If the time is absolute, the value returned is the current unix time (seconds since Jan 1st, 1970). DIS exercises may use either form of time.

Clients call Initialize_Clock_Relative or Initialize_Clock_Absolute, depending upon the form of time they want returned.

Initialize_Clock_Absolute( )
This method initializes the Federation Time Clock to return absolute time (Unix time in seconds since Jan 1st, 1970).

All subsequent calls to the Clock method will return absolute time.

Initialize_Clock_Relative( )

This method initializes the Federation Time clock by establishing and RTI epoch time which is considered "Time Zero" for this simulation run of the RTI.

All subsequent calls to the Clock method will return a time relative to the simulation epoch time.

Get_Corresponding_ID(DIS_ID : Entity_Identifier_Record) : Object_ID

This method returns the Object_ID corresponding to the DIS Entity ID of a DIS Entity. It returns Null_Object_ID if the DIS ID isn't recognized.

Request_ID(The_Count : Object_ID_Count, First_ID : access Object_ID, Last_ID : access Object_ID)

Returns HLA Object Ids. IDs are paired with DIS Entity IDs during a simulation. If for some reason, an external DIS Entity becomes stale and then reappears, the ID server will know to use the same ID. This method returns IDs from the pool of IDs reserved for Entities controlled by this federate. That is, corresponding DIS IDs will be fabricated for these entities by the RTI based upon the HLA Object ID.

Get_Corresponding_ID(The_ID : Object_ID) : Entity_Identifier_Record

This method returns the DIS Entity ID corresponding to the given Object ID.

If the Object ID is from the pool of IDs reserved for this federate's objects, then the Entity ID is fabricated from the Object_ID.

If the Object ID is from the pool of external IDs, then the Entity ID is looked up in the table of ID pairs.

Possible Exceptions:

Invalid_Object_ID. ID Hasn't been assigned yet from pool, or there is no corresponding DIS Entity ID in the pair array.

Request_ID(DIS_ID : Entity_Identifer_Record) : Object_ID

Returns an Object ID from the pool for this new DIS ID.

Possible Exceptions:

Invalid_Object_ID. Raised if DIS ID is already assigned to an Object ID.

ID_Supply_Exhausted.

Class name:

Socket Reader Task

Documentation:

This task continually reads DIS PDUs from a Unix BSD socket. It puts the PDUs into the PDU_Buffer which it shares with the Ada_Based interface.

Superclasses:

<none>

Associations:

Association Has_Socket_Reader_Task
Association Shares_PDU_Buffer

Attributes:

<none>

Operations:

Elaborate_With_Discriminants(Port_Number : Positive = 4000, PDU_Buffer_Ref : PDU_Buffer.Reference, Socket_Handle : Integer = 0, Socket_Address : sockaddr_ptr)

Elaborate the task with discriminants.
sockaddr_ptr is an access type which refers to sockaddr_in records (a BSD structure for manipulating sockets).

Kill( )

Allows parent task to kill this task.

Index

Thesis Appendix Note: Since this stand-alone design document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute C-5 for 5, C-11 for 11, etc.

Appendix D: SimWorx Iteration Two Analysis Model

SimWorx: Analysis Model

Iteration Two Analysis Model

Software Architectures in Ada 95
AFIT Software Engineering Group
Air Force Institute of Technology
Wright-Patterson AFB, OH 45434

This Page Intentionally Left Blank

Table Of Contents

Thesis Appendix Note: Since this stand-alone analysis document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute D-5 for 5, D-11 for 11, etc.

List of Figures

Thesis Appendix Note: Since this stand-alone analysis document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute D-5 for 5, D-11 for 11, etc.

* Denotes New/Updated Use Case for Iteration Two.

Introduction

This document reflects the second iteration of software analysis for the SimWorx application framework for distributed simulation.

The SimWorx framework enables application programmers (a.k.a. client programmers) to quickly and easily generate distributed simulation applications conforming to the DoD High Level Architecture (HLA) for Modeling and Simulation. In addition, the SimWorx framework provides a Surrogate HLA Run-Time Infrastructure (RTI) which performs Distributed Interactive Simulation (DIS) version 2.0.4 network input and output.

Initial versions of the SimWorx framework do not support either the full HLA nor full DIS standards. Initial versions supply the minimal functionality necessary for simple objects to interact for typical DIS interactions (entity movement and collision, weapons fire and detonation).

This first iteration of the SimWorx framework concentrates on receiving input from the DIS network. The application framework is divided into two main categories: Federate components with the behavior of an HLA Federate, and RTI components with the behavior of the HLA RTI.

In the first iterationof the SimWorx framework, DIS Protocol Data Units (PDUs) are received from the network by the Surrogate RTI Ambassador[1]. DIS Entities are modeled and their states are dead-reckoned between network updates. DIS Entity information is forwarded by the RTI Ambassador to the Federate Ambassador via the standard HLA ambassador interface, so the federate components may be reused with various sorts of RTIs--see the Federate Ambassador class for details.

The second iteration of the SimWorx framework adds the ability to send output to the DIS network. HLA_Objects of the Federation interact via interactions. Actions which affect HLA_Objects which aren't owned by the SimWorx Federate are sent to the RTI as HLA_Interactions. State changes of the HLA_Objects are sent to the RTI as attribute updates. The DIS Surrogate RTI Ambassador interprets the updates and interactions, produces appropriate DIS PDUs, and broadcasts the PDUs onto the network.

[1] The term *Ambassador* is an HLA term for a kind of interface software component. Just as a diplomatic ambassador represents a foreign country, an *Ambassador* presents a local application programming interface for a system component which may actually be physically located somewhere else (another computer, another state, etc.). Thus, the client can make software procedure calls to the Ambassador, rather than relying on some message passing interface, since the Ambassador "abstracts-out" the details of the actual interaction.

Use Cases

The following Use Cases describe the first iteration functionality of the SimWorx Application Framework. The Use Cases are described with Message Trace Diagrams. Each numbered item is a message from a sending class to a receiving class. The messages are ordered by time: later numbered messages are later in time than earlier numbered messages. Use Cases 1 and 10-13 are new for iteration two.

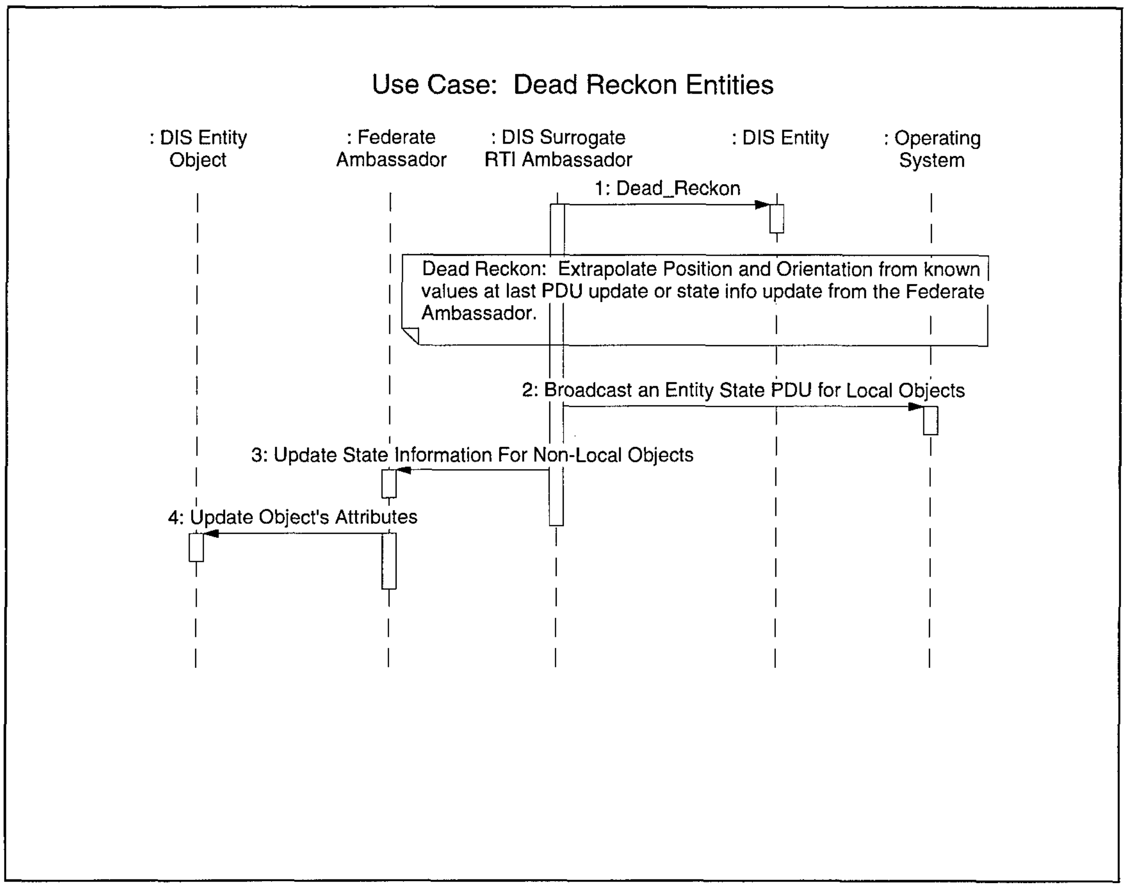

Use Case: Dead Reckon Entities
: DIS Entity Object
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: DIS Entity
: Operating System
1: Dead_Reckon
Dead Reckon: Extrapolate Position and Orientation from known values at last PDU update or state info update from the Federate Ambassador.
2: Broadcast an Entity State PDU for Local Objects
3: Update State Information For Non-Local Objects
4: Update Object's Attributes

Figure 1. Use Case: Dead Reckon Entities

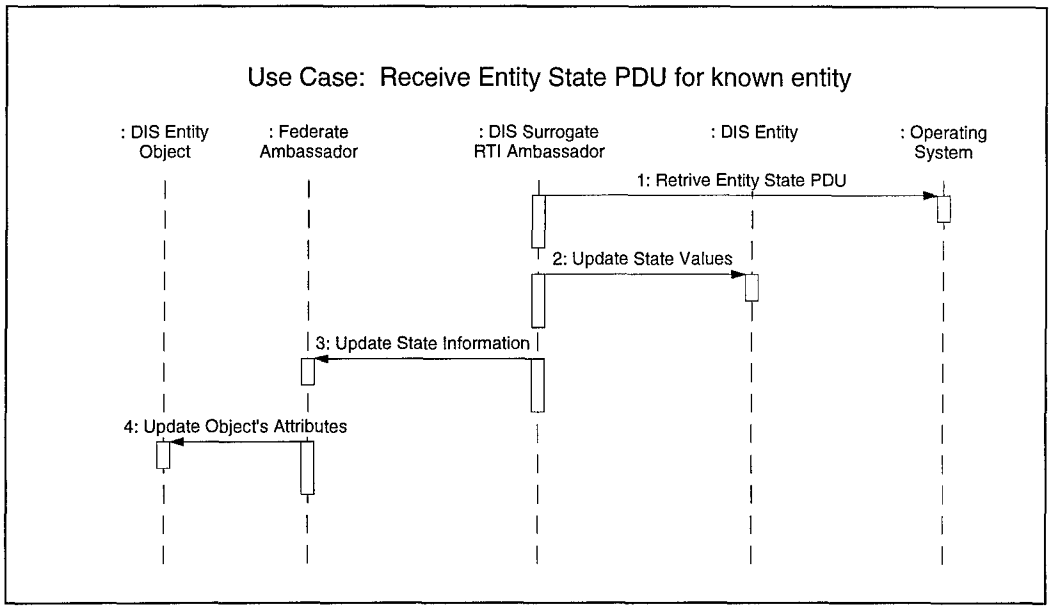

Use Case: Receive Entity State PDU for known entity
: DIS Entity Object
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: DIS Entity
: Operating System
1: Retrive Entity State PDU
2: Update State Values
3: Update State Information
4: Update Object's Attributes

Figure 2. Use Case: Receive Entity State PDU for Known Entity

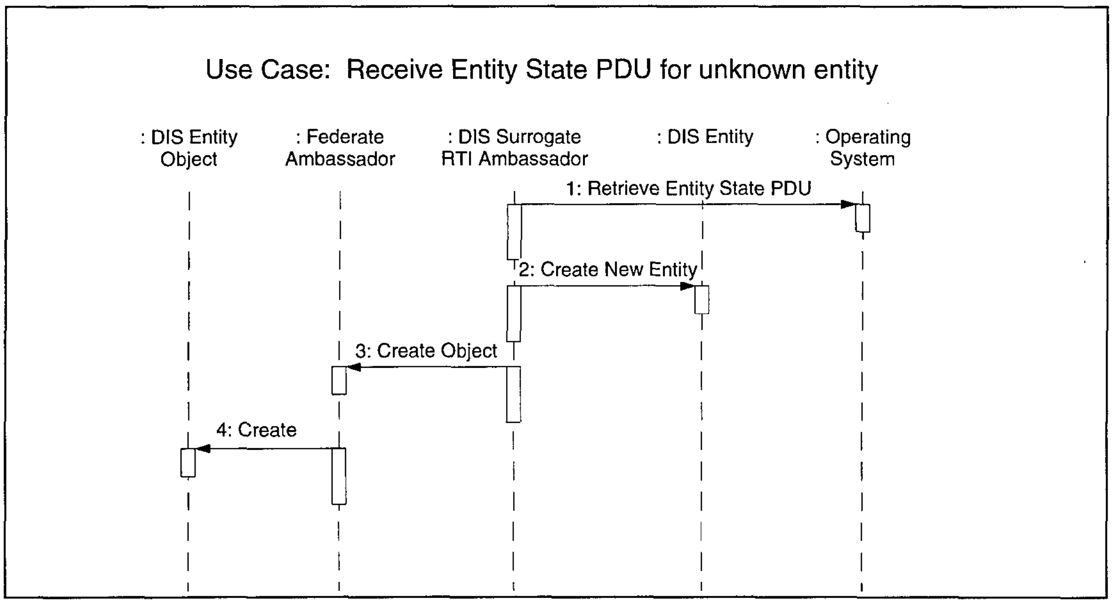


Figure 3. Receive Entity State PDU for Unknown Entity

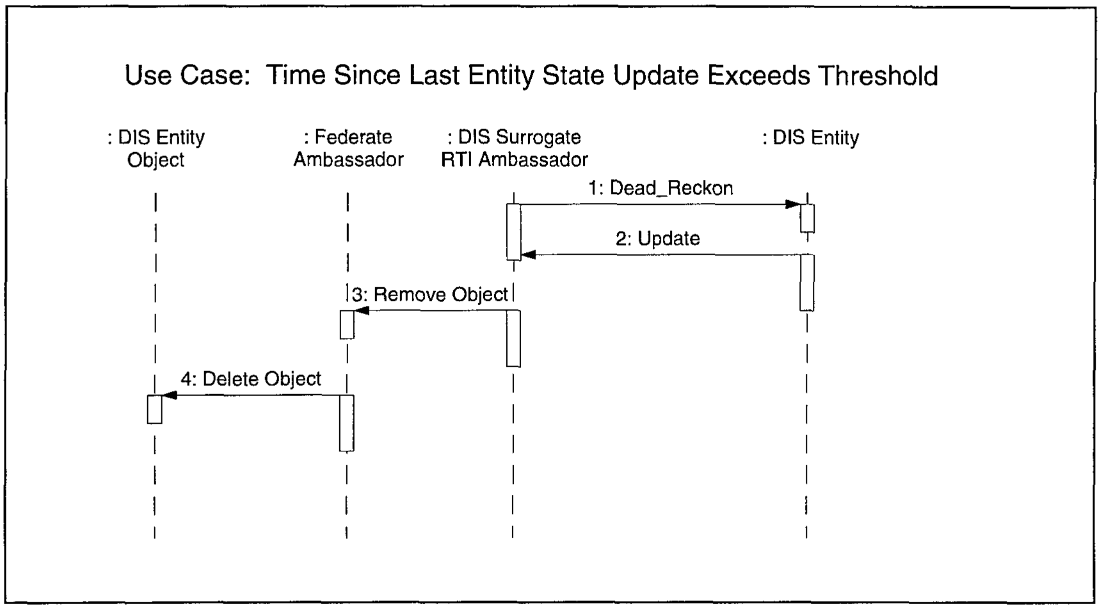


Figure 4. Use Case: Time Since Last Entity State Update Exceeds Threshold

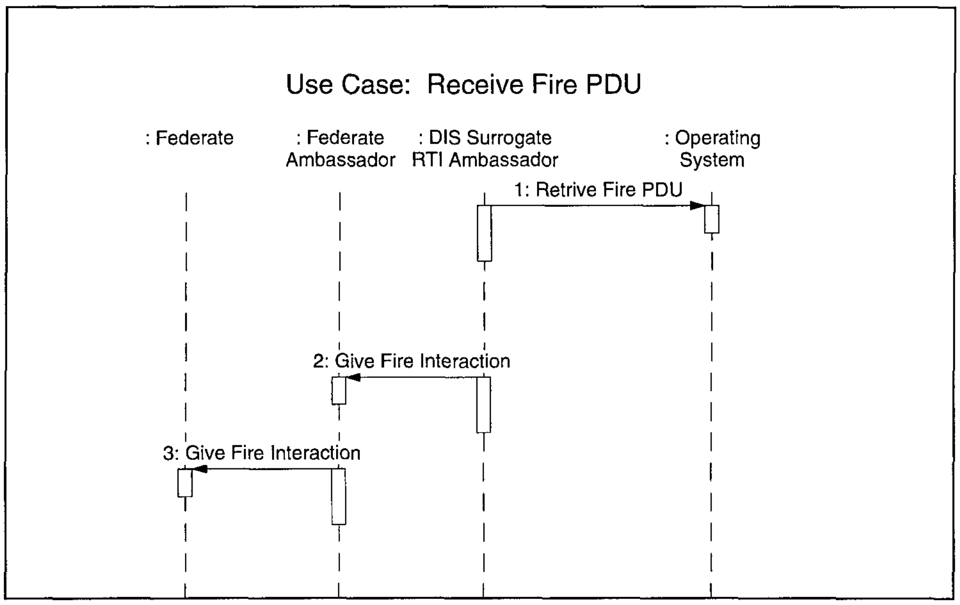


Figure 5. Use Case: Receive Fire PDU

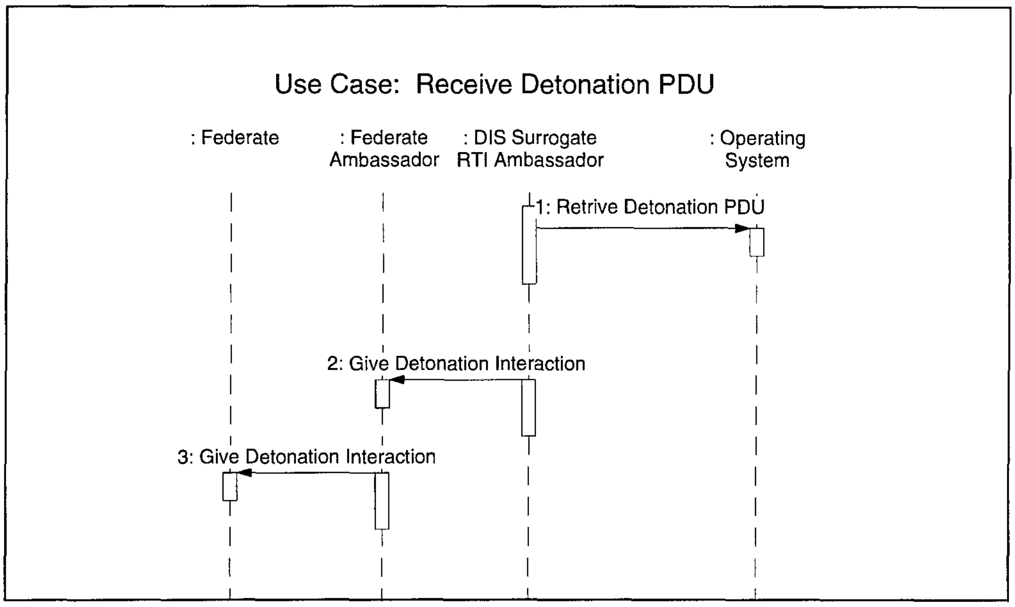

Use Case: Receive Detonation PDU
: Federate
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: Operating System
1: Retrive Detonation PDU
2: Give Detonation Interaction
3: Give Detonation Interaction

Figure 6. Use Case: Receive Detonation PDU

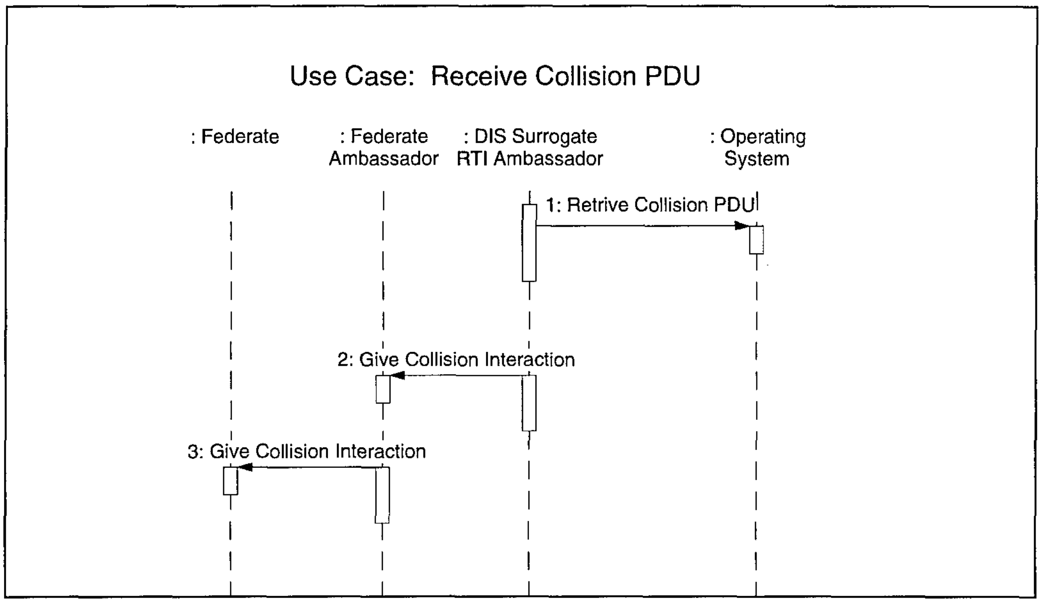
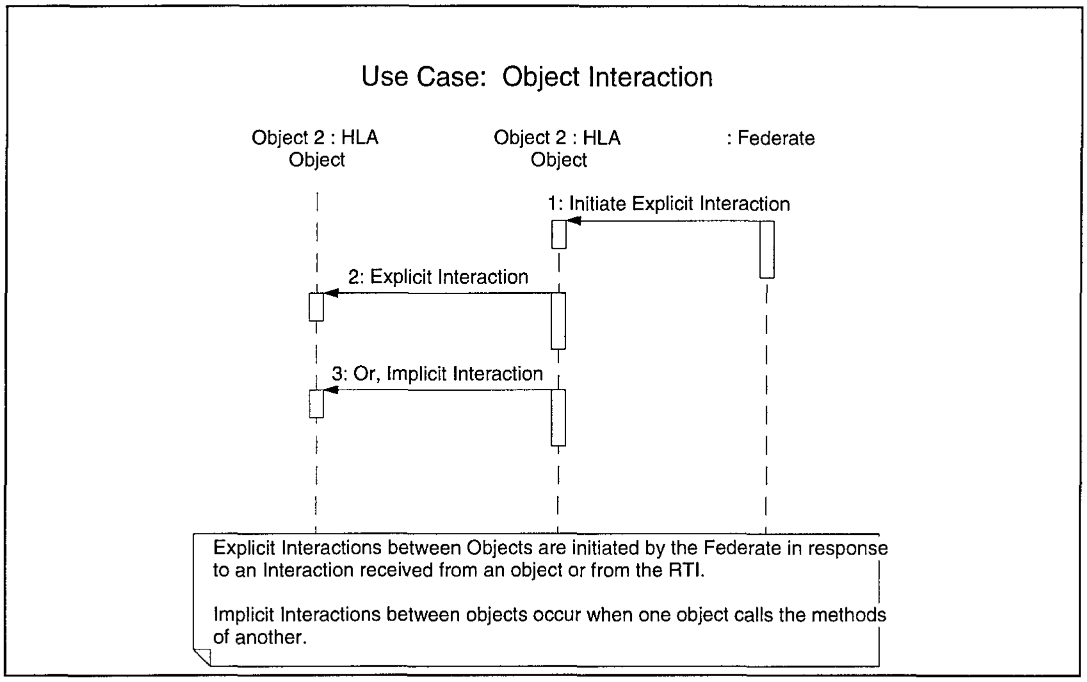

Use Case: Receive Collision PDU
: Federate
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: Operating System
1: Retrive Collision PDU
2: Give Collision Interaction
3: Give Collision Interaction

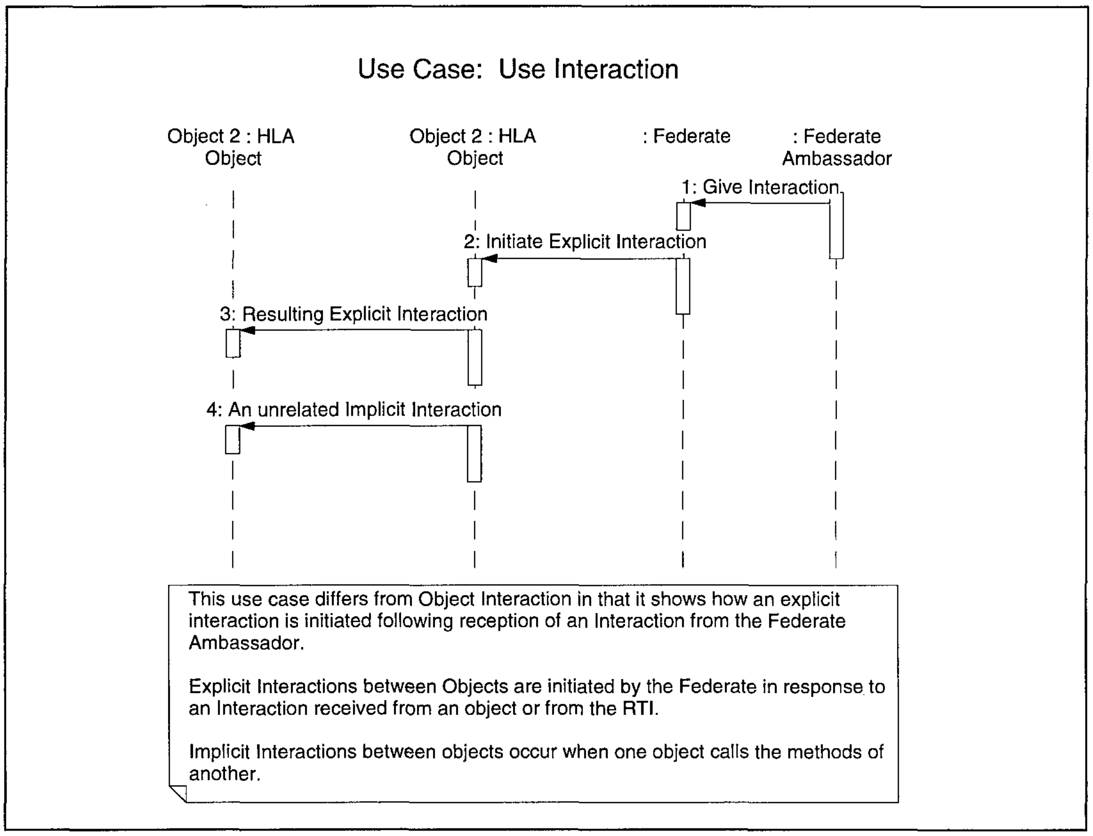
Figure 7. Use Case: Receive Collision PDU


Use Case: Object Interaction
Object 2 : HLA Object
Object 2 : HLA Object
: Federate
1: Initiate Explicit Interaction
2: Explicit Interaction
3: Or, Implicit Interaction

Explicit Interactions between Objects are initiated by the Federate in response to an Interaction received from an object or from the RTI.

Implicit Interactions between objects occur when one object calls the methods of another.

Figure 8. Use Case: Object Interaction


Use Case: Use Interaction
Object 2 : HLA Object
Object 2 : HLA Object
: Federate
: Federate Ambassador
1: Give Interaction
2: Initiate Explicit Interaction
3: Resulting Explicit Interaction
4: An unrelated Implicit Interaction

This use case differs from Object Interaction in that it shows how an explicit interaction is initiated following reception of an Interaction from the Federate Ambassador.

Explicit Interactions between Objects are initiated by the Federate in response to an Interaction received from an object or from the RTI.

Implicit Interactions between objects occur when one object calls the methods of another.

