DIS to HLA Translator

Marc Moreau

Fairfax, August 27, 1997

Contents

List of Figures

Chapter 1

Architecture

1.1 Rationale

In order to test our implementation of the HLA Run Time Infrastructure (see [PLM97]) at a real scale, we need a federate which is able to generate attribute updates and interactions, and to simulate a large number of entities.

The DIS to HLA translator is intended to make ModSAF work with the RTI and to provide the federate we need.

1.2 Software architecture

The DIS to HLA translator is an independent piece of software, and can be used without modifying the presentation and transport layers of ModSAF.

This translator sends/receives PDUs like a DIS simulation, converts them into RTI service requests and sends/receives RTI requests like a federate. This first version is only able to handle Entity State PDUs and a few others interaction PDUs. Experimental and other unprocessed PDUs are just forwarded through a "back-door".

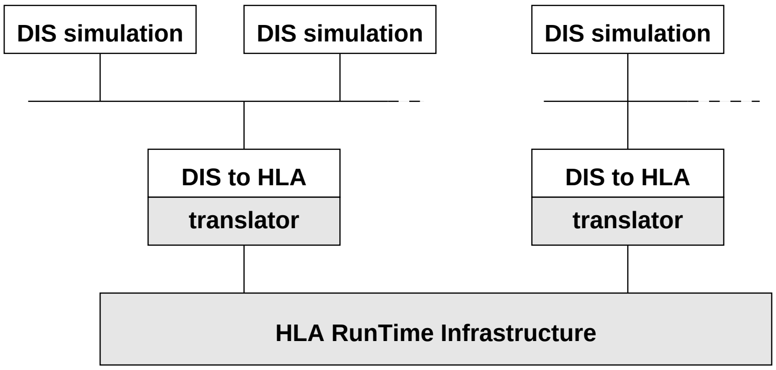

Figure 1.1: Architecture of the DIS to HLA translator

From a translator point-of-view, the simulations which send their PDUs to this translator are called "local"; the others simulations are "remote". The entities simulated by the local simulations are called "local entities". The entities simulated by remote simulations are referred to as remote entities or as “objects”, because they are visible through the HLA RTI.

Three different modules work together in the translator. The *RTI ambassador* and the *Federate ambassador* (called D2H ambassador) are the interface with the HLA RTI. The *DIS interface* receives information from the DIS simulations and turn them into HLA services. The back door transmits all the other information received from ModSAF.

The data flow between the components is described on figure 1.2.

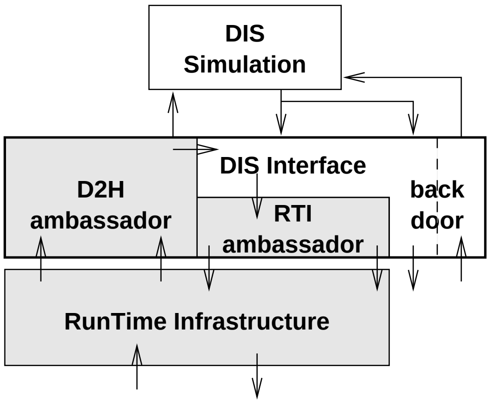

Figure 1.2: Exchange of data between the DIS simulation, the RTI and the different components of the translator

The software organization is not only based on an object oriented approach, but also on a distinction between FOM dependent and non dependent components.

The DIS data representation is well standardized, but the HLA object model is up to the programmer. An object model described in the `dis2hla.fed` file has been developed and the conversion between the DIS PDUs and this HLA object model has been hard coded. A limited number of methods are concerned and they have all been implemented in a file called `specific.cc`. An additional structure is also necessary to store different values provided by the HLA during the initialization phase. The corresponding declaration are in `specific.hh`.

This distinction makes the implementation a little awkward, but the modifications are easier to make. The translator can be adapted to a new FOM by changing only the `specific.cc` and `specific.hh` files and relinking the application.

