GENESIS: an integrated platform for designing and developing HLA applications

Jean Bourrely
Patrice Carle
Michel Barat
François Lévy
ONERA - Office National d'Études et Recherches Aérospatiales
29 av Division Leclerc
92320 Châtillon FRANCE
jean.bourrely@onera.fr, patrice.carle@onera.fr, michel.barat@onera.fr, francois.levy@onera.fr

Keywords:
HLA, code generation, design and development tools, genesis, C++

ABSTRACT: The first objective of the GENESIS project is the automatic design and development of HLA federates. This 3-years project started in January 2003. It is being conducted by two departments of ONERA (the French National Air and Space Agency). The French General Armament Directorate (DGA) funds this project. Its main purpose is to provide a framework consisting of a methodology and software tools. The basic requirements of this framework are as follows: (1) it must provide coherence between the object model and the federates software, (2) it should be able to handle already existing federates, objects models and federations, (3) it should cover the whole area of federates development, from the earliest stage of description, to the end of realization, (4) it should provide fully functional federates without any additional development.

To satisfy these requirements, our approach was to develop a formal language called the Genesis Description Language (GenDL) to describe the entire object model, federates and federation. The GenDL language is linked with code generation techniques, in order to finally and automatically produce the resulting software. The formal language has been established to translate every HLA concept. New concepts have been designed to ensure our objectives. For example, we introduced the concept of “importation” for object classes in a federate. This new concept specifies the use of instances of this class in the federate, by adding behaviours to these instances (discover, reflect, remove...) and by verifying the consistency of sharing attributes among federates.

The current version of GENESIS produces FOM and SOM files in the form of HTML tables. It also generates C++ source code of all HLA data types, classes, federates, and also makefiles to compile these sources into libraries and executables; it also produces federation configuration files (.fed" for RTI NG1.3, "xml" for RTI IEEE 1516) and OMT DIF files. The C++ code produced is not an "empty shell"; everything is done to minimize the work of future users who have only to give the relationships between their simulation models and the HLA part.

Recent uses of GENESIS include the retro engineering of the real-time platform reference object model (RPR-FOM), the development of a Federation to model activities, communications, loads and time delays in a Ballistic Defence System, and several other benchmarking & testing federations.

1. Introduction

GENESIS is a project of the French National Air and Space Agency (ONERA), sponsored by the French General Armament Directorate (DGA). ONERA has long and extensive experience in the domain of distributed processing & simulation (actors languages, multi-agents systems, DIS and HLA).

1.1. Inherent complexity of distributed simulations

Let us define a federation that emits, receives and prints “hello world” and which is composed of two federates. Why is such a simple federation so difficult to develop? The reason lies in the structure of current programming languages. They are not adapted to the description of distributed applications. Indeed, they miss all the inherent concepts.
There are two ways of writing software: procedural programming or declarative programming\(^1\). In the HLA standard, concepts are expressed in a declarative way, but they are programmed using procedural languages (C++, Java, Ada...).

The concepts to be expressed are the modes of information sharing among the federates. If we could define these modes in a simple, declarative way, the “hello world” federation would only require a few lines of code to be created. We shall see that, in fact, such applications consist of a few lines of code if an appropriate language is used.

1.2. An integrated development environment?

Let us use the building of a graphic interface as an example. People now use graphical formalisms and rapid application development (RAD) tools instead of writing subroutines to create buttons, checkboxes and other interface items. Such objects are now automatically generated in the appropriate programming language by a tool associated to the graphical application or are represented by resources used by the program.

A parallel can be drawn with the HLA concepts of distribution and sharing of information. RTI is then the means for doing the distribution, it is the programming language of our metaphor; the FOM/SOM design files are the visible part of the graphic interface. In a graphic interface, the links between the graphic objects (widgets) and the actions that are associated with them are usually described as the local methods of the windows or the interface frame. The interface generator produces the decoding code of the mouse events using the coordinates of the mouse’s position as keys, and it ensures coherence between an event and the action associated with the graphic object involved. The user need only to worry about the actions attached to the graphic objects without worrying about how these actions will be activated.

