

Mission Simulation Facility Documentation

Generating HLA based communication classes from a UML description

Lorenzo Flückiger

January 2002

Contents

1	Introduction	2
2	Mappings	3
2.1	Overview	3
2.2	Object Classes	4
2.3	Object Instances	6
2.4	Interaction Classes	7
2.5	Messages	8
2.6	Classes Details	9
3	The tool: uml2hla	10
3.1	Concept	11
3.2	Installation	12
3.3	Usage	12
4	Problems and Limitations	14

Document updated on May 15, 2003

1 Introduction

This document describes a method using the Unified Modeling Language (UML) notation to represent communication entities in an object-oriented way, and how to map these entities to an HLA implementation¹. Communication entities are the concepts representing objects or messages shared or exchanged by the participants in a simulation (the federates). An example of one entity could be a *Vehicle*. The federates could change the properties of this vehicle (e.g. `setWeight`) and send messages to it (e.g. `startEngine`). These conceptual entities are represented by UML classes. Then implementation classes based on the HLA communication layer are extracted from these source classes.

In addition to the proposed method and mappings, a tool is provided to automate the process. This tool comes in the form of an Addin for Rational Rose UML modeling tool.

Motivation

The classical approach to design a simulation based on the High Level Architecture (HLA) starts with the description of the simulation using the Object Model Template (OMT) file format. Then each federate has to represent this information locally in its Local Run-Time-Infrastructure Component (LRC) and manage the HLA objects and HLA interactions existing in the federation. This process flow makes the code reflect directly the HLA philosophy rather than the designer view of what a simulation should be. In addition, writing the LRC could be very time-consuming and laborious without the help of some additional layer between the Run-Time-Infrastructure (RTI) layer and the application code.

The proposed approach starts with a UML description of the communication entities (as classes) needed in a simulation, and derives from them a set of implementation classes based on HLA. Section 2 on the following page presents the mappings between these conceptual classes and their implementation counterpart. Section 3 on page 10 presents the tool developed to automate this process. These mapping and the tool rely on the Federate ToolKit, which provides a layer on top of HLA.

The advantages of the proposed method are:

1. Provides a rigorous process to map communication entities to the HLA scheme.
2. Allows changes to the communication implementation independently from the communication concepts: only the code generator script will have to be adapted to the new requirements of the implementation, and all the needed classes will follow.
3. Creates independence between the communication concepts needed and their underlying implementation: if a communication layer other than HLA should be used, only the code generation script needs to be rewritten².
4. Drastically speeds up the process of generating code to implement the communication entities: a simple click will generate code for all the needed objects.
5. Removes the risk of human error in the code and facilitates the debugging process since all the generated classes are guaranteed to follow the same scheme.

¹The reader of this document should be familiar with OO, UML, HLA and the FTK

²This is assuming that a communication layer is available at a pretty high level

The FTK

The Federate ToolKit (FTK) has been designed to ease the creation of the LRC of each federate by providing generic objects and interactions classes with all the associated services needed to participate in a federation. The FTK relies mainly on four classes which capture the corresponding HLA concepts:

HLAObjectClass represents the HLA concept of Object Class, its attributes and inheritance relationships.

HLAInteractionClass represents the HLA concept of Interaction Class, its parameters and inheritance relationships.

HLAObjectInstance represents an instance of an HLA object (which could have been created or discovered by a federate).

HLAInteractionMessage represents a message sent or received in a HLA federation.

These base classes work closely with two important classes:

RFI, the Run-Time-Infrastructure – Federate Interface, which manages all the object/interaction classes, objects and instances

FTKFederateAmbassador provides all the services that can be provided in a generic way by a Federate Ambassador

2 Mappings

The simulation designer describes the different communication entities needed in the simulation. These concepts are captured by UML classes. Each class is composed of attributes and operations. The inheritance relationship is allowed between classes³. From this UML description, a new set of classes will be derived to implement these communication needs with the HLA mechanism.

2.1 Overview

To explain the proposed mappings between the concept description and the HLA implementation, we will follow a simple example composed of the two classes shown in Figure 1 on the following page. This example shows a simulation composed of *Vessels* and *Sailboats*, the latter inheriting all the properties of the former. The *Vessel* class has two attributes: the *speed* and the *heading* of the vessel. The sailing boat has one additional attribute: its *heeling*. The implementation classes derived from this description will provide the access methods to these attributes (*speedGet*, *speedSet*, etc). The *Vessel* class has two operations: *dropAnchor*, which takes two parameters and *weighAnchors* with no parameter. The *Sailboat* class provides an additional method: *setSail*, which takes only one parameter.

The general scheme is that each source class will be mapped to one HLA Object Class to handle the attributes of the source class, and that each operation of the source class will be mapped to one HLA Interaction Class.