Figure 9. Use Case Use Interaction

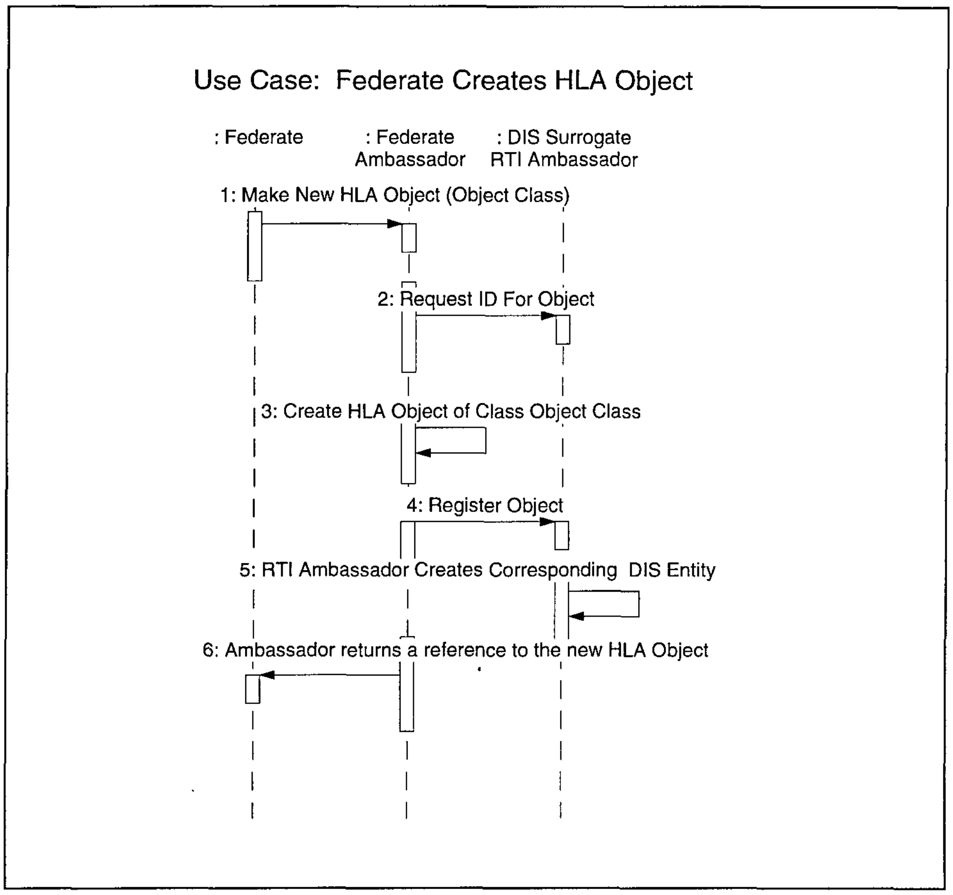

Use Case: Federate Creates HLA Object
: Federate
: Federate Ambassador
: DIS Surrogate RTI Ambassador
1: Make New HLA Object (Object Class)
2: Request ID For Object
3: Create HLA Object of Class Object Class
4: Register Object
5: RTI Ambassador Creates Corresponding DIS Entity
6: Ambassador returns a reference to the new HLA Object

Figure 10. Use Case: Federate Creates HLA_Object

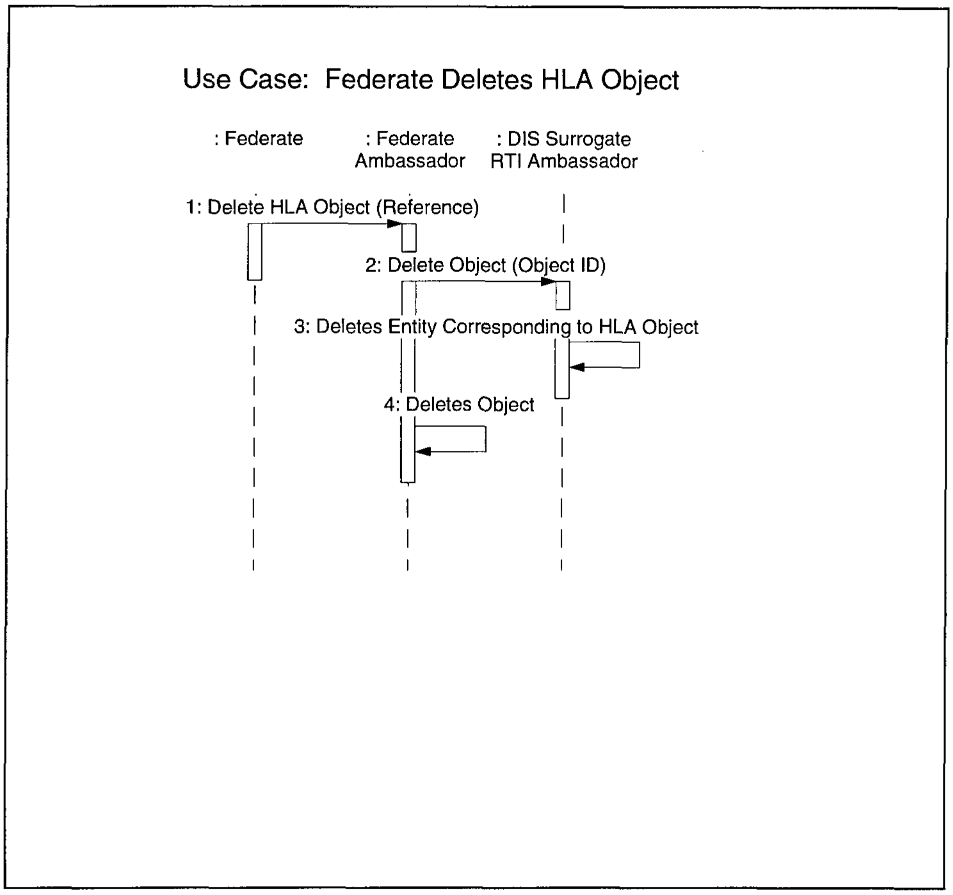

Use Case: Federate Deletes HLA Object
: Federate
: Federate Ambassador
: DIS Surrogate RTI Ambassador
1: Delete HLA Object (Reference)
2: Delete Object (Object ID)
3: Deletes Entity Corresponding to HLA Object
4: Deletes Object

Figure 11. Use Case: Federate Deletes HLA_Object

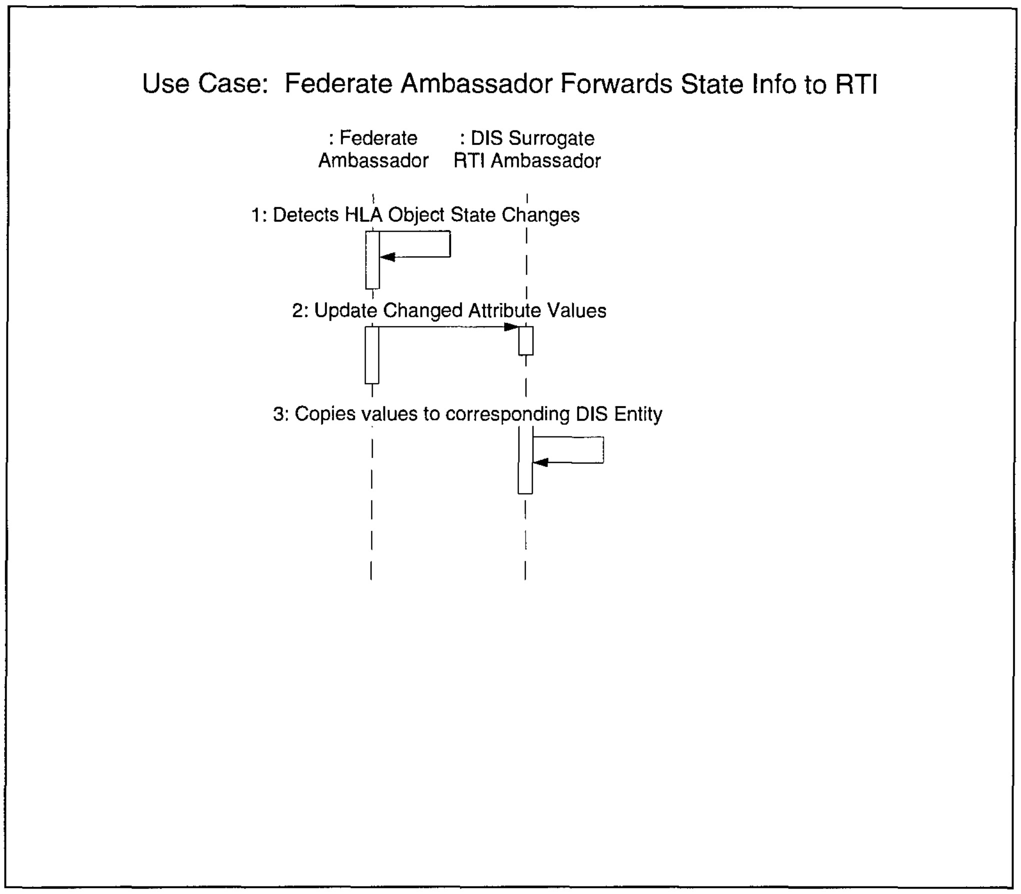

Use Case: Federate Ambassador Forwards State Info to RTI
: Federate Ambassador
: DIS Surrogate RTI Ambassador
1: Detects HLA Object State Changes
2: Update Changed Attribute Values
3: Copies values to corresponding DIS Entity

Figure 12. Use Case: Federate Ambassador Forwards State Info to RTI.

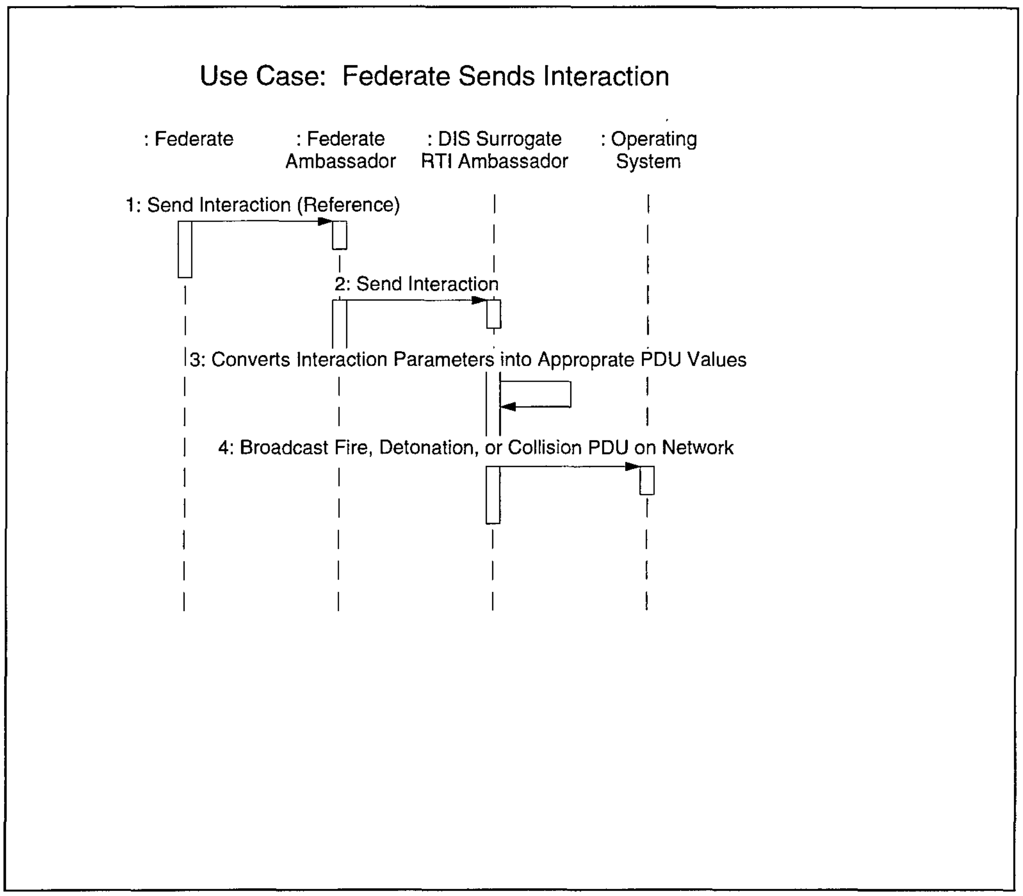

Use Case: Federate Sends Interaction
: Federate
: Federate Ambassador
: DIS Surrogate RTI Ambassador
: Operating System
1: Send Interaction (Reference)
2: Send Interaction
3: Converts Interaction Parameters into Approprate PDU Values
4: Broadcast Fire, Detonation, or Collision PDU on Network

Figure 13. Use Case: Federate Sends Interaction

Analysis Architecture

The following diagram shows the primary iteration analysis architecture in terms of a Rumbaugh Object Model Diagram.

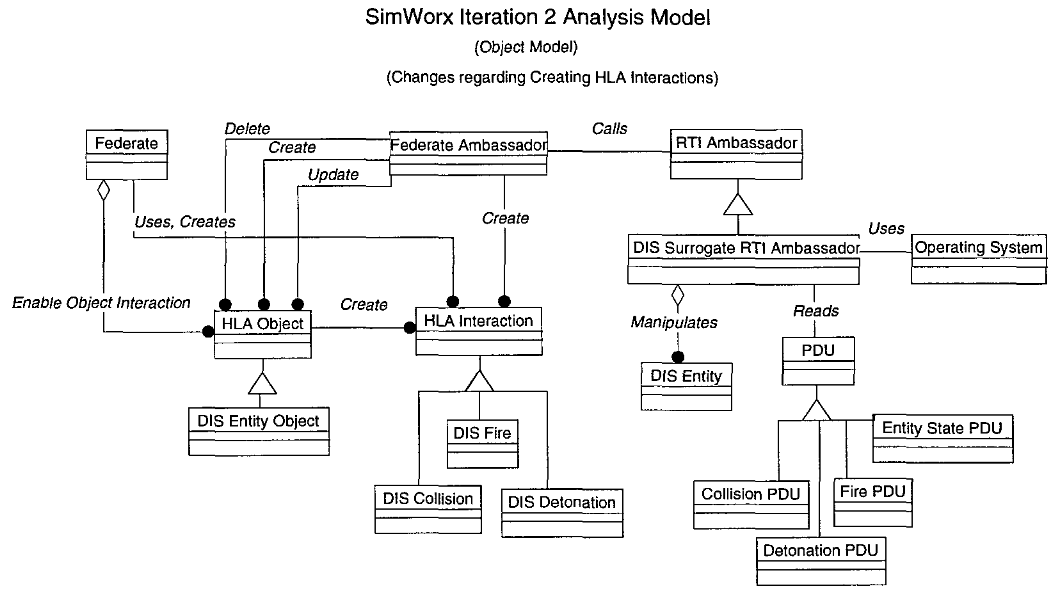


Figure 14. SimWorx Iteration 2 Analysis Model

Classes

Class name:

Collision PDU

Documentation:

This class represents the Collision PDU as defined by DIS standard 2.0.4 paragraph 5.4.3.2

Superclasses:

PDU

Associations:

<none>

Attributes:

Issuing Entity ID :
Colliding Entity ID :
Collision Event ID :
Collision Type :
Collision Velocity :
Mass :
Location Offset :

Operations:

<none>

Class name:

Detonation PDU

Documentation:

This class represents the Detonation PDU as defined by DIS standard 2.0.4 paragraph 5.4.4.2

Superclasses:

PDU

Associations:

<none>

Attributes:

Firing Entity ID :
Target Entity ID :
Munition ID :
Fire Event ID :
Fire Velocity :
Location In World :
Burst Descriptor :
Location Offset :
Detonation Result :
Articulation Parameter Count :
Articulation Parameters :

Operations:

<none>

Class name:

DIS Collision

Documentation:

This class represents a DIS Collision interaction between two DIS Entity Objects.

Superclasses:

HLA Interaction

Associations:

<none>

Attributes:

Collision_Event_ID :
Collision_Type :
Collision_Velocity :
Mass :
Location_Offset :

Operations:

<none>

Class name:

DIS Detonation

Documentation:

This class represents a DIS Detonation interaction between two DIS Entity Objects.

Superclasses:

HLA Interaction

Associations:

<none>

Attributes:

Munition_ID :
Fire_Event_ID :
Location :
Burst_Descriptor :
Fire_Velocity :
Location_Offset :
Result :

Operations:

<none>

Class name:

DIS Entity

Documentation:

This class represents an active DIS Entity of a DIS Simulation.

Superclasses:

<none>

Associations:

Association Manipulates

Attributes:

ID :
Force_ID :
Entity_Type :
Marking :
Dead_Reckoning_Parms :
Appearance :
Position_At_Update :
Orientation_At_Update :
Linear_Velocity :
Linear_Acceleration :
Angular_Velocity :
Location :
Orientation :

Operations:

<none>

Class name:

DIS Entity Object

Documentation:

This class is a DIS Entity which is an HLA Object. It is a "ready-made" HLA object for use with DIS-based simulations. Client programmers will subclass this object to create specific DIS entities such as F-15E aircraft and M-1A tanks.

Superclasses:

HLA Object

Associations:

<none>

Attributes:

Entity_ID : Entity Identifier Record
Force_ID :
Entity_Type :
Marking :
Appearance :
Position :
Orientation :

Operations:

<none>

Class name:

DIS Fire

Documentation:

This class represents a DIS Fire interaction between two DIS Entity Objects.

Superclasses:

HLA Interaction

Associations:

<none>

Attributes:

Munition_ID :
Fire_Event_ID :
Location :
Burst_Descriptor :
Fire_Velocity :
Fire_Range :
Fire_Mission_Index :

Operations:

<none>

Class name:

DIS Surrogate RTI Ambassador

Documentation:

This class is a form of RTI Ambassador which presents an HLA face to HLA simulations, but presents a DIS face to the network.

Superclasses:

RTI Ambassador

Associations:

Association Reads
Association Uses
Association Manipulates

Attributes:

<none>

Operations:

<none>

Class name:

Entity State PDU

Documentation:

This class represents the Entity State PDU as defined by DIS standard 2.0.4 paragraph 5.4.3.1

Superclasses:

PDU

Associations:

<none>

Attributes:

Entity_ID :
Force_ID :
Articulation_Parameter_Count :
Entity_Type :
Alternate_Entity_Type :
Entity_Linear_Velocity :
Entity_Location :
Entity_Orientation :
Entity_Appearance :
Dead_Reckoning_Parameters :
Entity_Marking :
Entity_Capabilities :
Articulation_Parameters :

Operations:

<none>

Class name:

Federate

Documentation:

This abstract class represents the client programmer's simulation application. The client programmer will subclass this class to handle the chores of her simulation. For example, a flight simulator subclass would model flight.

The Federate class enables the objects it has to interact with each other.

When it receives an interaction, it explicitly invokes the methods of the objects involved to carry out the interaction.

Superclasses:

<none>

Associations:

Association Uses, Creates
Association Enable Object Interaction

Attributes:

<none>

Operations:

<none>

Class name:

Federate Ambassador

Documentation:

This class represents the Federate's interface to the RTI. Rather than simply being an interface wrapper, in the SimWorx framework, the Federate Ambassador is responsible for performing the "mundane" tasks required by all HLA simulations: simulation pauses, simulation saves, simulation restores, etc.

Note that the SimWorx framework initially only supports four of the eighteen methods specified for the Federate Ambassador in the HLA Interface Spec. These methods are the minimum necessary to operate DIS simulations under the HLA (no initial need for advanced Declaration, Object, Ownership, or Time Management functions).

Superclasses:

<none>

Associations:

Association Create
Association Delete
Association Update
Association Create
Association Calls

Attributes:

<none>

Operations:

Discover_Object( )
From section 4.4 of the HLA Interface Spec v1.0
This service informs the federate that the RTI has discovered an object.

Remove_Object( )
From section 4.9 of the HLA Interface Spec v1.0
This method instructs the Federate Ambassador to remove the specified object since the object has been deleted from the federation execution.

Reflect_Attribute_Values(Object_ID : Object_ID, Attribute_Value_List : Name_Value_Pair_Set, The_Time : Federation_Time)
From section 4.5 of the HLA Interface Spec v1.0
Provides the federate with new values for a

discovered attribute. This service, coupled with the Update Attribute Values Service, forms the primary data exchange mechanism supported by the RTI.

Receive_Interaction( )

From section 4.7 of the HLA Interface Spec v1.0

Provides information about an action taken by one federation object potentially towards another object. Provides federate with information about an interaction between two objects.

Class name:

Fire PDU

Documentation:

This class represents the Fire PDU as defined by DIS standard 2.0.4 paragraph 5.4.4.1

Superclasses:

PDU

Associations:

<none>

Attributes:

Firing Entity ID :
Target Entity ID :
Munition ID :
Fire Event ID :
Fire Mission Index :
Location In World :
Burst Descriptor :
Fire Velocity :
Fire Range :

Operations:

<none>

Class name:

HLA Interaction

Documentation:

This class represents an abstract HLA Interaction. Framework client programmers will subclass this class for each interaction class specified by their simulation's SOM.

Superclasses:

<none>

Associations:

Association Uses, Creates
Association Create
Association Create

Attributes:

Class : Interaction_Class = 0
The interaction class this interaction instance belongs to.

ID : Interaction_ID = 0
The unique interaction ID of this interaction occurrence.

Initiator : Object_ID = 0
This is the object ID of the object which initiates the interaction.

Fed_Time : Federation_Time
The time the interaction is effective.

Operations:

<none>

Class name:

HLA Object

Documentation:

This class represents an HLA object. Objects interact with each other to carry out the duties of the simulation. Objects may be instantiated and controlled locally by this simulation and also instantiated on the behalf of some other federate. Object interactions may be implicit via method calls to other objects, or explicit via an Interaction invoked by the simulation.

Superclasses:

<none>

Associations:

Association Create
Association Delete
Association Update
Association Enable Object Interaction
Association Create

Attributes:

Class : Object_Class = 0
The object class this object belongs to.

ID : Object_ID = 0
The ID of this object. It is unique within the federation execution.

Operations:

Create(Object_ID : Object_ID, Object_Class : Object_Class)
Initialize an Object

Class name:

Operating System

Documentation:

This is not so much a class as a notion of the DIS Surrogate RTI Ambassador's dependence upon the operating system to read PDUs.

Superclasses:

<none>

Associations:

Association Uses

Attributes:

<none>

Operations:

<none>

Class name:

PDU

Documentation:

This class represents the common components of all categories of DIS PDU.

Superclasses:

<none>

Associations:

Association Reads

Attributes:

PDU_Header : PDU Header Record
The PDU Header Record is defined by DIS 2.0.4 standard paragraph 5.3.24

Operations:

<none>

Class name:

RTI Ambassador

Documentation:

This class represents the RTI's interface to the Federate.

Provides an abstract interface (specified by RTI IDL specifications) for different kinds of RTIs. SimWorx initially implements a DIS surrogate RTI.

Note that iteration two of the SimWorx framework supports only six of the forty-six methods specified for the RTI Ambassador in the HLA Interface Spec. These methods are the minimum necessary to operate DIS simulations under the HLA (no need for advanced Declaration, Object, Ownership, or Time Management functions).

Superclasses:

<none>

Associations:

Association Calls

Attributes:

<none>

Operations:

Request_ID( )

Reference section 4.1 of HLA Interface Specification v1.0

Provides federate with a number of IDs which the federate can assign to objects.

Register_Object( )

Reference section 4.2 of HLA Interface Specification v1.0.

Links an object ID with an instance of an object class. (i.e., lets Federate tell the RTI that the Federate has instantiated an object of this class).

Send_Interaction( )

Reference section 4.6 of HLA Interface Specification v1.0.

Informs the federation of an action taken by one object, potentially towards another object.

Delete_Object( )

Reference section 4.8 of HLA Interface Specification v1.0.

Informs the federation that an object with that ID, owned by the federate, is to be removed from the federateion execution. Once the object is removed from the federation execution, its ID cannot be reused. The RTI will use the Remove Object service to inform the reflecting federates that the object has been deleted.

Update_Attribute_Values( )

Reference section 4.3 of HLA Interface Specification v1.0.

Provides the current attribute values to the federation for attributes owned by the federate. This service, coupled with the Reflect Attribute Values sercie, form the primary data exchange mechanism supported by the RTI.

Request Federation Time( )

Reference section 6.1 of HLA Interface Specification v1.0

Returns the current Federation time. For SimWorx, the Federation is considered the centralized time-keeper.

Index

Thesis Appendix Note: Since this stand-alone analysis document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute D-5 for 5, D-11 for 11, etc.

This Page Intentionally Left Blank

Appendix E: SimWorx Iteration Two Design Model

SimWorx: Design Model

Iteration Two Design Model

Software Architectures in Ada 95
AFIT Software Engineering Group
Air Force Institute of Technology
Wright-Patterson AFB, OH 45434

This Page Intentionally Left Blank

Table Of Contents

Thesis Appendix Note: Since this stand-alone design document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute E-5 for 5, E-11 for 11, etc.

List of Figures

Thesis Appendix Note: Since this stand-alone design document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute E-5 for 5, E-11 for 11, etc.

* Denotes New/Updated Figure for Iteration Two

Introduction

This document reflects the first iteration of software design for the SimWorx application framework for distributed simulation.

The SimWorx framework enables application programmers (a.k.a. client programmers) to quickly and easily generate distributed simulation applications conforming to the DoD High Level Architecture (HLA) for Modeling and Simulation. In addition, the SimWorx framework provides a Surrogate HLA Run-Time Infrastructure (RTI) which performs Distributed Interactive Simulation (DIS) version 2.0.4 network input and output.

Initial versions of the SimWorx framework do not support either the full HLA nor full DIS standards. Initial versions supply the minimal functionality necessary for simple objects to interact for typical DIS interactions (entity movement and collision, weapons fire and detonation).

This first iteration of the SimWorx framework concentrates on receiving input from the DIS network. The application framework is divided into two main categories: Federate components with the behavior of an HLA Federate, and RTI components with the behavior of the HLA RTI.

DIS Protocol Data Units (PDUs) are received from the network by the Surrogate RTI Ambassador[1]. DIS Entities are modeled and their states are dead-reckoned between network updates. DIS Entity information is forwarded by the RTI Ambassador to the Federate Ambassador via the standard HLA ambassador interface, so the federate components may be reused with various sorts of RTIs--see the Federate Ambassador class for details.

The second iteration of the SimWorx framework adds the ability to send output to the DIS network. HLA_Objects of the Federation interact via interactions. Actions which affect HLA_Objects which aren't owned by the SimWorx Federate are sent to the RTI as HLA_Interactions. State changes of the HLA_Objects are sent to the RTI as attribute updates. The DIS Surrogate RTI Ambassador interprets the updates and interactions, produces appropriate DIS PDUs, and broadcasts the PDUs onto the network.

[1] The term *Ambassador* is an HLA term for a kind of interface software component. Just as a diplomatic ambassador represents a foreign country, an *Ambassador* presents a local application programming interface for a system component which may actually be physically located somewhere else (another computer, another state, etc.). Thus, a client can make software procedure calls to the Ambassador, rather than relying on some message passing interface, since the Ambassador "abstracts-out" the details of the actual interaction.

Visualizing the SimWorx Architecture Via The 4+1 View Model.

Software is inherently hard to visualize since it's not of the physical world. Phillipe Kruchten's 4+1 View Model (IEEE Software, November 1995) promotes architectural understanding of a system via a collection of views. Each view describes the system from a different perspective--taken together, the views are a description of the overall system.

Kruchten's model relies on 4 different kinds views and a set of scenarios (or use cases). The *logical view* describes the systems functions and the services it supplies to end users--in SimWorx, we denote the logical view with Rumbaugh Object Model diagrams and Rational Rose Class Category diagrams.

The *process view* helps show how the system satisfies non-functional requirements like performance and availability. It shows the independent tasks and processes of the system and the links between them. SimWorx uses Rational Rose Module diagrams to show the relationships between the tasks of the framework.

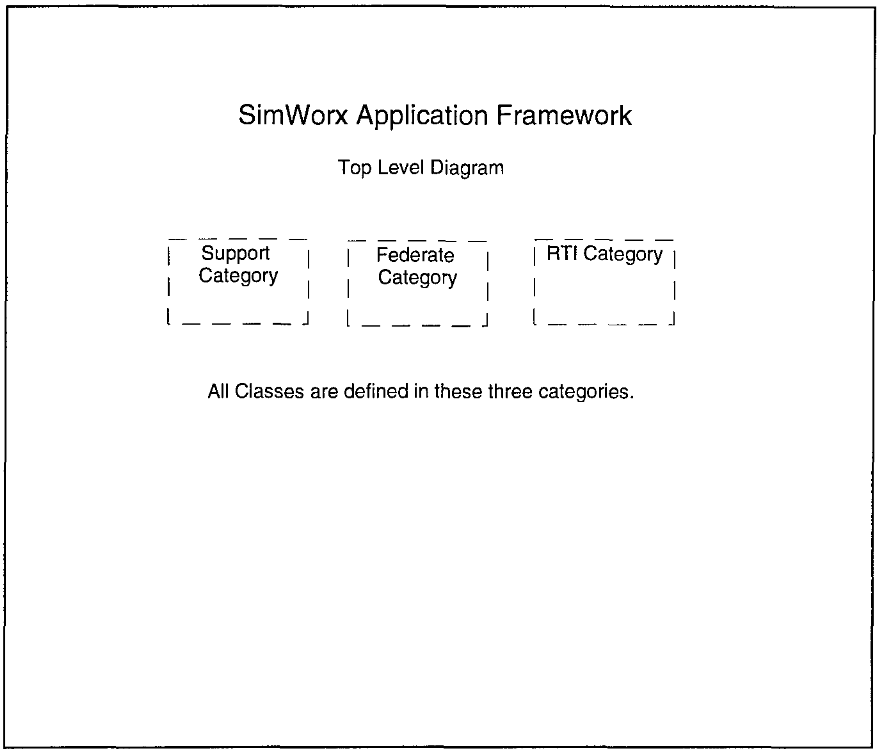
The *development view* shows the organization of the systems software modules. It helps developers, maintainers, and client developers of the SimWorx framework understand the layout of the source code. SimWorx uses Rational Rose Module diagrams to show the development view.

The *physical view* shows the mapping of logical processes to the actual hardware they run on. SimWorx uses Rational Rose Module diagrams to show the physical view.

Scenarios are the fifth (4+1) view. They show the dynamic interactions and behavior of the architecture. SimWorx uses Rational Rose Message Trace Diagrams (MTDs) to display scenarios. The arrows in a MTD represent method invocation or message passing between the objects. The arrows are ordered in time (1. happens before 2., etc.).

SimWorx Logical View

The following diagrams show the Logical View for the SimWorx Application Framework for Distributed Simulation in terms of a Rational Rose Class Category Diagram and Rumbaugh Object Model Diagrams.



