JSC XXXXX-DD.mm.20

Trick High Level Architecture TrickHLA Product Specification

Simulation and Graphics Branch (ER7)
Software, Robotics and Simulation Division
Engineering Directorate

Package Release TrickHLA v3.0.0 - Beta

Document Revision 1.0
June 2020

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

JSC XXXXX-DD.mm.20

Trick High Level Architecture
TrickHLA
Product Specification

Document Revision 1.0
June 2020

Edwin Z. Crues
and
Daniel E. Dexter

Simulation and Graphics Branch (ER7)
Software, Robotics and Simulation Division
Engineering Directorate

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

Abstract

The TrickHLA model provides a simplified interface to the IEEE-1516 High Level Architecture (HLA) [12] for use in the Trick Simulation Environment. It allows developers to concentrate on simulation development without needing to be an HLA expert. The model is heavily data driven and based on a simple API C++ making it relatively easy to take an existing Trick simulation and make it HLA aware. This document provides a high-level overview of the model.

Contents

Chapter 1

Introduction

The objective of TrickHLA is to simplify the process of providing simulations built with the Trick Simulation Environment[8] with the ability to participate in distributed executions using the High Level Architecture (HLA)[13]. This allows a simulation developer to concentrate on the simulation and not have to be an HLA expert. TrickHLA is data driven and provides a simple API making it relatively easy to take an existing Trick simulation and make it HLA capable.

1.1 Identification of Document

This document describes the design of the TrickHLA developed for use in the Trick Simulation Environment. This document adheres to the documentation standards defined in NASA Software Engineering Requirements Standard [7].

1.2 Scope of Document

This document provides information on the algorithms used in and the design of the source code associated with TrickHLA. This includes references to relevant Trick and HLA documents.

1.3 Purpose and Objectives of Document

The purpose of this document is to provide a thorough understanding of the TrickHLA product and its underlying architecture and design.

1.4 Documentation Status and Schedule

The information in this document is current with the TrickHLA v3.0.0 - Beta implementation of TrickHLA. Updates will be kept current with module changes.

Author	Date	Description
Edwin Z. Crues	June 2020	TrickHLA Version 3

Revised by	Date	Description

1.5 Document Organization

This document is organized into the following sections:

Chapter 1: Introduction - Identifies this document, defines the scope and purpose, present status, and provides a description of each major section.

Chapter 2: Related Documentation - Lists the related documentation that is applicable to this project.

Chapter 3: Architectural Design - Presents the top-level concepts behind TrickHLA.

Chapter 4: Mathematical Formulations - Presents the mathematical formulations implemented by TrickHLA.

Chapter 5: Interface Design - Describes the main interfaces to the TrickHLA model.

Chapter 6: Functional Design - Presents the detailed design of the model.

Chapter 7: Version Description - Identifies the configuration-managed items that comprise TrickHLA.

Bibliography - Informational references associated with this document.

Chapter 2

Related Documentation

2.1 Parent Documents

The following documents are parent to this document:

- *Trick High Level Architecture (TrickHLA)* [3]

2.2 Applicable Documents

The following documents are referenced herein and are directly applicable to this document:

- *TrickHLA Product Requirements* [4]
- *TrickHLA User Guide* [5]
- *TrickHLA Inspection, Verification, and Validation* [2]
- *Trick Simulation Environment: Installation Guide* [9]
- *Trick Simulation Environment: Tutorial* [10]
- *Trick Simulation Environment: Documentation* [8]
- *NASA Software Engineering Requirements* [7]

2.3 Information Documents

The following documents provide supporting material for understanding the concepts in this document:

- *Asynchronous distributed simulation via a sequence of parallel computations* [1]
- *Parallel Discrete Event Simulation* [6]

Chapter 3

Architectural Design

This chapter presents the high-level concepts behind TrickHLA and summarizes the model's architecture.

3.1 Summary

The TrickHLA model provides developers a relatively simple means of integrating HLA into Trick simulations. The model code is written in C++, since the HLA APIs are in C++. The architecture reflects this C++ foundation. In addition, since the model is intended for use with Trick, the architecture is also based on Trick-specific concepts, namely a Trick `sim_object` and default data input files.

3.2 Background

3.2.1 Purpose and Scope

Purpose. The TrickHLA model presents a high-level means by which simulation developers may integrate HLA into a simulation, thereby incorporating it into a distributed federation of simulations. In particular, the model makes this integration easier than would otherwise be the case, since the HLA programming interface is fairly complex and difficult to master.

Scope. The TrickHLA model is intended for use only in Trick-based simulations. It consists of

- several C++ classes,
- a Trick `sim_object`, and
- a default data file for that `sim_object`.

3.2.2 Goals and Objectives

Goals. The general goal of the TrickHLA model is to enable developers to build distributed HLA-based simulations with a minimum of HLA expertise.

Objectives. The specific objective of TrickHLA is to provide programming tools that allow simulation developers to easily integrate HLA into a simulation.

The module includes C++ classes and a `sim_object` which provide a high level interface to HLA — one that is easier to use and understand. Furthermore, the tight integration of HLA *publish* and *subscribe* capabilities with Trick enables developers to automatically send and receive HLA data using Trick input files instead of code. Other HLA capabilities (e.g., sending and receiving *interactions* and managing attribute *ownership*) are handled by TrickHLA classes and functions which must be invoked explicitly but are easier to work with than the standard HLA functions.

3.2.3 Concepts and Terminology

This section summarizes some key Trick and HLA terminology and concepts in order to make the subsequent presentation of the TrickHLA model easier to understand.

Trick. Trick is a simulation environment used in NASA and developed at Johnson Space Center. The architectural model for this software is as defined in, and required by, the Trick Simulation Environment. See references [8] and [10] for more information on Trick. Trick-based simulations consist of *models* which roughly speaking are sets of related data and functions which operate on them. The TrickHLA model is a Trick model in this sense, and it consists of sets of data and functions which simplify access to HLA.

Data Driven Mechanisms. In what follows, we sometimes refer to data driven mechanisms for accomplishing a task. By this we mean that the task may be specified in a Trick input file rather than by writing C or C++ code to handle the task. One of the useful aspects of Trick is that the simulations it creates are aggressively data-driven, making it possible to change many simulation parameters through input files instead of changing source code and rebuilding the simulation binary. The TrickHLA model presents data-driven mechanisms for some (but not all) HLA tasks.

High Level Architecture (HLA). The High Level Architecture (HLA) is an IEEE standard (IEEE-1516) which defines a set of *services* to facilitate the development of distributed systems. More information is available in references [13], [12], [14], and [11]. The TrickHLA model presents a Trick-friendly interface to the relatively complex HLA application programming interface (API).

Federates and Federations. The distributed system in HLA is refered to as a *federation.* Each federation has a name, and there may be many federations in existence at the same time. Each participant in a particular federation is refered to as a *federate.* A federate may *join* one or more federations. Federates have names, which must be unique in each federation to which they belong. The TrickHLA model provides a data-driven mechanism for turning a Trick simulation into an HLA federate and joining the appropriate HLA federation.

Publish and Subscribe. The publish/subscribe model of data distribution is one in which the data in a system are given names which allow *publishers* to set new values for the data and make them available to *subscribers* who must explicitly indicate their interest in specific data (by name) in order to receive them. Publish/subscribe systems allow a single publisher to distribute data to many subscribers without being aware of the specific recipients, since subscribers usually register their interest with the underlying middleware instead of directly contacting the publisher. In HLA, the term *publish* is used rather unconventionally to indicate a federate's intent to eventually generate data values. The actual act of generating those data values is called *reflection*. The TrickHLA model provides a data-driven mechanism for associating Trick simulation data with HLA, allowing simulations to publish/reflect and subscribe data without writing any code to do so.