For each source class, the set of derived classes is composed as follows:

³Multiple inheritance is not allowed since there is no mechanism to support this concept in HLA

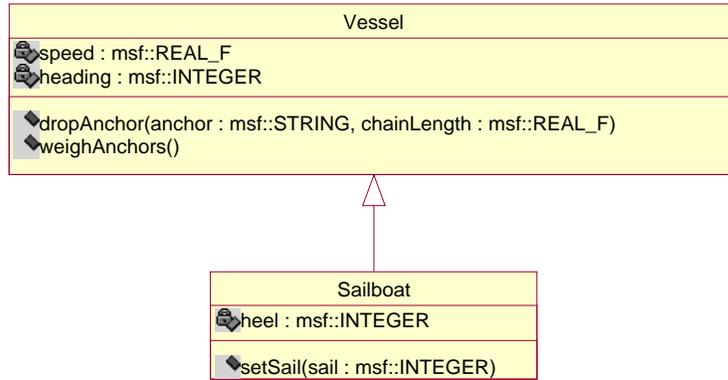


Figure 1: Source classes describing the communications objects needed in a simulation.

- one class to represent the HLA object class concept (based on the `HLAObjectClass` of FTK)
- one class to represent the the actual instance of these objects (based on the `HLAObjectInstance` of FTK)
- one class to represent the HLA interaction class concept⁴
- one class to represent the HLA interaction class concept for each operation of the source class (based on the `HLAInteractionClass` of FTK)
- one class to represent the actual message of the interaction class sent to or received by a federate (based on the `HLAInteractionMessage` of FTK)

In the case of the example, the two initial concept classes will be mapped to 12 implementation classes as shown in Figure 2 on the next page. The relationship between the classes will be discussed individually in the following sections.

2.2 Object Classes

The concept of HLA Object Classes is captured by classes inheriting from the `HLAObjectClass` class of the FTK package. Each conceptual source class is mapped to one and only one specialized `HLAObjectClass`, which maintains information about each class (name, handle), its inheritance relationship (name of parent) and each of its attributes (name, handle, publish, subscribe). Figure 3 on page 6 shows the details of the object classes derived from the example source classes.

Each specialized object class inherits directly from the `HLAObjectClass`. The inheritance relationship from the source classes (which is also present in the OMT description of a federation) is expressed with the `PARENT` attribute of the class. Source classes which have no superclass, will inherit from the 'ObjectRoot' HLA class: this is expressed with the initialization of the `PARENT` attribute to `HLAObjectClass::BASE_NAME`.

⁴This class will not be implemented with a message and is simply here to regroup all the operations of the source class under one unique HLA interaction class.