SimWorx Application Framework

Top Level Diagram

Support Category

Federate Category

RTI Category

All Classes are defined in these three categories.

Figure 1. SimWorx Application Framework

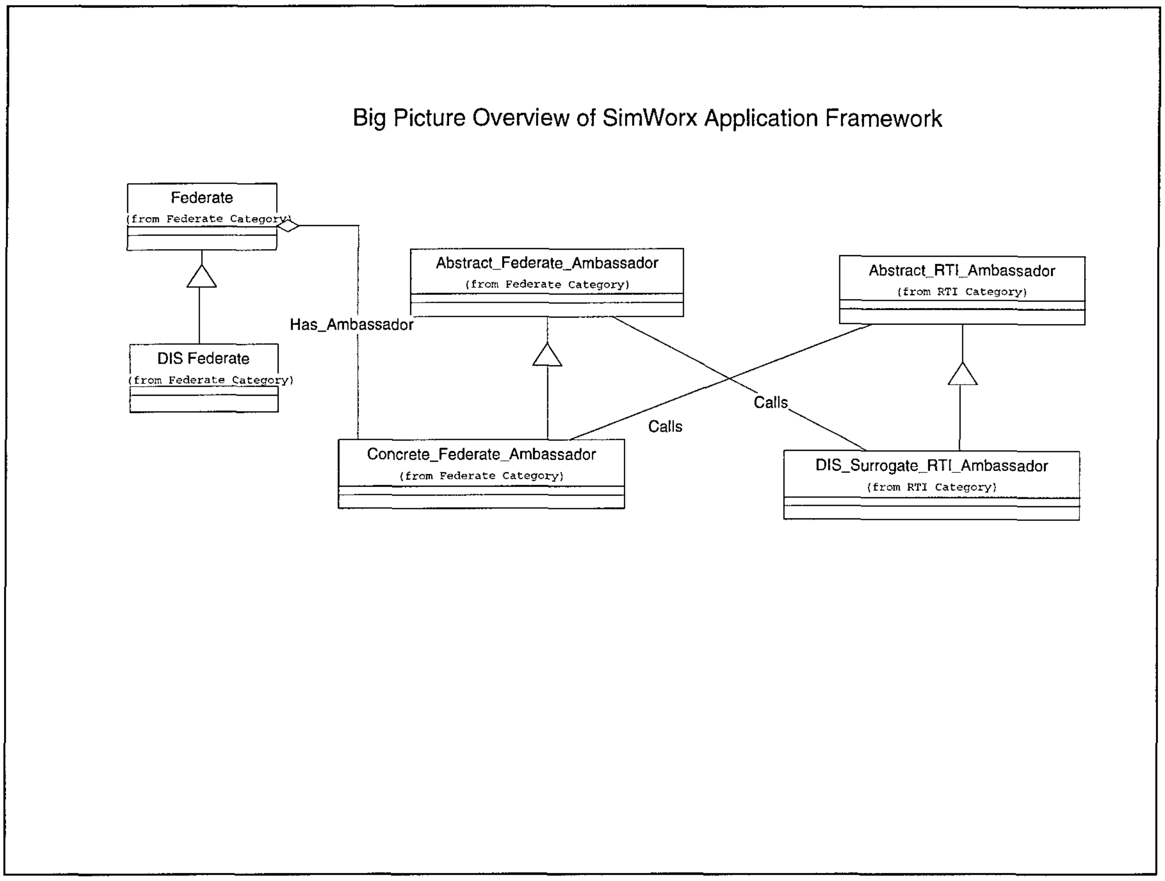


Figure 2. Big Picture Overview of SimWorx Application Framework

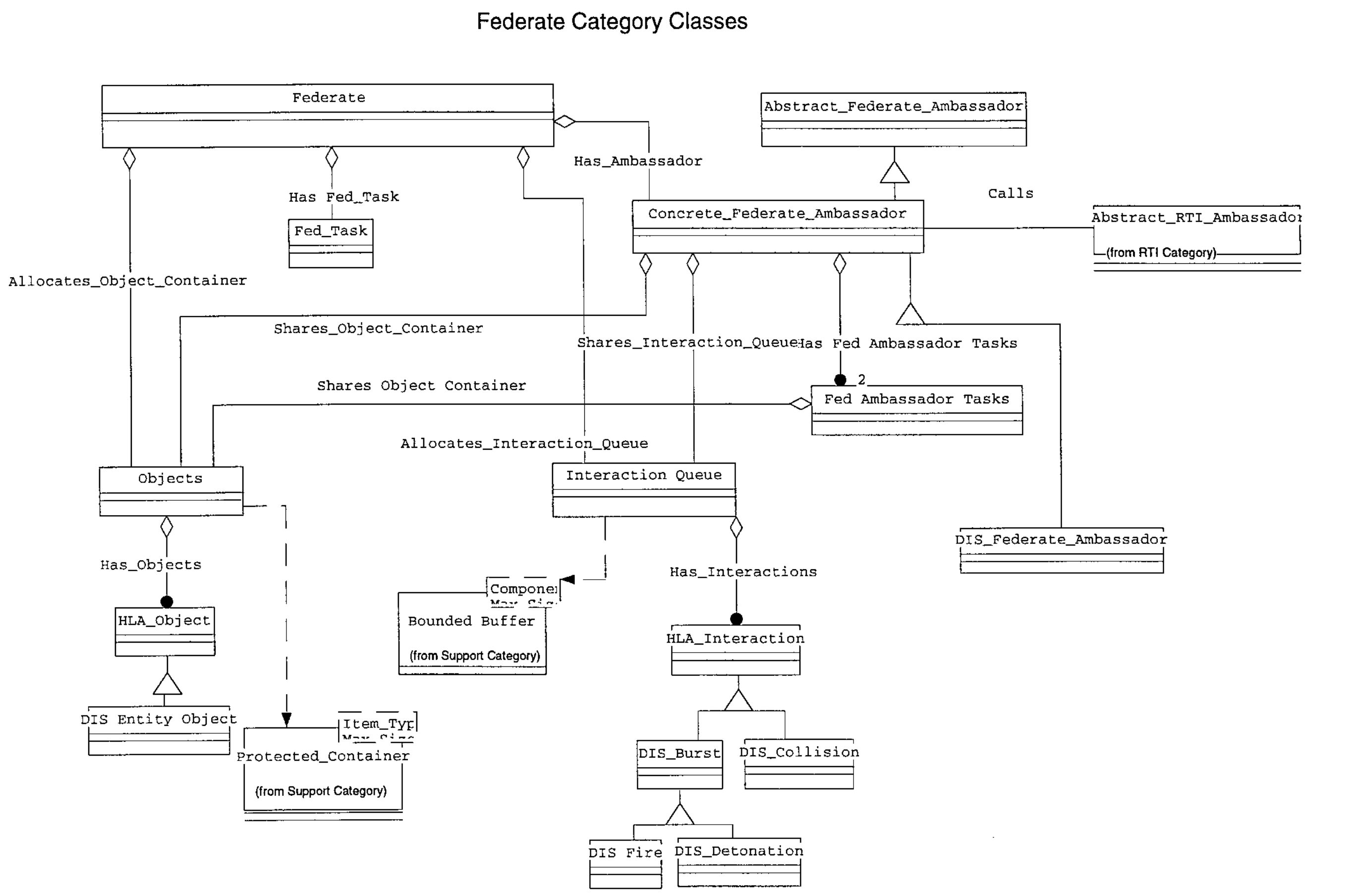


Figure 3. Federate Category Classes

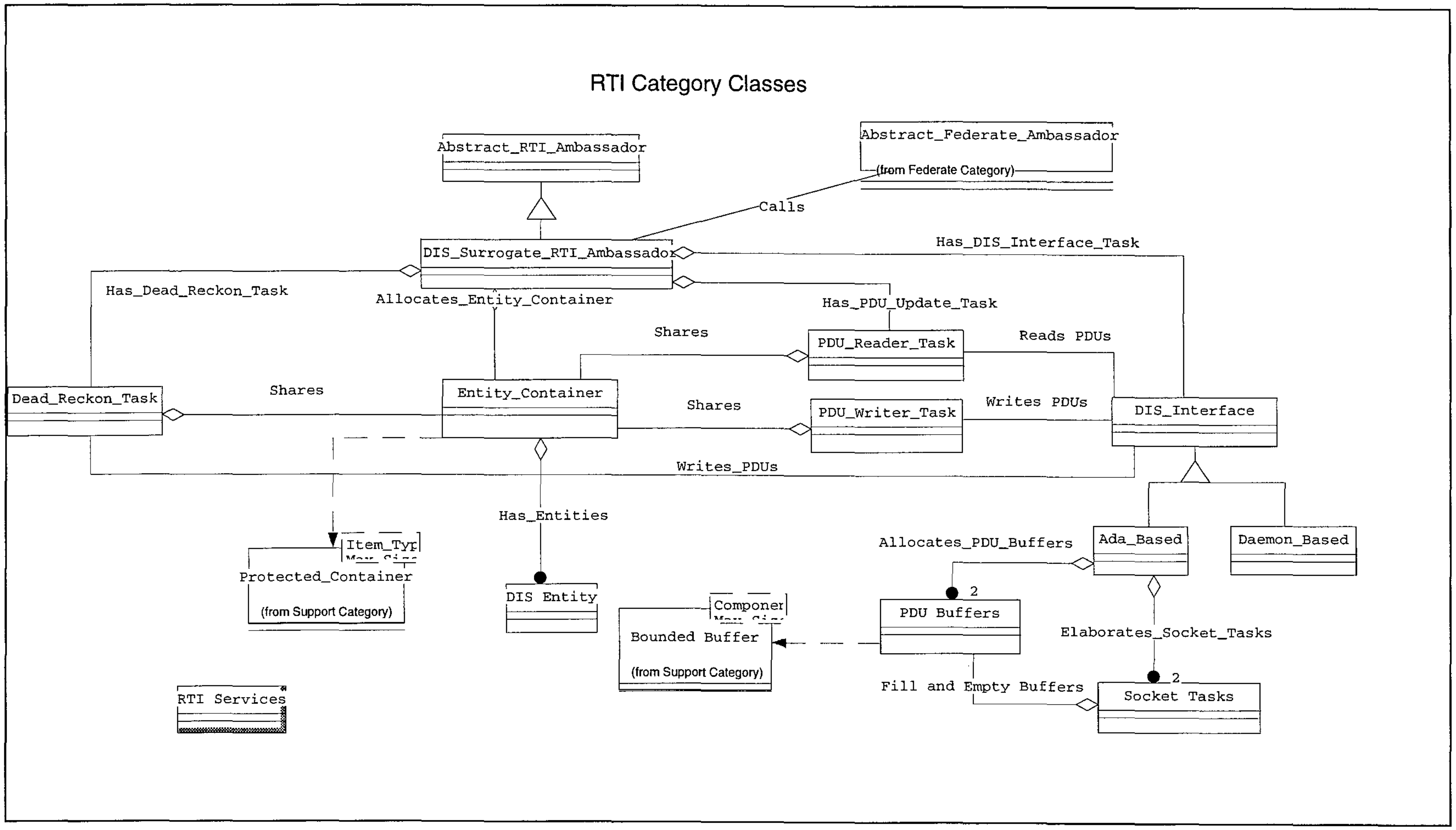


Figure 4. RTI Category Classes

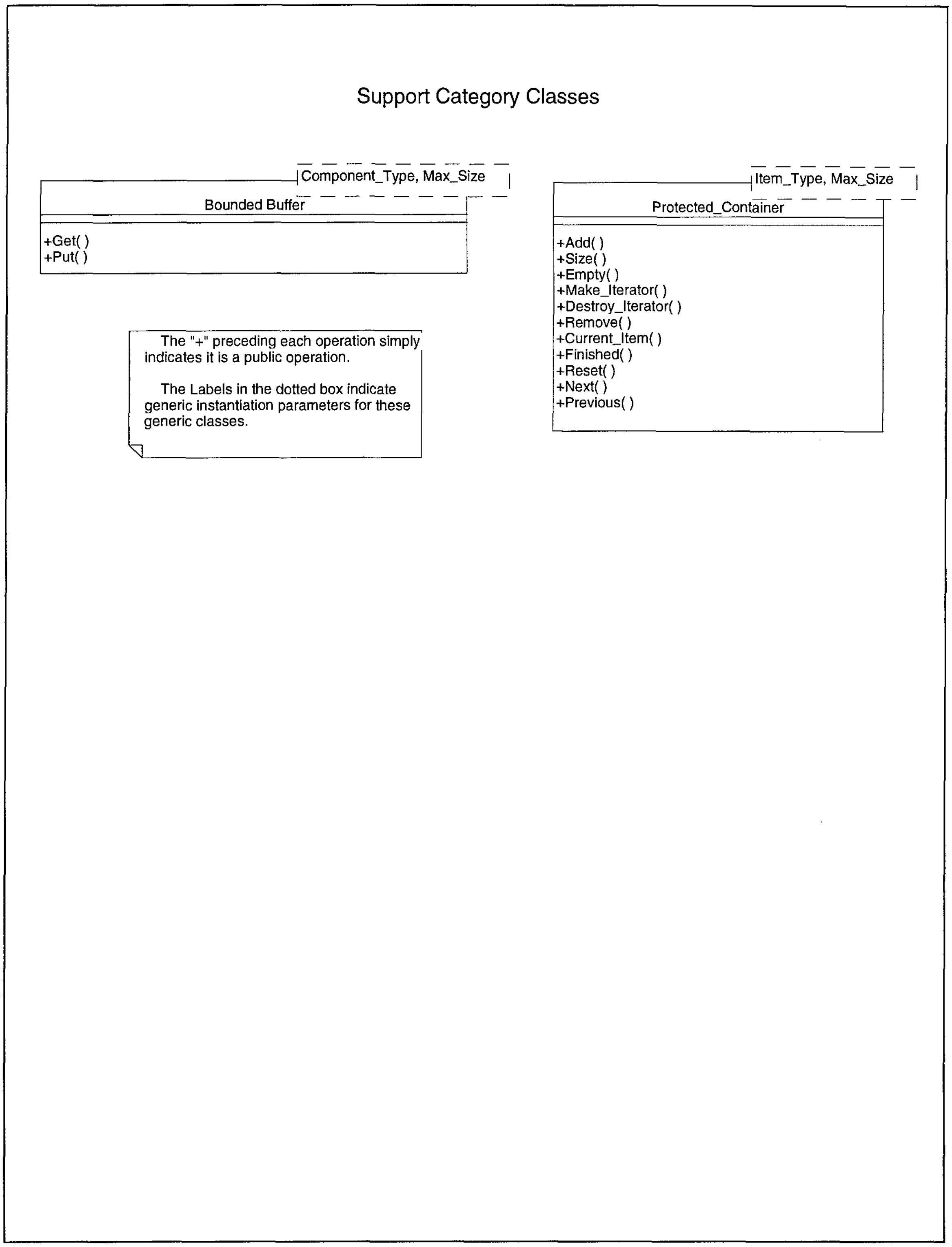


Figure 5. Support Category Classes

Scenarios

The following scenarios describe the functionality of the SimWorx Application Framework. They expand upon the use cases of the analysis model, and are described with Message Trace Diagrams. Each numbered item is a message from a sending class to a receiving class. The messages are ordered by time: later numbered messages are later in time than earlier numbered messages. Scenarios denoted by an asterisk (*) are new/modified for iteration two.

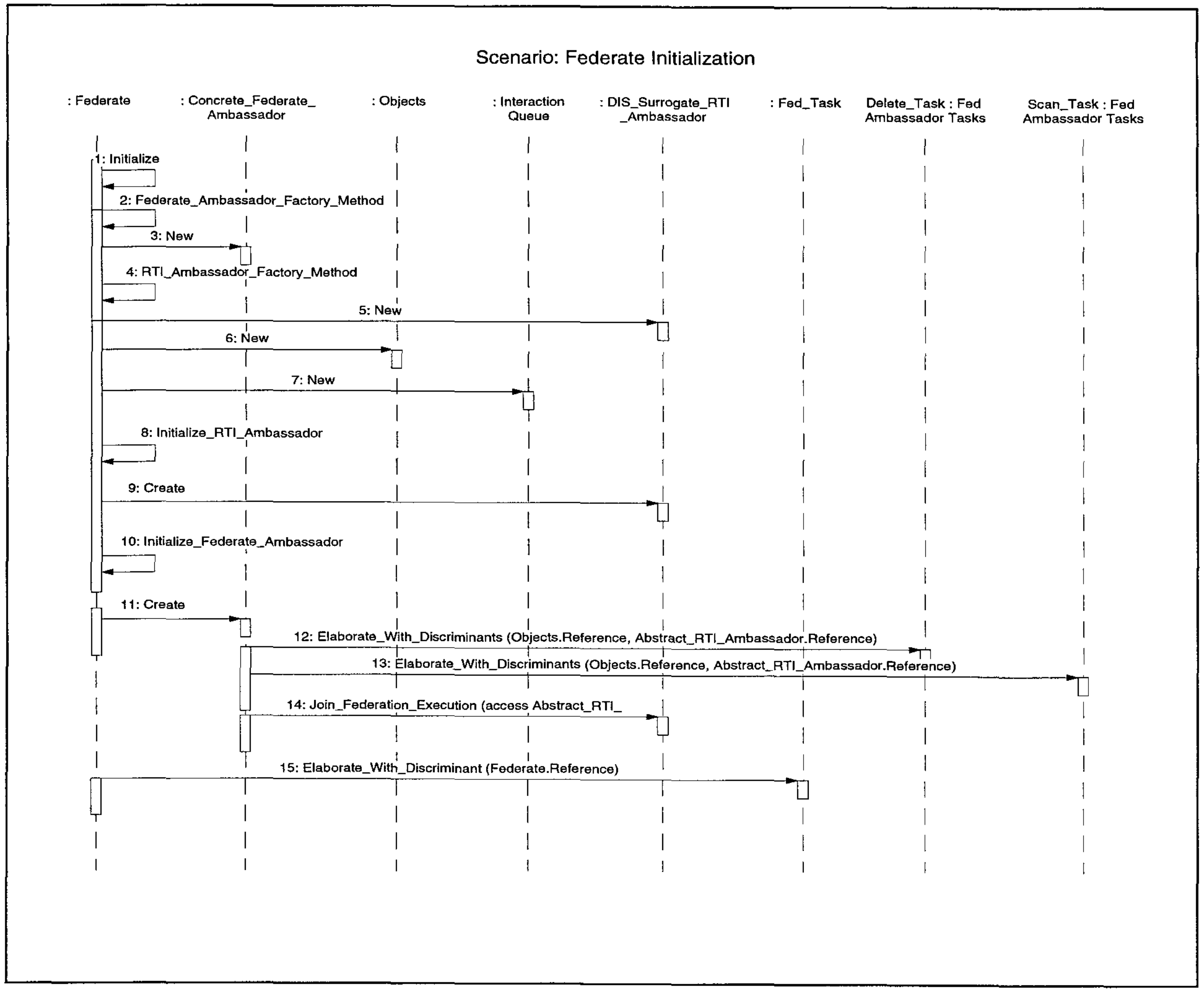

Scenario: Federate Initialization
: Federate
: Concrete_Federate_ Ambassador
: Objects
: Interaction Queue
: DIS_Surrogate_RTI _Ambassador
: Fed_Task
Delete_Task : Fed Ambassador Tasks
Scan_Task : Fed Ambassador Tasks
1: Initialize
2: Federate_Ambassador_Factory_Method
3: New
4: RTI_Ambassador_Factory_Method
5: New
6: New
7: New
8: Initialize_RTI_Ambassador
9: Create
10: Initialize_Federate_Ambassador
11: Create
12: Elaborate_With_Discriminants (Objects.Reference, Abstract_RTI_Ambassador.Reference)
13: Elaborate_With_Discriminants (Objects.Reference, Abstract_RTI_Ambassador.Reference)
14: Join_Federation_Execution (access Abstract_RTI_
15: Elaborate_With_Discriminant (Federate.Reference)

Figure 6. Scenario: Federate Initialization

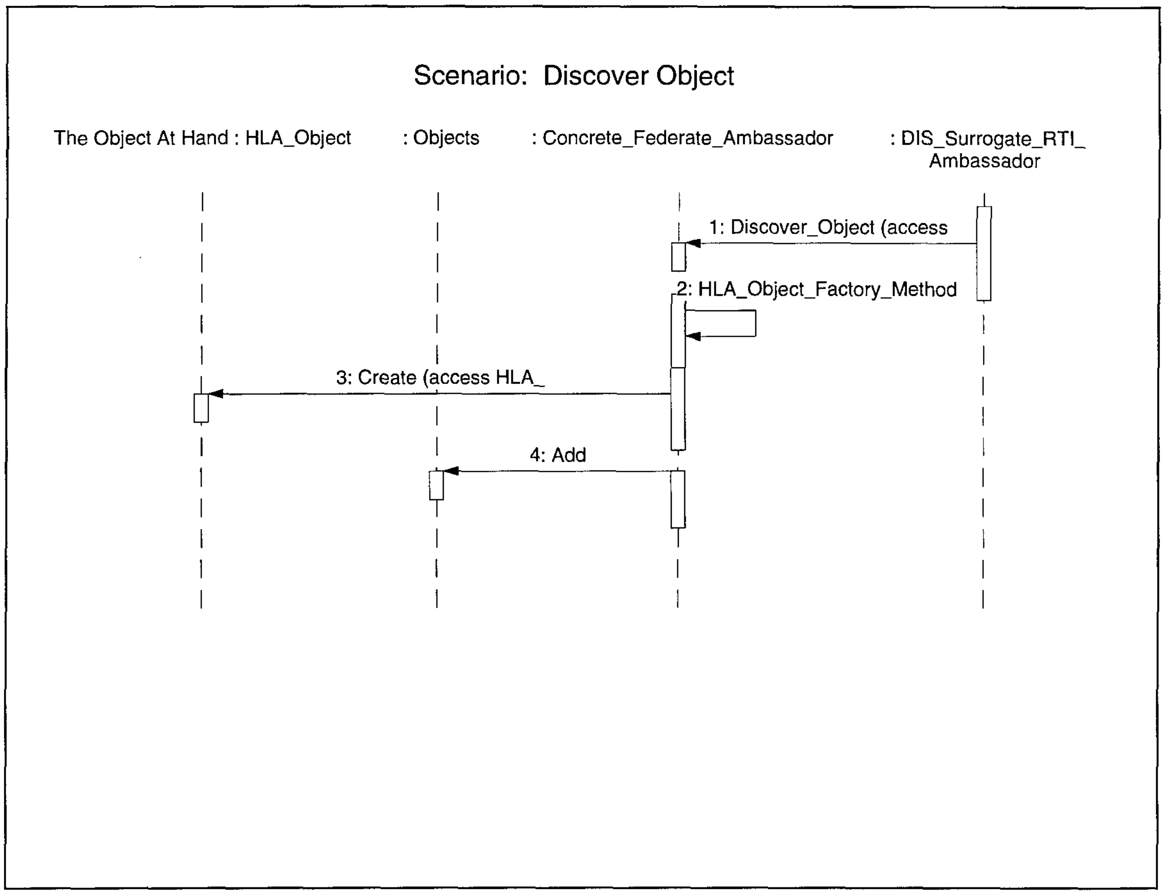

Scenario: Discover Object
The Object At Hand : HLA_Object
: Objects
: Concrete_Federate_Ambassador
: DIS_Surrogate_RTI_ Ambassador
1: Discover_Object (access
2: HLA_Object_Factory_Method
3: Create (access HLA_
4: Add

Figure 7. Scenario: Discover Object

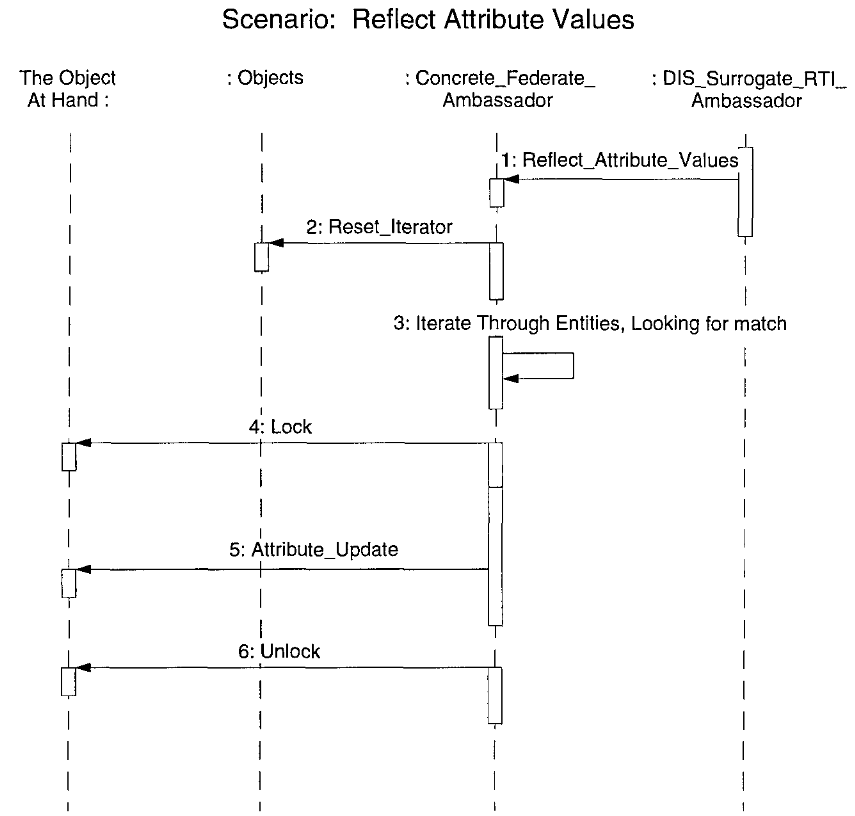

Scenario: Reflect Attribute Values
The Object At Hand :
: Objects
: Concrete_Federate_ Ambassador
: DIS_Surrogate_RTI_ Ambassador
1: Reflect_Attribute_Values
2: Reset_Iterator
3: Iterate Through Entities, Looking for match
4: Lock
5: Attribute_Update
6: Unlock