Classes, Objects and Attributes. In HLA, an *object class* is an abstract specification for how some data may be organized. A class consists of various constituents called *attributes*. Instances of a particular HLA object class are called *object instances*. Is is these instances which actually hold data values. In HLA classes and their instances are meant to be model persistent data, i.e., things in the simulation domain that exist and have values that continue to exist over time. The TrickHLA model provides a data-driven mechanism to associate attributes with Trick simulation data.

Interactions and Parameters. In HLA, messages can be sent between federates that are not associated with things that exist over time in the simulation domain. These are called *interactions*. Like classes, they consist of various constituents, but interaction constituents are called *parameters*. Interactions and their parameters are used to model transient data. The TrickHLA model provides a data-driven mechanism to associate parameters with Trick simulation data. The TrickHLA model has functions which may be used to send interactions from a simulation to other federates in the federation. It also provides an abstract C++ class which may be subclassed by simulation developers in order to specify simulation-specific handling of incoming interactions.

Attribute Ownership. In HLA, object attributes may be *owned* by a particular federate, which means that the data values for that attribute are the responsibility of that federate. Only one federate may own an attribute at a time; however, the ownership of an attribute may be transferred from federate to federate during the execution of the federation. This transfer of ownership is refered to as *ownership management*. The TrickHLA model has functions which may be used to initiate an ownership transfer. (A current attribute owner may *push* ownership away, and a non-owner may *pull* ownership from the current owner.) The model also provides an abstract C++ class which may be subclassed by simulation developers in order to specify simulation-specific handling of ownership transfers.

Federation Object Model (FOM). Each federation must have a single FOM which specifies the object classes, attributes, interactions, and parameters which are exchanged during the federation's execution. This information is captured in an XML file refered to as the *FOM file*. The TrickHLA model provides a data-driven mechanism to associate a particular FOM file with a simulation.

Time Management and Lookahead. One of the challenges of distributed computing is how one goes about defining a consistent value of time. One of the major aspects of HLA is its *time management* services which allow distributed simulations to proceed concurrently yet have a consistent notion of time. The time management services are typically implemented on top of the Chandy-Misra-Bryant protocol[1] The performance of this protocol is affected significantly by a *time lookahead* parameter[6] which is a time interval *into the future* to which a federate can predict (or extrapolate) its current state. The use of lookahead is a fundamental aspect of HLA simulations. It is how simulations can reliably proceed in parallel without deadlocking. The TrickHLA model provides a data-driven mechanism for specifying the lookahead to be associated with the simulation.

3.3 Architectural Overview

This section provides a very high-level view of the TrickHLA model. Finer details of this model are presented in Chapters 5 and 6. The model can be described as three layers as depicted in Figure 3.1

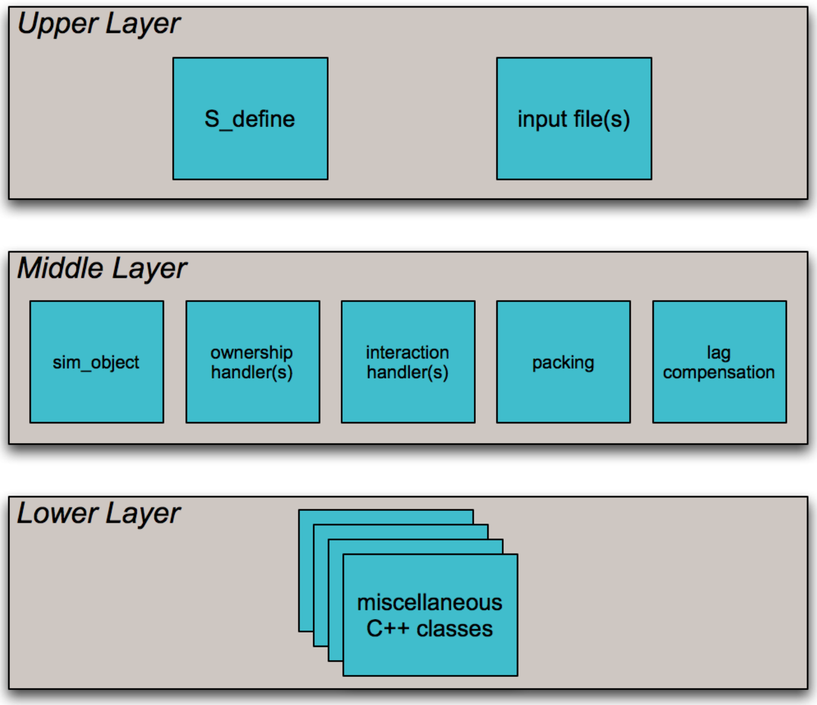

Figure 3.1: TrickHLA Layers

3.3.1 The Upper Layer

The upper layer of the TrickHLA model consists of simulation-specific artifacts[1] which must be modified in order to use the model. These are discussed in the following sections.

The `S_define` File

To integrate TrickHLA into a Trick simulation, the simulation's `S_define` file must be modified to include HLA-specific data structures and jobs. To a large extent, this can be accomplished by inserting the default TrickHLA `sim_object` (see below) into the `S_define` file. However, in order to use HLA interactions or ownership management, additional *handler* data structures (see below) must be inserted into the `S_define` in addition to the `sim_object`.

The Input Files

As discussed on page 5, Trick is a data-driven system. Accordingly, most parameters related to the TrickHLA model are initialized in the Trick input file fashion. For example, the name of the federation which the simulation is joining as well as the hostname and port of the HLA runtime system are specified in input files. In addition, if the simulation publishes/reflects any simulation data to other federates or subscribes to data from other federates, the names of the data are specified in the input files.

3.3.2 The Middle Layer

The model's middle layer consists of the default `sim_object` and several other classes that allow developers to customize their handling of interactions, ownership transfer, packing and unpacking of transmitted data, actions to run upon the deletion of a federate, saving the federation, sending data conditionally and lag compensation.

The Default TrickHLA `sim_object`

In order to simplify the process of integrating TrickHLA into existing (non-HLA) simulations, the model includes a pre-built `sim_object` which may be inserted into the `S_define` verbatim, i.e., the HLA-specific data structures and jobs can be mostly contained to this single object. The object itself mainly consists of three interrelated data structures and jobs related to them:

TrickHLA::Manager: This is a C++ class which is mainly responsible for handling the flow of data traffic between the core simulation and HLA. In particular, it has data structures and methods (which are invoked as jobs) responsible for sending and receiving HLA data to/from remote federates.

[1] In a sense, these are not elements of the model, since they are not artifacts delivered as part of the model, but since they are important in understanding how the model works, we have chosen to explicitly include them in this discussion.

TrickHLA::Federate: This is a C++ class which has methods for coordinating HLA with the Trick simulation time and freeze/unfreeze. In particular, these methods are responsible for handling HLA time management and making sure it is properly synchronized with the simulation time.

TrickHLA::FedAmb: This is a C++ class which extends the abstract HLA class, `FederateAmbassador`. The federate ambassador implements many methods required by the HLA API. In particular, the HLA runtime infrastructure asynchronously invokes these methods in order to notify the federate that some event occured (for example when an object attribute value has been updated by a remote federate).

TrickHLA::ExecutionControl: This is a C++ class which extends the abstract TrickHLA class, `ExecutionControlBase`. This class is used to control the execution of the federate in a defined HLA federation execution. This class will implement the execution control strategy common to all members of a federation execution. TrickHLA provides a muber of available execution control implementations. For instance, the SISO standard Space Refernce Federation Object Model (SpaceFOM) execution control stategy is provided in the `SpaceFOM::ExecutionControl` class.