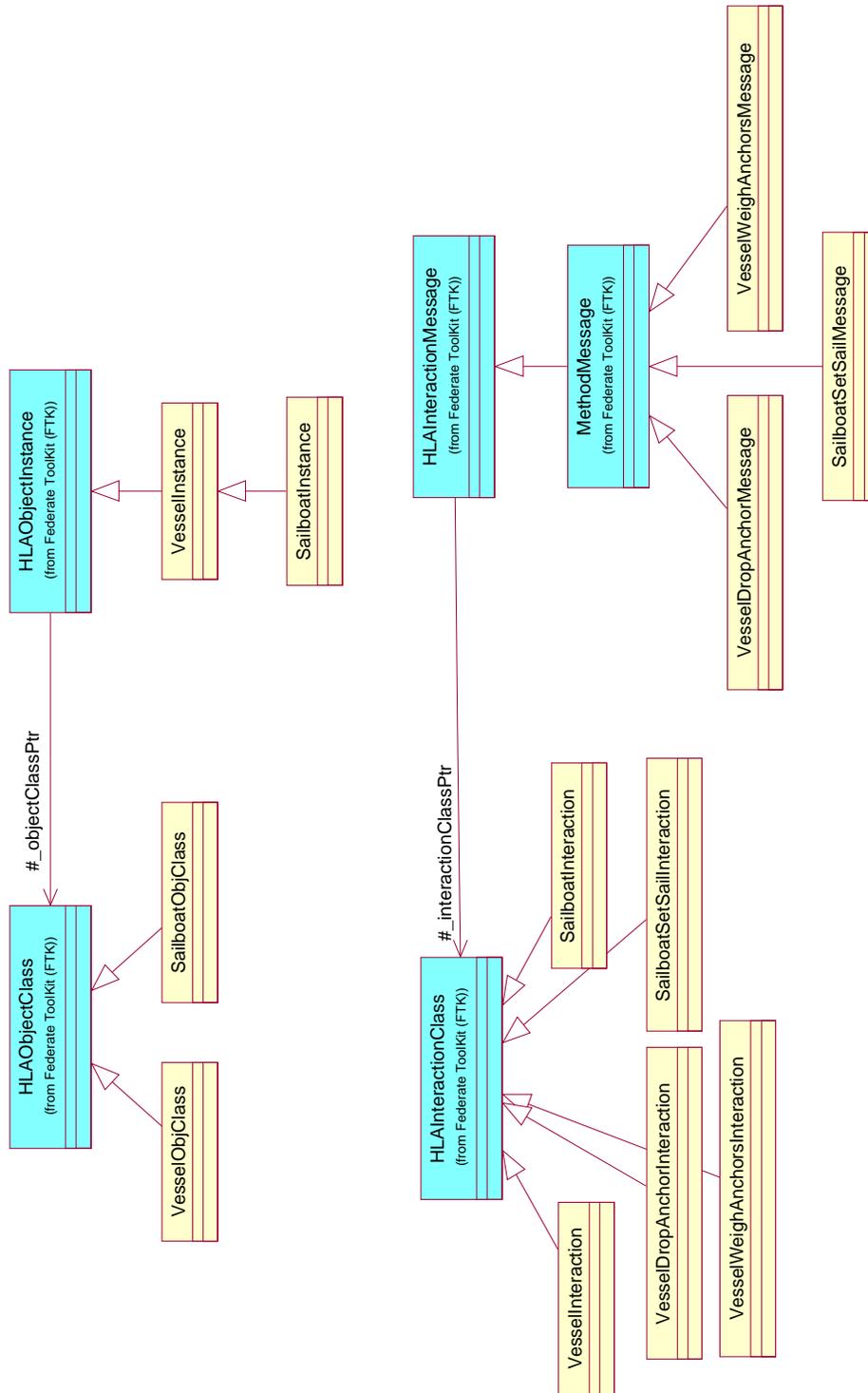


Figure 2: The set of classes used to implement the desired concept with the HLA mechanism (the blue classes are not generated but come from the FTK package).

For each attribute of the source class – which corresponds to an HLA attribute in the simulation – one attribute is created in the implementation class. The value of the generated attribute is the name of the attribute of the source class and its name is its value in all capital letters. This scheme maintains the information from the OMT in the LRC. The federate developer can then use in his program a reference to any attribute using its name. For example:

```
ftk::HANDLE h = objclass->getAttributeHandle(VesselObjClass::SPEED);
```

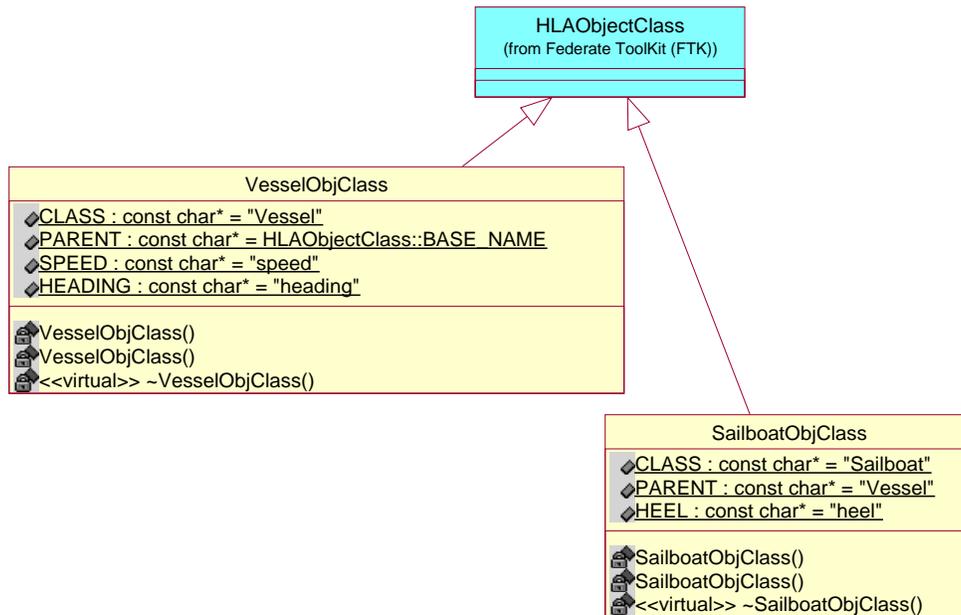


Figure 3: Object Classes.

2.3 Object Instances

The local representations of HLA objects inside a federate (LRC) are instances of a class inheriting from `HLAObjectInstance`. The inheritance between the HLA object classes is directly reflected by the same inheritance in the `HLAObjectInstance` classes as shown by Figure 4 on the next page. This allows subclasses to inherit all the operations of their super-classes. For example, the method `speedGet` of the class `VesselInstance` will be available for any object of the class `SailboatInstance`. Specialized object instance classes which are derived from source classes without superclass will inherit directly from the base `HLAObjectInstance` class which provides all the necessary services to produce, discover, update and reflect these HLA objects.

Each object instance class has private attributes to store data about the particular instance in the simulation. The type of the attribute reflects the type defined in the source class. Methods to access these attribute (`Set` and `Get`) are also present.