We propose creating the same type of mechanism in the field of distributed simulation. Here the designer must focus on the actions triggered in the federates to manage the objects and the interactions, disregarding the encoding/decoding of the information and keys for accessing the objects.

GENESIS applies these principles. We will see later that it is possible to design and make a complete federation without using anything other than the GENESIS tool itself.

1.3. Links with Federation Development and Execution Process (FEDEP)

The HLA FEDEP [1] is a recommended practice describing a process for designing, developing, reusing and executing HLA federations.

GENESIS is useful for middle steps 2 to 4 of FEDEP which concern the design and development of the federation. It is not involved in step 1 (specification of the federation), nor steps 5 to 7 (tests and evaluation of the results of the federation).

During step 2, the tool is used to represent the conceptual model of the federation by reusing existing models or by creating new ones.

In step 3, the GENESIS tool can be used for the selection of federates or to reuse existing federates.

In step 4, the tool produces the documentation of the federation (FOM) and the federates (SOM), and the federate configuration files (FED/FDD). It generates the C++ source code of the federates and the makefiles for compiling it (see [9] for a previous analog approach).

2. GENESIS Description Language

2.1. Objectives of the language

The description language of GENESIS (GenDL) is the unique point of entrance for the user during all the phases of conception, then during the phase of development of the federation or federates. It is thanks to this language that the user specifies all the HLA notions present in the DoD 1.3 [2] and IEEE 1516 [3] standards. But it is also thanks to this language that the user makes the link with the internal logic, the simulation part of his federates.

GenDL was established by taking into account specifications of the two standards. However, as the IEEE 1516 standard is richer and includes somewhat the previous standard, the user will feel he is describing his application according to this standard, rather than the former. The HLA concepts present in these standards were enhanced with notions specific to GENESIS which are described hereinafter.

Why a new language? There are two reasons: first, it was necessary to analyze in detail all the concepts related to the HLA distributed simulation, their mutual interactions,
and to transcribe them without constraints in an entirely new language, which would have been more difficult in a pre-existing language. The second reason is that by creating a language, we create the compiler associated with it, and we control the generation actions that can be performed from code written in this language. For example, generation of the C++ code is only a "plug-in" that is added to that of the documentation generation.

2.2. Syntax of the GenDL language

The syntax of the GenDL language is derived from C++. For example to declare a new record type we write:

```c
RecordType VariableDatumStruct
{
    DatumIdentifierEnum datumID;
    HLAinteger64BE datumLength;
    HLAopaqueData datumValue;
};
```

Here "RecordType" is a keyword of the GenDL language, which means that we begin a declaration of a new complex type.

The syntax of GenDL language is expressed using Backus-Naur Form (BNF) grammar [4].

```
<recordType> ::= RECORDTYPE <defType> [<note>]
   \[ [ <semantic> ] ]
   [ ENCODING \{ ( HLAFixedRecord
   | UserFixedRecord ) \} ]
   [<note>] \};
   \[ ( <refType> <defNameField> [<note>] \};
   [ <semantic> ] ]
   ]*
\};
```

Figure 1 : example of GenDL grammatical rule, the case of Record Types

This grammatical rule means that a record data type consists of a name, semantics, an encoding mode\(^2\), and several fields having each their semantics. Some of these characteristics are optional; in fact only the name of the type, the opening and closing brackets and the final semicolon are compulsory. Notes can be collected in several places.

Another more explicit way, to represent the previous grammatical rule is to use a graphic formalism:

```
Figure 2: graphical representation of the record type grammatical rule
```

The grammar of the GenDL language includes approximately 170 keywords (tokens) and 130 rules. All the concepts present in the DoD 1.3 and IEEE 1516 standards are represented, none is left aside.