TrickHLA::ExecutionConfiguration: This is a C++ class which extends the abstract TrickHLA class, `ExecutionConfigurationBase`. This class is used to provide configuration data across a coordinated federation execution. An execution configuration class is usually paired with an execution control class. For instance, the `SpaceFOM::ExecutionConfiguration` class is used with the `SpaceFOM::ExecutionControl` class.

Interaction Handlers

Interactions are not part of the infrastructure in the default `sim_object`. In order to send or receive interactions, you must explicitly insert an interaction handler data structure into the `S_define` file. The handler class has a method which may be invoked from the `S_define` file as a scheduled job. The handler is specified in the input file for each interaction the simulation expects to receive.

Ownership Handlers

Ownership transfer is also managed by a handler class. In order to change ownership of an attribute, you must explicitly insert an ownership handler data structure into the `S_define` file. The handler class has methods which "schedule" ownership transfers at some future time. By default, no such transfers exist, but by defining one of these ownership handlers you may arrange for ownership changes using either a *push* or *pull* model. When a push or pull occurs, a job in the default `sim_object` handles the low-level details of coordinating the HLA ownership transfer transaction.

Data Packing and Unpacking

The TrickHLA model includes an abstract class which may be subclassed by simulation developers in order to incorporate simulation-specific packing and unpacking of data as they arrive from and are sent to remote federates.

Object Deleted

The TrickHLA model includes an abstract class which may be subclassed by simulation developers in order to incorporate simulation specific action(s), which would take place upon the deletion of an object.

Sending Data Conditionally

The TrickHLA model includes an abstract class which may be subclassed by simulation developers in order to incorporate decision(s) when to send an individual simulation attribute across the wire. If not overidden, the attribute is sent over the wire on each data cycle.

Lag Compensation

The underlying HLA time management mechanisms involve a protocol which involves extrapolation of the system's state forward into the future. It is these forward predictions of the system state that are sent to remote federates. This mechanism is what permits the distributed federates to operate concurrently but at the same time maintain a consistent notion of the current simulation time. However, protocol introduces simulation-time lags in the data each time attribute ownership is transfered from one federate to another. In order to compensate for this lag, sending or receiving federates may subclass an abstract lag compensation class included in the TrickHLA model. This class includes methods that act as hooks allowing the federates to extrapolate the state in time in order to compensate for the protocol-introduced time lag.

3.3.3 The Lower Layer

This layer consists of a number of C++ classes that implement HLA constructs such as objects, attributes, interactions, parameters, and synchronization points. In addition there are some low level classes that serve primarily to hold utility methods, e.g., string and byte-swapping methods.

Chapter 4

Mathematical Formulations

Not Applicable.

Chapter 5

Interface Design

This chapter introduces the main interfaces to the TrickHLA model. Since the model is intended to be used in a Trick simulation, these interfaces are all Trick-related.

5.1 Summary

The main interface between the TrickHLA model and Trick is the default `sim_object`. Inclusion of this `sim_object` in an `S_define` file results in the simulation automatically joining a distributed HLA federation, sending certain specified simulation data to the federation as they change, and receiving certain changing data from other simulations in the federation. The details of this process (i.e., which federation to join, which data to send, and which data to subscribe to) are configured through standard Trick input files.

Interfaces to other HLA capabilities (e.g., interactions and ownership management) are through TrickHLA classes and their methods, which may be invoked as Trick jobs or directly from simulation-specific code.

This section summarizes these interfaces.

5.2 The Main TrickHLA Interface

The main TrickHLA interface is the default `sim_object` that is distributed with the model. This `sim_object` contains data and jobs that handle

- initialization of the HLA infrastructure,
- joining the specified federation and waiting for all expected federates to join,
- sending simulation data to other federates,
- receiving simulation data from other federates,
- handling the HLA time advancement logic and

- saving the federation.

Using this default `sim_object` involves two steps: (a) inserting the object in the simulation's `S_define` file, and (b) specifying the relevant configuration data in Trick input files. These steps are discussed below.

5.2.1 The TrickHLA `sim_object`

In most cases a Trick simulation may be integrated with HLA by simply inserting the default TrickHLA `sim_object` to the simulation's `S_define` file and then configuring the object appropriately using Trick input files. An abbreviated version of the `sim_object` is shown in Figure 5.1.

The contents of this `sim_object` include:

- data declarations,
- scheduled jobs,
- initialization jobs,
- freeze related jobs, and
- shutdown jobs.

The data declarations are partially explained in the discussion of input files, below.[1] The initialization jobs handle the initialization of the internal TrickHLA classes and need not concern most simulation developers. The freeze related jobs are included so that freezing a Trick job does not completely disable underlying HLA-related threads. And the shutdown jobs handle resignation from the HLA federation when the Trick simulation completes.

Other than the requirement that certain data be initialized consistently with the needs of the simulation, this `sim_object` may be treated as a black box – i.e., there is usually no need to modify it after copying it verbatim into the simulation's `S_define` file. The following section elaborates on how to appropriately set the input data.

5.2.2 Input Data Files

The various data structures declared in the TrickHLA default `sim_object` must be initialized through the Trick simulation input file. The following sections discuss the various initialization parameters.

HLA Runtime and Federation-Related Parameters

This section describes the input parameters necessary to connect the simulation to the HLA runtime infrastructure and other data used to set up the federate. The parameters are discussed in Table 5.1.

[1]Other than knowing how to initialize the significant components of these data, there is no need to understand their internal structure in detail.

Time-Related Parameters

This section describes the time-related input parameters. This includes parameter that govern HLA time management as well as parameters related to whether or not the simulation attempts to keep HLA simulation time synchronized with the "wallclock." The parameters are discussed in Table 5.2.

Object and Attribute-Related Parameters

This section describes the input parameters that govern HLA object instances and their constituent attributes. The parameters are discussed in Table 5.3.

[3]The `typedef`, `TrickHLA::TransportationEnum`, is an enum with the following values: `TRANSPORT_SPECIFIED_IN_FOM`, `TRANSPORT_TIMESTAMP_ORDER`, and `TRANSPORT_RECEIVE_ORDER`.