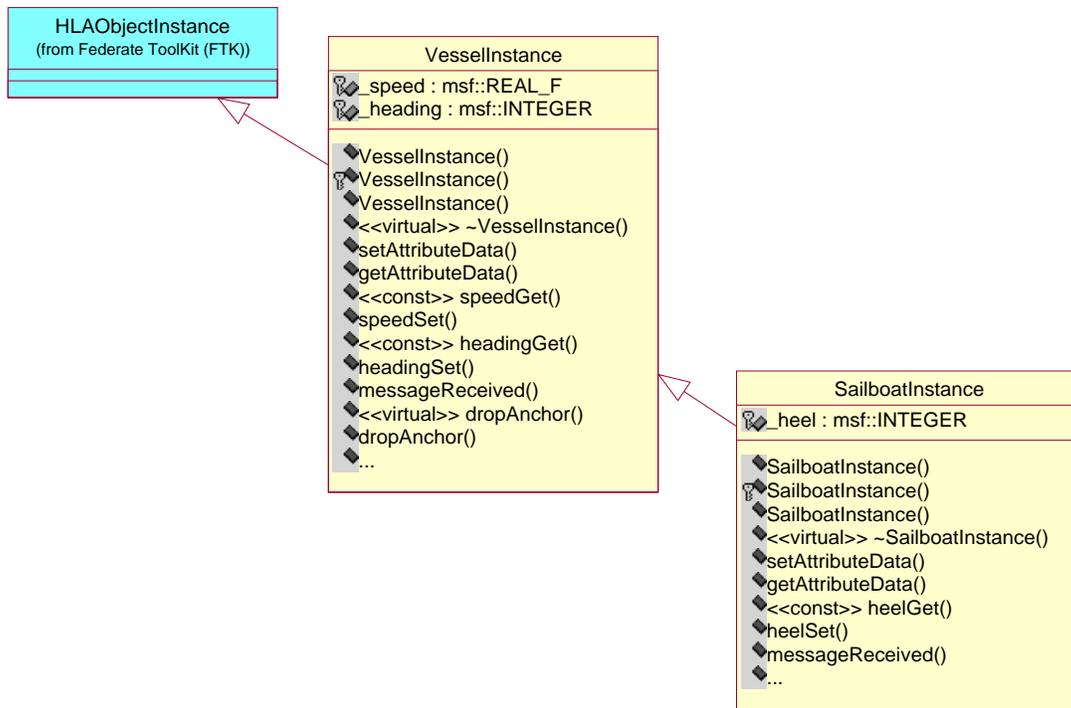


Figure 4: Object Instances.

2.4 Interaction Classes

The concept of HLA Interaction is captured by classes inheriting from the `HLAInteractionClass` class of the FTK package. Each conceptual source class is mapped to one and only one specialized `HLAInteractionClass` which maintains information about each interaction class (name, handle, publish/subscribe), its inheritance relationship (name of parent) and each of its parameters (name, handle). Figure 5 on the following page shows the details of the interaction classes derived from the example source classes.

Each specialized interaction class inherits directly from the `HLAInteractionClass`. The inheritance relationship from the source classes (which is also present in the OMT description of a federation) is expressed with the `PARENT` attribute of the class. For each source class, an interaction class with no parameter is created. This empty interaction has for its parent class⁵ the `MethodInteraction` class. This latter class from the FTK package provides one additional parameter to all its subclasses: the handle of the destination object⁶. This enables the regrouping of all the interactions representing operations of the source class under the same interaction class. No message of this grouping class will be sent or received in the simulation.

For each parameter of an operation in the source class – which corresponds to an HLA parameter in

⁵This `PARENT` relationship is inheritance in the HLA scheme, but not in the implementation classes.

⁶HLA does not support the concept of operations acting on the object they belongs to. This is mimicked with a HLA interaction with its first parameter describing the target object of this message.

the simulation – one attribute is created in the implementation class. The value of the generated attribute is the name of the parameter of the source class and its name is its value in all capital letters. This scheme maintains the information from the OMT in the LRC. The federate developer can then use in his program a reference to any parameter using its name. For example:

```
ftk::HANDLE h = interaction->getParameterHandle(VesselInteraction::ANCHOR);
```