Figure 8. Scenario: Reflect Attribute Values

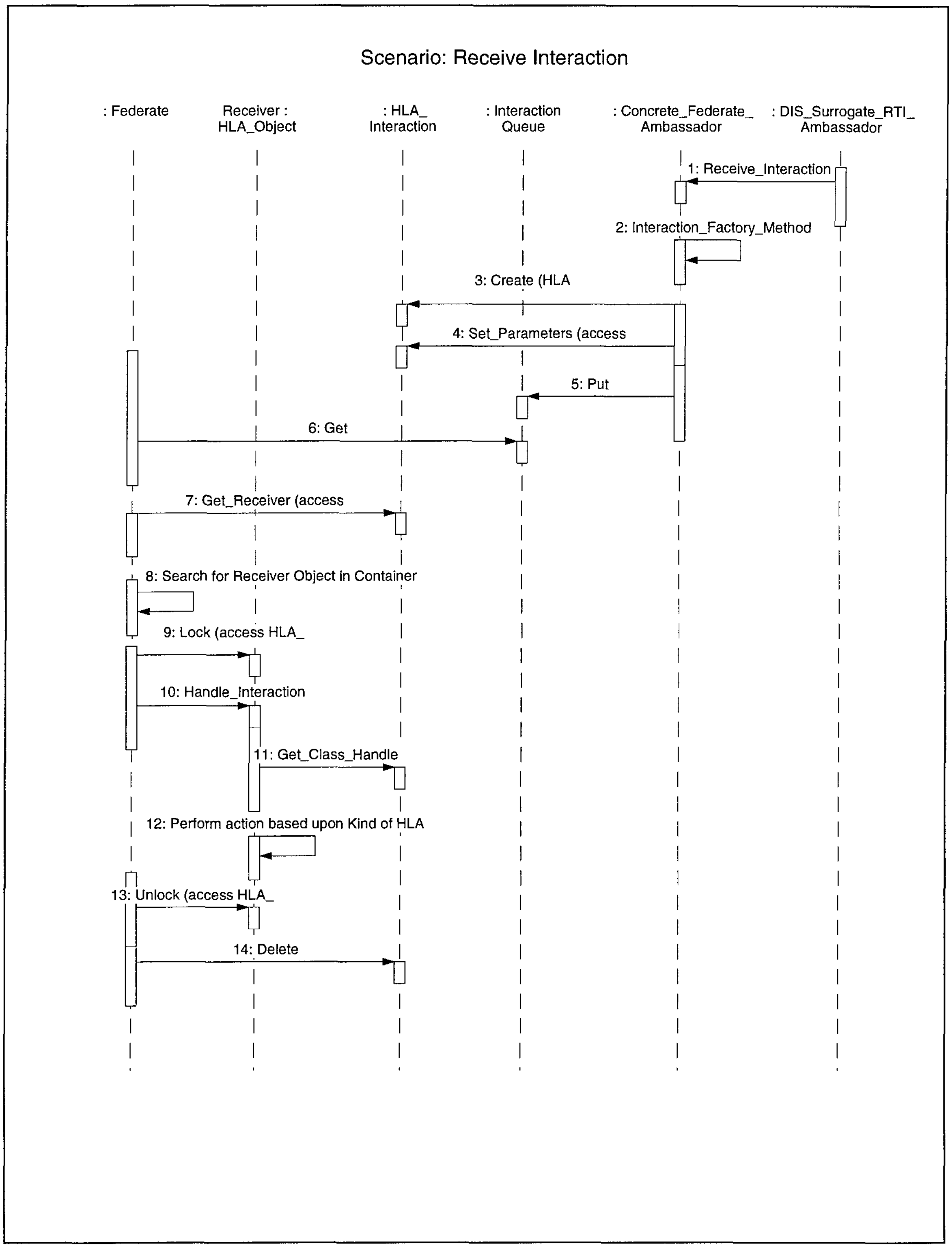


Figure 9. Scenario Receive Interaction

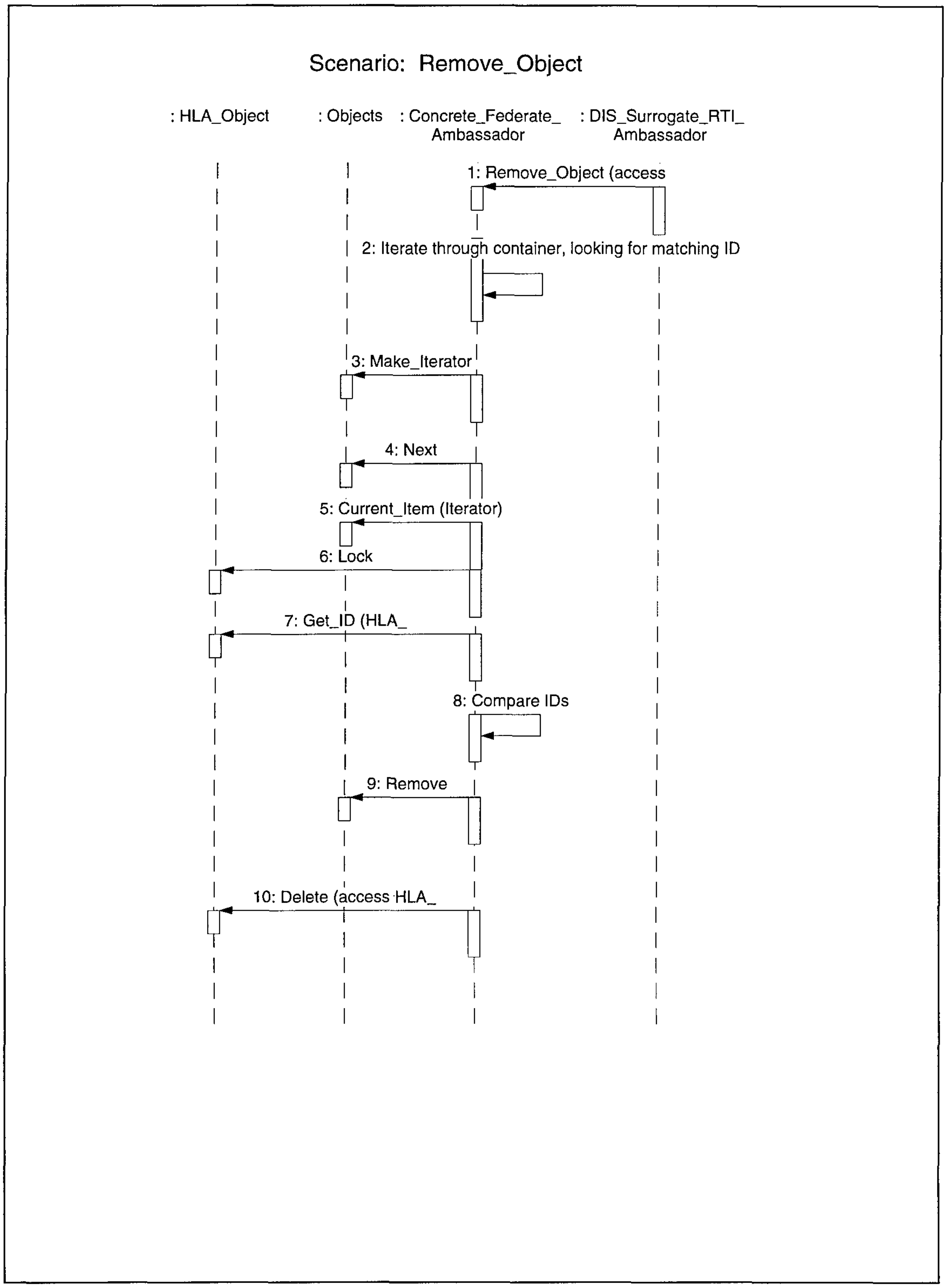


Figure 10. Scenario: Remove Object

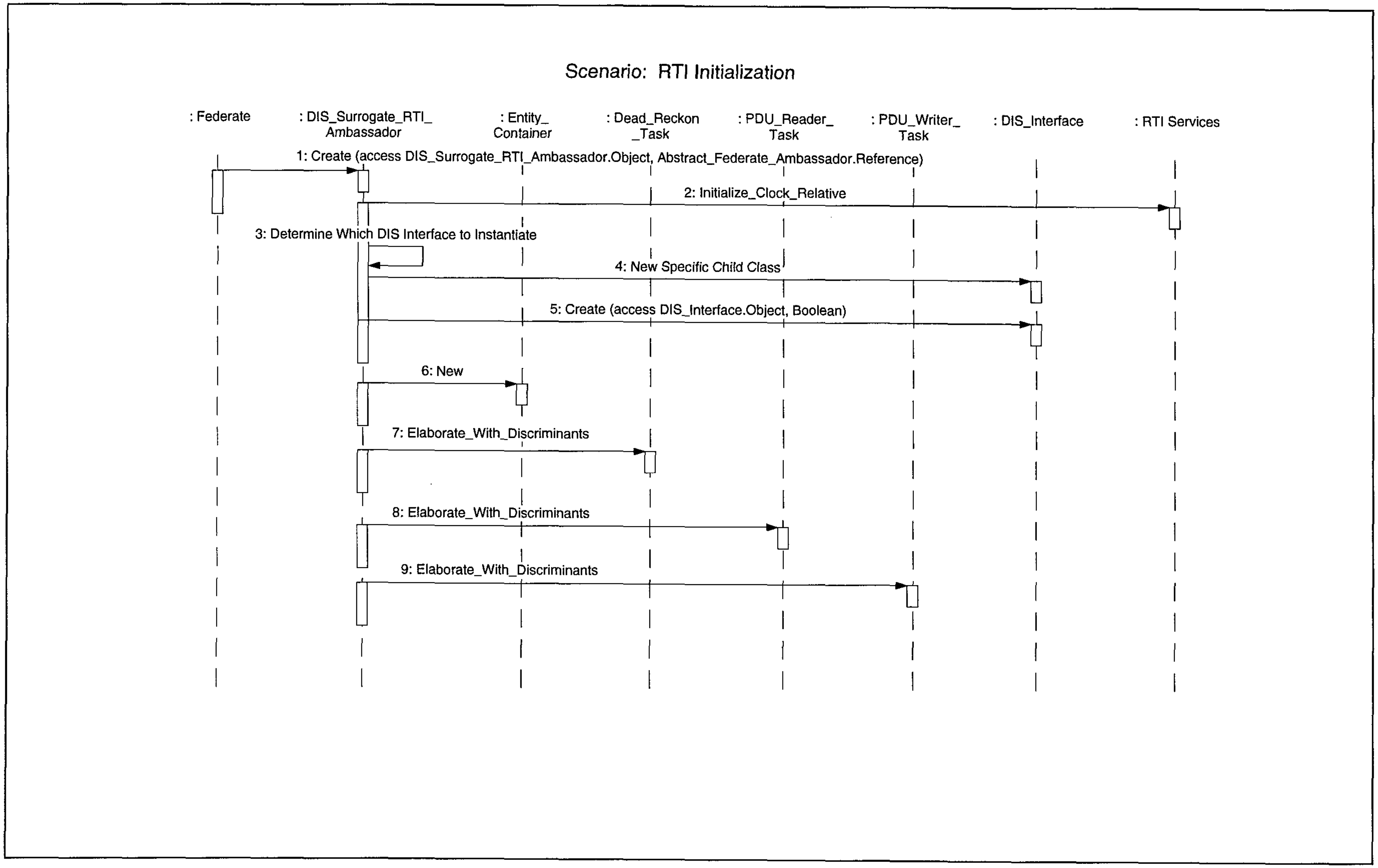

Scenario: RTI Initialization
: Federate
: DIS_Surrogate_RTI_ Ambassador
: Entity_ Container
: Dead_Reckon _Task
: PDU_Reader_ Task
: PDU_Writer_ Task
: DIS_Interface
: RTI Services
1: Create (access DIS_Surrogate_RTI_Ambassador.Object, Abstract_Federate_Ambassador.Reference)
2: Initialize_Clock_Relative
3: Determine Which DIS Interface to Instantiate
4: New Specific Child Class
5: Create (access DIS_Interface.Object, Boolean)
6: New
7: Elaborate_With_Discriminants
8: Elaborate_With_Discriminants
9: Elaborate_With_Discriminants

Figure 11. Scenario: RTI Initialization

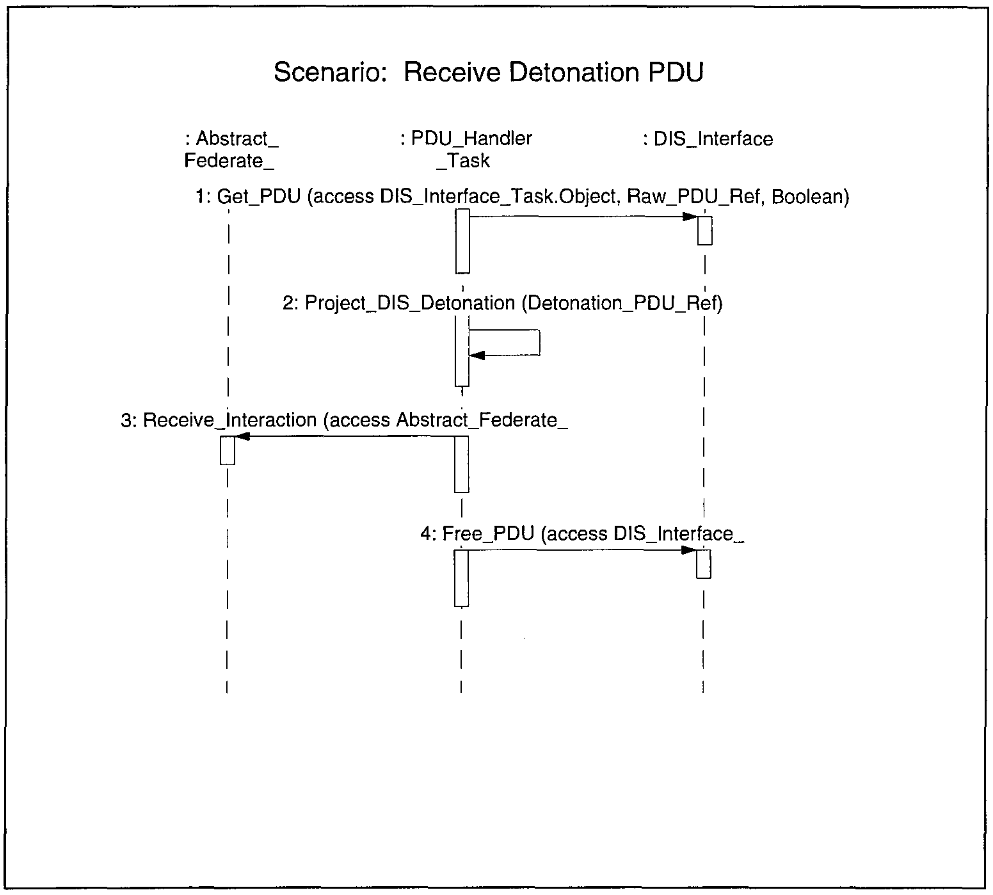

Scenario: Receive Detonation PDU
: Abstract_
Federate_
: PDU_Handler
_Task
: DIS_Interface
1: Get_PDU (access DIS_Interface_Task.Object, Raw_PDU_Ref, Boolean)
2: Project_DIS_Detonation (Detonation_PDU_Ref)
3: Receive_Interaction (access Abstract_Federate_
4: Free_PDU (access DIS_Interface_

Figure 12. Scenario: Receive Detonation PDU

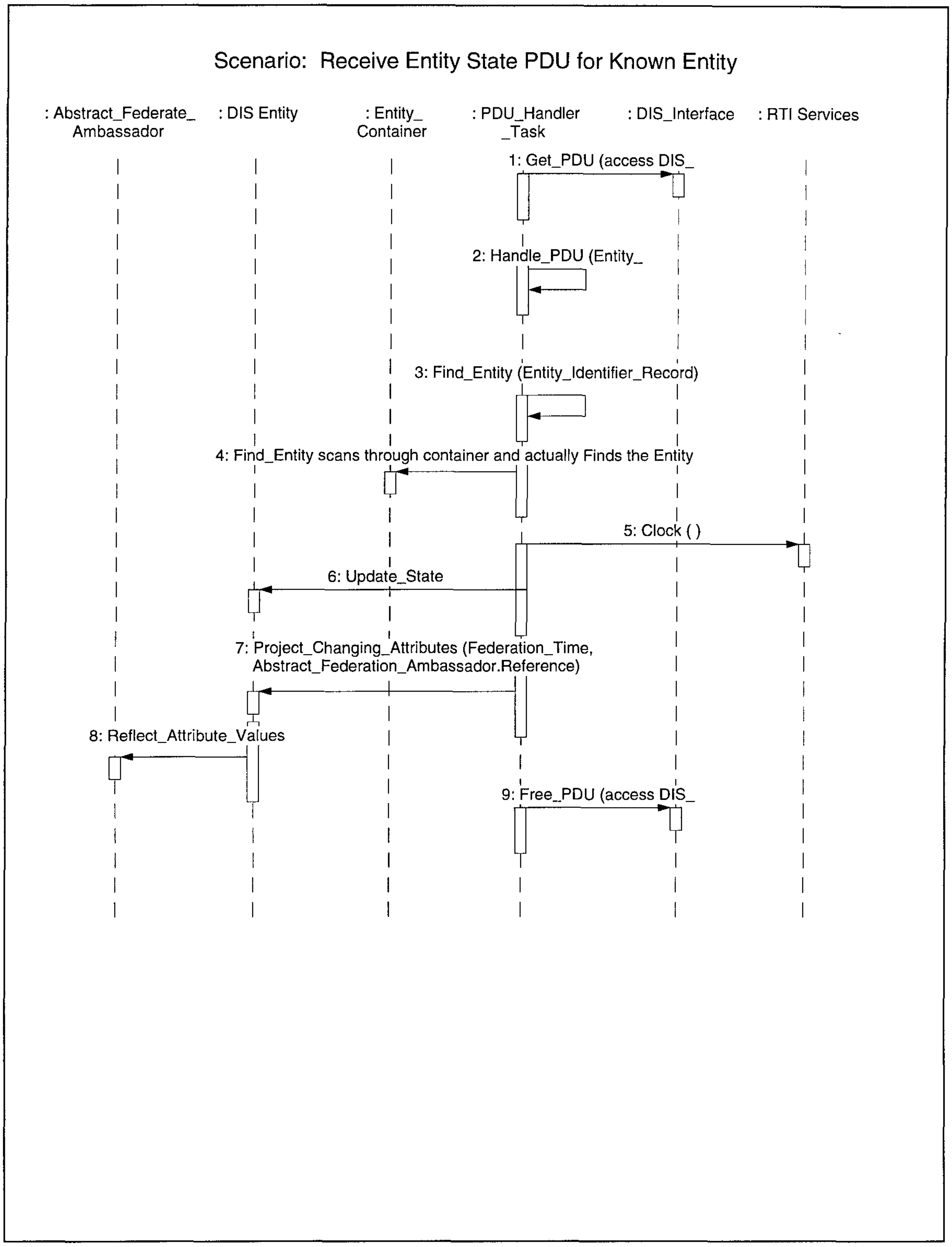

Scenario: Receive Entity State PDU for Known Entity
: Abstract_Federate_ Ambassador
: DIS Entity
: Entity_ Container
: PDU_Handler _Task
: DIS_Interface
: RTI Services
1: Get_PDU (access DIS_
2: Handle_PDU (Entity_
3: Find_Entity (Entity_Identifier_Record)
4: Find_Entity scans through container and actually Finds the Entity
5: Clock ( )
6: Update_State
7: Project_Changing_Attributes (Federation_Time, Abstract_Federation_Ambassador.Reference)
8: Reflect_Attribute_Values
9: Free_PDU (access DIS_

Figure 13. Scenario: Receive Entity State PDU for Known Entity

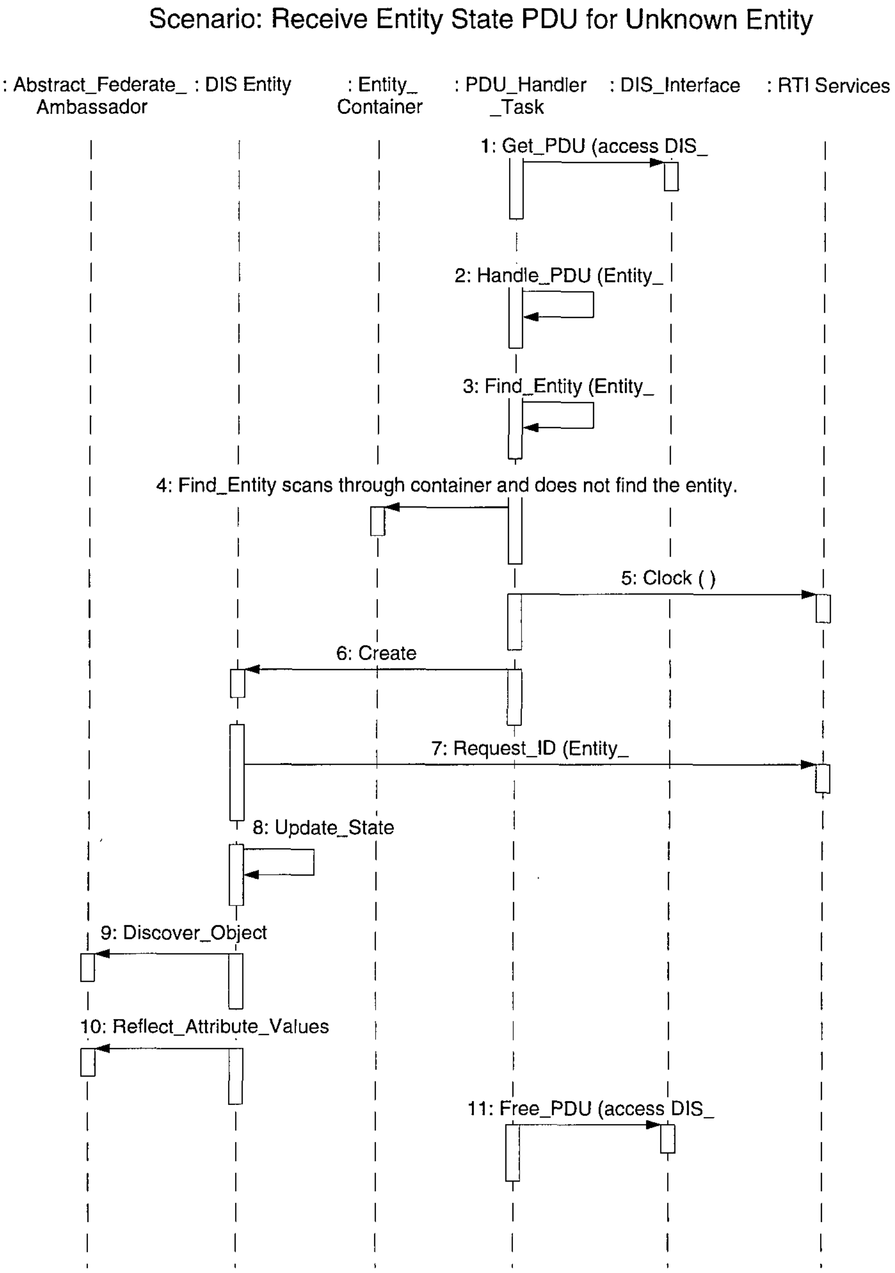


Figure 14. Scenario: Receive Entity State PDU for Unknown Entity

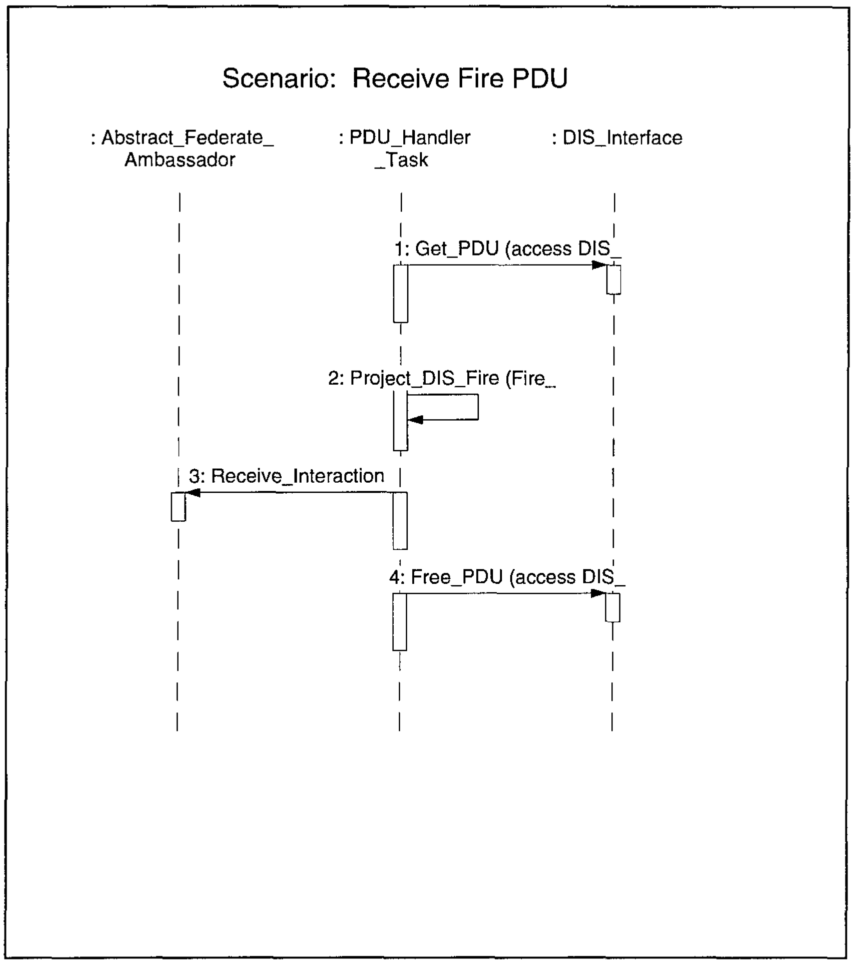


Figure 15. Scenario: Receive Fire PDU

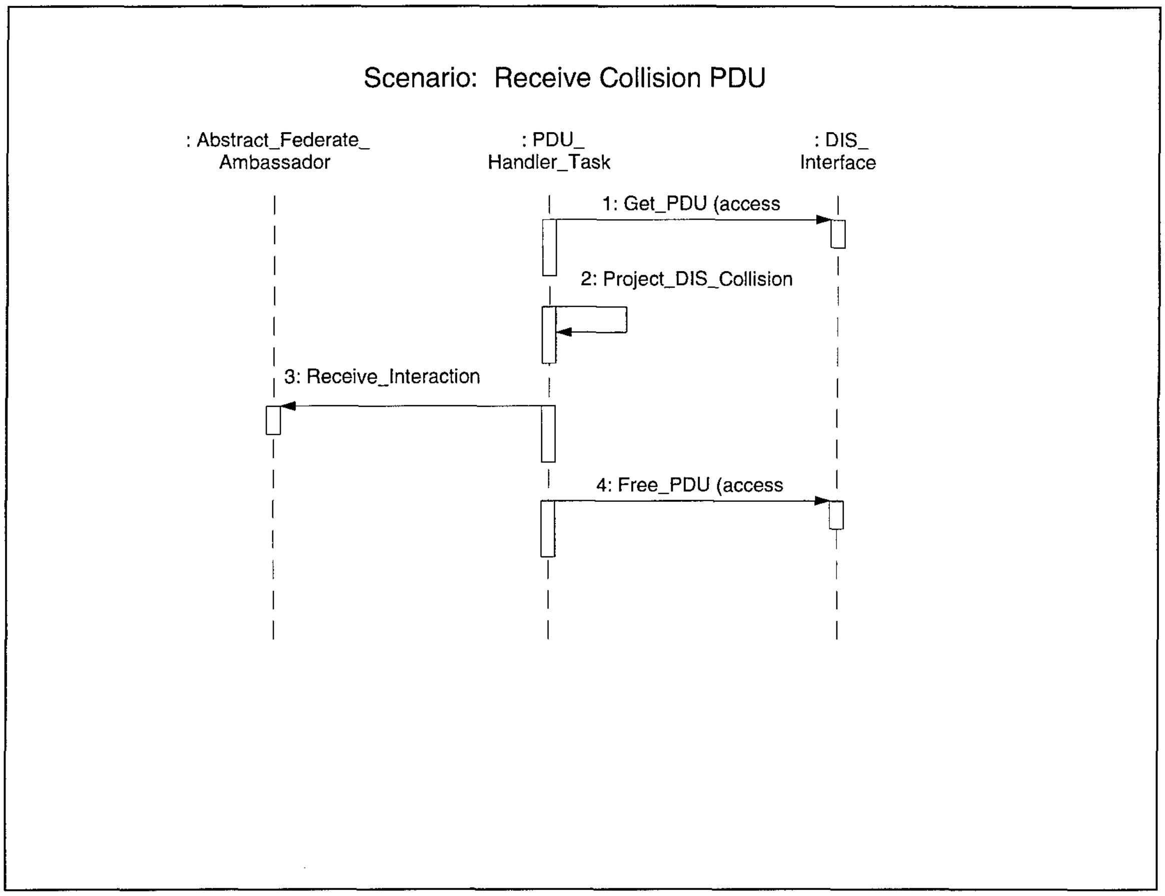


Figure 16. Scenario: Receive Collision PDU

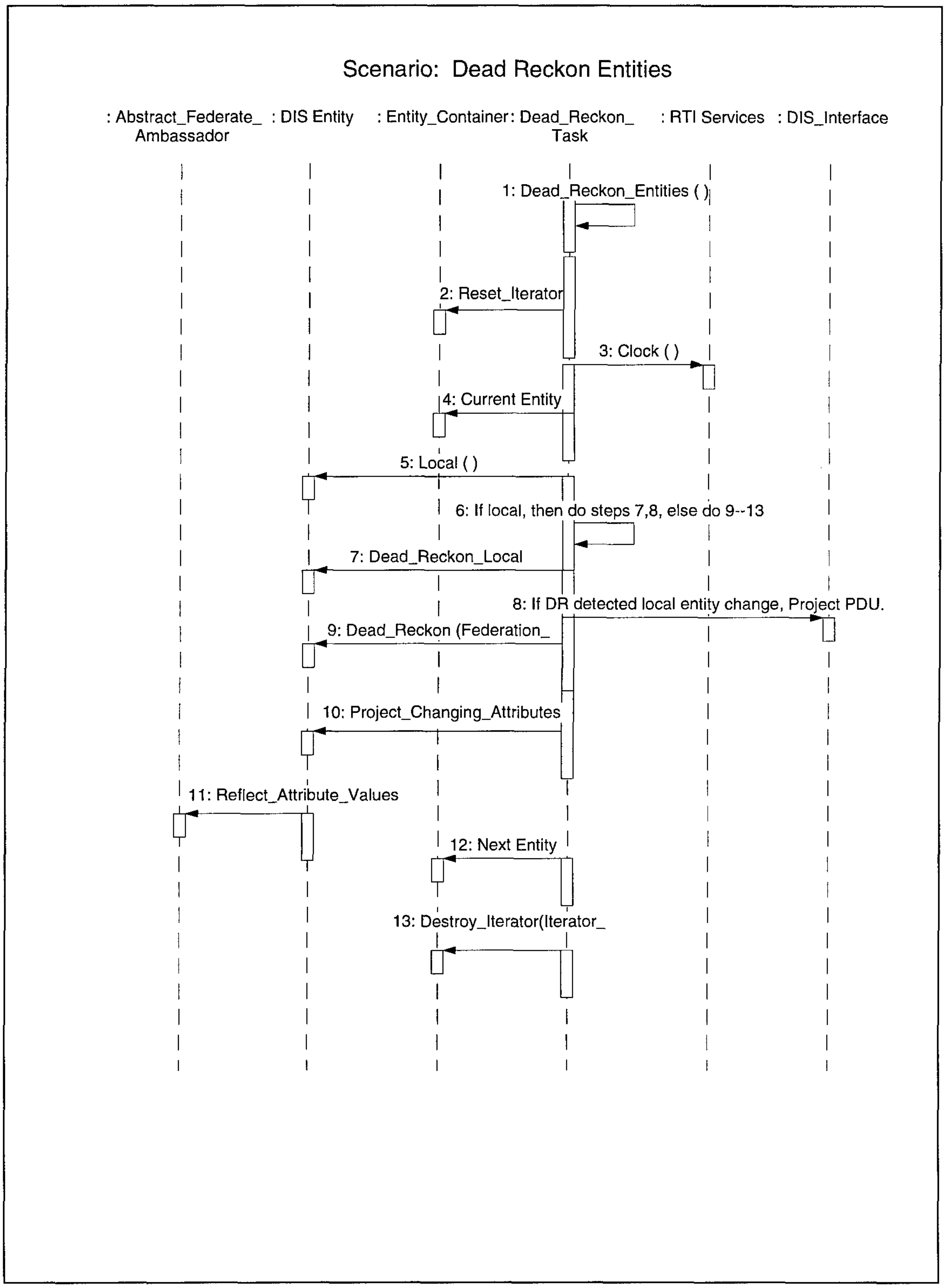

Scenario: Dead Reckon Entities
: Abstract_Federate_ Ambassador
: DIS Entity
: Entity_Container
: Dead_Reckon_ Task
: RTI Services
: DIS_Interface
1: Dead_Reckon_Entities ( )
2: Reset_Iterator
3: Clock ( )
4: Current Entity
5: Local ( )
6: If local, then do steps 7,8, else do 9--13
7: Dead_Reckon_Local
8: If DR detected local entity change, Project PDU.
9: Dead_Reckon (Federation_
10: Project_Changing_Attributes
11: Reflect_Attribute_Values
12: Next Entity
13: Destroy_Iterator(Iterator_