[3]The `typedef`, `TrickHLA::EncodingEnum`, is an enum with the following values: `ENCODING_UNKNOWN`, `ENCODING_BIG_ENDIAN`, `ENCODING_LITTLE_ENDIAN`, `ENCODING_LOGICAL_TIME`, `ENCODING_C_STRING`, `ENCODING_UNICODE_STRING`, `ENCODING_ASCII_STRING`, `ENCODING_OPAQUE_DATA`, `ENCODING_BOOLEAN`, and `ENCODING_NO_ENCODING`.

```
sim_object {
   TrickHLA: TrickHLAFreezeInteractionHandler freeze_ih;
   TrickHLA: TrickHLAFedAmb   federate_amb;
   TrickHLA: TrickHLAFederate federate;
   TrickHLA: TrickHLAManager  manager;
   double checkpoint_time;
   char   checkpoint_label[256];

   P1 (initialization) TrickHLA: THLA.manager.print_version();

   P1 (initialization) TrickHLA: THLA.federate.fix_FPU_control_word();

   P60 (initialization) TrickHLA: THLA.federate_amb.initialize(
      In TrickHLAFederate * federate = &THLA.federate,
      In TrickHLAManager  * manager  = &THLA.manager );

   P60 (initialization) TrickHLA: THLA.federate.initialize(
      Inout TrickHLAFedAmb * federate_amb = &THLA.federate_amb );
   P60 (initialization) TrickHLA: THLA.manager.initialize(
      In TrickHLAFederate * federate = &THLA.federate );

   P65534 (initialization) TrickHLA: THLA.manager.initialization_complete();

   P65534 (initialization) TrickHLA: THLA.federate.check_pause_at_init(
      In const double check_pause_delta = THLA_CHECK_PAUSE_DELTA );

   P1 (checkpoint)          TrickHLA: THLA.federate.setup_checkpoint();
   (freeze)                 TrickHLA: THLA.federate.perform_checkpoint();
   P1 (pre_load_checkpoint) TrickHLA: THLA.federate.setup_restore();
   (freeze)                 TrickHLA: THLA.federate.perform_restore();

   (freeze)   TrickHLA: THLA.federate.check_freeze();
   (unfreeze) TrickHLA: THLA.federate.exit_freeze();

   P1 (THLA_DATA_CYCLE_TIME, environment) TrickHLA: THLA.federate.wait_for_time_advance_grant();

   P1 (THLA_INTERACTION_CYCLE_TIME, environment) TrickHLA: THLA.manager.process_interactions();

   P1 (THLA_DATA_CYCLE_TIME, environment) TrickHLA: THLA.manager.process_deleted_objects();

   P1 (0.0, environment) TrickHLA: THLA.manager.start_federation_save(
      In const char * file_name = THLA.checkpoint_label );

   P1 (0.0, environment) TrickHLA: THLA.manager.start_federation_save_at_sim_time(
      In double freeze_sim_time = THLA.checkpoint_time,
      In const char * file_name = THLA.checkpoint_label );

   P1 (0.0, environment) TrickHLA: THLA.manager.start_federation_save_at_scenario_time(
      In double freeze_sim_time = THLA.checkpoint_time,
      In const char * file_name = THLA.checkpoint_label );

   P1 (THLA_DATA_CYCLE_TIME, environment) TrickHLA: THLA.manager.receive_cyclic_data(
      In double current_time = sys.exec.out.time );

   P65534 (THLA_DATA_CYCLE_TIME, logging) TrickHLA: THLA.manager.send_cyclic_data(
      In double current_time = sys.exec.out.time );

   P65534 (THLA_DATA_CYCLE_TIME, logging) TrickHLA: THLA.manager.send_requested_data(
      In double current_time = sys.exec.out.time );

   P65534 (THLA_DATA_CYCLE_TIME, logging) TrickHLA: THLA.manager.process_ownership();

   P65534 (THLA_DATA_CYCLE_TIME, logging) TrickHLA: THLA.federate.time_advance_request();

   P65534 (THLA_DATA_CYCLE_TIME, logging) TrickHLA: THLA.federate.check_freeze_time();

   P65534 (THLA_DATA_CYCLE_TIME, THLA_CHECK_PAUSE_JOB_OFFSET, logging) TrickHLA: THLA.federate.check_pause(
      In const double check_pause_delta = THLA_CHECK_PAUSE_DELTA );

   P65534 (THLA_DATA_CYCLE_TIME, THLA_CHECK_PAUSE_JOB_OFFSET, logging) TrickHLA: THLA.federate.enter_freeze();

   P65534 (shutdown) TrickHLA: THLA.manager.shutdown();
} THLA;
```

parameter name	type	description
`THLA.federate.local_settings`	string	RTI vendor specific string that specifies the CRC hostname or IP address with port number where the HLA runtime executive is running
`THLA.federate.name`	string	federate name to be assigned to the simulation
`THLA.federate.enable_FOM_validation`	true or false	true to enable FOM validation or false to disable it (default: true)
`THLA.federate.FOM_modules`	string	name of the FOM file when using the IEEE 1516-2000 and SISO-STD-004-2004 standards, or a comma separated list of FOM-module filenames when using the HLA Evolved IEEE 1516-2010 standard
`THLA.federate.MIM_module`	string	name of the MOM and Initialization Module (MIM) file for the HLA Evolved IEEE 1516-2010 standard
`THLA.federate.federation_name`	string	name of the federation to join
`THLA.federate.enable_known_feds`	true or false	should the simulation wait for all the other (known) federates in the federation before it begins?
`THLA.federate.known_feds_count`	integer	how many known federations are there?
`THLA.federate.known_feds`	alloc(N_{feds})	allocate an array with N_{feds} elements, where N_{feds} is the value of **`THLA.federate.known_feds_count`**
`THLA.federate.known_feds[`i`].name`	string	the name of known federate $i \in 0...N_{feds} - 1$ where N_{feds} is the value of **`THLA.federate.known_feds_count`**
`THLA.federate.known_feds[`i`].required`	true or false	is federate $i \in 0...N_{feds} - 1$ required to be present before this federate begins, where N_{feds} is the value of **`THLA.trick_fed.-known_feds_count`**
`THLA.federate.use_preset_master`	true or false	true to force the use of the preset THLA.federate.master federate flag value or false to automatically determine the master federate during the multiphase initialization process (default: false)
`THLA.federate.master`	true or false	true when this federate is the master federate for the multiphase initialization process. When THLA.federate.use_preset_master is set to true then the THLA.federate.master flag is used to preset the master federate, otherwise the THLA.federate.master flag is ignored (default: false)
`THLA.federate.can_rejoin_federation`	true or false	true to signal this federate to resign from the federation in a manner which makes rejoining the running federation possible. See Section 16 of the IMSim_Multiphase_Init_Design_Document.pdf for more details. (default: false)
`THLA.federate.scenario_timeline`	TrickHLA::Timeline	A pointer to a TrickHLA::Timeline implementation for a scenario timeline. If nothing is specified TrickHLA will use the TrickHLA::SimTimeline implementation, which uses the Trick simulation time as the default scenario timeline and is only valid if all federates are Trick based simulations. (default: TrickHLA::SimTimeline)
`THLA.federate.multiphase_init_sync_points`	string	A comma-separated list of the simulation specific multiphase initialization synchronization-point labels as specified in the Federation Agreement.

Table 5.1: HLA Runtime and Federation-Related Parameters

parameter name	type	description
`THLA.federate.lookahead_time`	float	lookahead time for HLA time management
`THLA.federate.time_regulating`	true or false	is this a *time regulating* federate?
`THLA.federate.time_constrained`	true or false	is this a *time constrained* federate?
`THLA.federate.time_management`	true or false	enables / disables HLA time management
`sys.exec.in.rt_software_frame`	float	realtime frame (in seconds). The Trick timing loop will synchronize HLA simulation time with the wallclock at the end of this interval. Usually that means suspending the simulation until the specified interval has elapsed on the wallclock. Set this to a very large number to disable realtime execution.
`sys.exec.in.rt_itimer_frame`	float	itimer frame (in seconds). If realtime is enabled, set this value to the same value as sys.exec.in.rt_software_frame.
`sys.exec.in.rt_timer`	Yes or No	should Trick use itimers when synchronizing HLA simulation time with the "wallclock"? To avoid a spin lock (which consumes CPU cycles) while the simulation waits for simulation/wallclock time synchronization, this value should be set to Yes.