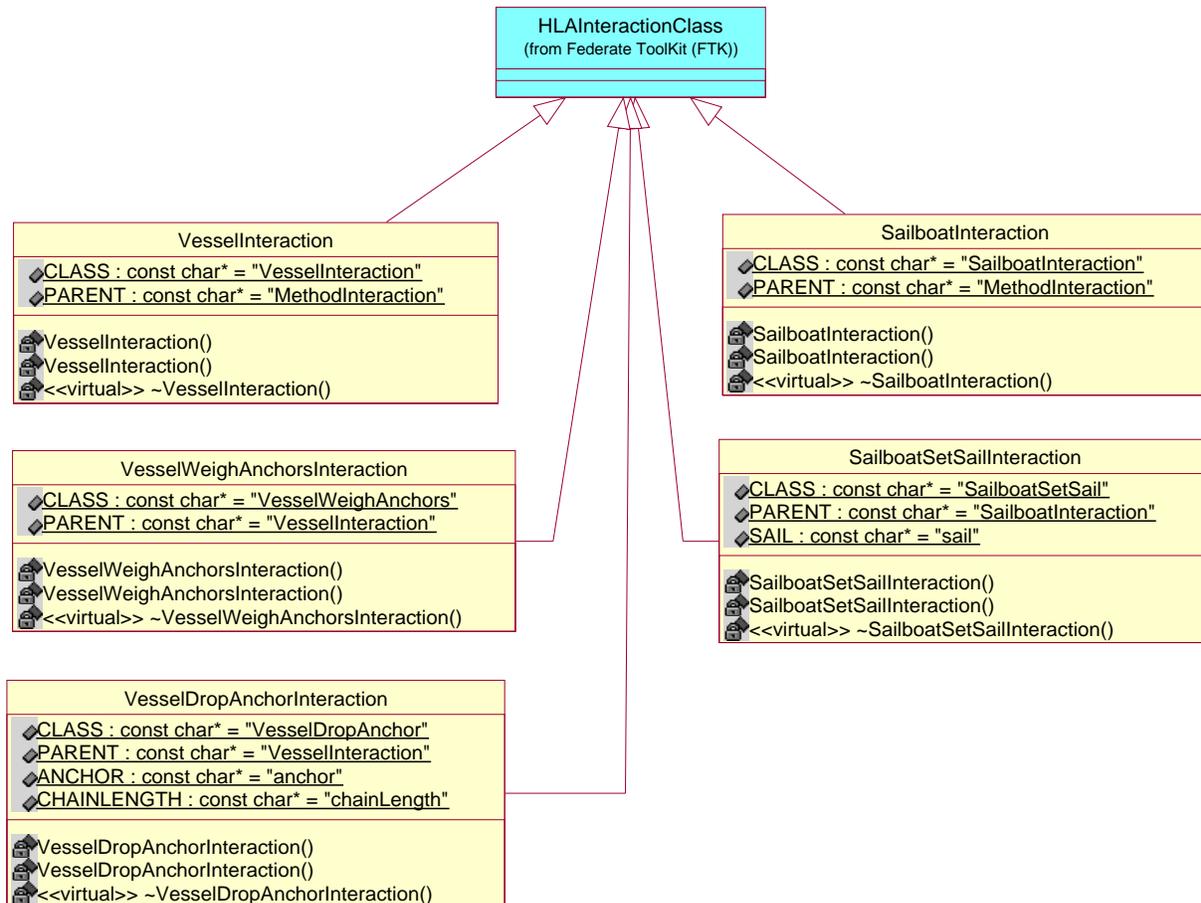


Figure 5: Interactions Classes.

2.5 Messages

In a simulation, each federate can send and receive interactions. These entities are represented in the LRC by objects of class `HLAInteractionMessage` (in fact a specialized version of it). The inheritance in the HLA interaction classes is not reflected at all in the messages hierarchy: unlike in HLA, an operation of a source class does not inherit the parameters of the operations in its parent class. The hierarchy of the message classes is maintained by the `HLAInteractionClass` classes, but all the actual messages inherit from the same message class: `MethodMessage`. This latter class provides facilities related to the destination object of any message.

Each specialized `HLAInteractionMessage` class has private attributes to store the parameters a message will carry in the simulation. The type of the attribute reflects the parameter type of the operation in the source class. Methods to access these attributes (`Set` and `Get`) are also present.

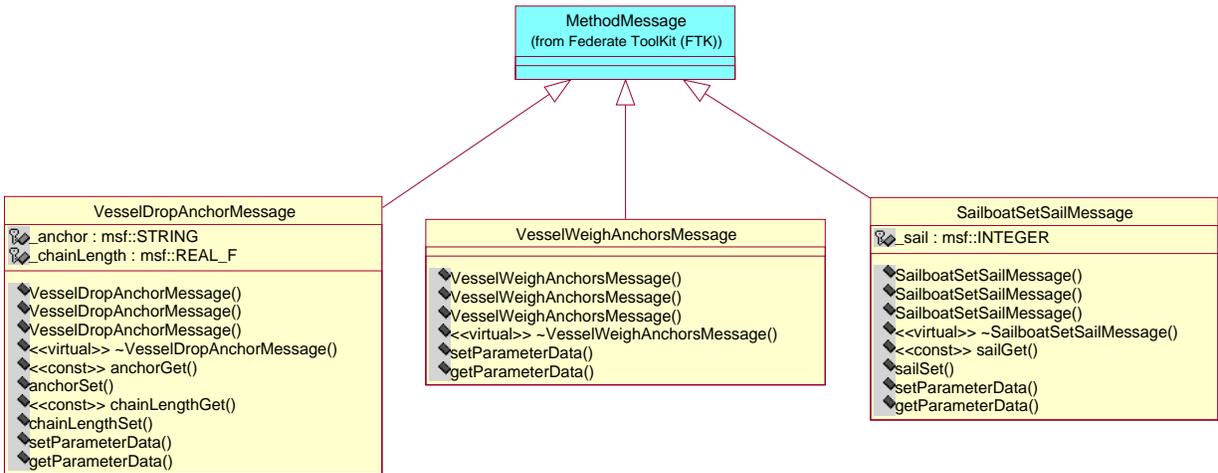


Figure 6: Messages.