Figure 17. Scenario: Dead Reckon Entities

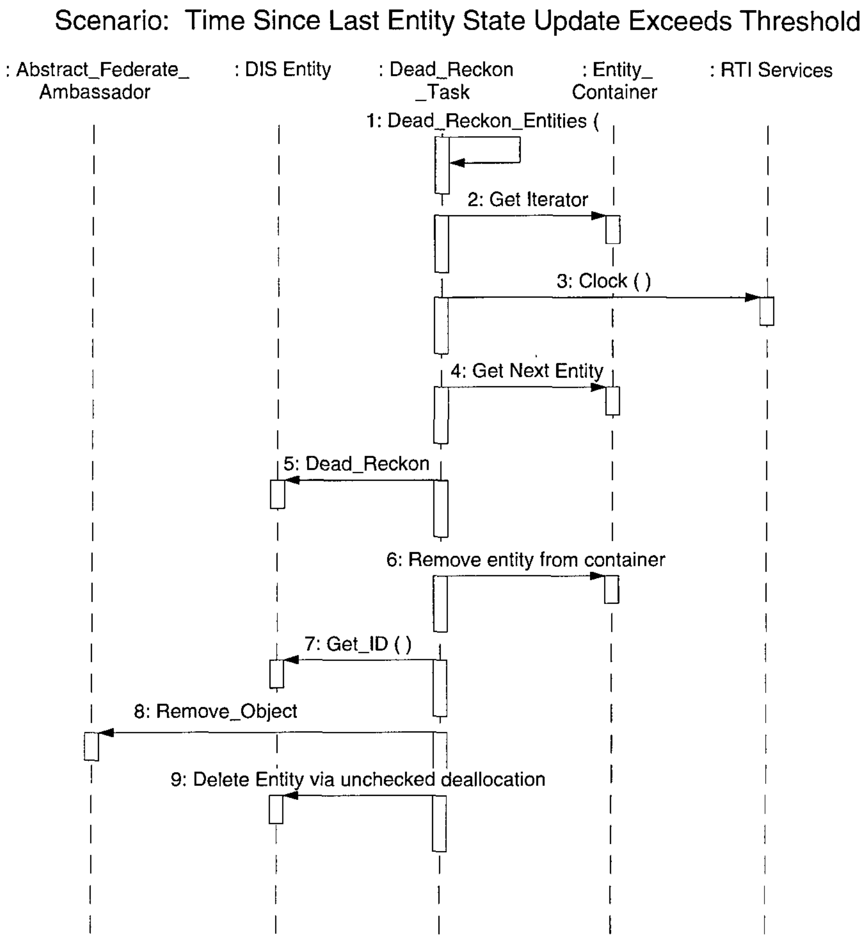

Scenario: Time Since Last Entity State Update Exceeds Threshold
: Abstract_Federate_ Ambassador
: DIS Entity
: Dead_Reckon _Task
: Entity_ Container
: RTI Services
1: Dead_Reckon_Entities (
2: Get Iterator
3: Clock ( )
4: Get Next Entity
5: Dead_Reckon
6: Remove entity from container
7: Get_ID ( )
8: Remove_Object
9: Delete Entity via unchecked deallocation

Figure 18. Scenario: Time Since Last Entity State Update Exceeds Threshold

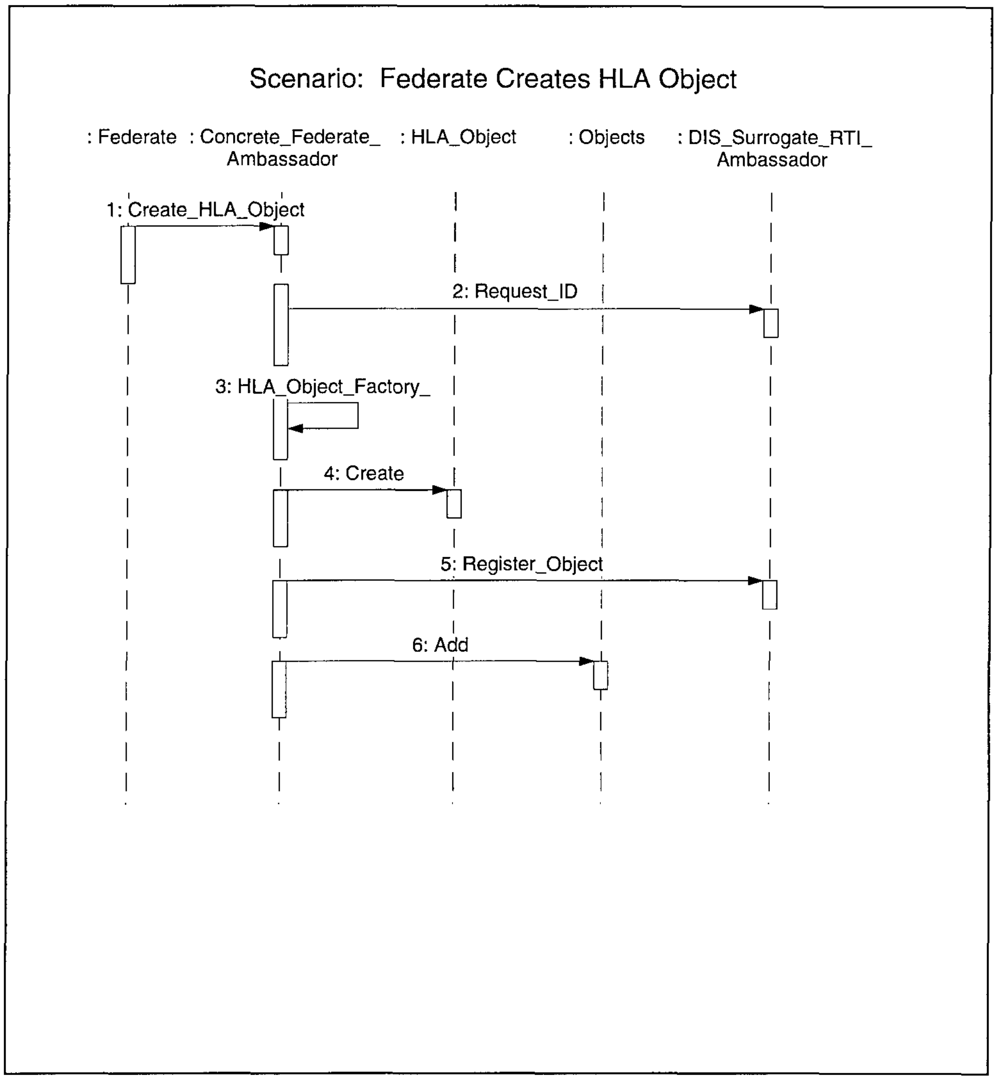


Figure 19. Scenario: Federate Creates HLA Object

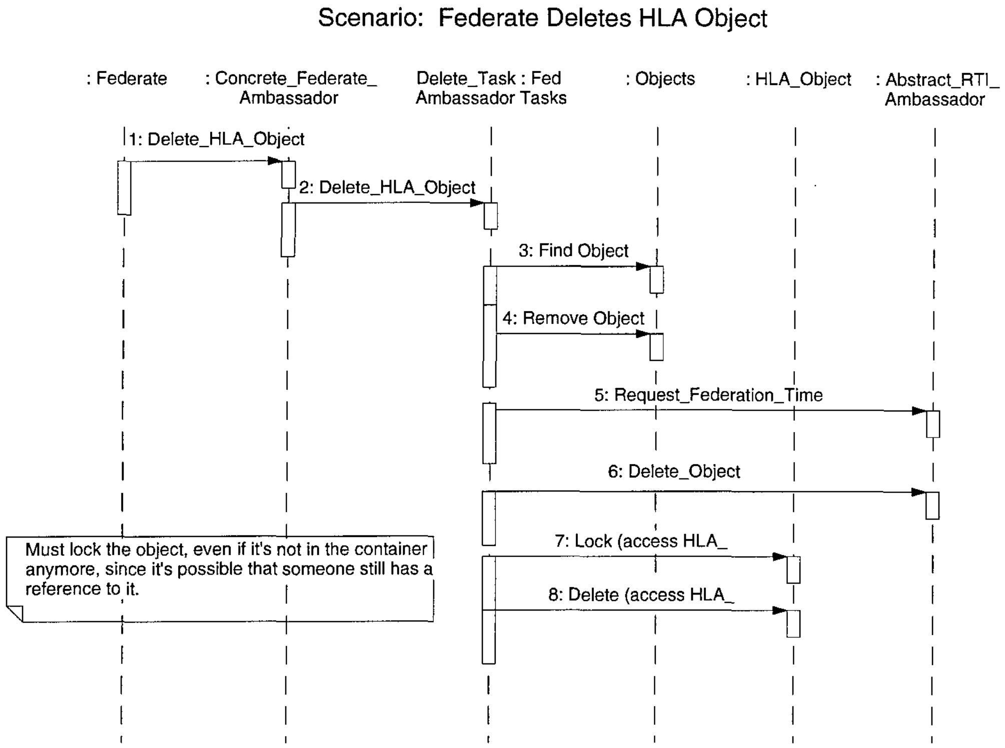

Scenario: Federate Deletes HLA Object
: Federate
: Concrete_Federate_ Ambassador
Delete_Task : Fed Ambassador Tasks
: Objects
: HLA_Object
: Abstract_RTI_ Ambassador
1: Delete_HLA_Object
2: Delete_HLA_Object
3: Find Object
4: Remove Object
5: Request_Federation_Time
6: Delete_Object
7: Lock (access HLA_
8: Delete (access HLA_
Must lock the object, even if it's not in the container anymore, since it's possible that someone still has a reference to it.

Figure 20. Scenario: Federate Deletes HLA Object

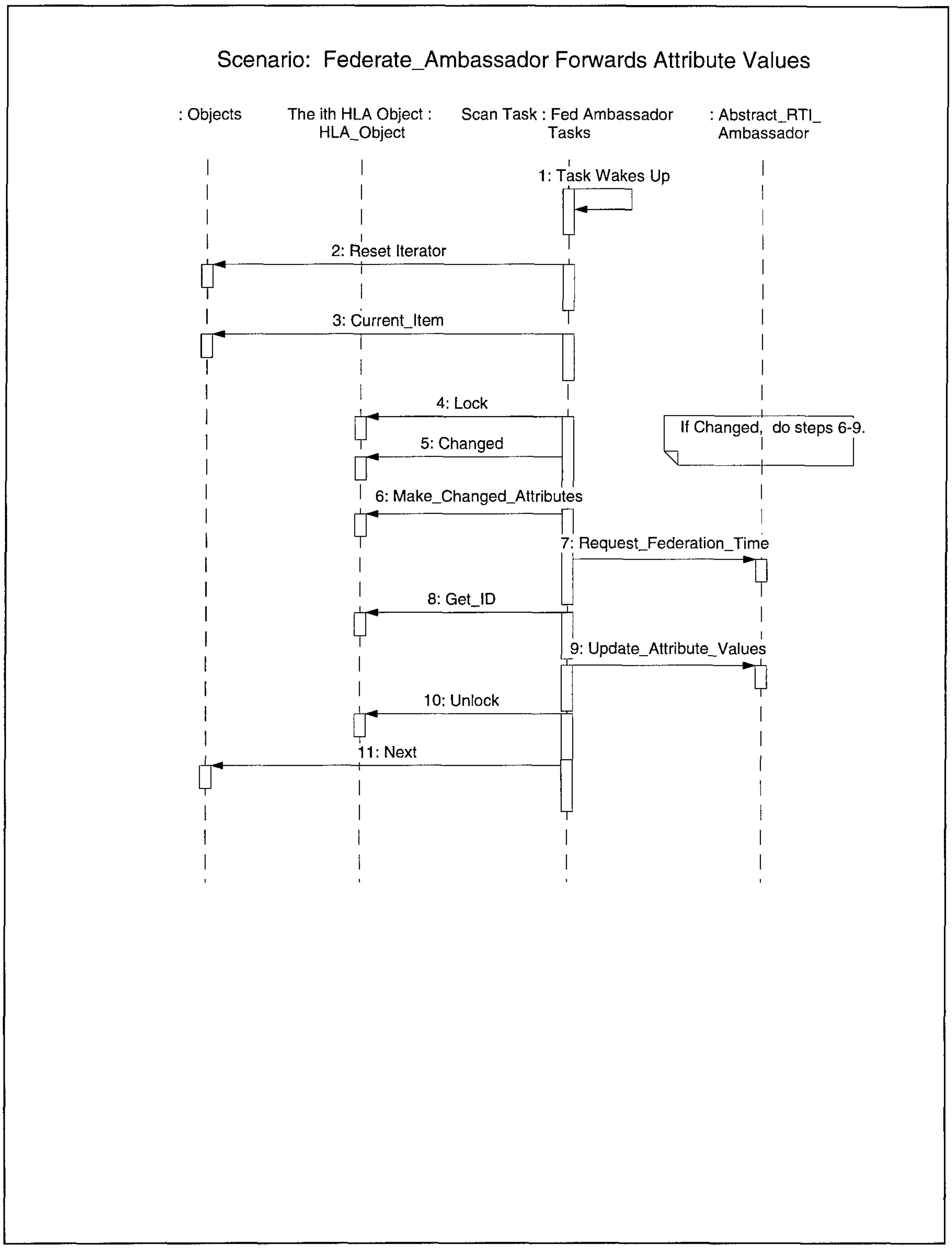

Scenario: Federate_Ambassador Forwards Attribute Values
: Objects
The ith HLA Object : HLA_Object
Scan Task : Fed Ambassador Tasks
: Abstract_RTI_ Ambassador
1: Task Wakes Up
2: Reset Iterator
3: Current_Item
4: Lock
5: Changed
If Changed, do steps 6-9.
6: Make_Changed_Attributes
7: Request_Federation_Time
8: Get_ID
9: Update_Attribute_Values
10: Unlock
11: Next

Figure 21. Scenario: Federate_Ambassador Forwards Attribute Values

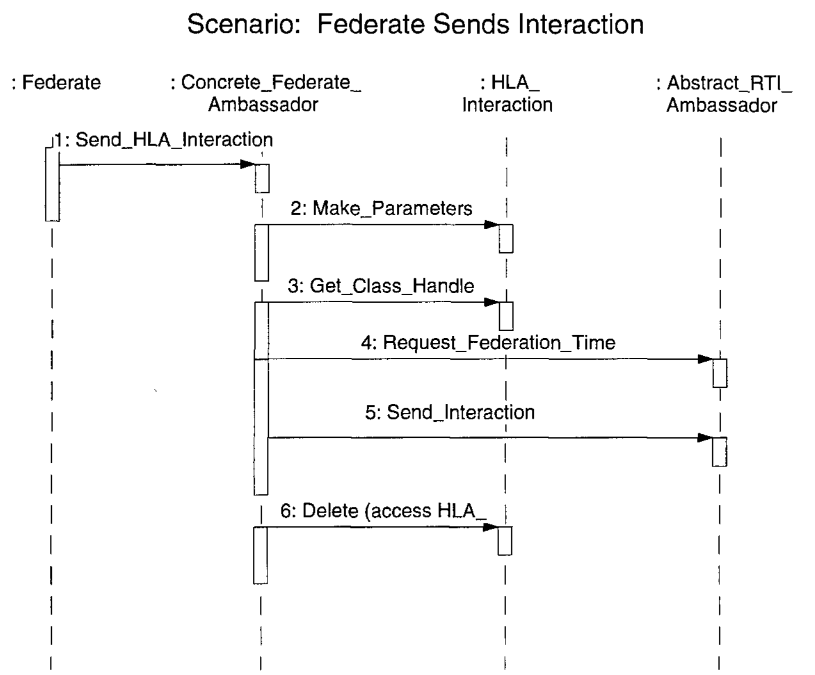


Figure 22. Federate Sends Interaction

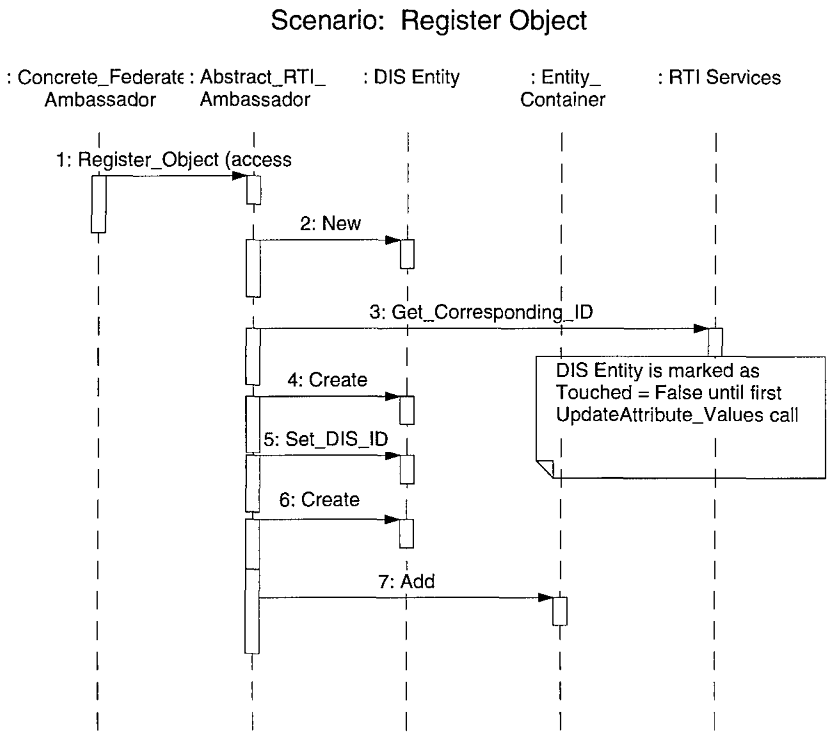
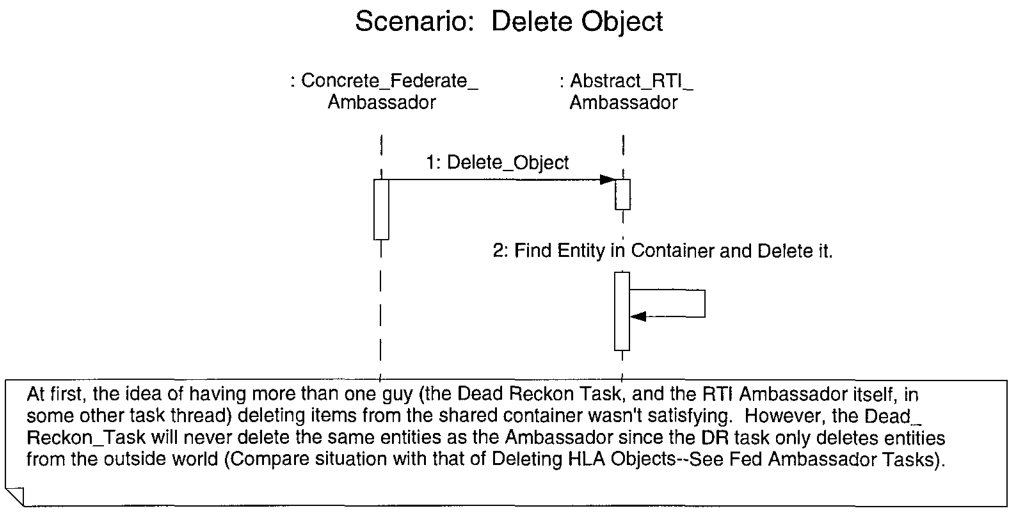

Scenario: Register Object
: Concrete_Federate Ambassador
: Abstract_RTI_ Ambassador
: DIS Entity
: Entity_ Container
: RTI Services
1: Register_Object (access
2: New
3: Get_Corresponding_ID
DIS Entity is marked as Touched = False until first UpdateAttribute_Values call
4: Create
5: Set_DIS_ID
6: Create
7: Add

Figure 23. Scenario: Register Object


Scenario: Delete Object
: Concrete_Federate_ Ambassador
: Abstract_RTI_ Ambassador
1: Delete_Object
2: Find Entity in Container and Delete it.

At first, the idea of having more than one guy (the Dead Reckon Task, and the RTI Ambassador itself, in some other task thread) deleting items from the shared container wasn't satisfying. However, the Dead_ Reckon_Task will never delete the same entities as the Ambassador since the DR task only deletes entities from the outside world (Compare situation with that of Deleting HLA Objects--See Fed Ambassador Tasks).

Figure 24. Scenario: Delete Object

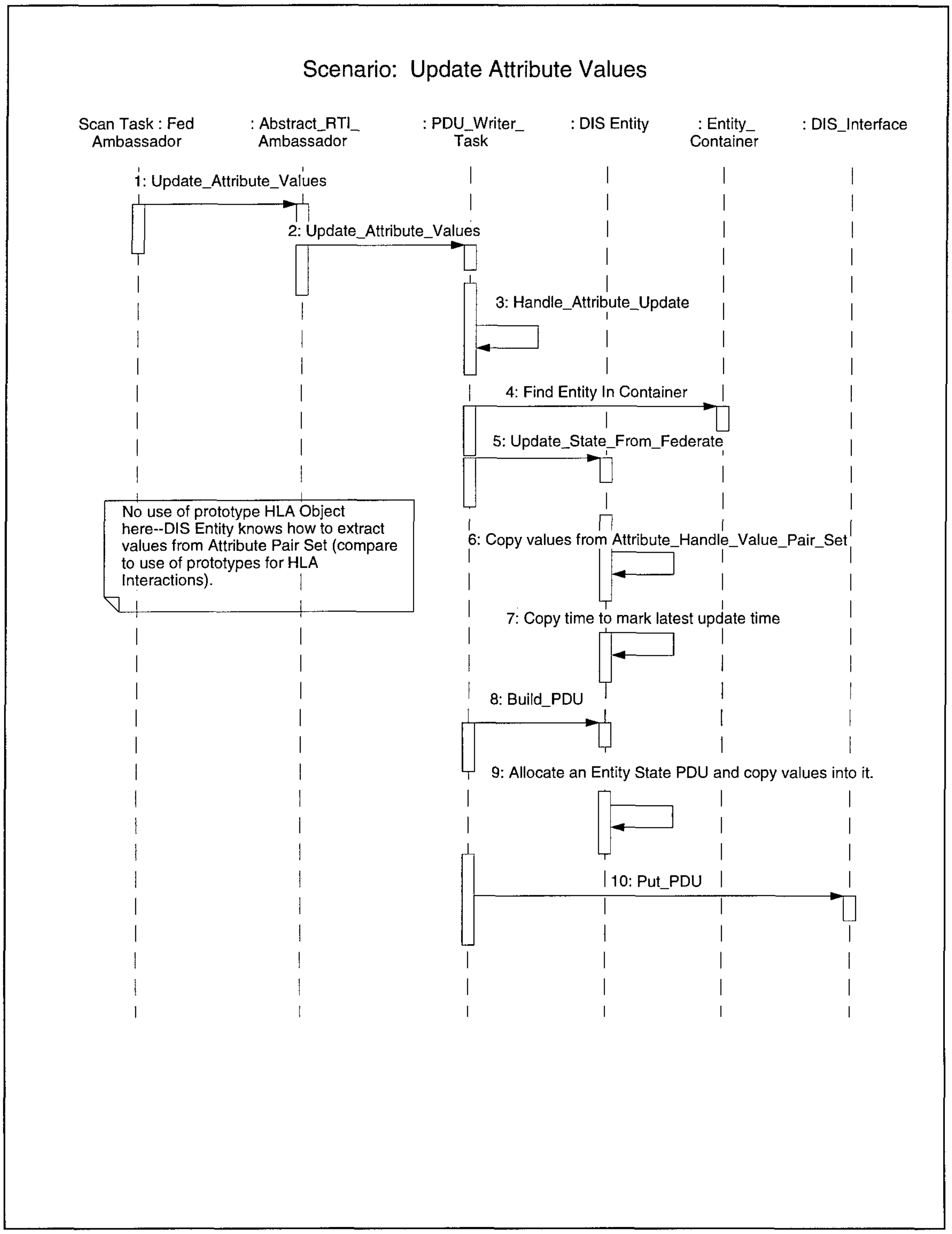

Scenario: Update Attribute Values
Scan Task : Fed Ambassador
: Abstract_RTI_ Ambassador
: PDU_Writer_ Task
: DIS Entity
: Entity_ Container
: DIS_Interface
1: Update_Attribute_Values
2: Update_Attribute_Values
3: Handle_Attribute_Update
4: Find Entity In Container
5: Update_State_From_Federate
No use of prototype HLA Object here--DIS Entity knows how to extract values from Attribute Pair Set (compare to use of prototypes for HLA Interactions).
6: Copy values from Attribute_Handle_Value_Pair_Set
7: Copy time to mark latest update time
8: Build_PDU
9: Allocate an Entity State PDU and copy values into it.
10: Put_PDU

Figure 25. Scenario Update Attribute Values

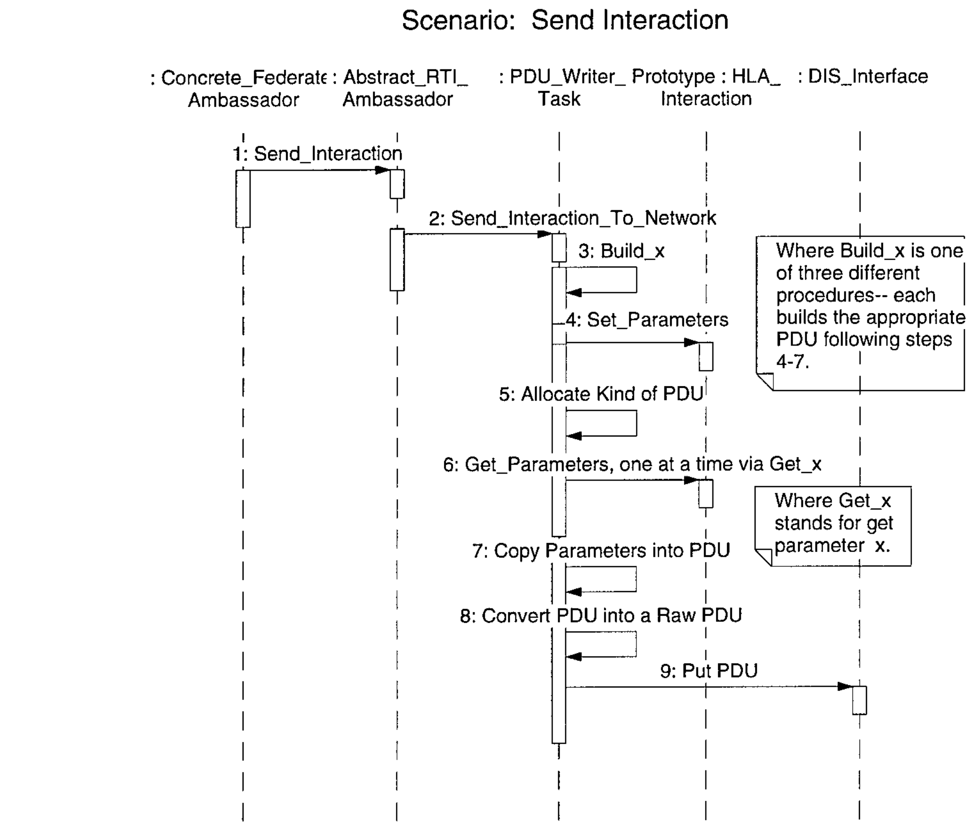

Scenario: Send Interaction
: Concrete_Federate Ambassador
: Abstract_RTI_ Ambassador
: PDU_Writer_ Task
Prototype : HLA_ Interaction
: DIS_Interface
1: Send_Interaction
2: Send_Interaction_To_Network
3: Build_x
4: Set_Parameters
5: Allocate Kind of PDU
6: Get_Parameters, one at a time via Get_x
7: Copy Parameters into PDU
8: Convert PDU into a Raw PDU
9: Put PDU
Where Build_x is one of three different procedures-- each builds the appropriate PDU following steps 4-7.
Where Get_x stands for get parameter x.

Figure 26. Scenario: Send Interaction

SimWorx Process View

The following diagram shows the Process View for the SimWorx Application Framework for Distributed Simulation in terms of Rational Rose Class Module Diagrams.