The use of Data Distribution Management can be enabled during the compilation by defining `USE_DDM`. In that case, the translator will use the interface of the RTI that I have developed to declare its interests and send attribute updates and interactions.

If `USE_DDM` is not set, the translator will not use the DDM and can be executed, for example, with the RTI F.0 provided by the DMSO.

Chapter 2

Object model

2.1 Basic classes

The following classes represent the data manipulated by the translator.

2.1.1 Real Time

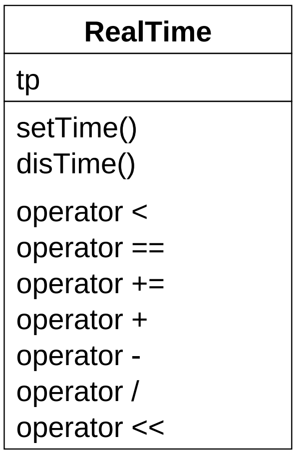

Figure 2.1: Class for the manipulation of real time (declared in `real-time.hh`)

Since we are working with real time simulation, the translator needs to access and manipulate time information. The `RealTime` class is mostly an encapsulation of the `gettimeofday()` system call (used in the `setTime()` method). The time is stored internally as a `timeval` structure called `tp`. This class also defines several operators which perform basic arithmetic operations, and a `disTime()` method which returns the time using the DIS format.

2.1.2 Classes

The object and interaction classes have two different representations. An object class is defined, with a `handle` and a set of attributes `ahs`, but the interaction

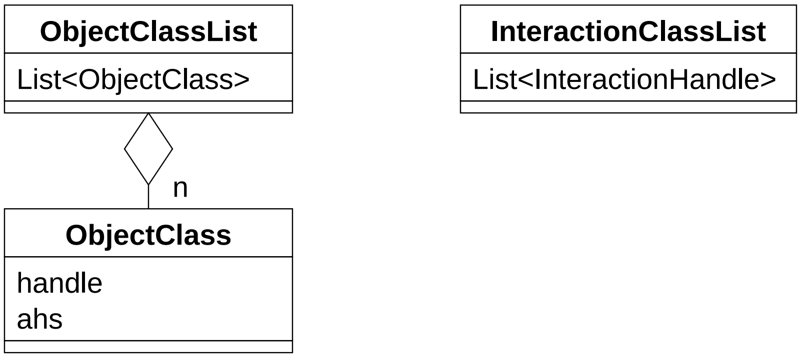

Figure 2.2: Representation of the object and interaction classes in the translator (declared in `class.hh`)

class can be represented with only a handle, because it is not allowed to manipulate a subset of all the parameters.

The two containers `ObjectClassList` and `InteractionClassList` are defined.

2.1.3 Parser for the initialization file

The `DMFile` class is similar to the `FED` class described in the documentation of our RTI. It is created with the name of a file containing classes that the translator is going to publish or subscribe (see 3.3.3).

The only method `readDMFile()` reads this initialization file, using the same parsing mechanism as `readFED`, and returns the list of object and interaction classes.

2.1.4 Entity

The translator must keep track of all the local entities and remote objects. The class `Entity` stores the entity state in the PDU format and implements methods to manipulate this state (like the necessary conversions between the DIS and HLA representations).

Some of these methods, like the `deadReckon()`, are independent of the Federation Object Model. Some others, like `buildAttributeList()` or `updateFields()` which makes conversion between the HLA data format and the DIS data format, must be changed whenever the FOM is changed. The latest are implemented in `specific.cc`.

The `Entity` class has a lot of methods which perform different tasks:

– `getAttributes()` get the attributes from its `AttributeHandleValuePairSet` parameter and set the corresponding value of the entity state: it performs the HLA to DIS conversion.

– `buildAttributeList()` returns an `AttributeHandleValuePairSet` with the complete object state.

– `updateFields` takes an ES-PDU as a parameter and update the `Entity state`. It returns an `AttributeHandleValuePairSet` containing only the modified values.