Table 5.2: Time-Related Parameters

parameter name	type	description
`THLA.manager.obj_count` (N_{objs})	integer	the number of object instances to be handled by this federate
`THLA.manager.objects`	alloc(N_{objs})	allocate an array with N_{objs} elements
let $i \in 0...N_{objs} - 1$		
`THLA.manager.objects[`i`].FOM_name`	string	the HLA class name for object[i] as specified in the FOM file
`THLA.manager.objects[`i`].name`	string	the HLA object instance name for object[i]
`THLA.manager.objects[`i`].-name_required`	true or false	specifies whether the object instance name is required. The default is true, which requires an object instance name.
`THLA.manager.objects[`i`].-create_HLA_instance`	true or false	specifies whether an HLA object instance is created
`THLA.manager.objects[`i`].required`	true or false	specifies whether the object is required for startup of the federation. The default is true, which requires the object intance to exist for startup of the federation.
`THLA.manager.objects[`i`].-blocking_cyclic_read`	true or false	should a cyclic read for this object block waiting for data. The default is to not block on cyclic reads.
`THLA.manager.objects[`i`].packing`	string	the Trick name of a C++ packing object for attributes of object[i]. If no packing is associated with attributes of this object, this parameter may be omitted.
`THLA.manager.objects[`i`].ownership`	string	the Trick name of a C++ ownership handler object for attributes of object[i]. If no ownership transfer is associated with attributes of this object, this parameter may be omitted.
`THLA.manager.objects[`i`].deleted`	string	the Trick name of a C++ object deleted callback object for object [i]. If no object deleted callback is associated with attributes of this object, this parameter may be omitted.
`THLA.manager.objects[`i`].attr_count` (N_{attrs})	integer	the number of attributes associated with object[i]
`THLA.manager.objects[`i`].attributes`	alloc(N_{attrs}),	allocate an array with N_{attrs} elements
let $j \in 0...N_{attrs} - 1$		
`THLA.manager.objects[`i`].-attributes[`j`].FOM_name`	string	the HLA name of attribute[j] according to the FOM file
`THLA.manager.objects[`i`].-attributes[`j`].trick_name`	string	the name of the Trick variable with which this attribute is associated. When Trick publishes values for this attribute, the values sent out are those of the associated Trick variable. Similarly, when Trick subscribes to this attribute, incoming data are assigned to the associated Trick variable.
`THLA.manager.objects[`i`].-attributes[`j`].config`	`TrickHLA::DataUpdateEnum`	Use the enum value `CONFIG_CYCLIC` to allow this attribte to be cyclically sent or received.
`THLA.manager.objects[`i`].-attributes[`j`].locally_owned`	true or false	is attribute[j] initially owned by this federate?
`THLA.manager.objects[`i`].-attributes[`j`].publish`	true or false	is this federate publishing values for attribute[j]? Federates may only publish values for attributes that are locally owned. If this is set to true, TrickHLA will automatically publish values for this attribute as specified in the `sim_object`.
`THLA.manager.objects[`i`].-attributes[`j`].subscribe`	true or false	is this federate subscribing values for attribute[j]? If this is set to true, TrickHLA will automatically subscribe to values for this attribute as specified in the `sim_object`.
`THLA.manager.objects[`i`].-attributes[`j`].preferred_order`	`TrickHLA::TransportationEnum`[2]	override the preferred send order of this attribute. Use the default enum value `TRANSPORT_SPECIFIED_IN_FOM` to use the order specified in FOM. Use the enum value `TRANSPORT_TIMESTAMP_ORDER` to override the FOM and use a timestamp order. Use the enum value `TRANSPORT_RECEIVE_ORDER` to override the FOM and use a receive order.
`THLA.manager.objects[`i`].-attributes[`j`].rti_encoding`	`TrickHLA::EncodingEnum`[3]	the wire format of the values of this attribute. This specifies how the values are converted to/from Trick variables. Big and little endian formats specify numerical data with the associated byte order. C strings are null-terminated strings of bytes. The Unicode and ASCII string formats are documented in the IEEE Standard 1516.2-2010, section 4.13.4. Opaque data is for non-numeric, non-string data and is also documented in 1516.2-2010, section 4.13.6.
`THLA.manager.objects[`i`].-attributes[`j`].cycle_time`	integer	`default:` 1. Sends cyclic attributes at the specified cycle-time provided that the time is an integer multiple of the core job cycle time of the send_cyclic_data job. Ex: A cycle-time of 4 * THLA_DATA_CYCLE_TIME would send this attribute every fourth time the send_cyclic_data job was called.
`THLA.manager.objects[`i`].-attributes[`j`].conditional`	`TrickHLAConditional *`	`default: NULL`. If overridden, must point to subclass which makes the decision if this attribute is to be sent over the wire in each frame. Otherwise, this attribute is sent over the wire for each simulation frame.

Table 5.3: Object and Attribute-Related Parameters

5.3 Other HLA Interfaces

5.3.1 Interactions

Input Data

This section describes the input parameters that govern HLA interactions and their parameters. The parameters are discussed in Table 5.4.

parameter name	type	description
`THLA.manager.inter_count` (N_{ints})	integer	the number of interactions to be handled by this federate
`THLA.manager.interactions`	alloc(N_{ints}),	allocate an array with N_{ints} elements
let $i \in 0...N_{ints} - 1$		
`THLA.manager.-interactions[`i`].FOM_name`	string	the HLA name for interaction[i] as specified in the FOM file
`THLA.manager.-interactions[`i`].publish`	true or false	can this federate send interaction[i] to other federates?
`THLA.manager.-interactions[`i`].subscribe`	true or false	can this federate receive interaction[i] from other federates?
`THLA.manager.-interactions[`i`].preferred_order`	`TrickHLA::TransportationEnum`	override the preferred send order of this interaction. Use the default enum value `TRANSPORT_SPECIFIED_IN_FOM` to use the order specified in FOM. Use the enum value `TRANSPORT_TIMESTAMP_ORDER` to override the FOM and use a timestamp order. Use the enum value `TRANSPORT_RECEIVE_ORDER` to override the FOM and use a receive order.
`THLA.manager.-interactions[`i`].handler`	string	the Trick name of a C++ interaction handler object for interaction[i].
`THLA.manager.-interactions[`i`].param_count` (N_{params})	integer	the number of parameters associated with interaction[i]
`THLA.manager.-interactions[`i`].parameters`	alloc(N_{params})	allocate an array with N_{params} elements
let $j \in 0...N_{params} - 1$		
`THLA.manager.-interactions[`i`].parameters[`j`].-FOM_name`	string	the name of parameter[j] as specified in the FOM file
`THLA.manager.-interactions[`i`].parameters[`j`].-trick_name`	string	the name of the Trick variable with which this parameter is associated.
`THLA.manager.-interactions[`i`].parameters[`j`].-rti_encoding`	`TrickHLA::EncodingEnum`	the wire format of the values of parameter[j]. The encoding values have the same meaning as the attribute encodings discussed in Table 5.3.

Table 5.4: Interaction-Related Parameters

C++ Handler Classes

The TrickHLA model includes a base class, `TrickHLA::InteractionHandler`, which forms the foundation of how developers handle HLA interactions. The class implements two `send_interaction()` methods (one for receive order and the other for timestamp order transmission of the interaction), and it defines a virutal method, `receive_interaction()`, which must be implemented in subclasses. (The default implementation of the method does nothing.)

The base class `send_interaction()` methods may be used directly to send interactions to remote federates. You may invoke these methods directly in the simulation `S_define` file as scheduled jobs or from internal simulation code.

In order to receive interactions generated remotely, you must subclass the base class, overriding the `receive_interaction()` method with application-specific logic. Assuming that the handler(s) have been associated with the appropriate interactions as discussed in Table 5.4, the TrickHLA infrastructure will invoke the appropriate `receive_interaction()` method for each arriving interaction.

In both cases (sending and receiving) the HLA interaction parameter values are associated with Trick variables as discussed in Table 5.4.

5.3.2 Ownership Transfer

The TrickHLA model includes an ownership handler class, `TrickHLA::OwnershipHandler`, which may be used to transfer ownership of specific HLA attributes of a particular object from one federate to another. There are two sets of methods that do this.

The `push_ownership()` methods may be used by a developer to divest ownership of the specified attributes. The `pull_ownership()` methods may be used by a developer to take over ownership of the specified attributes. The methods enqueue push/pull requests which are processed by the jobs in the default TrickHLA `sim_object`.

Ownership handlers are associated with particular objects through the input files, as discussed in Table 5.3.