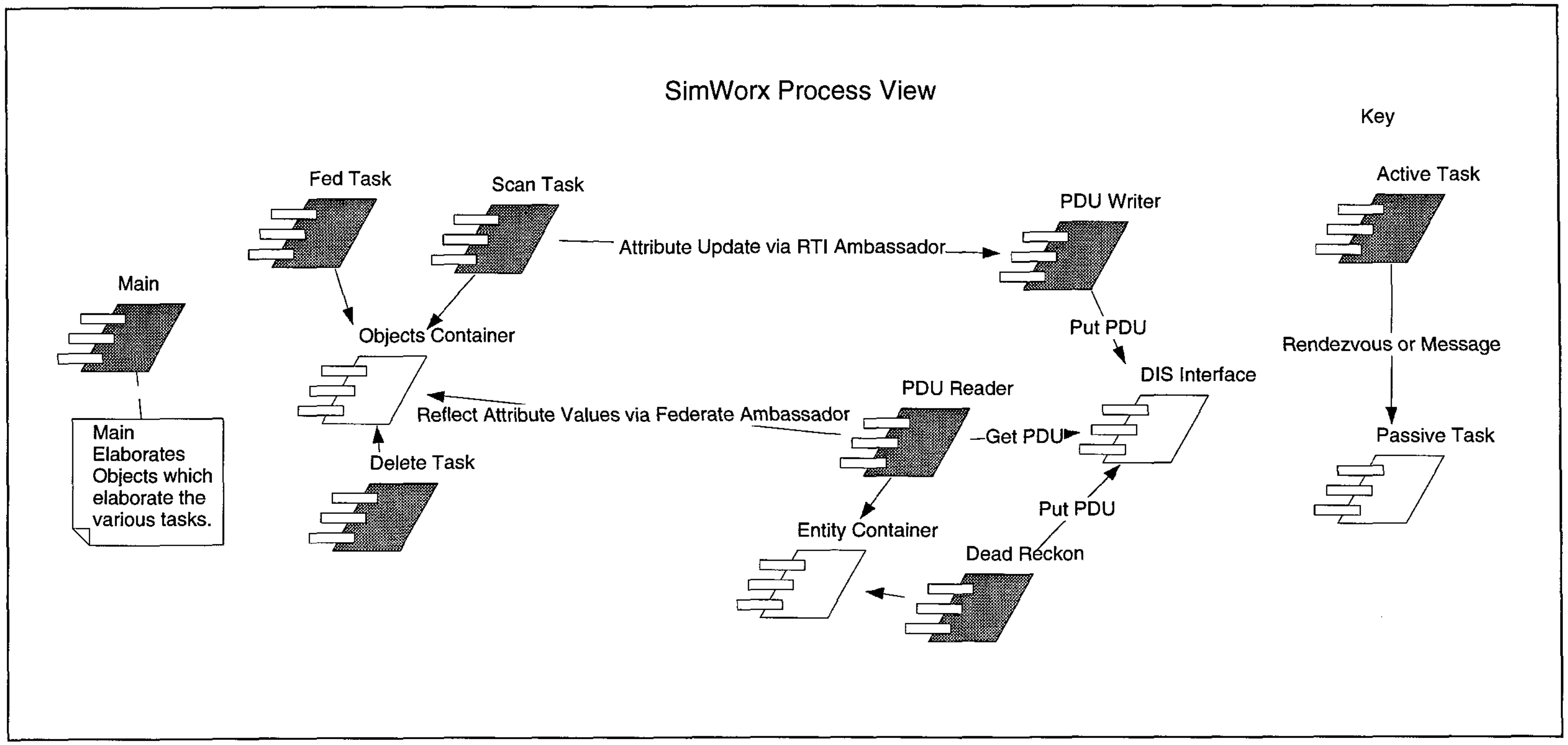


Figure 27. SimWorx Process View

SimWorx Development View

The following diagrams show the Development View for the SimWorx Application Framework for Distributed Simulation in terms of Rational Rose Module diagrams. The arrows represent package hierarchy membership, NOT package withing relationships.

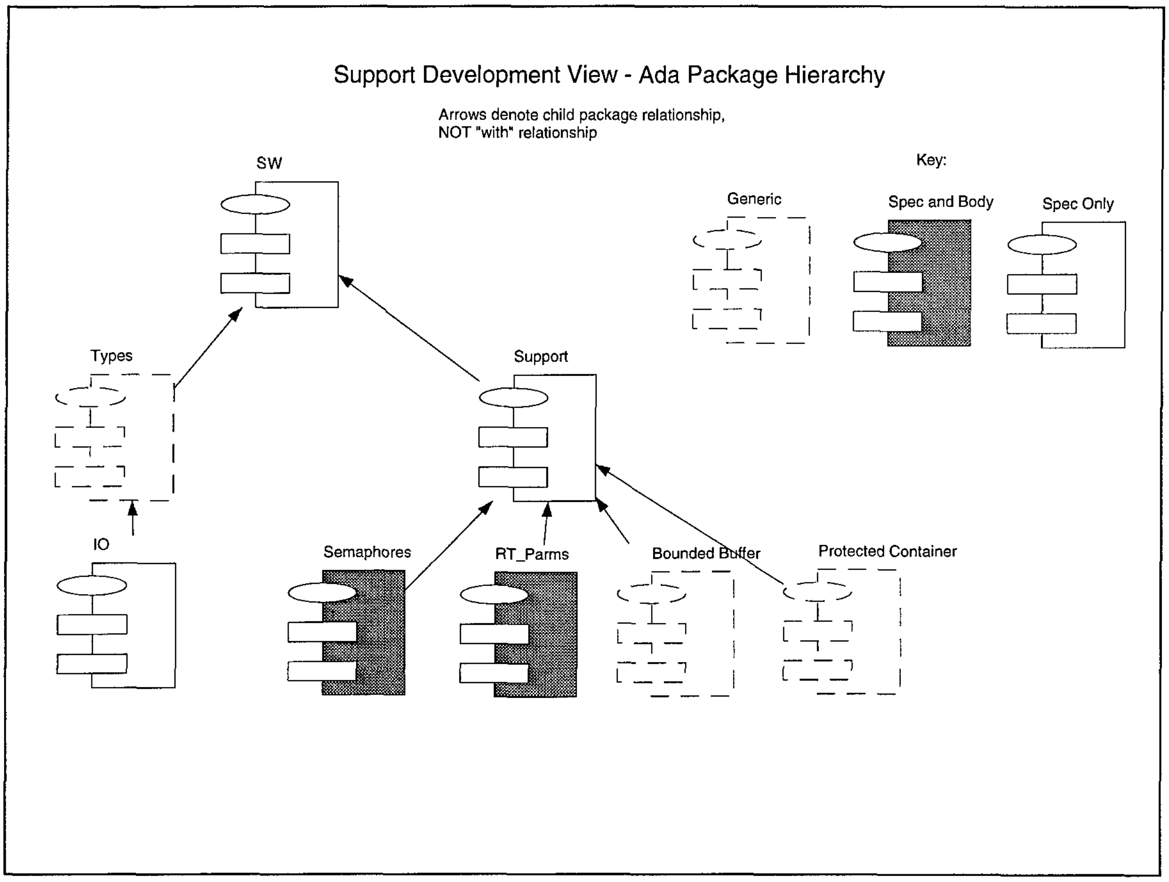

Support Development View - Ada Package Hierarchy
Arrows denote child package relationship, NOT "with" relationship
SW
Key:
Generic
Spec and Body
Spec Only
Types
Support
IO
Semaphores
RT_Parms
Bounded Buffer
Protected Container

Figure 28. Support Development View - Ada Package Hierarchy

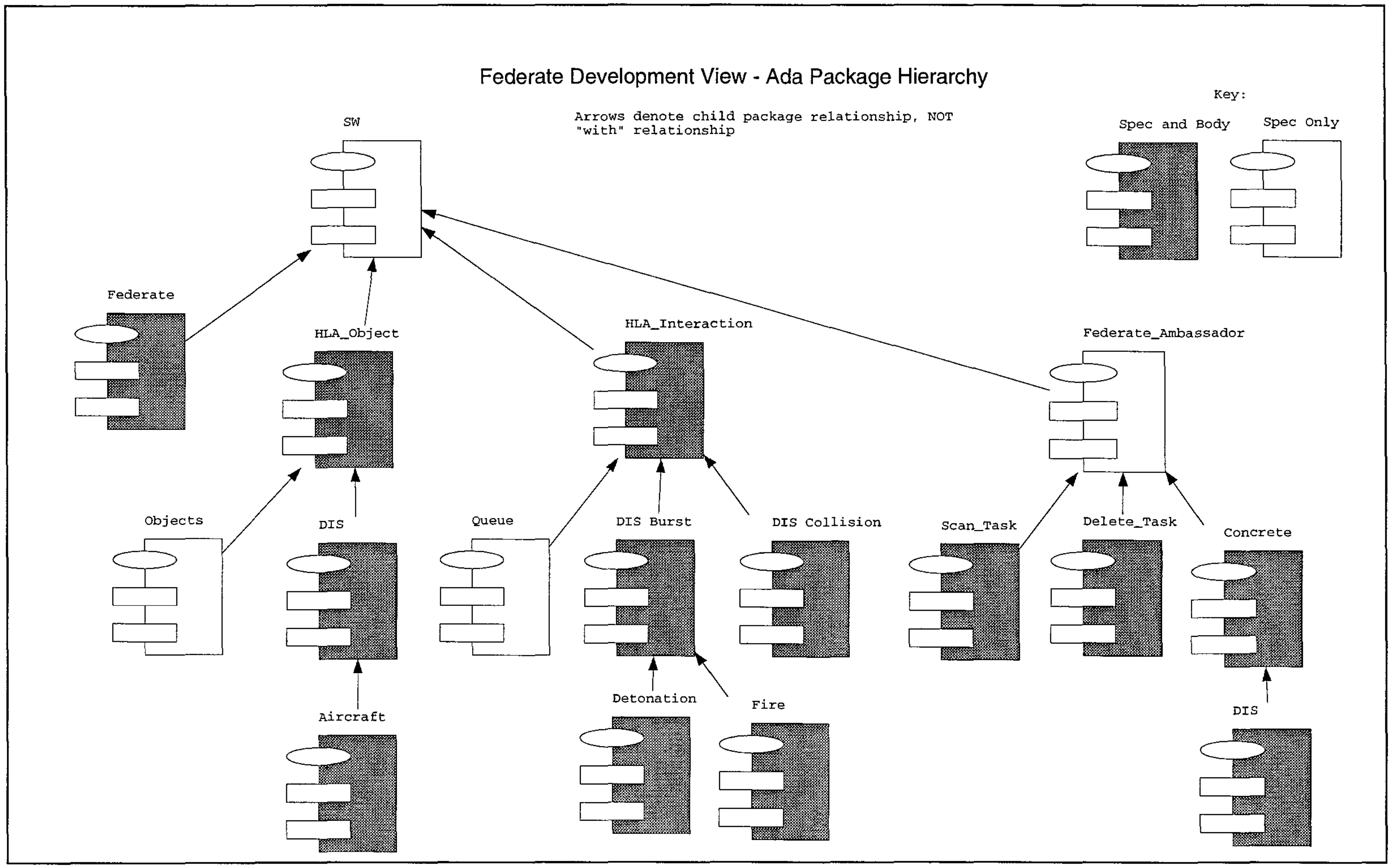


Figure 29. Federate Development View - Ada Package Hierarchy

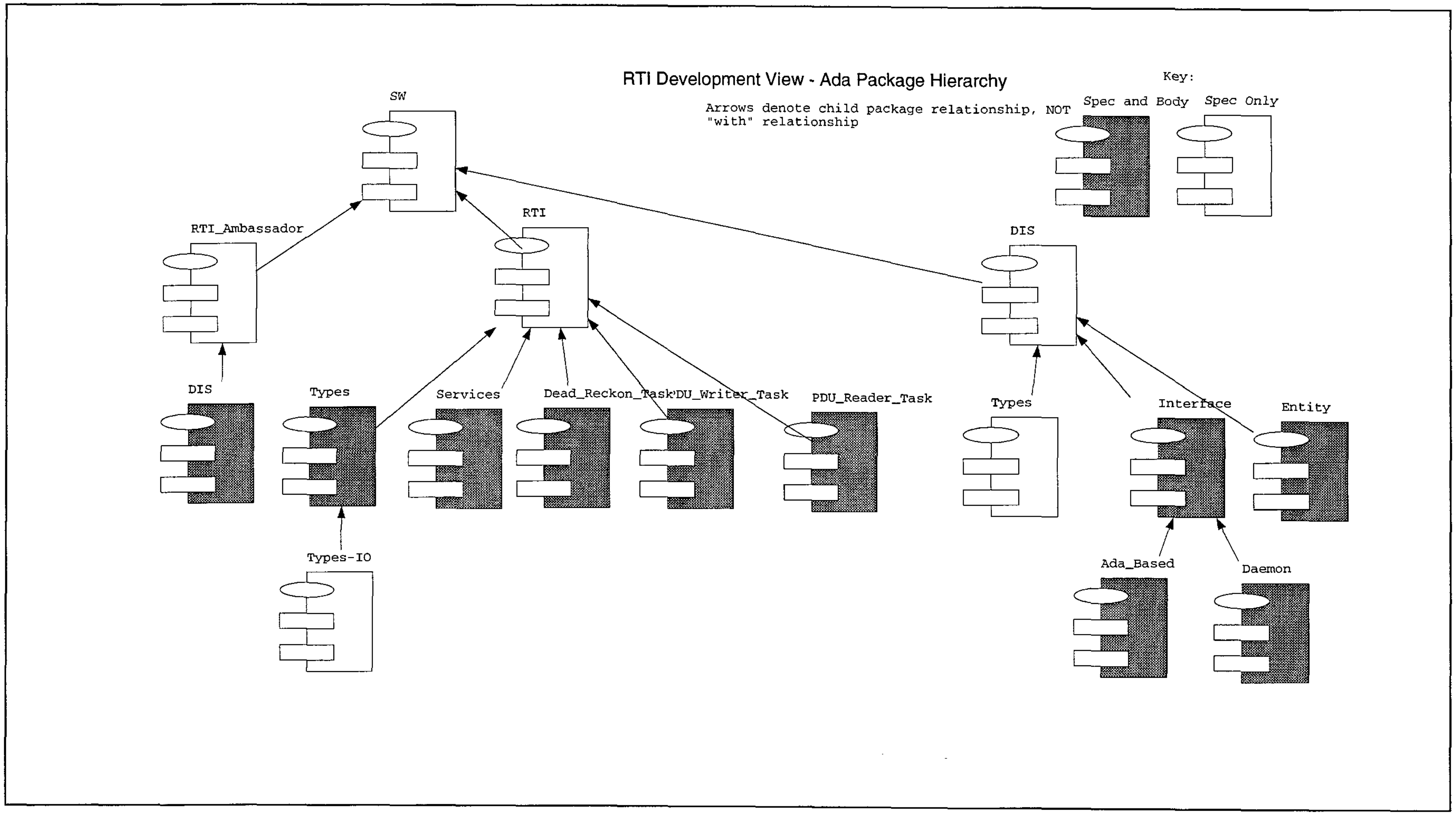
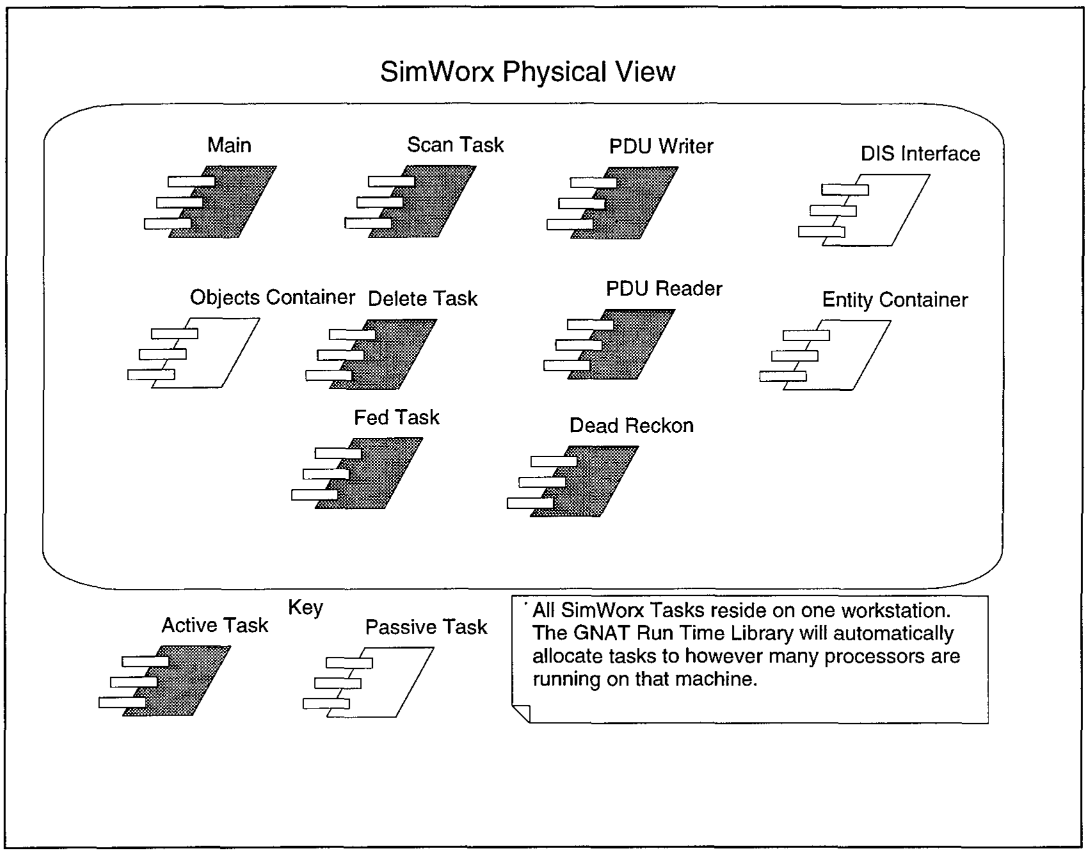


Figure 30. RTI Development View - Ada Package Hierarchy

SimWorx Physical View

The following diagram shows the Physical View for the SimWorx Application Framework for Distributed Simulation in terms of a Rational Rose Module diagram.



All SimWorx Tasks reside on one workstation. The GNAT Run Time Library will automatically allocate tasks to however many processors are running on that machine.

Figure 31. SimWorx Physical View

Classes

This section of the document describes each class of the SimWorx Application Framework.

Class name:

Abstract_Federate_Ambassador

Documentation:

This class represents the Federate's interface to the RTI. It provides abstract methods for each of the methods which Federate's are supposed to provide the RTI.

This class is abstract since Ada does not allow "circular withs". In order for the Federate Ambassador and RTI Ambassador to call each other, each needs to be defined with no knowledge of the other. Given that, the concrete Child Class of Sim Ambassador will reference the operations of the Abstract RTI Ambassador and the concrete Child Class of the RTI Ambassador will reference the operations of the Abstract Federate Ambassador.

Note that the SimWorx framework initially only supports only four of the eighteen methods specified for Federate Ambassadors in the HLA Interface Spec. These methods are the minimal necessary to operate DIS simulations under HLA (no need for advanced Declaration, Object, Ownership, or Time Management functions). The other methods should be implemented at some future date.

Superclasses:

<none>

Roles/Associations:

Association Calls

Attributes:

<none>

Operations:

Discover_Object(Instance : access Abstract_Federate_Ambassador.Object, The_Object : Object_ID, The_Object_Class : Object_Class_Handle, The_Time : Federation_Time, The_Tag : User_Supplied_Tag, The_Handle : Event_Retraction_Handle)

From section 4.4 of the HLA Interface Spec v1.0

This service informs the federate that the RTI has

discovered an object.
Raises Exceptions:
Could_Not_Discover
Object_Class_Not_Known
Invalid_Federation_Time
Federate_Internal_Error

Reflect_Attribute_Values(Instance : access Abstract_Federate_Ambassador.Object, The_Object : Object_ID, The_Attributes : Attribute_Handle_Value_Pair_Set, The_Time : Federation_Time, The_Tag : User_Supplied_Tag, The_Handle : Event_Retraction_Handle)

From section 4.5 of the HLA Interface Spec v1.0

Provides the federate with new values for a discovered attribute. This service, coupled with the Update Attribute Values Service, forms the primary data exchange mechanism supported by the RTI.

Possible Exceptions:
Object_Not_Known
Attribute_Not_Known
Invalid_Federation_Time
Federate_Internal_Error

Receive_Interaction(Instance : access Abstract_Federate_Ambassador.Object, The_Interaction : Interaction_Class_Handle, The_Parameters : Parameter_Handle_Value_Pair_Set, The_Time : Federation_Time, The_Tag : User_Supplied_Tag, The_Handle : Event_Retraction_Handle)

From section 4.7 of the HLA Interface spec v1.0

Provides information about an action taken by one federation object potentially towards another object.

Possible Exceptions:
Interaction_Class_Not_Known
Interaction_Parameter_Not_Known
Invalid_Federation_Time
Federate_Internal_Error

Remove_Object(Instance : access Abstract_Federate_Ambassador.Object, The_Object : Object_ID, The_Reason : Object_Removal_Reason, The_Time : Federation_Time, The_Tag : User_Supplied_Tag, The_Handle : Event_Retraction_Handle)

From section 4.9 of the HLA Interface Spec v1.0

This method instructs the Simulation Ambassador to remove the object specified by Object_ID since the object has been deleted from the federation execution.

Possible Exceptions:
Object_Not_Known
Invalid_Federation_Time
Federate_Internal_Error

Finalize(Instance : access Abstract_Federate_Ambassador.Object)

Provides an abstract place-holder for a finalization method to "clean up" the instantiated Federate_Ambassador upon finalization.

Class name:

Abstract_RTI_Ambassador

Documentation:

This class represents the RTI's interface to the Federate Ambassador.

Provides an abstract interface (specified by RTI IDL specifications) for different kinds of RTIs. SimWorx initially uses a DIS surrogate RTI, since AFIT does not have an HLA RTI.

See comments about Ada & "circular withs" under class Abstract_Federate_Ambassador.

Note that the SimWorx framework initially only supports only seven of the forty-six methods specified for the RTI Ambassador in the HLA Interface Spec. These methods are the minimal necessary to operate DIS simulations under HLA (no need for advanced Declaration, Object, Ownership, or Time Management functions). The other methods should be implemented at some future date.

Superclasses:

<none>

Roles/Associations:

Association Calls

Attributes:

<none>

Operations:

Join_Federation_Execution(Instance : access Abstract_RTI_Ambassador.Object, Your_Name : Federate_Name, Execution_Name : Federation_Execution_Name) : Federate_Handle

From Section 2.3 of the HLA Interface Spec v1.0.

This service affiliates the federate with the named federation execution. Calling this method indicates a federate's intention to participate in the federation.

Possible Exceptions:

Federate_Already_Execution_Member
Federation_Execution_Does_Not_Exist
Concurrent_Access_Attempted
RTI_Internal_Error
Unimplemented_Service

Request_ID(Instance : access Abstract_RTI_Ambassador.Object, ID_Count : Object_ID_Count, First_ID : access Object_ID, Last_ID : access Object_ID)

Reference section 4.1 of HLA Interface Specification v1.0

Federate requests The_Count number of new IDs to assign to objects. RTI returns IDs in range between First_ID and Last_ID.

Possible Exceptions:
Too_Many_IDs_Requested
ID_Supply_Exhausted
Federate_Not_Execution_Member
RTI_Internal_Error
Unimplemented_Service

Register_Object(Instance : access Abstract_RTI_Ambassador.Object, The_Class : Object_Class_Handle, The_Object : Object_ID)

Reference section 4.2 of HLA Interface Specification v1.0

Links an object ID with an instance of an object class. (i.e., lets Federate tell the RTI that the Federate has instantiated an object of this class.)

Possible Exceptions:
Invalid_Object_ID
Object_ID_Already_Linked
Object_Class_Not_Defined
Object_Class_Not_Published
Federate_Not_Execution_Member
RTI_Internal_Error
Unimplemented_Service

Update_Attribute_Values(Instance : access Abstract_RTI_Ambassador.Object, The_Object : Object_ID, The_Attributes : Attribute_Handle_Value_Pair_Set, The_Time : Federation_Time, The_Tag : User_Supplied_Tag) : Event_Retraction_Handle

Reference section 4.3 of HLA Interface Specification v1.0

Provides the current attribute values to the federtion for attributes owned by the federate. This service, coupled with the Reflect Attribute Values service, form the primary data exchange mechanism supported by the RTI.

Possible Exceptions:
Object_Not_Known
Attribute_Not_Defined
Attribute_Not_Owned
Invalid_Federation_Time
Federate_Not_Execution_Member
Concurrent_Access_Attempted
RTI_Internal_Error
Unimplemented_Service

Send_Interaction(Instance : access Abstract_RTI_Ambassador.Object, The_Interaction : Interaction_Class_Handle, The_Parameters : Parameter_Handle_Value_Pair_Set, The_Time : Federation_Time, The_Tag : User_Supplied_Tag) : Event_Retraction_Handle

Reference section 4.6 of HLA Interface Specification v1.0

Informs the federation of an action taken by one object, potentialy towards another object. The service returns a federation-unique retraction handle.

Possible Exceptions:
Interaction_Class_Not_Defined
Interaction_Class_Not_Published
Interaction_Parameter_Not_Defined
Invalid_Federation_Time
Federate_not_Execution_Member
Concurrent_Access_Attempted
RTI_Internal_Error
Unimplemented_Service

Delete_Object(Instance : access Abstract_RTI_Ambassador.Object, Object_ID : Object_ID, The_Time : Federation_Time, The_Tag : User_Supplied_Tag) : Event_Retraction_Handle

Reference section 4.8 of HLA Interface Specification v1.0

Informs the federation that an object with that ID, owned by the federate, is to be removed from the federation execution. Once the object is removed from the federation execution, its ID cannot be reused. THe RTI will use the Remove Object service to inform the reflecting federates that the object has been deleted.

Possible Exceptions:
Object_Not_Known
Delete_Privilege_Not_Held
Invalid_Federation_Time
Federate_Not_Execution_Member
Concurrent_Access_Attempted
RTI_Internal_Error
Unimplemented_Service

Request_Federation_Time(Instance : access Abstract_RTI_Ambassador) : Federation_Time

Reference section 6.1 of HLA Interface Spec v1.0

Provides federate with current federation time.

Possible Exceptions:
Federate_Not_Execution_Member
RTI_Internal_Error
Unimplemented_Service

Finalize(Instance : access Abstract_RTI_Ambassador.Object)

Clients call this method prior to deallocating the RTI Ambassador. It is responsible for deallocating and cleaning up internal objects.

Class name:

Ada_Based

Documentation:

This class provides a native Ada network interface for reading and writing DIS PDUs (as opposed to using Bruce Clay's C-language Daemons).
Since this is a polymorphic child class of the DIS_Interface, it implements the same methods defined by that class.

It elaborates a Socket_Reader_Task to continuously read PDUs from the socket and put them in an input buffer, and a Socket_Writer_Task to continuously get PDUs from an output buffer and write them to the socket.

Calls to the Get and Put methods get and put PDUs from/to the buffers.

Superclasses:

DIS_Interface

Roles/Associations:

Association Allocates_PDU_Buffers
Association Elaborates_Socket_Tasks

Attributes:

<none>

Operations:

<none>

Class name:

Bounded Buffer

Documentation:

This class is a generic bounded buffer. It contains up to Max_Size items of Component_Type. It is essentially copied from Burns & Wellings "Concurrency in Ada", Cambridge, ISBN 0-521-41471.

The buffer provides buffering between a producer and a consumer. It's implemented as a protected object containing a simple array of size Max_Size.

Superclasses:

<none>

Roles/Associations:

<none>

Attributes:

Store : Buffer_Array
The array of Component_Type items.

First : an integer type
The index of the first valid item in the array.

Last : an integer type
The index of the last valid item in the array.

Current_Size : Count_Range
The current numbe of items in the buffer.

Operations:

Get(Item : Item_Type)
Put(Item : Item_Type)

Concrete_Federate_Ambassador

Documentation:

This class is the concrete Federate Ambassador. It exists for several reasons.

1. Created to resolve the Ada "circular with" problem (see Abstract Federate Ambassador class for more details).
2. It provides common services for the Federate and HLA Interactions.

3. Most Important: The philosophy of SimWorx is to reduce client programmer workload by "automatically" handling the dirty work of HLA simulations--the Concrete_Federate_Ambassador is intended to be the doer of much of the Dirty Work (like Federate Saves, Pauses, Restores, ect. to be implemented later).

This class overrides all of the methods of the Abstract_Federate_Ambassador, and adds the Create initialization method, and two abstract factory methods, and several services methods.

The abstract Factory methods raise exceptions if they are called, since the Concrete_Federate_Ambassador is not meant to be used as-is, it's meant to be overridden.

Superclasses:

Abstract_Federate_Ambassador

Roles/Associations:

Association Shares_Object_Container
Association Shares_Interaction_Queue
Association Has_Ambassador
Association Calls
Association Has Fed Ambassador Tasks

Attributes:

RTI_Amb : Abstract_RTI_Ambassador.Reference

This is an Ada classwide type which refers to the Federate_Ambassador's associated RTI. It's necessary to be able to dispatch method calls to the RTI.

The_Handle : Federate_Handle

This handle represents the ID of the Federate (and thus, it's associated Federate_Ambassador) during a federation execution. The Federate Ambassador receives the handle from the RTI via the Join_Federation_Execution method.

Operations:

Create(Instance : access Concrete_Federate_Ambassador, RTI : Abstract_RTI_Ambassador.Reference, Objects : Objects.Reference, Interaction_Queue : Interaction_Queue.Reference, Success : Boolean)

This method initializes the Concrete Federate Ambassador following allocation. It sets it's instance variables to the values passed from the Federate since the Federate is responsible for allocating and initializing the values. It elaborates the Scan and Delete Tasks. Then, it invokes the Join_Federation_Execution method to get the Federate's Handle for this execution.

Create_HLA_Object(Instance : access Concrete_Federate_Ambassador.Object, The_Object_Class : Object_Class_Handle) : HLA_Object.Reference

Creates an HLA_Object, initializes it, and puts it in the container. (Client is from the federate side, not the RTI). Enables HLA_Objects to create other HLA_Objects (like when an aircraft launches a missile), and provides a central point for HLA_Object creation within the Federate.

Delete_HLA_Object(Instance : access Concrete_Federate_Ambassador.Object, The_Object : Object_ID)

Deletes the HLA Object specified from the Objects container (via the Delete Task). Note this is different from the Remove_Object method. This method is meant to be called by Federate Clients, not from the RTI side of the house.

Send_HLA_Interaction(Instance : access Concrete_Federate_Ambassador.Object, The_Interaction : HLA_Interaction.Reference)

Send the HLA Interaction to the RTI. Builds Parameter Pair List via HLA_Interaction's Make_Parameters method, then sends list to RTI. This method is meant to be called by Federate clients, rather than from the RTI.

Clock(Instance : access Concrete_Federate_Ambassador.Object) : Federation_Time

Provides Federate with a centralized time source.

Objects_Reference(Instance : access Concrete_Federate_Ambassador.Object) : Objects.Reference

Provides clients within the Federate a reference to the shared, protected, Objects container of HLA_Objects. Enables objects to find and interact with each other.

HLA_Object_Factory_Method(Instance : access Concrete_Federate_Ambassador.Object, Object_Class : Object_Class_Handle) : HLA_Object.Reference

This method is a parameterized factory method which returns a tagged reference to an HLA_Object of the class corresponding to the Object_Class parameter. This method is meant to be overridden by a child class.