2.6 Classes Details

2.6.1 “Class” classes

The two classes derived from `HLAObjectClass` and `HLAInteractionClass` are very simple and share the exact same scheme, as shown on Figure 11 on page 15 and Figure 12 on page 16. These classes only store the properties of a class of entities, the mechanism to manage and access this data is embedded in their parent class in FTK.

As this type of classes represents a *Class* concept, only one instance of the class will exist in each federate. For this reason, the constructor and copy constructor of these classes are private: the only way to create an instance of such a class is using a *Singleton* pattern implemented through the FTK class `SingletonHolder`. The classes derived from `HLAObjectClass` and `HLAInteractionClass` have a friend which is an instantiated class of the `SingletonHolder` template (instantiated by the class itself). The user of these classes will get a reference to them by using the `Instance` method of the `SingletonHolder`. The `Instance` call will return a reference to the desired object class if it exists, or create first such an instance if it does not yet exist.

The attributes of these Class classes are all public and only store data relative to attribute and parameter names: it allows the federate programmer to reference these attributes/parameters without having to know their names in the federation or their handles (these latter being only defined at run-time and so not accessible before).

2.6.2 Instances

Instance classes, like `SailboatInstance` shown on Figure 11 on page 15 store attribute data of object instances existing in the simulation. In the above example, the federate developer can retrieve the parameter

heel thanks to the generated method `heelGet`. When the attribute is set with `wheelSet`, then, in addition to having the given value assigned to the private data member `_heel`, the attribute is queued in the RFI to be updated (which will cause an update in the RTI at the next tick).

All instance classes have three constructors (see FTK documentation for a detailed description). The default constructor is used by a federate which wants to create a new instance and declare it to the RTI: the instance is created directly with the correct default arguments. The constructor with the form `SailboatInstance(ftk::HANDLE h, HLAObjectClass* cl)` is used when the Federate Ambassador of a federate discovers a new object instance and wants to add it in its local LRC: the only things the RTI provides on discovery is the new instance handle and the handle on the class of object (which is used to retrieve the corresponding `HLAObjectClass`). Finally, the third constructor of the form `SailboatInstance(const char* className)` is called by a subclass constructor and so cannot be used directly by the federate developer.

Two key methods are also automatically generated for each Instance class: `setAttributeData` and `getAttributeData`. These methods cannot be implemented at the `HLAObjectInstance` level since they require knowledge about the type of the data used for each attribute of the Instance. It should be noted that these methods are not normally used by the federate developer because they are called by either the `FTKFederateAmbassador` or the RFI. Nevertheless they provide a very important service for the FTK:

`setAttributeData(ftk::HANDLE h, const char* data, unsigned int size)` is used by the generic Federate Ambassador, which receives a callback to reflect some attribute changes in the simulation: the Federate Ambassador provides the handle on the attribute to update and a buffer of characters containing the associated data (as well as its size). This method un-serializes the XDR encoded data and puts it in the correct data member of the instance.

`getAttributeData(ftk::HANDLE h, char*& data, unsigned int& size)` is used by the RFI when it needs to update some attributes of an instance (to notify the RTI of changes in the LRC of the federate): in this case, the RFI provides the handle of the attribute it needs to update and is returned a buffer with the corresponding data serialized as an XDR string.

2.6.3 Messages

Message classes like `VesselDropAnchorMessage` shown on Figure 12 on page 16, behave similarly to Instances as described in the previous section. For each of these classes, three constructors are generated for the same reasons mentioned above.

The `Get` methods return data associated with each parameter: they are used to read the parameters of a received interaction message. The `Set` methods assign the given values to the right data members but do not push them directly to the RTI: the `send` method will send an interaction message to the RTI with all the parameters of this message.

The functionalities of `getParameterData` and `setParameterData` are the same as the `getAttributeData` and `setAttributeData` defined above.

3 The tool: `uml2hla`

The tool `uml2hla` was created to fully automate the implementation of communication entities from their UML description. The goal of this tool is to generate directly compilable code from a UML description.

The current realization of this tool relies on Rose, a commercial modeling software from Rational, but the concepts could be easily applied to another environment⁷.

3.1 Concept

The work-flow of the uml2hla tool is shown on Figure 7 and summarized below:

1. Communication entities are designed with UML diagrams in Rose.
2. Using a script, implementation classes are generated according to the proposed mapping. The code for each method of the implementation classes is also generated by the script.
3. By forward engineering these Rose classes, C++ code is written to files which can be compiled without modification or addition.