2.3. New concepts introduced by the language

However, certain concepts are not present in the HLA standard and are additions of GENESIS. For example, the notion of equivalent types of a basic datatype exists only because it is necessary to link the HLA data types to types of the C++ language (predefined or defined by the user). For example, it is natural to say that the equivalent type of the type "HLAfloat64BE" of the IEEE 1516 standard is the "double" type of C++.

Among the additions we made, not present in the FOM/SOM standard:

- The possibility of declaring a datatype useful to represent time, either a moment (time-stamp) or a temporal duration (lookahead). In the current version of the GENESIS tool, this possibility is opened to simple datatypes, having for equivalent type the "double" type of C++;
- The possibility for the basic, record, array and variant datatypes to use the mode of encoding defined in the IEEE 1516 standard (§4.12.9 in [3]) or to define a particular mode;
- For a federate, the possibility of specifying its time advance policy: by steps of time or by events, the initial and final simulations times, the initial value of lookahead.

\(^2\) HLAfixedRecord means that the encoding of the record data type is made according to the specification of the standard IEEE on 1516 (§4.12.9 "Predefined encodings for constructed datatypes" in [3]). UserFixedRecord means that the encoding is left to the discretion of the user; he has to specify it in a note and to implement the encoding using a particular C++ code, directly in the generated sources codes.
For the federates, the notion of "import" was added to specify the behaviour with regard to objects, interactions or points of synchronization. This separation between the definition and the specialization by the import allows to clarify what is shared between the federates and what is not, and so ensuring the coherence of the federation (see [5] for development on reusable component objects).

The usual characteristics of publication or subscription are present in this import mechanism, with some additions in GENESIS:

- For a class of interactions, it is possible to define C++ actions when receiving an instance, by making the link with the internal code of the simulator;
- For a class of objects, we can specify discovery or disappearance behaviour of an object; we can define reception behaviour on updates of attributes;
- For objects attributes, we can define whether the updates are "pushed" by the federate which publishes the attribute, or are "pulled" by the one which subscribes to them;
- For a synchronization point, we can declare the condition to be performed by the federate in order to achieve the point, as well as the behaviour of the federate when the whole federation is synchronized.

As an example, in the following table, we represent every case of publication or subscription for an object attribute, in a given federate. The first three columns contain notions present in the HLA standard. The fourth is added by GenDL. The last column gives the interpretation and indicates in advance what actions will perform a federate produced by GENESIS:

<table>
<thead>
<tr>
<th>P/S</th>
<th>Update type</th>
<th>Update condition</th>
<th>Pushed/Pulled</th>
<th>Semantics for the federate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publ. Static</td>
<td>at init</td>
<td>pushed</td>
<td>Attribute emitted once, at initialization</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pulled</td>
<td>The federate awaits requests</td>
<td></td>
</tr>
<tr>
<td>Cond. at change</td>
<td>pushed</td>
<td>attribute emitted at each change</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pulled</td>
<td>federate awaits requests</td>
<td></td>
</tr>
<tr>
<td></td>
<td>user define</td>
<td>pushed</td>
<td>attribute emitted when condition is true</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pulled</td>
<td>federate awaits requests</td>
<td></td>
</tr>
<tr>
<td>Periodic period</td>
<td>pushed</td>
<td>attribute emitted periodically</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pulled</td>
<td>federate awaits requests</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: table of every possible combination for the "import" an attribute in a federate

Portions of C++ code can be introduced in many definitions of HLA elements written in the GenDL language. They allow the user to specify files to include (#include), to declare C++ attributes, or to declare methods to be added to the classes automatically generated by GENESIS.

Some other notions do not exist in the HLA standard, for example the notion of "demons" and "scenario" which we added to give a dynamic/acting dimension to the federate. Demons are processes which run during the life of the federate. They start on temporal or logical conditions, and allow the generation of actions. Among the demons that we wrote in our applications, we find for example:

- Demons emitting interactions when a condition is fulfilled;
- Demons periodically updating the attributes of HLA objects, with regard to values calculated in the internal part – not HLA – of the simulation;
- Demons creating instances of objects at a given period or when some conditions are met, and demons destroying them;
- Etc.