HLA_Interaction_Factory_Method(Instance : access Concrete_Federate_Ambassador.Object, Interaction_Class : Interaction_Class_Handle) : HLA_Interaction.Reference

Does for HLA_Interactions what the HLA_Object_Factory_Method does for Objects.

Class name:

Daemon_Based

Documentation:

This class is the C-based DIS network interface. Is interfaces with Bruce Clay's (SAIC Corp.) PDU Daemons. See comments in the DIS_Interface class. This class provides protected access to Getting PDUs by wrapping the calls to dsi_user within an Ada protected object.

This class overrides the Create, Finalize, Get_PDU, and Put_PDU methods of the abstract DIS Interface Class.

Superclasses:

DIS_Interface

Roles/Associations:

<none>

Attributes:

<none>

Operations:

<none>

Class name:

Dead_Reckon_Task

Documentation:

This task repeatedly dead-reckons the entities in the container--it "wakes up" periodically (as determined by the Dead-Reckon iteration rate parameter in the Run Time parameters file) and scans through the container, dead-reckoning each entity.

For each local entity dead reckoned, if the PDU Reference returned from the Dead_Reckon_Local method is non-null, then the Dead_Reckon_Task calls the Put method of the DIS Interface to send the PDU update to the network.

After each non-local entity is dead-reckoned, the Dead_Reckon_Task projects follows up with a call to the Entity to project it's values to the Federate, as long as the entity hasn't become stale. Stale entities are removed from the container.

The exception Program_Error may be raised by the protected container or by a protected DIS Entity if the object is deleted while this task is waiting to use it. Thus, Program_Errors are simply handled in the container loop iteration by doing nothing.

Superclasses:

<none>

Roles/Associations:

Association Has_Dead_Reckon_Task
Association Shares
Association Writes_PDUs

Attributes:

<none>

Operations:

Elaborate_With_Discriminants(Fed_Ambassador : Abstract_Federate_Ambassador.Reference, Entity_Container : Entity_Container.Reference, DIS_Interface : DIS_Interface.Reference)

This method represents elaborating the task with discriminants. The Federate_Ambassador Reference is necessary to enable DIS Entities to call the methods of the Federate_Ambassador. The DIS_Interface reference enables the task to call the Put_PDU method of the DIS_Interface. The task uses the Entity

Container reference to access the shared entity container.

Dead_Reckon_Entities( )

This method dead-reckons each entity in the Entity_Container. It is called repeatedly by the Dead_Reckon_Task.

```
For each entity:
 if it's local
   Dead-Reckon_Local the entity
   Write Entity State PDU update if necessary
 else (non local, i.e., came from net)
   if it's stale,
     remove it from the container,
     tell the Federate to delete the corresponding object,
     delete the entity.
   if it's not stale,
   then tell the entity to project the updated values to the Federate.
```

Kill( )

Kill the Dead_Reckon_Task.

Class name:

DIS Entity

Documentation:

This class represents an active DIS Entity of a DIS Simulation. It is implemented as an Ada protected object so concurrent operations are performed atomically.

Entities are modeled here in the RTI Ambassador principally because of the DIS requirement to Dead Reckon them. Dead Reckoning requirements are slightly different between entities of the local simulation (federate) and entities of external simulations (other federates).

Although the methods of the RTI which deal with HLA Interaction parameters use prototype HLA_Interactions to encapsulate knowledge of the interaction's parameters, the DIS Entity does not use a prototype HLA DIS Entity Object to manipulate attribute values. Why? Because DIS Entities and DIS Entity Objects are intimately tied together. There's no value in hiding one's representation from the other EXCEPT in minimizing future maintenance. So, in the interests of execution speed, the DIS Entity employs a copy of the code in the DIS Entity Object in order to manipulates attribute pair sets. (See Update_State_From_Federate and Dead_Reckon methods).

Superclasses:

<none>

Roles/Associations:

Association Has_Entities

Attributes:

Local : Boolean = False

Indicator of whether this DIS Entity is modeled locally, or in some other simulator. Necessary since Dead Reckoning is different for local entities, since the RTI Ambassador must send updates to the network when entity attribute thresholds are exceeded.

Touched : Boolean = False

An indicator which indicates whether the DIS Entity has been assigned instance values. DIS

Entities can't be Dead_Reckoned until they've had their instance values updated from the network or from the Federate Ambassador.

Fed : Federate_Ambassador.Reference = null
Each DIS Entity requires a reference to the Federate Ambassador so it can make the project HLA Object attributes for DIS Entity Objects to the Federate.

ID : Entity_Identifier_Record
This is the Entity's DIS ID as defined by DIS 2.0.4 paragraph 5.3.14.

Force_ID : Nat8
The Entity's Force ID as defined by DIS 2.0.4 standard paragraph 5.3.21.

Entity_Type : Entity_Type_Record
DIS Entity type as defined by DIS Standard paragraph 5.3.16.

Marking : Entity_Marking_Record
DIS Entity Marking as defined by DIS standard 2.0.4 paragraph 5.3.15.

Dead_Reckoning_Parms : Dead_Reckoning_Parameters_Record
DIS DR Parameters as defined by DIS standard 2.0.4 paragraph 5.4.3.1.11.

Appearance : Nat32
The DIS Entity's appearance as of the last PDU received, as defined by DIS Standard paragraph 5.3.12.

Position_At_Update : World_Coordinates_Record
The Entity's coordinates from the last PDU received. As defined by DIS Standard 2.0.4 paragraph 5.3.33.

Orientation_At_Update : Euler_Angles_Type
The Euler Angles (psi, theta, phi) defining the DIS Entity's orientation as of the last PDU received. As defined by DIS standard paragraph 5.3.1.

Linear_Velocity : Linear_Velocity_Record
The Linear Velocity of the DIS Entity as of the last PDU update. As defined by DIS standard paragraph 5.3.32.

Linear_Acceleration : Linear_Acceleration_Record
The entity's linear acceleration at the last PDU update, as defined by DIS Standard 2.0.4 paragraph 5.3.32.

Angular_Velocity : Angular_Velocity_Record

The DIS Entity's angular velocity as of the last PDU update. As defined by DIS standard paragraph 5.3.2.

Location : World_Coordinates_Record

The Entity's current coordinates. As defined by DIS Standard 2.0.4 paragraph 5.3.33.

Orientation : Euler_Angles_Type

The Euler Angles (psi, theta, phi) defining the DIS Entity's current orientation. As defined by DIS standard paragraph 5.3.1.

WW_ID : Object_ID = 0

This is the Entity's HLA Object ID, provided by the RTI.

Last_PDU_Stamp : DIS_Time_Stamp

The DIS Time Stamp from the PDU Header Record. This is the time the PDU was issued.

WB_Matrix : Flt32_Matrix

This matrix is the World Coordinates to Body Coordinates orientation matrix from section B3.6.1 of the DIS Standard.

It's updated with each new PDU received for entities employing Dead Reckoning models using orientation: RPW, RVW, FPB, RPB, RVP, FVB (see section B3.6.1).

Roll_Squared : Flt32

The entity's roll, squared. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Pitch_Squared : Flt32

The entity's pitch, squared. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Yaw_Squared : Flt32

The entity's yaw, squared. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Roll_Pitch : Flt32

The entity's roll * pitch. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Roll_Yaw : Flt32

The entity's roll * yaw. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Pitch_Yaw : Flt32

The entity's pitch * yaw. This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update.

Omega_Magnitude_Squared : Flt32

This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update. See paragraph B3.6.1.1

Omega_Magnitude : Flt32

This value is calculated once with each new PDU update, and then used in each dead-reckoning cycle until the next update. See paragraph B3.6.1.1

Last_PDU_Received : Federation_Time

The time the last PDU was received by the DIS_Surrogate_RTI_Ambassador.

DR_Matrix : Flt32_Matrix

The [DR] Matrix from section B3.6.1 of the DIS standard. It's updated with each new PDU and with each dead-reckoning cycle.

New_WB_Matrix : Flt32_Matrix

The [R]w->B matrix from section B3.6.1 of the DIS standard. It's updated with each new PDU and with each dead-reckoning cycle.

Last_PDU_Built : Federation_Time

Marks the time when the last PDU was built. Used to keep track of how long it's been since we broadcast this entity's state to the network.

Operations:

Create(Federate : Federate_Ambassador.Reference, New_ID : Object_ID, Is_Local : Boolean)

Performs initialization of an allocated DIS Entity.

Set the Entity's Federate Ambassador reference used to dispatch calls to the Federate Ambassador.

Set the Entity's WW HLA Object ID (the DIS ID is either set with a different method call, Set_DIS_ID, for local entities, or from an Entity State PDU, for non-local entities.

Set the Entity's Locality (a boolean). Set True if this entity represents an object of the local is not derivable like a tagged type, and thus can't be overridden. But on the plus side, at least the semantics of the Ada 95 language weren't complicated by this feature (dripping sarcasm).

Build_PDU(The_Time : Federation_Time, Output_PDU : Entity_State_PDU_Ref)

Allocate an Entity State PDU and copy the Entity's state information into it. We're relying on the DIS Interface to deallocate the PDUs storage.

Copies The_Time to Last_PDU_Built to indicate when the last PDU was built and sent to the network.

Dead_Reckon(Current_Time : Federation_Time, Is_Stale : Boolean)

Dead_Reckon the Entity based upon it's dead-reckoning parameters (but don't do anything unless this entity has been touched, see Touched attribute). If the max time between updates has been exceeded, then return Is_Stale = True to indicate the entity has become stale.

Dead_Reckon_Local(Current_Time : Federation_Time, Output_PDU : Entity_State_PDU_Ref)

Dead_Reckon the local entity based upon it's dead-reckoning parameters (but don't do anything if this entity hasn't been touched yet).

If the entity's position or orientation has changed by more than a certain threshold (1 meter for position, 3 degrees for orientation, ref DIS Standard paragraph 4.4.2.1.2.1), or it's been a while without an update (must broadcast an Entity State PDU at least every 5 seconds, ref DIS Standard paragraph 4.4.2.1.3(3).), then call Build_PDU to generate a PDU for broadcast to the network.

If no PDU has been generated, then null is returned in the Output_PDU parameter.

NOTE: The DR_Hertz rate must be greater than 12 DR cycles/minute to ensure an Entity State PDU is broadcast at least every 5 seconds.

Print( )

Print the current values of the Entity's attributes on standard output.

Project_All_Attributes(Current_Time : Federation_Time, Federate : Abstract_Federate_Ambassador.Reference)

Send the current attribute values to the Federate via the Reflect_Attribute_Values method.

Project_Changing_Attributes(Current_Time : Federation_Time, Fed_Ambassador : Abstract_Federation_Ambassador.Reference)

Send the attributes which change during each dead-reckoning cycle to the Federate via the Reflect_Attribute_Values method.

Propagate_Discovered_Object(Current_Time : Federation_Time, Fed_Ambassador : Abstract_Federate_Ambassador.Reference)

Tell the Federate about this new entity. Contains both Discover_Object and Reflect_Attribute_Values method calls.

Delete(Instance : DIS_Entity.Reference)

Delete the DIS Entity.

Class name:

DIS Entity Object

Documentation:

This class is a DIS Entity which is an HLA Object. It is a "ready-made" HLA object for use with DIS-based simulations (it contains all of the attributes of a DIS Simulation Entity). Client programmers will subclass this object to create specific DIS entities such as F-15E aircraft and M-1A tanks.

It overrides all of the dispatching methods of HLA_Object (i.e., the ones that aren't class-wide, the ones that don't have a parameter of type Reference).

Superclasses:

HLA_Object

Roles/Associations:

<none>

Attributes:

Entity_ID : Entity_Identifier_Record
This is the DIS Entity ID. Reference DIS Standard 2.0.4 paragraph 5.3.14

Force_ID : Nat8
This is the DIS Force ID, as specified by DIS Standard 2.0.4 paragraph 5.3.21

Entity_Type : Entity_Type_Record
DIS Entity type as defined by DIS Standard paragraph 5.3.16.

Marking : Entity_Marking_Record
DIS Entity Marking as defined by DIS Standard 2.0.4 paragraph 5.3.15.

Appearance : Nat32
DIS Appearance as defined by DIS Standard 2.0.4 paragraph 5.3.12.

Position : World_Coordinates_Record
The DIS Entitiy's coordinates as defined by DIS Standard 2.0.4 paragraph 5.3.33.

Orientation : Euler_Angles_Type

The Euler Angles (psi, theta, phi) defining the DIS Entitiy's orientation as specified by DIS Standard 2.0.4 paragraph 5.3.1.

DR_Parms : Dead_Reckoning_Parameters_Record

Specifies the Dead Reckoning Parameters simulations will employ to Dead Reckon this entity. See the DIS Standard for a full description of Dead Reckoning Parameters.

Linear_Velocity : Linear_Velocity_Record

The Entity's current linear velocity.

Angular_Velocity : Angular_Velocity_Record

The Entity's current angular velocity.

Linear_Acceleration : Linear_Acceleration_Record

The entity's current Linear Acceleration.

Objects : Objects.Reference

A reference to the Objects container of HLA Objects. Enables DIS Entities to iterate through the container (look for enemy planes in range, etc.).

This attribute is set in the overridden DIS Entity Object's Create method via a call to the Federate Ambassador.

Operations:

Set_Attribute_X(Instance : DIS_Entity_Object.Reference, New_Value : Attribute Value)

DIS Entity Object provides a Set operation for each of it's attributes.

Get_Attribute_X(Instance : DIS_Entity_Object.Reference) : Attribute Value

DIS Entity Object provides a Get operation for each of it's attributes.

Class name:

DIS Fire

Documentation:

This class represents a DIS Fire interaction between two DIS Entity Objects.

It overrides the Set_Parameters, Make_Parameters, Show, and Delete methods of its parent.

Superclasses:

DIS_Burst

Roles/Associations:

<none>

Attributes:

Fire_Range : Flt32
The range of the Fire action.

Fire_Mission_Index : Nat32
The mission index of the fire action.

Operations:

Create(Instance : access DIS_Fire.Object, Class : Interaction_Class_Handle, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag, Event_Retraction_Handle : Event_Retraction_Handle, Initiator : Object_ID, Receiver : Object_ID, Munition_ID : Entity_Identifier_Record, Fire_Event_ID : Event_ID_Record, Location : World_Coordinates_Record, Burst_Descriptor : Burst_Descriptor_Record, Fire_Velocity : Linear_Velocity_Record, Fire_Range : Flt32, Fire_Mission_Index : Nat32)
Initialize a DIS Fire interaction following allocation.

Set_Fire_Range(Instance : access DIS_Fire.Object, New_Value : Flt32)
Set the DIS_Fire interaction's fire range.

Get_Fire_Range(Instance : access DIS_Fire.Object) : Flt32
Get the DIS_Fire Interaction's fire range.

Set_Fire_Mission_Index(Instance : access DIS_Fire.Object, New_Value : Nat32)
Set the DIS_Fire Interaction's mission index.

Get_Fire_Mission_Index(Instance : access DIS_Fire.Object) : Nat32
Get the DIS_Fire Interaction's mission index.

Class name:

DIS_Burst

Documentation:

This class results from factoring the common elements out of the DIS Fire and DIS Detonation subclasses.

It overrides the Set_Parameters, Make_Parameters, Show, and Delete methods of its parent.

Superclasses:

HLA_Interaction

Roles/Associations:

<none>

Attributes:

Initiator : Object_ID
The DIS Entity doing the firing.

Receiver : Object_ID
The DIS Entity being fired upon.

Munition_ID : Entity_Identifier_Record
An Entity_Identifier_Record which refers to the munition being fired (if it is a "tracked" munition, record is zeros otherwise.)

Fire_Event_ID : Event_ID_Record
The DIS event id of the fire interaction.

Location : World_Coordinates_Record
The location from which the munition was fired.

Burst_Descriptor : Burst_Descriptor_Record
Interaction's burst descriptor.

Fire_Velocity : Linear_Velocity_Record
The initial velocity of the munition.

Operations:

Create(Instance : access DIS_Burst.Object, Class : Interaction_Class_Handle, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag, Event_Retraction_Handle : Event_Retraction_Handle, Initiator : Object_ID, Munition_ID : Entity_Identifier_Record, Fire_Event_ID : Event_ID_Record, Location : World_Coordinates_Record, Burst_Descriptor : Burst_Descriptor_Record, Fire_Velocity : Linear_Velocity_Record)

Initialize a DIS_Burst object following allocation.

Set_Munition_ID(Instance : access DIS_Burst.Object, New_Value : Entity_Identifier_Record)
Set the DIS Burst's munition id.

Get_Munition_ID(Instance : access DIS_Burst.Object) : Entity_Identifier_Record
Get the DIS_Burst's munition id.

Set_Fire_Event_ID(Instance : access DIS_Burst.Object, New_Value : Event_ID_Record)
Set the DIS_Burst's fire event id.

Get_Fire_Event_ID(Instance : access DIS_Burst.Object) : Event_ID_Record
Get the DIS_Burst's fire event id.

Set_Location(Instance : access DIS_Burst.Object, New_Value : World_Coordinates_Record)
Set the DIS_Burst's location.

Get_Location(Instance : access DIS_Burst.Object) : World_Coordinates_Record
Get the DIS_Burst's location.

Set_Burst_Descriptor(Instance : access DIS_Burst.Object, New_Value : Burst_Descriptor_Record)
Set the DIS_Burst's descriptor.

Get_Burst_Descriptor(Instance : access DIS_Burst.Object) : Burst_Descriptor_Record
Get the DIS_Burst's descriptor.

Set_Fire_Velocity(Instance : access DIS_Burst.Object, New_Value : Linear_Velocity_Record)
Set the DIS_Burst's initial velocity.

Get_Fire_Velocity(Instance : access DIS_Burst.Object) : Linear_Velocity_Record
Get the DIS_Burst's initial velocity.

Class name:

DIS_Collision

Documentation:

This class represents a DIS Collision interaction between two DIS Entity Objects. The DIS Collision is described in paragraph 4.4.5 of the DIS Standard.

It overrides the Set_Parameters, Make_Parameters, Show, and Delete methods of its parent.

Superclasses:

HLA_Interaction

Roles/Associations:

<none>

Attributes:

Initiator : Object_ID
The DIS Entity issuing the collision PDU...the collider.

Receiver : Object_ID
The "collidee" DIS Entity.

Collision_Event_ID : Event_ID_Record
The DIS Collision Event ID

Collision_Type : Collision_Types
The Collision type of the collision.

Collision_Velocity : Linear_Velocity_Record
The velocity vector of the issuing entity (the initiator).

Mass : Flt32
The mass of the issuing entity.

Location_Offset : Entity_Coordinate_Record
The location of impact in entity coordinates of the entity with which the issueing entity collided.

Operations:

Set_Initiator(Instance : access DIS_Collision.Object, New_Value : Object_ID)
Set the Collision Interaction's Initator.

Create(Instance : access DIS_Collision.Object, Class : Interaction_Class_Handle, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag,

Event_Retraction_Handle : Event_Retraction_Handle, Initiator : Object_ID, Receiver : Object_ID, Collision_Event_ID : Event_ID_Record, Collision_Type : Collision_Types, Collision_Velocity : Linear_Velocity_Record, Mass : Flt32, Location_Offset : Entity_Coordinate_Record)

Initialize a Collision Interaction following allocation.

Get_Initiator(Instance : access DIS_Collision.Object) : Object_ID

Get the Collision Interaction's initiator.

Set_Receiver(Instance : access DIS_Collision.Object, New_Value : Object_ID)

Set the Collision Interaction's receiver.

Get_Receiver(Instance : access DIS_Collision.Object) : Object_ID

Get the Collision Interaction's receiver.

Set_Collision_Event_ID(Instance : access DIS_Collision.Object, New_Value : Event_ID_Record)

Set the Collision Interaction's Event_ID.

Get_Collision_Event_ID(Instance : access DIS_Collision.Object) : Event_ID_Record

Get the Collision Interaction's event id.

Set_Collision_Type(Instance : access DIS_Collision.Object, New_Value : Collision_Types)

Set the Collision Interaction's collision type.

Get_Collision_Type(Instance : access DIS_Collision.Object) : Collision_Types

Get the Collision Interaction's Collision Type

Set_Collision_Velocity(Instance : access DIS_Collision.Object, New_Value : Linear_Velocity_Record)

Set the Collision Interaction's collision velocity.

Get_Collision_Velocity(Instance : access DIS_Collision.Object) : Linear_Velocity_Record

Get the Collision Interaction's collision velocity.

Set_Mass(Instance : access DIS_Collision.Object, New_Value : Flt32)

Set the Collision Interaction's mass.

Get_Mass(Instance : access DIS_Collision.Object) : Flt32

Get the Collision Interaction's mass.

Set_Location_Offset(Instance : access DIS_Collision.Object, New_Value : Entity_Coordinate_Record)

Set the Collision Interaction's location offset.

Get_Location_Offset(Instance : access DIS_Collision.Object) : Entity_Coordinate_Record

Get the Collision Interaction's location offset.

Class name:

DIS_Detonation

Documentation:

This class represents a DIS Detonation interaction between two DIS Entity Objects.

It overrides the Set_Parameters, Make_Parameters, Show, and Delete methods of its parent.

Superclasses:

DIS_Burst

Roles/Associations:

<none>

Attributes:

Location_Offset : Entity_Coordinate_Record
The location of the detonation relative to the receiving entity's origin.

Result : Nat8
The result of the detonation.

Operations:

Create(Instance : access DIS_Detonation.Object, Class : Interaction_Class_Handle, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag, Event_Retraction_Handle : Event_Retraction_Handle, Initiator : Object_ID, Receiver : Object_ID, Munition_ID : Entity_Identifier_Record, Fire_Event_ID : Event_ID_Record, Location : World_Coordinates_Record, Burst_Descriptor : Burst_Descriptor_Record, Fire_Velocity : Linear_Velocity_Record, Location_Offset : Entity_Coordinate_Record, Result : Nat8)
Initialize a DIS_Detonation interaction following allocation.

Set_Location_Offset(Instance : access DIS_Detonation.Object, New_Value : Entity_Coordinate_Record)
Set the DIS_Detonation Interaction's location offset.

Get_Location_Offset(Instance : access DIS_Detonation.Object) : Entity_Coordinate_Record
Get the DIS_Detonation Interaction's location offset.

Set_Result(Instance : access DIS_Detonation.Object, New_Value : Nat8)
Set the DIS_Detonation Interaction's result.

Get_Result(Instance : access DIS_Detonation.Object) : Nat8
Get the DIS_Detonation Interaction's result.

Class name:

DIS_Federate_Ambassador

Documentation:

The DIS Federate ambassador is a Concrete Federate Ambassador with specific knowledge about the DIS Object and Interaction types.

It overrides the Object and Interaction Factory methods of the Concrete_Federate_Ambassador class to provide DIS specific HLA Object's representing various DIS Entity types and DIS specific interactions (Collision, Fire, Detonation).

Superclasses:

Concrete_Federate_Ambassador

Roles/Associations:

<none>

Attributes:

<none>

Operations:

<none>

Class name:

DIS_Interface

Documentation:

This class represents the abstract DIS interface. It is a protected object (well, sort of... as a minimum, it's children must be implemented via protected objects or tasks to enable protected concurrent operations) which encapsulates an abstact interface to Get or Put PDUs from/to the DIS network.

Since this class is abstract, it enables polymorphic calls for getting and putting PDUs, enablling the actual DIS interface to be specified at run time.

The abstract interface component has two child classes. The Daemon_Based interface uses Bruce Clay's PDU Daemons, as defined by the interface in dsi_user.h (dsi_user.ads). The daemons are themselves two sprocs: one for reading PDUs and one for writing PDUs. The interface provided by dsi_user.h is an AFIT Graphics Lab standard, used by various applications. Future DIS network enhancements, such as ATM, will be built using the same interface, allowing applications such as SimWorx to interface without internal changes.

The Ada_Based interface relies on socket reading and writing tasks to directly read and write to a BSD socket. It was the original SimWorx DIS interface, built before the SimWorx requirement to use dsi_user.h existed.

Either interface may be used at run-time, based upon run-time parameters in the ww_parms.dat file.

Superclasses:

<none>

Roles/Associations:

Association Reads PDUs
Association Has_DIS_Interface_Task
Association Writes_PDUs
Association Writes PDUs

Attributes:

<none>

Operations:

Create(Instance : access DIS_Interface.Object, Success : Boolean)

This method initializes the interface after the client allocates it via the new operator. Returns True if successful.

Get_PDU(Instance : access DIS_Interface.Object, PDU_Reference : Raw_PDU_Ref, PDU_Available : Boolean)

This method returns a reference to the next available PDU. If no PDU is immediately available, PDU_Available is False. When the client is finished with the PDU, he calls Free_PDU to indicate he's finished.

Put_PDU(Instance : access DIS_Interface.Object, PDU_Reference : Raw_PDU_Ref)

This method sends the PDU pointed to by PDU_Reference out to the network, and then deallocates the PDU.

Finalize(Instance : access DIS_Interface.Object)

Clients call this method prior to deallocating the DIS_Interface_Task

Class name:

DIS_Surrogate_RTI_Ambassador

Documentation:

This class is a form of RTI Ambassador which presents an HLA face to HLA simulations, but presents a DIS face to the network. It implements all of the methods of the Abstract RTI Ambassador, and adds a Create method for class-specific initialization.

A client class (subclass of Federate) allocates the DIS_Surrogate_RTI_Ambassador and then calls the Create method. Following the initializing call to Create, the ambassador is ready to accept the method calls inherited from the Abstract_RTI_Ambassador.

Further, following initialization, the DIS_Surrogate_RTI_Ambassador is able to generate calls to the Federate's interface based upon actions within the DIS network.

Superclasses:

Abstract_RTI_Ambassador

Roles/Associations:

Association Allocates_Entity_Container
Association Has_Dead_Reckon_Task
Association Has_PDU_Update_Task
Association Calls
Association Has_DIS_Interface_Task

Attributes:

Fed_Amb : Abstract_Federate_Ambassador.Reference = null

This is an Ada classwide type which refers to the RTI_Ambassador's associated Federate_Ambassador. It's necessary to enable dispatching calls to the Federate Ambasssador.

Operations:

Create(Instance : access DIS_Surrogate_RTI_Ambassador.Object, Fed_Amb : Abstract_Federate_Ambassador.Reference)

This method initializes the DIS_Surrogate_RTI_Ambassador following allocation by the new operator. It initializes the RTI Clock. It allocates and initializes the Entity Container, the PDU_Reader_Task, the PDU_Writer_Task, the Dead_Reckon_Task, and the parameter-file-specified DIS_Interface.

Remember, in Ada, as soon as each task is elaborated, it starts to run concurrently.

Class name:

Entity_Container

Documentation:

This class is an instance of the generic Protected_Container. It contains references to DIS_Entities. The container lets the two tasks maintain multiple iterators into it.

Superclasses:

<none>

Roles/Associations:

Association Allocates_Entity_Container
Association Shares
Association Shares
Association Has_Entities

Association Shares

Attributes:

<none>

Operations:

<none>

Class name:

Fed Ambassador Tasks

Documentation:

This class represents two tasks owned by the Concrete Federate Ambassador. (It has to represent two tasks since the demo version of the Rational Rose tool I'm using only supports up to 30 classes at a time).

The first task scans the Objects object container at some run-time parameter specified rate, and checks to see if there have been any changes to the object. If there have been any changes to an object, then it sends the updated attributes to the RTI via the Update_Attribute_Values method.

The second task exists soley to manage deletion of HLA_Objects in the Objects container. One centralized deleter is necessary to prevent deadlock from multiple concurrent Object Removals of the same object in the container.

One might think we could save a task, and have one task handle both of these operations. That is a bad idea, however, since it violates Gomaa's task assignment critera: namely, you shouldn't have one task doing both periodic (the update thing) and aperiodic (the deletion thing) activities.

Superclasses:

<none>

Roles/Associations:

Association Has Fed Ambassador Tasks
Association Shares Object Container

Attributes:

<none>

Operations:

Elaborate_With_Discriminants(Objects_Container : Objects.Reference, The_RTI : Abstract_RTI_Ambassador.Reference)

Both tasks require a reference to the Objects container in order to manipulate the objects in it.

Both tasks require a reference to the RTI in order to call the RTI's methods.

Delete_HLA_Object(The_ID : Object_ID)

An Ada entry which deletes the HLA_Object from the Objects container.

Scan_Container( )

Examines each HLA object in the Objects container to see if any attribute values have changed (HLA_Object.Changes = True). Calls Make_Attributes for HLA_Objects which have changed. Then sends attributes to RTI via Update_Attribute Values.

Class name:

Fed_Task

Documentation:

This task realizes the overall scheme behind the Federate class. It loops forever, at some periodic rate, calling a client specified routine over and over until it is killed. Most simulation applications have a "main loop"--The Fed_Task is the SimWorx default main loop. If client simulations don't have a "main loop", they can override the Federate.Initialize method, (simply leaving the Fed_Task reference null) and do some other form of overall simulation control.

The Fed_Task is created by the Federate.Initialize method. Upon initialization, it calls the Federate.Client_Initialize subprogram. Following that, it loops forever (or until the Kill entry is called).

Upon acceptance of the Kill entry, the Fed_Task calls Federate.Client_Finalize to perform client code wrap-up duties.

This task guarantees the concurrency of the Federate's operations. It's implemented as a task to collaborate with the tasks of the RTI Ambassador.

Superclasses:

<none>

Roles/Associations:

Association Has Fed_Task

Attributes:

Fed : Federate.Reference

This attribute is an Ada classwide type which refers to the Fed_Task's associated Federate--necessary to allow Fed_Task to call the dispatching methods of the Federate: Client_Initialize, Client_Loop_Body, etc.

Operations:

Kill( )

The Kill entry causes the task to exit after it calls the Client_Finalize method.