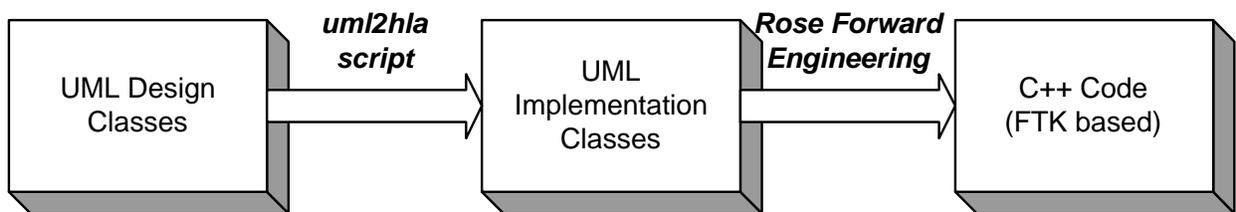


Figure 7: Work-flow of the uml2hla tool.

The uml2hla tool comes as a Rational Rose Addin. This particular Addin is composed of:

- A property sheet which allows the designer to select HLA specific properties for packages, classes, attributes and methods.
- A menu file which enables a direct access to the tool functionalities.
- A set of scripts (written in Basic-Script) which generates implementation classes and associated code from a set of design classes.

Most of the work is done by the main script (which calls sub-scripts) which parses the given UML design class diagram and generates the implementation classes. These implementation classes follow the mappings described in the previous chapter. They contain inheritance relationships, attributes and methods. In addition, the C++ code for each method is directly embedded in the class by the script. These classes are added to the current Rose model in a new package and can then be forward engineered. The implementation classes rely on the Federate ToolKit (FTK) which is also part of the same Rose model: this allows the forward engineering process to resolve the correct relationships between the classes.

⁷The usage of Rational Rose brought two advantages for the rapid implementation of the tool. First, Rose fully exposes its API to the developer with a set of Basic-Script classes, which allows easy parsing of UML models as well as the creation of new model elements. Second, the ANSI C++ forward engineering capabilities of Rose is used to generate the skeleton of the code as well as the file dependencies (#include statements).

3.2 Installation

A simple installation program is provided to setup correctly the uml2hla tool for Rose. This program, called `install.sh` is a shell script ('sh') which make use of the 'sed' program. It should then be usable on any Unix platform and under Windows with the Cygwin utilities.

If this script does not work, a manual installation could always be performed as follows (a look at the previously mentioned script will give a detailed explanation of the procedure):

1. Create the directory where to put the hla Addin. You could also simply use the standard distribution directory (`$MSF_HOME/uml2hla`) and then skip steps 2 and 3
2. Copy or link the menu and property (`hla.mnu` and `hla.pty`) files into this directory
3. Copy or link the 3 scripts (`uml2hla.ebs`, `objgen.ebs` and `msggen.ebs`) into this directory
4. Change the `hla.mnu` file to reflect where the scripts are located
5. Change the `hla.reg` file to reflect where the Addin is located
6. Import the `hla.reg` into the registry using `regedit`

After having done the instalation automatically or manually, a new group of commands named HLA should appear in the Tool Menu in Rose. In addition, a package specification should present a new HLA tab. If these two checks are OK, then the installation was probably successful.

3.3 Usage

3.3.1 Requirements

The first requirement to use the automatic generation tool is to open a Rose model, which, contains the FTK package that is needed by the scripts (since the generated classes are FTK based). It is also possible to create a new model and to import the `$MSF_HOME/FTK/ftk.cat` package in it.

3.3.2 Define the source classes

The design of the communication entities should be put in a Rose 'package'⁸. For this, create a new package in the Logical View. Then open the specification of the package and configure it to be used with the uml2hla tool. As shown in Figure 8 on the following page, several fields in the HLA tab should be set according to your needs:

FedFileName is the file name of the federation file which will be generated from the classes in this package.

SubSystemName is the name of the generated Subsystem (package in the Component View) which will contain all the components for the generated classes.

⁸Multiple packages containing design classes are allowed, but at least one should exist because the script act on a complete package.

CategoryName specifies the name of the generated Package (in the Logical View) where all the generated classes will be produced.

CodeGenerationDir specifies the directory (on your filesystem) where the generated code will be written. This directory should exist before running the code generation. Rose variables are allowed in this field.

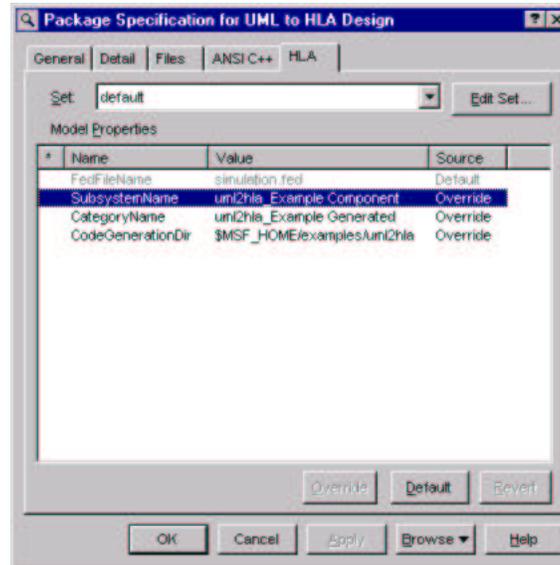


Figure 8: Specifications of a package containing design classes for communication entities.