5.3.3 Packing and Unpacking

The TrickHLA model includes a base class, `TrickHLA::Packing`, which is used for packing HLA data before it is sent to remote federates and unpacking data just after it has been received. The class consists of two virtual functions, `pack()` and `unpack()`, the default implementations will terminate the simulation, therefore you must subclass the base class in order to use it, overriding the two methods.

Packing objects are associated with particular objects through the input files, as discussed in Table 5.3.

5.3.4 Object Deleted

The TrickHLA model includes a base class, `TrickHLA::ObjectDeleted`, which is triggered when the object is deleted from the federation. The class consists of one virtual function, `deleted()`, the default implementation of which does nothing, therefore you must subclass the base class in order to use it, overriding the method.

Object Deleted are associated with particular objects through the input files, as discussed in Table 5.3.

5.3.5 Sending Data Conditionally

The TrickHLA model includes a class, `TrickHLA::Conditional`, which must be subclassed by the user's simulation for each attribute that they wish to send across the wire based upon user-determinted criteria.

The class consists of a method, `should_send()`, which must be filled in by the user to provide the aforementioned criteria and return true if it has been met.

5.3.6 Lag Compensation

C++ Classes

The TrickHLA model includes a base class, `TrickHLA::LagCompensation`, which provides methods that allow federates to compensate for HLA-induced lags. The virtual methods, `send_lag_compensation()` and `receive_lag_compensation()` allow this compensation to be executed by the sender of data or by the receivers. The default methods do nothing, therefore you must subclass the base class in order to use it, overriding the two methods.

Input Data

To associate a lag compensation class with specific HLA objects, the input data described in Table 5.3 must be expanded to include an additional parameter as shown in Table 5.5.

parameter name	type	description
`THLA.manager.objects[`*i*`].lag_comp`	string	the Trick name of an application-specific lag compensation object. This is an instance of a subclass of `TrickHLA::LagCompensation` which has been declared in the `S_define` file.
`THLA.manager.objects[`*i*`].-lag_comp_type`	`TrickHLA::LagCompensation`	an enumerated type the specified whether the federate uses send-side lag compensation, receive-side lag compensation, or none at all. This flag tells the TrickHLA infrastructure whether to invoke the `send_lag_compensation()` method of the specified lag compensation class, the `receive_lag_compensation()` method, or neither.

Table 5.5: Lag Compensation Parameters

5.3.7 Requesting Data Updates

The TrickHLA model includes a class, `TrickHLA::Manager`, which manages the HLA data transfers in and out of the user's simulation.

The `TrickHLA::Manager` class includes a method, `request_data_update()`, which may be called by the user's simulation to request a data update for a specific named object instance.

5.3.8 Multiple Verbosity Levels

The TrickHLA model includes a class, `TrickHLA::DebugHandler`, which is to be used to specify the verbosity level of the TrickHLA software. It also allows the user the ability to narrow down the specific code modules that are to emit debug messages.

The settings made to this class need to be done once in the input file, as described below, and the values are automatically propagated to all subsystems of the TrickHLA software.

Table 5.6 lists the verbosity levels available to the user, found in the `TrickHLA::DebugSourceEnum` enumeration, while table 5.7 discusses the code section control available to the user, found in the `TrickHLA::DebugSourceEnum` enumeration. Both enumerations are defined in the `TrickHLA/Types.hh` file.

Setting of the verbosity level is accomplished by setting the class' `debug_level` value, found inside `THLA.manager.debug_handler` object, within the simulation's input file. Note: It is recommended that the user utilize the enumerated values to specify the desired verbosity level. If the user specifies the `debug_level` value via an integer, the value will be range-checked and any integer value outside of the enumerated values range will recieve a warning message on the console.

For example, the following command will enable debug level 3 messages to get emitted on the console:

```
// Show or hide the TrickHLA debug messages.
THLA.manager.debug_handler.debug_level = DEBUG_LEVEL_3_TRACE;
```

Specifying which code sections are to emit debug messages is accomplished by setting the class' `code_section` value, found inside `THLA.manager.debug_handler` object, within the simulation's input file. Note: These values are binarily unique so it is recommended that the user add all desired code sections' enumarated value when specifying them in the simulation's input file.

For example, the following command will enable debug messages to be emitted only from `TrickHLA::Manager`, `TrickHLA::Federate` and `TrickHLA::FedAmb` source code modules:

```
THLA.manager.debug_handler.code_section = DEBUG_SOURCE_MANAGER +
                                          DEBUG_SOURCE_FEDERATE +
                                          DEBUG_SOURCE_FED_AMB;
```

enumeration name	value	description
`DEBUG_LEVEL_NO_TRACE`	0	`default`: Disables all TrickHLA messages and no TrickHLA messages will be displayed on the user's console. Any messages emitted by the user's simulation will still be emitted.
`DEBUG_LEVEL_0_TRACE`	0	`default`: Disables all TrickHLA messages and no TrickHLA messages will be displayed on the user's console. Any messages emitted by the user's simulation will still be emitted.
`DEBUG_LEVEL_1_TRACE`	1	Adds initialization complete and Time Advance Grant messages.
`DEBUG_LEVEL_2_TRACE`	2	Adds initialization messages as well as the standard complement of execution messages.
`DEBUG_LEVEL_3_TRACE`	3	Adds Ownership Transfer messages.
`DEBUG_LEVEL_4_TRACE`	4	Adds HLA Time Advancement and Freeze job messages.
`DEBUG_LEVEL_5_TRACE`	5	Adds Interaction, InitSyncPts and SyncPts messages.
`DEBUG_LEVEL_6_TRACE`	6	Adds Packing / LagCompensation subclass messages.
`DEBUG_LEVEL_7_TRACE`	7	Adds the names of all Attributes / Parameters sent to other federates.
`DEBUG_LEVEL_8_TRACE`	8	Adds FederateAmbassador and RTI callback messages.
`DEBUG_LEVEL_9_TRACE`	9	Adds Trick Ref-Attributes and RTI Handles (both during initialization).
`DEBUG_LEVEL_10_TRACE`	10	Adds internal state of all Attributes and Parameters.
`DEBUG_LEVEL_11_TRACE`	11	Adds buffer contents of all Attributes and Parameters.
`DEBUG_LEVEL_FULL_TRACE`	11	Adds "all of the above".

Table 5.6: TrickHLA::DebugLevelEnum values

enumeration name	description
`DEBUG_SOURCE_FED_AMB`	Adds `TrickHLA::FedAmb` debug messages to the console.
`DEBUG_SOURCE_FEDERATE`	Adds `TrickHLA::Federate` debug messages to the console.
`DEBUG_SOURCE_MANAGER`	Adds `TrickHLA::Manager` debug messages to the console.
`DEBUG_SOURCE_OBJECT`	Adds `TrickHLA::Object` (and subclass) debug messages to the console.
`DEBUG_SOURCE_INTERACTION`	Adds `TrickHLA::Interaction` (and subclass) debug messages to the console.
`DEBUG_SOURCE_ATTRIBUTE`	Adds `TrickHLA::Attribute` debug messages to the console.
`DEBUG_SOURCE_PARAMETER`	Adds `TrickHLA::Parameter` debug messages to the console.
`DEBUG_SOURCE_SYNCPOINT`	Adds `TrickHLA::SyncPnt` debug messages to the console.
`DEBUG_SOURCE_OWNERSHIP`	Adds `TrickHLA::OwnershipHandler` debug messages to the console.
`DEBUG_SOURCE_PACKING`	Adds `TrickHLAPacking` (and subclass) debug messages to the console.
`DEBUG_SOURCE_LAG_COMPENSATION`	Adds `TrickHLA::LagCompensation` (and subclass) debug messages to the console.
`DEBUG_SOURCE_ALL_MODULES`	`default`: Adds debug messages from all code modules to the console.