– `fullState()` returns true if we have received all the attribute values from the RTI (it means we are ready to send our first ES-PDU).

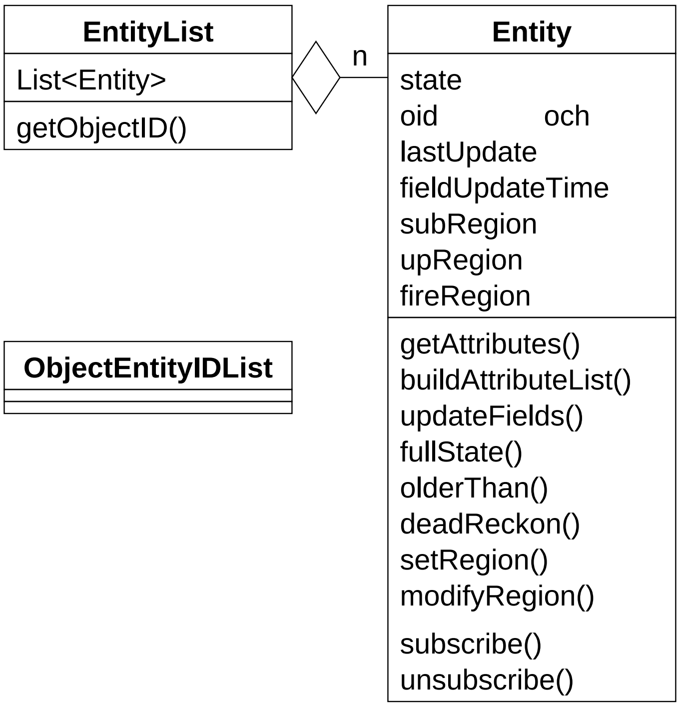

Figure 2.3: The `Entity` class (used to store and convert the entities state) and associated containers (declared in `entity.hh`)

– `olderThan()` returns true if the last update occured before the given date. It is used to know when to send the heartbeats or to remove the old entities.

– `setRegion()` and `modifyRegion()` create and modify the update and subscription regions associated to the entity (see 3.1).

– `subscribe()` and `unsubscribe()` declare the federate interests when `USE_DDM` is set: they use the `upRegion`, `subRegion` and `fireRegion` attributes.

Two containers are defined.

The `EntityList` container uses the DIS entity ID as a key to store and retrieve the entities. It has also an extra method `getObjectID()` which returns the HLA object ID of the entity.

The `ObjectEntityIDList` is only a hash-table which performs the reverse operation and returns the DIS ID corresponding to a given HLA ID.

2.1.5 Interaction

Interactions do not need to be stored. The `Interaction` class (see figure 2.4) essentially performs the conversion between the HLA and DIS format. When the interaction object is created using a DIS PDU, the constructor initializes the `ich`, `phvps` and `ft` attributes which represent the interaction for the HLA. When the interaction is created with a parameter set, the constructor initializes the `type`, `pdu` and `size` attributes.

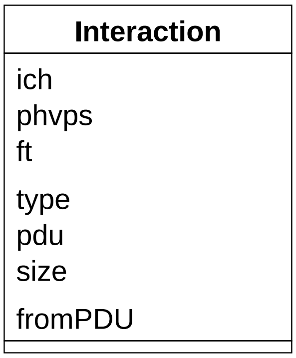

Figure 2.4: The `Interaction` class (declared in `interaction.hh`)

2.1.6 Coordinate

When the translator is compiled with the `USE_DDM` option, it needs to manipulate geographic coordinates and to convert geocentric coordinates (used by DIS) into topocentric coordinates which can be easily used to implement a geographic filtering mechanism.

The `Coordinate` class is an encapsulation of the `libcoordinates` library from ModSAF. The library initialization is done by the `init()` method, and the definition of the topocentric coordinates system by the `define()` method.