In addition to the design package, one component (in the Component View) should also be created and have the HLA Language assigned to it (one example is shown in Figure 9 on the next page). This way, the design classes can then be assigned to this component⁹.

The next step is to populate the design package with the classes representing the communication entities. Each added classes should be assigned to a component being of HLA language.

The attributes of the class should be simply defined as usual. They could be left as *public* since the script generating the implementation classes will convert them to private data members. There is an additional HLA tab in the attribute specification (as shown in Figure 10 on the following page) which allows to specify the characteristics of this attribute seen as an HLA attribute (Transport method and Delivery mode).

The operations of the class should also be defined as usual. Their arguments will become HLA parameters in the generated code.

3.3.3 Generate implementation classes

Once all the design classes have been defined, it is possible to generate automatically the implementation classes. To do so, the icon of the design package should appear on a class diagram. Select this icon and then

⁹It is possible to create one component with the HLA classes for each design classes, but it is not necessary since it is only used to set the HLA language

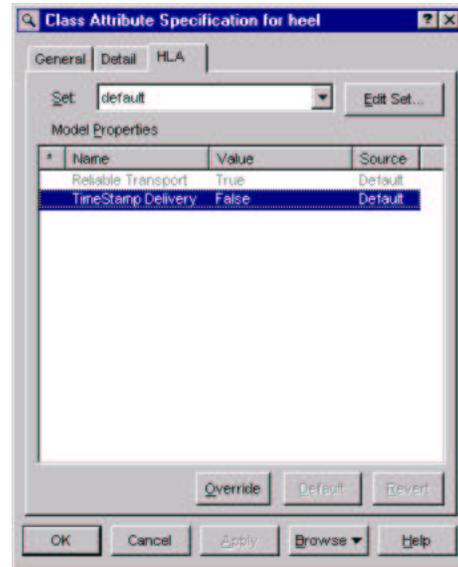
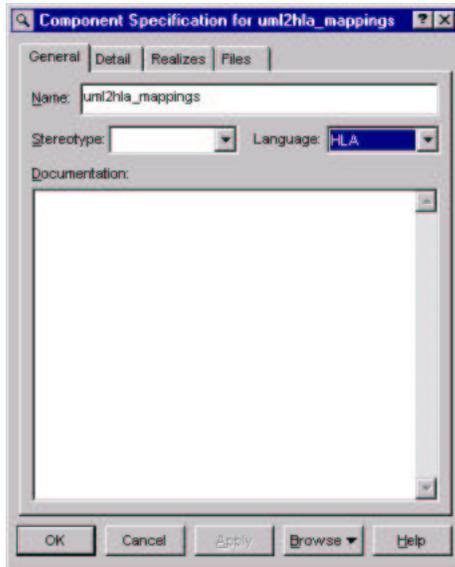


Figure 9: Specifications of a component with the Language set to HLA. Figure 10: Additional specifications of an attribute for the HLA language.

go under *Tools -> HLA -> Generate HLA classes for package_name* (if this menu is not available, it means you did not select a package or you selected one but in the browser view).

The “Generate HLA classes” command will launch the appropriate script to process the selected package. The script opens a window which displays some information about the process. The script will create one new subsystem and one new package according to the names defined in the design package specification (see previous section). If this subsystem or package already exists in the Rose model, they will not be deleted. The script then generates all the necessary implementation classes. If a class with the same name already exists in the implementation package, then it will be completely cleaned up (remove attributes, operations, relationships), but not deleted: this allows to keep coherency in existing diagrams showing it. The script also creates one component for each generated class. These components are configured to generate C++ code with the correct dependencies.

After the script completes, you should have a new package containing a set of generated classes (which are inside the msf name space). Now, you can select individual classes and generate code for them using the Rose forward engineering capability (*ANSI C++ -> Generate Code*). Note that the code generation could also be done from the component view. Finally, care must be taken to generate code for the classes in the order of their inheritance tree (this applies only for the dependencies inside the generated classes): first begin to generate code for the parent classes and then their children. In the presented example, there is inheritance only between `VesselInstance` and `SailboatInstance`: this means you should first generate code for the class `VesselInstance` and then for the class `SailboatInstance`. Otherwise, the generated file `SailboatInstance.h` will not contain the required include directive `#include "VesselInstance.h"`.

4 Problems and Limitations

- The uml2hla.ebs script will crash when it tries to generate a class with a name which already exists in another package.



Figure 11: Object Class and Instance details.



Figure 12: Interaction and Message details.