Elaborate_With_Discriminant(Federate : Federate.Reference)

This method represents elaborating the Fed_Task with a reference to it's associated Federation object.

Class name:

Federate

Documentation:

This abstract class represents the client programmer's simulation application. The client programmer will subclass this class to handle the chores of her simulation. For example, a flight simulator subclass would model flight.

The Federate class enables the objects it has to interact with each other.

When it receives an interaction, it explicitly invokes the methods of the objects it owns to handle the interaction.

The Federate Class allocates and initializes the RTI, the Object container, the Interaction_Queue, the Fed_Task, and some version of the Concrete Federate Ambassador (specified by the client-overridden Federate_Ambassador_Factory_Method).

This class is the DIS-specific version of the Federate. It's a simple DIS Federate which just periodically prints information about the HLA_Objects and HLA_Interactions of the federation.

It overrides the Federate_Ambassador_Factory_Method to allocate a DIS-specific Federate_Ambassador which has knowledge of the DIS Objects and DIS Interactions. It also overrides the Initialize_Federate_Ambassador method for initialization.

It also overrides the RTI_Ambassador_Factory_Method to allocate the DIS_Surrogate_RTI_Ambassador. Also overrides Initialize_RTI_Ambassador.

It overrides Client_Initialize with a null operation since there is nothing special to do here.

It overrides Client_Loop_Body to periodcally do two things. First, it loops through all of the HLA_Objects in the Objects container, invoking the Show method on each. Second, it pulls each queued HLA_Interaction off of the Interaction_Queue and invokes Show on it.

It overrides Client_Finalize with a null operation since nothing special needs to be done to clean up.

Superclasses:

<none>

Roles/Associations:

Association Has Fed_Task

Association Has_Ambassador
Association Allocates_Object_Container
Association Allocates_Interaction_Queue

Attributes:

<none>

Operations:

Initialize(Instance : Federate.Object)

This subprogram initializes the Federate object following allocation by the new operator. It is automatically called since the Federate is a child class of Ada.Finalization.Limited_Controlled.

Finalize(Instance : Federate.Object)

This method overrides the Ada Controlled type Finalize method. It "cleans up" the Federate object when it is deallocated.

Client_Initialize(Instance : access Federate.Object)

At task initialization, the Sim_Task calls this client supplied initialization subprogram to perform initialization duties for the client.

This subprogram is meant to be overridden.

Client_Loop_Body(Instance : access Federate.Object)

For each iteration of it's main loop, Sim_Task calls this client supplied subprogram to do a "frame's worth" of simulation work. Note that this implies the SimWorx framework has a main loop which repeats some client-specified action over and over. If you don't like that, then over-ride the Initialize subprogram to define a different behavior for a simulation!

Client_Finalize(Instance : access Federate.Object)

When the Sim_Task is terminated, it calls this subprogram (overridden by the client programmer, of course) to perform clean-up duties within the client simulation.

Federate_Ambassador_Factory_Method(Instance : Federate.Object) : Abstract_Federate_Ambassador.Reference

This method is called by the Initialize method to determine the actual instance of Federate Ambassador the Federate will be using. Client programmers override this method in their child Federate class, specifying what kind of Federate Ambassador to use. It is an example of the Factory Method pattern from the book Design Patterns: Elements of Reusable Object-Oriented Software. ISBN 0-201-63361-2

RTI_Ambassador_Factory_Method(Instance : Federate.Object)

This method is called by the Initialize method to determine the actual instance of RTI Ambassador the Federate will be using. Client programmers override this method in their child Federate class, specifying what kind of RTI Ambassador to use.

Initialize_Federate_Ambassador(Instance : Federate.Object)

Called by the Initialize method to initialize the Federate_Ambassador created by the factory method. Initialization can't be embedded in the factory method since both Ambassadors must be allocated before either is initialized. Meant to be overridden by client programmers.

Initialize_RTI_Ambassador(Instance : Federate.Object)

Called by Initialize method to initialize the RTI Ambassador. Same comment about initilization applies to the RTI Ambassador as noted above in Initialize_Federate_Ambassador. Meant to be overridden by client programmers.

Class name:

HLA_Interaction

Documentation:

This class represents an abstract HLA Interaction. Framework client programmers will subclass this class for each interaction class specified by their simulation's SOM.

Interactions are a form of the Command Behavioral Pattern. See pate 233 of "Design Patterns: Elements of Reusable Object-Oriented Software", ISBN 0-201-63361-2. Commands are used to "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."

However, unlike the Design Pattern's Command, an interaction is not responsible for executing itself--that would require it to have knowledge of different HLA_Objects (aaaa, the circular with thing again!!), instead, we'll make the HLA_Objects have knowledge of the different interactions they can be subjected to. HLA_Objects are then responsible for converting an explicit HLA_Interaction into the implicit effects (method calls) the interaction has on them.

Superclasses:

<none>

Roles/Associations:

Association Has_Interactions

Attributes:

Class_Handle : Interaction_Class_Handle = 0

The interaction class this interaction instance belongs to. The handle is unique for each kind of HLA_Interaction.

Fed_Time : Federation_Time

The time the interaction is effective.

User_Tag : User_Supplied_Tag

A user supplied tag--implemented as a pointer to string in C.

Event_Retraction_Handle : Event_Retraction_Handle

An event retraction handle. This is probably for use in "undoing" or "recalling" an event during event driven sims. Not currently used in SimWorx since DIS and SimWorx support real-time sims.

Initiator : Object_ID = Null_Object_ID

The initiating HLA object of the HLA Interaction. The value for this attribute is updated via the Set_Parameters method since the HLA Interface Spec didn't specifiy this as an explicit method parameter.

Receiver : Object_ID = Null_Object_ID

The receiving HLA object of the HLA Interaction (not that every HLA_Interaction will necessarily have a specific receiver). The value for this attribute is updated via the Set_Parameters method since the HLA Interface Spec didn't specifiy this as an explicit method parameter.

Operations:

Create(Instance : HLA_Interaction.Reference, Fed_Time : Federation_Time, User_Tag : User_Supplied_Tag, Event_Retraction_Handle : Event_Retraction_Handle)

Initialize an interaction following allocation.

Set_Parameters(Instance : access HLA_Interaction.Object, Parameters : Parameter_Handle_Value_Pair_Set)

Set the parameters of the Interaction by copying the parameters from the Parameter_Handle_Value_Pair_Set.

Make_Parameters(Instance : access HLA_Interaction.Object, Parameters : Parameter_Handle_Value_Pair_Set)

Build a Parameter_Handle_Value_Pair_Set from the parameters (attributes) of this interaction. Allows a client class to use Set_Attribute calls to initialize an HLA Interaction, then call this method to create the pair set for transmission to the RTI (or Federate).

Get_Class_Handle(Instance : access HLA_Interaction.Object) : Interaction_Class_Handle

This is a dispatching method meant to be overridden by each subclass. It returns the Interaction_Class_Handle for this kind of interaction.

Note that there is no corresponding Set_Class_Handle method since each kind of interaction has a constant Class_Handle, set upon instantiation.

Delete(Instance : access HLA_Interaction.Object)

This method safely deallocates the storage for this interaction. It is meant to be overriden by every child interaction class defined by client programmers.

Set_Time(Instance : access HLA_Interaction.Object, New_Value : Federation_Time)

Set the Interaction's Federation Time attribute. I believe this is the time the Interaction is effective.

Get_Time(Instance : access HLA_Interaction.Object) : Federation_Time

Get the HLA Interaction's time.

Set_Initiator(Instance : access HLA_Interaction.Object, New_Value : Object_ID)

Set the HLA Interaction's initiator.

Get_Initiator(Instance : access HLA_Interaction.Object) : Object_ID

Get the HLA Interaction's initiator.

Set_Receiver(Instance : access HLA_Interaction.Object, New_Value : Object_ID)

Set the Interaction's receiver. Note that HLA_Interactions do not necessarily have a receiver, but many child classes of the HLA_Interaction do. This method is meant to be overridden.

Get_Receiver(Instance : access HLA_Interaction.Object) : Object_ID

This method returns the Interaction's receiver. If the interaction does not have a specific receiver, then the method returns Null_Object_ID. Note that HLA_Interactions do not necessarily have a receiver, but many child classes of the HLA_Interaction do. This method is meant to be overridden.

Show(Instance : access HLA_Interaction.Object)

Show the interaction (print it, although printing could be overridden for graphical applications).

Class name:

HLA_Object

Documentation:

This class represents an HLA object. Objects interact with each other to carry out the duties of the simulation. Objects may be instantiated and controlled locally by this simulation and also instantiated on the behalf of some other federate.

Object interactions may be implicit via method calls to other objects, or explicit via an Interaction invoked by the simulation. The method Handle_Interaction is meant to be overridden for each kind of HLA_Object in order to handle the different kinds of interactions each kind of Object can be subjected to.

Superclasses:

<none>

Roles/Associations:

Association Has_Objects

Attributes:

Class : Object_Class_Handle = 0

The object class this object belongs to. Each child class has a unique constant value for this attribute.

ID : Object_ID = Null_Object_ID

The ID of this object. It is unique within the federation execution.

Time_Stamp : Federation_Time

Time which something was last done to the HLA_Object.

Changed : Boolean

Indicates whether any of the HLA Object's attributes have changed. Necessary to avoid sending duplicate, unnecessary information to the RTI.

This flag is set by the Set_X methods of this and any child classes. (thus, a derived requirement is access to instance variables should be via Set_X in order to correctly set the Changed Flag, else client programmer has to remember to set Change := True).

Reset by the Make_Attributes methods.

Owned : Boolean

A flag which indicates whether this object is Owned by this Federate. Since SimWorx does not currently support Ownership Management operations, this attribute is set upon object creation and not changed during the object's lifetime.

Fed_Amb : Federate_Ambassador.Reference

Provides each HLA Object with the identity of the Federate Ambassador so it can call the ambassador's services.

Operations:

Create(Instance : access HLA_Object.Object, Object_ID : Object_ID, Federate_Amb : Federate_Ambassador.Reference, Owned : Boolean)

Initialize an Object following allocation.

Simply sets instance variables. Note that the Class ID is not passed in since each HLA Object child class will know what it's class is and set its instance value accordingly.

Delete(Instance : access HLA_Object.Object)

This method safely de-allocates the storage for this object. It is meant to be overridden by every child class defined by client programmers.

Attribute_Update(Instance : access HLA_Object.Object, Time_Stamp : Federation_Time, Attribute_Value_List : Attribute_Handle_Value_Pair_Set)

This method copies the attributes from the Attribute_Handle_Value_Pair_Set to the Object's corresponding attributes. It is meant to be overridden by every child class defined by client programmers.

Handle_Interaction(Instance : access HLA_Object.Object, Interaction : HLA_Interaction.Reference)

This method handles an interaction. That is, it takes an explicit HLA_Interaction and performs the necessary actions to react to it (usually by calling other methods, depending upon the kind of interaction).

Lock(Instance : access HLA_Object.Object)

Lock the Object for exclusive access. All clients must Lock the HLA_Object prior to executing a method which could change the HLA_Object's state.
The protocol is...

```
 Lock
 Do_Some_Method
 Unlock
```

NOTE: Children of the HLA_Object do not call the Lock method since that will lead to deadlock.

Unlock(Instance : access HLA_Object.Object)
The Unlock operation. See Lock method for more details.

Set_ID(Instance : HLA_Object.Reference, Object_ID : Object_ID)
Set the Object's Object_ID.

Get_ID(Instance : HLA_Object.Reference) : Object_ID
Return the Object's ID.

Get_Class(Instance : access HLA_Object.Object) : Object_Class_Handle
Return the Object's Class. Note that there is no corresponding Set_Class method since each subclass will have its own constant value.

Show(Instance : access HLA_Object.Object)
Show the Object. Meant to be overridden for each child class. For example, a simple data logger may simply print the attribute values. A graphical program could draw the object.

Changed(Instance : HLA_Object.Reference) : Boolean
Passes the value of the Changed Attribute to clients.

Make_Changed_Attributes(Instance : access HLA_Object.Object, Attributes : Attribute_Handle_Value_Pair_Set)
Copies the values of attributes which have changed into the Attributes set. Resets the Changed flag to False, indicating attributes have stabilized.

Make_All_Attributes(Instance : access HLA_Object.Object, Attributes : Attribute_Handle_Value_Pair_Set)
Copies all of the HLA Object's attributes to the Attribute Set. Sets Changed to False to indicate stability of attributes.

Owned(Instance : HLA_Object.Reference) : Boolean
Provides clients access to the Owned attribute.

Get_Time_Stamp(Instance : HLA_Object.Reference) : Federation_Time
Provides clients access to Time_Stamp attribute.

Action(Instance : access HLA_Object.Object)
A standardized method for invoking an HLA Object's behavior. Clients programmers will override this method for each different object's different behaviors.

Class name:

Interaction Queue

Documentation:

This class is an instance of the generic Bounded Buffer. It buffers interactions received by the Concrete_Federate_Ambassador so they can be used by the Federate when he gets around to them.

Superclasses:

<none>

Roles/Associations:

Association Shares_Interaction_Queue
Association Has_Interactions
Association Allocates_Interaction_Queue

Attributes:

<none>

Operations:

<none>

Class name:

Objects

Documentation:

This class is an instance of the generic Protected_Container. It manages concurrent access to the HLA_Objects it contains (actually, it contains References to Objects) from the Fed_Task and the tasks of the RTI_Ambassador (via the Concrete Federate Ambassador).

Since it is a generic, it has all of the operations and type definitions of the Protected_Container class.

Superclasses:

<none>

Roles/Associations:

Association Shares_Object_Container
Association Has_Objects
Association Allocates_Object_Container
Association Shares Object Container

Attributes:

<none>

Operations:

<none>

Class name:

PDU Buffers

Documentation:

The PDU Buffers (one for input and one for output) are instantiations of the generic Bounded_Buffer. They're meant to handle the slack between PDU consumers and producers.

Note the use of these buffers implies if the Socket_Reader_Task can't keep up with PDUs coming from the network, it will "lose" the most recent PDUs on the network since the Socket_Reader task will not be able to put them in the "full" bounded buffer and they will be overwritten in the socket by newer PDUs.

This is in contrast to the Daemon based implementation which will overwrite waiting PDUs with newer ones as they are received from the network.

Superclasses:

<none>

Roles/Associations:

Association Allocates_PDU_Buffers
Association Fill and Empty Buffers

Attributes:

<none>

Operations:

<none>

Class name:

PDU_Reader_Task

Documentation:

This task object repeatedly reads PDUs from the DIS_Interface, performing different actions depending upon the type of PDU read.

Entity State PDU: If this is for an unrecognized entity, the task creates a new DIS Entity and puts it in the Entity Container, and updates the entity's state based upon data in the PDU. If this is for a recognized entity, the task simply updates the entity's state from the PDU.

Fire PDU: Propagates a DIS Fire Interaction to the Federate with data from the PDU.

Detonation PDU: Propagates a DIS Detonation Interaction to the Federate with data from the PDU.

Collision PDU: Propagates a DIS Collision Interaction to the Federate with data from the PDU.

The exception Program_Error may be raised by the protected container or by a protected DIS Entity if the object is deleted while this task is waiting to use it. Thus, Program_Errors are simply handled in the task loop iteration by doing nothing.

Superclasses:

<none>

Roles/Associations:

Association Has_PDU_Update_Task
Association Shares
Association Reads PDUs

Attributes:

Prototype_Collision : DIS_Collision

A prototype HLA Interaction used in this manner to encapsulate the interaction's knowledge of the parameters:

1. Copy parameters into it via Set_Parameters
2. Get the parameters out one at a time via Get_x, where x is some method name.

Prototype_Fire : DIS_Fire

A prototype HLA Interaction used in this manner to encapsulate the interaction's knowledge of the parameters:

1. Copy parameters into it via Set_Parameters

2. Get the parameters out one at a time via Get_x, where x is some method name.

Prototype_Detonation : DIS_Detonation

A prototype HLA Interaction used in this manner to encapsulate the interaction's knowledge of the parameters:

1. Copy parameters into it via Set_Parameters
2. Get the parameters out one at a time via Get_x, where x is some method name.

Operations:

Elaborate_With_Discriminants(Fed_Ambassador : Abstract_Federate_Ambassador.Reference, Entity_Container_Handle : Entity_Container.Reference, DIS_Interface : DIS_Interface.Reference)

This method represents elaborating the task with the above discriminants.

The Federate Ambassador Reference is necessary to allow the task to call the methods of the Federate Ambassador.

The DIS_Interface reference is the necessary handle to the DIS Interface for making method calls.

Handle_PDU(Entity_State_Ptr : Entity_State_PDU_Ref)

This method handles each PDU received. See the main documentation section above for details.

Project_DIS_Collision(Collision_Ptr : Collision_PDU_Ref)

This method sets up for, and then calls the Federate Receive_Interaction method for the DIS Collision interaction. It builds the Parameter_Value_Handle_Pair_Set (from values in the Collision PDU).

Project_DIS_Fire(Fire_Ptr : Fire_PDU_Ref)

This method sets up for, and then calls the Federate Receive_Interaction method for the DIS Fire interaction. It builds the Parameter_Value_Handle_Pair_Set (from values in the Fire PDU).

Project_DIS_Detonation(Detonation_Ptr : Detonation_PDU_Ref)

This method sets up for, and then calls the Federate Receive_Interaction method for the DIS Detonation interaction. It builds the Parameter_Value_Handle_Pair_Set (from values in the Detonation PDU).

Find_Entity(Entity_ID : Entity_Identifier_Record) : DIS_Entity.Reference

This method scans through the Entity_Container, looking for an entity with a matching DIS Entity ID. If found, it returns a reference to the Entity. If not found, it returns null.

Kill( )

Kill the PDU_Handler_Task.

Class name:

PDU_Writer_Task

Documentation:

Responsible for writing PDUs to the network: Collision, Fire, and Detonation PDUs resulting from Federate initiated HLA Interactions, and Entity State PDUs resulting from Attribute Updates.

This is a separate task from the PDU_Reader_Task to ensure the act of reading PDUs from the network (a blocking activity) does not prevent act of writing PDUs to the network.

Note that the Dead_Reckon_Task also writes PDUs resulting from the Dead Reckoning Process.

Superclasses:

<none>

Roles/Associations:

Association Writes PDUs
Association Shares

Attributes:

Prototype_Collision : DIS_Collision

A prototype HLA Interaction used in this manner to encapsulate the interaction's knowledge of the parameters:

1. Copy parameters into it via Set_Parameters
2. Get the parameters out one at a time via Get_x, where x is some method name.

Prototype_Fire : DIS_Fire

A prototype HLA Interaction used in this manner to encapsulate the interaction's knowledge of the parameters:

1. Copy parameters into it via Set_Parameters
2. Get the parameters out one at a time via Get_x, where x is some method name.

Prototype_Detonation : DIS_Detonation

A prototype HLA Interaction used in this manner to encapsulate the interaction's knowledge of the parameters:

1. Copy parameters into it via Set_Parameters
2. Get the parameters out one at a time via Get_x, where x is some method name.

Operations:

Elaborate_With_Discriminants(Fed_Ambassador : Federate_Ambassador.Reference, Entity_Container : Entity_Container.Reference, DIS_Interface : DIS_Interface.Reference)

Elaborate the PDU_Writer_Task with the discriminants--these are object identies necessary for the task to interact with other objects.

Send_Interaction_To_Network(The_Interaction : Interaction_Class_Handle, The_Parameters : Parameter_Handle_Value_Pair_Set, The_Time : Federation_Time)

This entry converts an HLA Interaction into it's corresponding DIS PDU and then sends the PDU to the network.

Prototype HLA Interactions are used to get the parameter values out of the Parameter_Handle_Value_Pair_Set, in order to encapsulate knowledge of the different parameters and to simplify maintenance. This is in contrast to the lack of use of prototypes for HLA Objects (see DIS Entity). This is mainly a function of the variety of DIS HLA_Interactions.

Update_Attribute_Values(The_Object : Object_ID, The_Attributes : Attribute_Handle_Value_Pair_Set, The_Time : Federation_Time)

Updates a DIS Entity's attribute values on behalf of the RTI Ambassador. Then builds an Entity State PDU to reflect the value update and writes the PDU to the DIS Interface.

Kill( )

Enables parent task to terminate this task at will.

Build_Collision(The_Parameters : Parameter_Handle_Value_Pair_Set, Raw_PDU : Raw_PDU_Ref)

Allocate a Collision PDU, copy the Collision Interaction's parameters from the parameter set (via the prototype Collision) into it, and return a reference to it.

Build_Detonation(The_Parameters : Parameter_Handle_Value_Pair_Set, Raw_PDU : Raw_PDU_Ref)

Fabricates a Detonation PDU in same manner as Build_Collision.

Build_Fire(The_Parameters : Parameter_Handle_Value_Pair_Set, Raw_PDU : Raw_PDU_Ref)

Does for Fire PDUs what Build_Collision does for Collisions.

Handle_Attribute_Update(The_Object : Object_ID, The_Attributes : Attribute_Handle_Value_Pair_Set, The_Time : Federation_Time, Entity_Container : Entity_Container.Reference, Raw_PDU : Raw_PDU_Ref)

Called by the Update_Attribute_Values entry to do the dirty work of updating the DIS Entity (via DIS_Entity.Update_State_From_Federate), and building the Entity State PDU (via DIS_Entity.Build_PDU) for shipment to the DIS Interface.

Class name:

Protected_Container

Documentation:

This is the generic Protected_Container Class. It is implemented as a protected type.

This container allows several different iterations to be performed concurrently via "Iterators". An iterator is an external cursor, or marker, marking a client's current position within the container.

For more information on iterators, see the Behavioral Design Pattern "Iterator" on page 257 of "Design Patterns: Elements of Reusable Object-Oriented Software", ISBN0-201-63361-2

Note that the only tricky bit here is that there could be a dead-lock condition when two separate clients try to remove an item at the same time (that their iterator's are both pointed at). This shouldn't be a problem in the SimWorx framework since only one client will have the responsiblity to remove items from the container. (Thomas Kofler provides a solution to this problem in the Paper "Robust Iterators in ET++", in Structured Programming, 14:62-85, March 1993, however his approach requires use of an atomic Visit method which removes much of the flexiblity of the external iterator).

Superclasses:

<none>

Roles/Associations:

<none>

Attributes:

Iterators : Iterator_Indices_Table_Type
This table stores the current values of the iterators of the container.

Store : Buffer_Array
A simple array of Item_Type--the actual container.

Current_Size : Count_Range
The current number of Item_Type in the container.

Next_Slot : Index_Range
The next available empty position in the container.

Operations:

Add(Item : Item_Type)
Add an Item to the Container

Size( ) : Natural
Returns the current size of the container.

Empty( ) : Boolean
Returns True if the container is empty.

Make_Iterator( ) : Iterator_Type
This method returns an iterator (or external cursor) for use in iterating through the items in the container.

Destroy_Iterator(Iterator : Iterator_Type)
Destroy the iterator. Following this call, the iterator is no longer valid.

Remove(Iterator : Iterator_Type, Success : Boolean)
Remove the item pointed to by the iterator from the container. Returns Success = False if the item couldn't be removed (happens when some other iterator is currently pointing to that item--try again later).

Note: Waiting on Sucess could lead to dead-lock if more than one client is responsible for removing items. (But no one would ever create such an irresponsible, dangerous design, would they!?.)

Current_Item(Iterator : Iterator_Type) : Item_Type
Returns a copy of the current item in the container.

Finished(Iterator : Iterator_Type) : Boolean
Returns True if iterator has exhausted the list (in either the forward or reverse direction.)

Reset(Iterator : Iterator_Type)
Reset the iterator to the beginning of the container.

Next(Iterator : Iterator_Type)
Advances the Iterator to the next item in the container. When you're at the end of the container, and you call Next, the iterator's position will no longer be valid, and Finished will return True. (So test Finished before you do anything with Current_Item).

Previous(Iterator : Iterator_Type)
Move the iterator to the previous item in the container. When you're at the beginning of the

container, and you call Previous, the iterator's position will no longer be valid, and Finished will return True. (So test Finished before you do anything with current item).

Class name:

RTI Services

Documentation:

This class represents several RTI services (see attributes below). These are encapsulated here mainly because of the 30 class limit to the demonstration version of the Rational Rose tool used to capture the design. Otherwise, they'd be represented as separate classes.

Superclasses:

<none>

Roles/Associations:

<none>

Attributes:

Clock : Protected_Clock

This clock represents the ada protected object which controls concurrent access to the Clock.

ID_Server : Protected_ID_Server

This is the Ada protected object which controls concurrent access to the ID Server

Operations:

Clock( ) : Federation_Time

Returns the current Federation_Time in seconds. Federation Time may be relative, or absolute (wall clock time). If the time is relative, the value returned is relative to the initialization time of the RTI. If the time is absolute, the value returned is the current unix time (seconds since Jan 1st, 1970). DIS exercises may use either form of time.

Clients call Initialize_Clock_Relative or Initialize_Clock_Absolute, depending upon the form of time they want returned.

Time_Stamp( ) : DIS_Time_Stamp

Returns a DIS_Time_Stamp (an integer number of time hacks since the top of the hour--see DIS Types package) corresponding to the current time.

Initialize_Clock_Absolute( )

This method initializes the Federation Time Clock to return absolute time (Unix time in seconds since Jan 1st, 1970).

All subsequent calls to the Clock method will return absolute time.

Initialize_Clock_Relative( )

This method initializes the Federation Time clock by establishing and RTI epoch time which is considered "Time Zero" for this simulation run of the RTI.

All subsequent calls to the Clock method will return a time relative to the simulation epoch time.

Get_Corresponding_ID(DIS_ID : Entity_Identifier_Record) : Object_ID

This method returns the Object_ID corresponding to the DIS Entity ID of a DIS Entity. It returns Null_Object_ID if the DIS ID isn't recognized.

Request_ID(The_Count : Object_ID_Count, First_ID : access Object_ID, Last_ID : access Object_ID)

Returns HLA Object Ids. IDs are paired with DIS Entity IDs during a simulation. If for some reason, an external DIS Entity becomes stale and then reappears, the ID server will know to use the same ID. This method returns IDs from the pool of IDs reserved for Entities controlled by this federate. That is, corresponding DIS IDs will be fabricated for these entities by the RTI based upon the HLA Object ID.

Get_Corresponding_ID(The_ID : Object_ID) : Entity_Identifier_Record

This method returns the DIS Entity ID corresponding to the given Object ID.

If the Object ID is from the pool of IDs reserved for this federate's objects, then the Entity ID is fabricated from the Object_ID.

If the Object ID is from the pool of external IDs, then the Entity ID is looked up in the table of ID pairs.

Possible Exceptions:

Invalid_Object_ID. ID Hasn't been assigned yet from pool, or there is no corresponding DIS Entity ID in the pair array.

Request_ID(DIS_ID : Entity_Identifer_Record) : Object_ID

Returns an Object ID from the pool for this new DIS ID.

Possible Exceptions:

Invalid_Object_ID. Raised if DIS ID is already assigned to an Object ID.

ID_Supply_Exhausted.

Class name:

Socket Tasks

Documentation:

This class represents the Socket_Reader_Task and the Socket_Writer_Task of the Ada_Based DIS Interface.

The Socket_Reader_Task continuously reads PDUs from a socket and places them in an input buffer.

The Socket_Writer_Task continuously gets PDUs from an output buffer and writes them to the socket.

This implementation relies upon the operating system to serialize I/O on the Internet Port used by the exercise (done within the sendto and receivefrom calls).

Superclasses:

<none>

Roles/Associations:

Association Elaborates_Socket_Tasks
Association Fill and Empty Buffers

Attributes:

<none>

Operations:

Elaborate_With_Discriminants(Port_Number : Positive = 4000, PDU_Buffer_Ref : PDU_Buffer.Reference, Socket_Handle : Integer = 0, Socket_Address : sockaddr_ptr)

Elaborate the task with discriminants.
sockaddr_ptr is an access type which refers to sockaddr_in records (a BSD structure for manipulating sockets).

Kill( )

Allows parent task to kill this task.

Index

Thesis Appendix Note: Since this stand-alone design document was imported into this thesis, page numbers in its Table of Contents, Table of Figures, and Index refer only to this appendix. Substitute E-5 for 5, E-11 for 11, etc.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1996

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
SIMWORX: AN ADA 95 DISTRIBUTED SIMULATION APPLICATION FRAMEWORK SUPPORTING HLA AND DIS

5. FUNDING NUMBERS

6. AUTHOR(S)
Earl Conrad Pilloud, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2750 P Street
WPAFB, OH 45433-7126

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/96D-23

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
ATTN: Mr. Gary Shupe
DISA/DVSW/JEXSV
5600 Columbia Pike
Falls Church, VA 22041

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research consisted of the analysis, design, and implementation of a reusable application framework for distributed simulation which is compliant with both the DoD High Level Architecture (HLA) for Modeling and Simulation and the Distributed Interactive Simulation (DIS) standards. The goal was to create an Ada-based system for experimentation in distributed simulation. A subsidiary goal was to integrate the system with an existing Air Force Institute of Technology (AFIT) application framework for virtual simulations, Easy_Sim.

The application framework was designed using object-oriented techniques to enable experimenters to customize it via inheritance extension. The application framework, named SimWorx, consists of two sections: an HLA Federate skeleton, and a surrogate HLA Run-Time Infrastructure (RTI) which has an HLA "front-end" and a DIS "back end" to provide DIS compatibility. The SimWorx framework was successfully integrated with Easy_Sim to provide an Ada-based joint simulation system for distributed virtual simulations.

14. SUBJECT TERMS
Modeling, Simulation, Distributed Simulation, High Level Architecture (HLA), Distributed Interactive Simulation (DIS), Software Engineering, Application Framework, Object-Oriented, Design Patterns

15. NUMBER OF PAGES
372

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet *optical scanning requirements.*

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	-	Contract	PR	-	Project
G	-	Grant	TA	-	Task
PE	-	Program Element	WU	-	Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

★U.S.GPO:1993-0-336-043