A new object is created with the three geocentric coordinates. The topocentric coordinates are returned by the `convert()` method which encapsulates the `coord_convert()` ModSAF function.

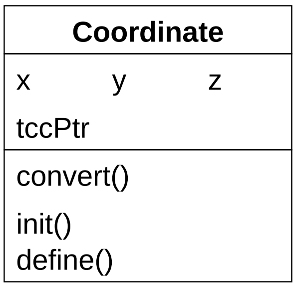

Figure 2.5: `Coordinate` class for the manipulation of geographic coordinates (declared in `coordinate.hh`)

2.1.7 Region

A region is identified with an `ID` and described with an `extentSet` and a `thresholdSet`.

The `setNewExtentSet()` method returns `true` if the variation of the extents exceeds the thresholds. In that case, the RTI must be informed with the `modifyRegion()` service call.

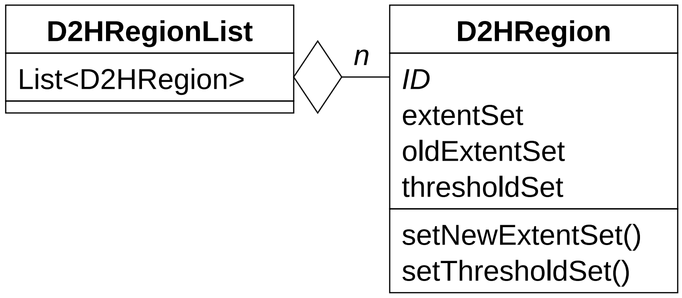

Figure 2.6: Representation of the region in the DIS to HLA translator (declared in `region.hh`)

2.2 Main components of the translator

Five main objects are used by the DIS to HLA translator.

2.2.1 The FOM

The Federation Object Model is represented by a `FOM` object. All the data are stored in a `FOMdata` structure declared in `specific.hh` (see 3.2). The initialization is done with `setFOM()`. Typically, the `FOM` object is used to store and provide the values of the parameters, attributes and class handles as defined by the RTI.

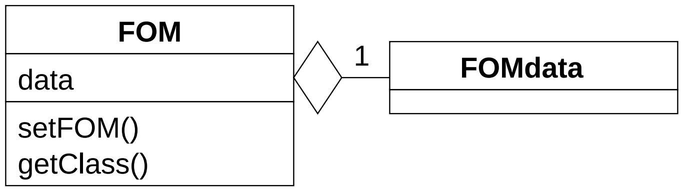

Figure 2.7: Representation of FOM specific data (declared in `fom.hh`)

2.2.2 The Connection Manager

The objects described in the RTI documentation are used by the translator: several `Connection` are defined and manipulated by a `ConnectionManager`.

2.2.3 The RTI ambassador

The RTI ambassador is the implementation of the RTI services, implemented according to the HLA Interface specification. It is provided by the RTI itself (by the RTI F.0 or by our own RTI).