Table 5.7: TrickHLA::DebugSourceEnum values

Chapter 6

Functional Design

This chapter presents the detailed design of the TrickHLA model. We present the model in terms of its C++ classes, grouping them into low, middle, and high-level classes. We proceed from the bottom up.

6.1 Low Level Classes and Types

These low level classes are not particularly interesting. The are mainly miscellaneous utilities and low level data types used in other parts of the system.

6.1.1 `TrickHLA::Utilities`

Description. This class contains miscellaneous utility methods. In the current version, these are all static byteswapping routines.

Files. The class files are shown below.

file name	description
`TrickHLA/Utilities.hh`	standard C++ header file
`TrickHLA/Utilities.cpp`	implementation of the byteswap methods

6.1.2 `TrickHLA::StringUtilities`

Description. A set of static string utilities. In the current version, these convert between C, C++ and wide string representations.

Files. The class files are shown below.

file name	description
`TrickHLA/StringUtilities.hh`	standard C++ header file (all the utilities are defined inline)

6.1.3 `TrickHLA::KnownFederate`

Description. A class that holds information about the known federates in a federation. This class is used by the TrickHLA model to wait for all known federates before starting the simulation. The class is really meant as a structure. Its data fields are public. The fields specify the known federate's *name* and whether or not the federate is required before the simulation starts.

Files. The class files are shown below.

file name	description
`TrickHLA/KnownFederate.hh`	standard C++ header file

6.1.4 `TrickHLA::Int64Interval`

Description. An implementation of HLA time intervals based on a 64 bit integer that stores microseconds. It is a subclass of the HLA class, `LogicalTimeInterval`. It implements all the methods required by that class.

Files. The class files are shown below.

file name	description
`TrickHLA/Int64Interval.hh`	standard C++ header file
`TrickHLA/Int64Interval.cpp`	method implementation file

6.1.5 `TrickHLA::Int64Time`

Description. An implementation of HLA time based on a 64 bit integer that stores microseconds. It is a subclass of the HLA class, `LogicalTime`. It implements all the methods required by that class.

Files. The class files are shown below.

file name	description
`TrickHLA/Int64Time.hh`	standard C++ header file
`TrickHLA/Int64Time.cpp`	method imlemenation file

6.1.6 `TrickHLA::Types`

Description. enums and typedefs used by various parts of the system

Files. The class files are shown below.

file name	description
`TrickHLA/Types.hh`	enum definitions

6.1.7 `TrickHLA::DebugHandler`

Description. A class which defines the verbosity level and the code sections which are to emit messages. Unless overridden in the input file, no messsages are emitted by the TrickHLA software – only the simulation messages, if any, will get emitted.

Files. The class files are shown below.

file name	description
`TrickHLA/DebugHandler.hh`	standard C++ header file

6.2 Middle Level Classes

The middle level classes represent core HLA abstractions: synchronization points, objects, attributes, interactions, and parameters.

6.2.1 `TrickHLA::SyncPntListBase`

Description. This class provides a mechanism for storing and managing HLA synchronization points. A set of public methods is exported that enable synchronization points to be added to the system, achieved, acknowledged, and queried.

Files. The class files are shown below.

file name	description
`TrickHLA/SyncPntListBase.hh`	standard C++ header file
`TrickHLA/SyncPntListBase.cpp`	method implementation file

6.2.2 `TrickHLA::Object`

Description. This class represents HLA objects. It is large class with many methods. The higher level classes delegate many tasks to this class. The public methods associate the object with the high-level `TrickHLA::Manager` class, handle name reservation, publishing, subscribing, object registration, instance removal, publishing and subscribing of Trick variables, ownership transfer, object deletion, and lag compensation. Lag compensation and ownership transfer are delegated to associated helper classes. It also exports method related to its constituent attributes which are referenced through a private array of `TrichHLA::Attribute` instances.

Files. The class files are shown below.

file name	description
`TrickHLA/Object.hh`	standard C++ header file
`TrickHLA/Object.cpp`	method implemetation file

6.2.3 `TrickHLA::Attribute`

Description. This class represents an HLA attribute.

Files. The class files are shown below.

file name	description
`TrickHLA/Attribute.hh`	standard C++ header file
`TrickHLA/AttributeNoICG.hh`	C++ header file that is not preprocessed by Trick
`TrickHLA/Attribute.cpp`	method implementation file

6.2.4 `TrickHLA::Interaction`

Description. This class represents an HLA interaction. Its methods are responsible for sending interactions to remote federates and receiving remotely generated interations. The logic for handling incoming interactions is delegated to a helper class.

Files. The class files are shown below.

file name	description
`TrickHLA/Interaction.hh`	standard C++ header file
`TrickHLA/Interaction.cpp`	method implemetation file

6.2.5 `TrickHLA::Parameter`

Description. This class represents HLA parameters of an interaction. The public methods support manipulation of the parameter's FOM name, HLA handle, and encoded and decoded values.

Files. The class files are shown below.

file name	description
`TrickHLA/Parameter.hh`	standard C++ header file
`TrickHLA/Parameter.cpp`	method imlementation file

6.3 Upper Level Classes

The high level classes represent TrickHLA functionality. Some of these classes must be subclasses by simulation developers and others are not intended for direct use by developers at all but are rather invoked as Trick jobs from the default TrickHLA `sim_object`.

6.3.1 `TrickHLA::Packing`

Description. This class is a base class from which simulation developers may incorporate their own pack/unpack functionality into the TrickHLA model. The virtual methods `pack()` and `unpack()` have default implementations that will terminate the simulation, therefore you must subclass the base class in order to use it, overriding the two methods

The TrickHLA infrastructure invokes the `pack()` method prior to sending data to remote federates, and it invokes `unpack()` upon receiving data from remote federates.

Files. The class files are shown below.

file name	description
`TrickHLA/Packing.hh`	standard C++ header file
`TrickHLA/Packing.cpp`	method imlementation file

6.3.2 `TrickHLA::OwnershipHandler`

Description. This class handles HLA attribute ownerhip transfer. It does so by queueing up transfer requests. The queued requests are processed by a job scheduled in the default TrickHLA `sim_object`. Ownership may be divested using the `push_ownership()` method. It may be acquired by useing the `pull_ownership()` method. The actual work of pushing/pulling ownership is delegated to ownership-related methods in the associated `TrickHLA::Object` instance.

Files. The class files are shown below.

file name	description
`TrickHLA/OwnershipHandler.hh`	standard C++ header file
`TrickHLA/OwnershipHandler.cpp`	method implementation file

6.3.3 `TrickHLA::LagCompensation`

Description. This is a base class that allows simulation developers to insert logic that compensates for HLA-induced lags when attribute ownership is transfered from one federate to another. The two methods, `send_lag_compensation()` and `receive_lag_compensation()` are mutually exclusive. An object is configured to have send-side, receive-side, or no lag compensation. A simulation developer must subclass this base class, create an instance of it in the simulation `S_define` file, associate that instance with a particular object, and indicate whether the compensation is send-side or receiving-side compensation for the TrickHLA compensation logic to work. This compensation logic is invoked jobs in the default TrickHLA `sim_object`. These jobs defer to the `TrickHLA::Object` class which in turn invokes application-specific lag compensation methods (if any).

Files. The class files are shown below.

file name	description
`TrickHLA/LagCompensation.hh`	standard C++ header file
`TrickHLA/LagCompensation.cpp`	method implementation file

6.3.4 `TrickHLA::InteractionHandler`

Description. This class handles the sending and receiving of HLA insteractions.