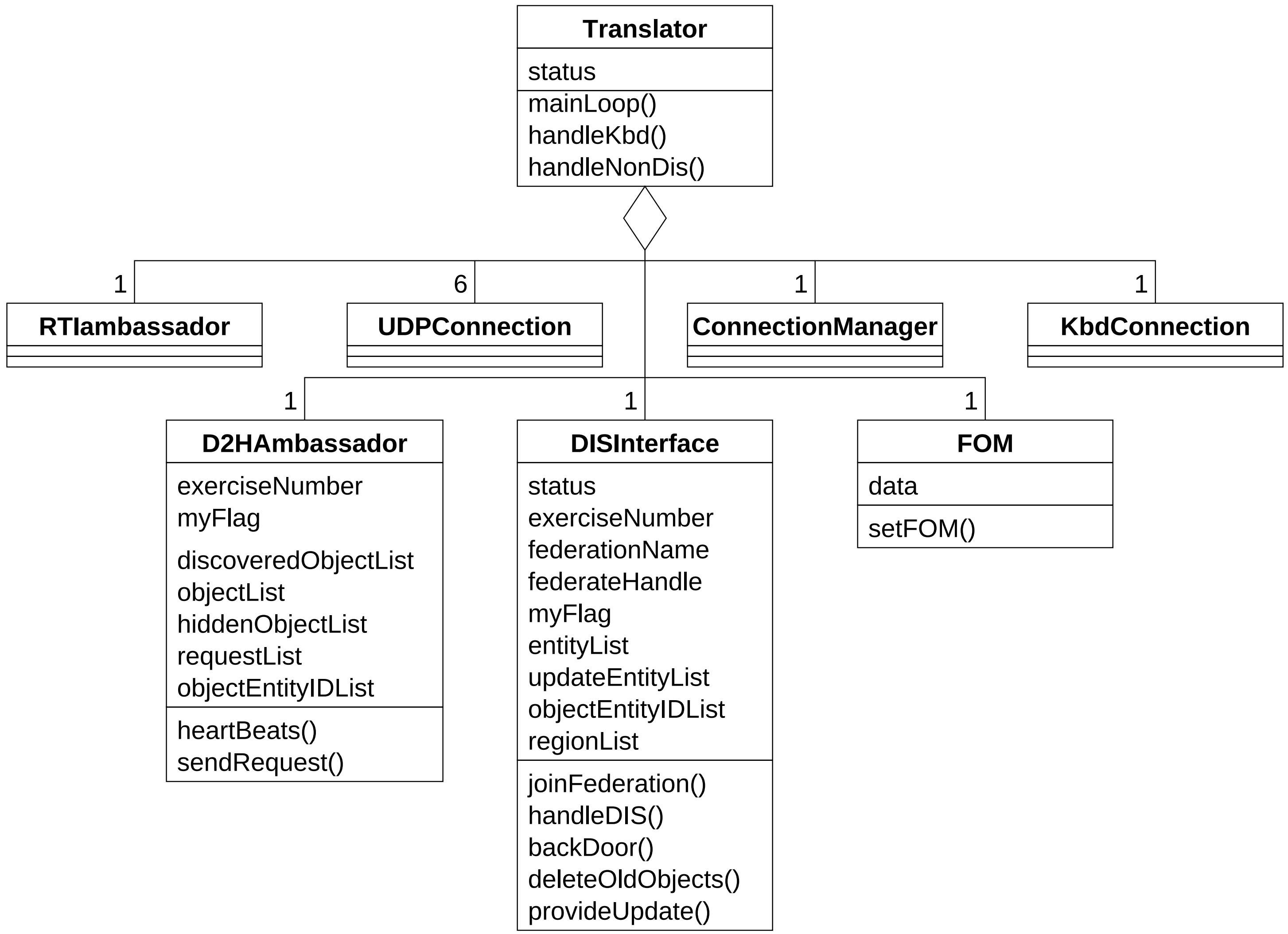

Figure 2.8: Aggregation of the different components in the translator

2.2.4 The federate ambassador

It receives and handles data and service requests from the RTI. Our federate ambassador is called the `D2HAmbassador`. This class is derivated from the default `RTI::FederateAmbassador` class and declared in `d2hAmbassador.hh`.

The Object Management mostly consists in receiving attributes and interactions. This is handled directly by the ambassador which converts them into DIS PDUs.

When `discoverObject()` is called, a new `Entity` is created and stored in the `discoveredObjectList`. In order to send an ES-PDU, the translator needs to have all the attribute values and will therefore ask for a full update: the new object is added to the `requestList`. The `sendRequest()` method is used whenever `requested` is true to send the update requests to the RTI.

As soon as all the attributes have been received with `reflectAttributeValues()`, the `Entity` is transfered to the `objectList` and the translator starts sending ES-PDUs for this entity. The `hearBeats()` method can be used to send a dead reckoned ES-PDU if the last update was received more than `heartbeatsDelay` ago.

The `hiddenObjectList` is used to store objects which came out-of-scope but have not been removed from the federation: they can be transfered back in the `objectList` as soon as they come back in scope.

The Declaration and Data Distribution Management service calls are transfered to the DIS Interface: they concern the control of the attribute updates and interactions, and must be taken in account by the module which sends the information.

The Federation, Ownership and Time Management services have not been implemented.

2.2.5 The DIS interface

This object, defined in `d2hdis.hh`, turns the DIS PDUs into RTI service requests.

The initialization is done by the `joinFederation()` method, which attempts to create the federation if it does not exist already, joins the federation and registers the interests of the simulator.

The received PDUs are handled by the `handleDIS()` callback. They are turned into attribute updates and interactions, or ignored and forwarded to the LAN if they are experimental PDUs. The `entityList` is used to store all the entity states.

The `backDoor()` callback receives non processed PDUs from the WAN and forwards them to the LAN.

Several methods are mere implementations of RTI services: as explained before, the control of the interactions and attribute updates generation can only be performed within the DIS Interface. The corresponding classes are stored in the `updatedObjectClass` and `updatedInteractionClass` lists. `handleDIS()` will not send an attribute update or an interaction unless the corresponding class is in one of these lists.

In a similar way, the DIS Interface receives the `provideAttributeValueUpdate()` request from the RTI through the D2H ambassador and adds the entity to the `updateEntityList` list. The `provideUpdate()` method can then be used to send the complete entity state for all the enqueued entities.

The Declaration Management is performed globally by the private methods `publish()`, `subscribe()`, `unpublish()` and `unsubscribe()` which declare the interests of the local simulations according to the provided initialization files (see 3.3.3). These methods are called at the beginning and at the end of the execution.

DIS simulations do not explicitly destroy entities, they just stop sending updates. The translator must detect this and issue a `deleteObject()` RTI call; this task is performed by the DIS interface with the `deleteOldEntities()` method.

All the simulated entities are destroyed at the end by `deleteAllEntities()`.

2.2.6 The translator

The `Translator` is the heart of the DIS to HLA translator. All the previously mentioned objects are integrated into it (see figure 2.8).

During the creation, the different components (like the `ConnectionManager`, the `RTIAmbassador`, `DISInterface`...) are instantiated. Then the translator joins the federation.

Two callbacks `handleKbd()` and `handleNonDIS()` are defined to execute the user's commands and forward the PO and Stealth PDUs sent by ModSAF.

The main loop uses the Connection Manager's `netScan()` method to receive and handle the PDUs. `netScan()` will call `handleDIS()`, `backDoor()`, `handleKbd()` or `handleNonDIS()`, depending on the file descriptor the message was received from.

The main loop also uses the RTI ambassador's `tick()` method to allow the RTI to perform several internal tasks, the `heartBeats()` method to send heartbeats, the DIS Interface's `deleteOldEntites()` and `provideUpdate()` methods as often as necessary.

Chapter 3

Implementation

3.1 The DDM

The Data Distribution Management is used by the DIS to HLA translator when `USE_DDM` is set. It has been designed to work with our implementation of the RTI and has been tested successfully.

3.1.1 Defining the regions

Three regions are associated to each entity:

– `upRegion` is used to send the attribute updates, and also to subscribe and publish the collision interactions.

– `subRegion` is used as a subscription region for all the others interactions and for the object classes specified in the initialization file.

– `fireRegion` is used to send the fire and detonation interactions.