Incoming interactions are dispatched from the federate ambassador to the manager class (see below) which in turn delegates responsibility to a corresponding instance of the `TrickHLA::Interaction` class which finally delegates to the associted application-specific interaction handler. Application-specific handlers must subclass this class.

This class implements several methods for sending interactions. These may be called from simulation code, or they may be scheduled as Trick jobs in the `S_define` file.

Files. The class files are shown below.

file name	description
`TrickHLA/InteractionHandler.hh`	standard C++ header file
`TrickHLA/InteractionHandler.cpp`	method implementation file

6.3.5 `IMSim::FreezeInteractionHandler`

Description. NOTE: This section needs to be removed. This is now part of the DSES and IMSim `ExecutionControl` classes.

This class handles the sending and receiving of Freeze Interactions in TimeStamp Order. It is available only when using version 2 of the multiphase initialization (by specifiying "`THLA.manager.sim_initialization = THLA_MULTIPHASE_INIT_V2;`" in the input file).

It is a subclass of `TrickHLAInteractionHandler`, specialized to send and receive an interaction that will suspend the federation execution. It requires that the `THLA.federate.check_freeze_time()` routine is present in the `S_define` file (see Figure 5.1).

When the user wishes to send a `Freeze Interaction`, this class will check if the supplied time is valid. If the supplied time is a valid future time, this class sends out an interaction for this future time.

If no time was supplied, the next available cooridinated `freeze` time is computed and sent out as the `interaction time`. Note: This time computation takes into account interaction times sent by late-joining federates.

All federates will go into `freeze` mode at the bottom of the frame specified by the interaction time.

The user is responsible for `un-freezing` the sim after the `freeze` mode work has completed.

This class implements methods for sending and receiving of these specialized interactions. These may be called from simulation code, or they may be scheduled as Trick jobs in the `S_define` file.

Files. The class files are shown below.

file name	description
`IMSim/FreezeInteractionHandler.hh`	standard C++ header file
`IMSim/FreezeInteractionHandler.cpp`	method implementation file

6.3.6 `TrickHLA::Federate`

Description. This class acts as a bridge between the manager and federate ambassador classes. It provides basic services for connecting a Trick simulation to an HLA federation. It also verifies the simulation data against the FOM and may terminate the simulation when discrepancies are found.

Files. The class files are shown below.

file name	description
`TrickHLA/Federate.hh`	standard C++ header file
`TrickHLA/Federate.cpp`	method implementation file

6.3.7 `TrickHLA::FedAmb`

Description. This class implements the HLA `FederateAmbassador` interface. The methods is implements are callbacks invoked by the HLA runtime infrastucture.

Files. The class files are shown below.

file name	description
`TrickHLA/FedAmb.hh`	standard C++ header file
`TrickHLA/FedAmb.cpp`	method implementation file

6.3.8 `TrickHLA::Manager`

Description. This class connects the TrickHLA model to Trick variables. It handles the sending and receiving of HLA objects, associating them with the Trick variables specified in the input file. And it also handles the processing of received interactions. Much of this logic is delegated to the relevant `TrickHLA::Object` and `TrickHLA::Interaction` objects.

Files. The class files are shown below.

file name	description
`TrickHLA/Manager.hh`	standard C++ header file
`TrickHLA/Manager.cpp`	method implementation file

6.3.9 `TrickHLA::ObjectDeleted`

Description. This class is a base class from which simulation developers may incorporate their own actions to take when an object is deleted from the federation into the TrickHLA model. The virtual `delete()` method is a do-nothing implemenation in this base class. The TrickHLA infrastructure invokes the `delete()` method when an object is deleted from the federation.

Files. The class files are shown below.

file name	description
`TrickHLA/ObjectDeleted.hh`	standard C++ header file

6.3.10 `TrickHLA::Conditional`

Description. This class is a base class from which simulation developers may incorporate the decisions when to send an attribute.
The virtual `should_send()` method is a do-nothing implemenation in this base class, returning true.
The TrickHLA infrastructure, on each data cycle, checks for if user wired a subclass to this attribute.

If found, the should_send() method is called; when it returns true, this attribute is sent. If not found, the attribute is always sent.

Files. The class files are shown below.

file name	description
`TrickHLA/Conditional.hh`	standard C++ header file
`TrickHLA/Conditional.cpp`	method implementation file

6.3.11 `TrickHLA::Timeline`

Description. This class is a base class from which simulation developers may provide access to the Scenario Timeline for their simulation.
The virtual `get_time()` method is abstract and the user must provide an implementation when they extend this class.
The TrickHLA infrastructure will call get_time() as needed to coordinate a federation freeze (pause) on the scenario timeline.

Files. The class files are shown below.

file name	description
`TrickHLA/Timeline.hh`	standard C++ header file
`TrickHLA/Timeline.cpp`	method implementation file

Chapter 7

Version Description

This section identifies the versions of TrickHLA described in the current release of the TrickHLA documentation.

7.1 Inventory

The inventory and detailed code derived documentation can be found in the following document:

TrickHLA Doxygen generated reference material.

This document is autogenerated using the Doxygen tool [15]. Doxygen generates a variety of document formates directly from an annotated code base. The TrickHLA code base has Doxygen compliant annotations. In addition, the Trick comments are also Doxygen compliant.

7.2 Change Status

This is the base implementation of the software suite. A change status is not provided.

7.3 Adaptation Data

This is the first release (base version) of the TrickHLA software suite. No adaptation data is applicable.

Bibliography

[1] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4), April 1981.

[2] Edwin Z. Crues. *TrickHLA Inspection, Verification, and Validation*. National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, June 2020.

[3] Edwin Z. Crues. *Trick High Level Architecture (TrickHLA)*. National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, June 2020.

[4] Edwin Z. Crues. *TrickHLA Product Requirements*. National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, June 2020.

[5] Edwin Z. Crues. *TrickHLA User Guide*. National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, June 2020.

[6] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10), October 1990.

[7] NASA. *NASA Software Engineering Requirements*. Technical Report NPR-7150.2C, National Aeronautics and Space Administration, NASA Headquarters, Washington, D.C., August 2019.

[8] National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch (ER7). Trick simulation environment: Documentation. Trick Wiki Site: https://nasa.github.io/trick/documentation/Documentation-Home, May 2020. Accessed 18 May 2020.

[9] National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch (ER7). Trick simulation environment: Installation guide. Trick Wiki Site: https://nasa.github.io/trick/documentation/install_guide/Install-Guide, May 2020 (accessed May 18, 2020). Accessed 18 May 2020.

[10] National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch (ER7). Trick simulation environment: Tutorial. Trick Wiki Site: https://nasa.github.io/trick/tutorial/Tutorial, May 2020 (accessed May 18, 2020). Accessed 18 May 2020.

[11] Simulation Interoperability Standards Organization (SIS0). *IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP).* Technical Report IEEE-1730-2010, The Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, January 2011.

[12] Simulation Interoperability Standards Organization/ Standards Activities Committee (SISO/SAC). *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Inferface Specification.* Technical Report IEEE-1516.1-2010, The Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, August 2010.

[13] Simulation Interoperability Standards Organization/ Standards Activities Committee (SISO/SAC). *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules.* Technical Report IEEE-1516-2010, The Institute of Electrical and Electronics Engineers, 2 Park Avenue, New York, NY 10016-5997, August 2010.

[14] Simulation Interoperability Standards Organization/ Standards Activities Committee (SISO/SAC). *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Template (OMT) Specification.* Technical Report IEEE-1516.2-2010, The Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, August 2010.

[15] Dimitri van Heesch. Doxygen. Doxygen Site: `https://www.doxygen.nl/index.html`, May 2020. Accessed 29 May 2020.