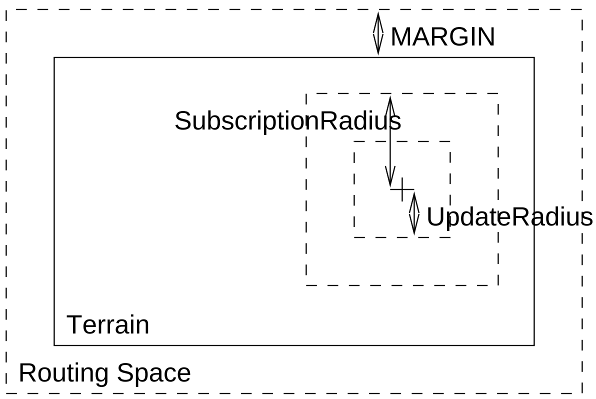

Figure 3.1: Regions and Routing Space for the translator

The regions are defined by `setRegions()` as squares centered on the entity position in a two dimensional geographic routing space. The dimensions of the square are returned respectively by the `getUpdateRadius()`, `getSubscriptionRadius()` and `getFireRadius()` methods.

The current values are 80 meters for the update region region and 2000 meters for the subscription and fire regions. The update radius has been chosen small enough to allowed a fine region-based filtering, but high enough to avoid frequent and costly modifications. The other radius value seems to be reasonable, at least for the first tests. A more realistic behavior can be obtained by changing the radius value according to the entity type.

Actually, if we define the efficiency of the filter

$$\eta = \frac{xy}{(x+\delta x)(y+\delta y)}$$

where x and y are the dimensions of the subscription region and δx and δy the dimensions of the update region (see [HLA97b]), we have $\eta = 0.96$.

Assuming that the speed of the entities is at most $60km/h = 16.6m/s$ (this is a reasonable assumption for armored vehicles but an obvious underestimation for aircrafts and rockets), and that the thresholds are 10% of the dimensions of the region, the translator must update the region at most once per second.

The extents of the routing space are the width and height used to define the TCC coordinates system according to ModSAF database, increased by `2 * MARGIN`. This safety margin is intended to avoid an error if an entity goes out of the map.

The value of `MARGIN` has been chosen equal to the biggest radius.

3.1.2 Using the Declaration Management with regions

`handleDIS()` will make the entity create the three regions and will associate all the entity attributes for update with the update region. It will let the entity subscribe to all the classes defined in `subscribedObjects` and `subscribedInteractions` with its subscription region.

The classes will be unsubscribed and the regions deleted when the object will be removed from the federation.

3.2 The dis2hla FOM

The actual FOM is defined in `dis2hla.fed`.

All the entities are defined as HLA subclasses of `BaseEntity`. The different object classes correspond to the DIS entity kinds and the attributes correspond to the different fields of the ES-PDU.

Three interactions are actually available. The Collision, Detonation and Fire interactions are directly associated to the corresponding PDUs, and the parameters correspond to the fields of the PDUs.

The conversion is done simply by copying the values from the PDU fields to the attributes.

As the FOM is described by names and the RTI manipulates handles, it is necessary to know the mapping between both representations: the `FOMdata` structure (see figure 3.2) is initialized during the creation of the translator using the RTI support services, and contains all the values we need.

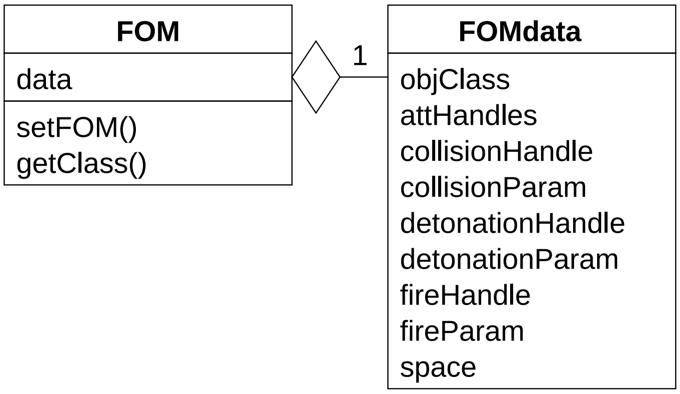

Figure 3.2: The `FOMdata` structure which represents data specific to the dis2hla FOM

Another developer may find it convenient to have different or additional pieces of information in this structure.

3.3 Initialization Files

Four initialization files are used by the translator. They must be in the `TRANS_CONFIG` directory or in the current directory if this variable is not set.

3.3.1 ModSAF initialization files

In order to use ModSAF `libcoordinates` library, `coord.rdr` must be available.

3.3.2 Network and connectivity

The `translator.rid` file describes the multicast groups, port numbers and other parameters used by the translator. The following is an example of such a file:

```
# the federation name
FEDE_NAME dis2hla

# the exercise number
EXERCISE 1

# the terrain
TERRAIN knox

# parameters for the LAN
LAN_ADDR 224.5.5.5
LAN_TTL 1
DIS_LAN_PORT 5000
STEALTH_LAN_PORT 5001
```

```
PO_LAN_PORT 5002

# parameters for the WAN
WAN_ADDR 224.7.7.7
WAN_TTL 1
DIS_WAN_PORT 7000
STEALTH_WAN_PORT 7001
PO_WAN_PORT 7002
```

3.3.3 Interest declaration

Two files called `<federationName>.pub` and `<federationName>.sub` contain the list of published and subscribed classes. Each class is described with the published/subscribed attributes, using the same LISP-like format as the FED file (see the RTI documentation).

3.4 Other implementation details

The translator is supposed to forward the non processed and other Stealth PDUs. In order to work properly, it must be able to make the difference between the PDUs it has sent, and the PDUs sent by the DIS simulations.

A filtering mechanism based on the sending host is not appropriate, since the translator may run on the same host as a ModSAF simulation. The DIS PDUs do not contain any serial number or special tag which allows an easy recognition.

For this reason, the translator uses the 16 unused bits of the PDU header to make the distinction by copying in that field the `myFlag` value (`myFlag` is defined as the host ID).

Chapter 4

Future work

4.1 New interactions

Only four kinds of PDU can be processed by the actual version of translator. An important task will therefore be to extend the capabilities of the software, and to work on converting the other PDUs into interactions or service requests.

4.2 Realistic values

The translator actually uses an approximative and schematic approach to implement the DDM. The values used to define the regions could be refined, and adapted according to the type of the entity: an aircraft has obviously a greater detection range than a tank, and the fire radius is not the same for all the weapons.

Bibliography

[BH97] Wesley K. Braudaway and Susan M. Harkrider.
"Implementation of the High Level Architecture into DIS-based Legacy Simulations".
In *1997 Spring Simulation Interoperability Workshop*, paper number 97S-SIW-089. Simulation Interoperability Standards Organization, March 1997.

[CWP+96] Andy Cox, Douglas D. Wood, Mikel D. Petty, , and Kenneth A. Juge.
"Integrating DIS and SIMNET into HLA with a gateway".
In *Proceedings of the 15th DIS Workshop*, volume 1, pages 517–525, paper number 96-15-082. Institute for Simulation and Training, University of Florida, September 1996.

[HLA97a] "High Level Architecture Interface Specification Version 1.1".
DMSO, February 1997.
http://www.dmso.mil/projects/hla/tech/ifspec/ifsp11.pdf.

[HLA97b] "HLA Data Distribution Management: Design Document Version 0.5".
DMSO, February 1997.
http://www.dmso.mil/projects/hla/tech/ifspec/ddm_05.pdf.

[PLM97] J. Mark Pullen, Vincent P. Laviano, and Marc O. Moreau.
"Creating a light-weight RTI using selectively reliable transmission as an evolution of dual-mode multicast".
In *1997 Fall Simulation Interoperability Workshop*, paper number 97F-SIW-149. Simulation Interoperability Standards Organization, September 1997.

[Smi96] Joshua E. Smith.
"Adapting MODSAF applications to comply with the HLA".
In *Proceedings of the 15th DIS Workshop*, volume 1, pages 339–348, paper number 96-15-048. Institute for Simulation and Training, University of Florida, September 1996.

Index