The design and development of a federation "from scratch" is illustrated by the following example written in the GenDL language. Assuming that we intend to produce two federates for a very simple performance benchmark: the first one, called "FedClock" emits instances of an interaction class called "IC1", the other federate, "FedReflect" subscribes to this interaction class and prints a message when it receives an instance.

The first step is to define the data types used by this very simple federation. In order to do this, we declare in a file the following types, which are sufficient for this federation:

```c++
#include "FedReflect"
```

---

3 These two concepts are present in the programming interface (API) of the RTIs, but are not in the object model (FOM/SOM). The "pushed" mode corresponds to the use of the couple of services: Update/Reflect attributes values. The "pulled" mode corresponds to the use of four successive services: Request/Provide/Update/Reflect attributes values.
Because the time management services are used by default in GENESIS federates, we need to declare a simple type to represent time. This type appears in the Representation Time Table of FOM/SOM:

```cpp
BasicType HLAinteger32BE {
    bitssize(32);
    endian(big);
    equivalentType(int);
};

BasicType HLAfloat64BE {
    bitssize(64);
    endian(big);
    equivalentType(double);
};
```

SimpleType MyTime{
    timeType();
    deltaTimeType();
    representation(HLAfloat64BE);
    unit("s");
};

TimeStampType {
    representation(MyTime);
};

LookAheadType {
    representation(MyTime);
};

At this step, GENESIS can generate several outputs: the FOM tables, some C++ sources and an object-code library:

The second step is to complete our object (conceptual) model (see [5]) by giving the interactions (here we complicate voluntarily by adding an inheriting subclass and some parameters):

```cpp
InteractionClass IC1{
    @semantics This interaction is used to emit the clock time@
    parameters:
    HLAinteger32BE clockLong;
};

InteractionClass IC2 : IC1{
    parameters:
        HLAinteger32BE dummy1;
        HLAfloat64BE dummy2;
};
```

Interaction class tables, interactions definitions table and parameters table are changed by those declarations. For example, the parameter table is now:

```
<table>
<thead>
<tr>
<th>Interaction</th>
<th>Parameter</th>
<th>Datatype</th>
<th>Available dimensions</th>
<th>Transportation</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC1</td>
<td>clockLong</td>
<td>HLAinteger32BE</td>
<td>NA</td>
<td>HLA reliable</td>
<td>receive</td>
</tr>
<tr>
<td>IC2</td>
<td>dummy1</td>
<td>HLAinteger32BE</td>
<td>NA</td>
<td>HLA reliable</td>
<td>reserve</td>
</tr>
<tr>
<td></td>
<td>dummy2</td>
<td>HLAfloat64BE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The third step consists of writing the code of a first federate. This definition starts by declaring the federate constrained or/and regulating and by choosing the (initial) time advance characteristics:

```cpp
Federate FedClock{
    regulating(true);
    constrained(true);
    limitTime(100.0);
    initialLookahead(0.1);
    advance(bytime, 1.0);
}
```

The definition of federate FedClock goes on by giving the GenDL code for publishing the IC1 interaction. A small process (demon) runs in the federate to emit IC1 interaction at each step time:

```cpp
importedInteractions:
    IC1 { sharing(publish);
    order(timestamp); }

Demons:
    emitIC1Demon {
        rule /{ true }/ action /{
            FedClockIC1 *out = new FedClockIC1(getFederate());
            out->set_clockLong(clock());
            out->send();
        }/}
```

Scenario:
When the federate is defined, GENESIS produces:
- SOM documentation in HTML format,
- source code of the federate,
- makefiles to compile the sources,
- OMT-DIF SOM file (according to DoD.3),
- XML SOM file (according to IEEE 1516).

The fourth step involves the second federate, FedReflect, which must be able to receive the interaction and react:

```cpp
Federate FedReflect {
    constrained(true);
    initialTime(0.0);
    limitTime(100.0);
    advance(bytime,1.0);

    MainInit /{
        std::cout << "Press a key+return";
        char start;
        std::cin >> start;
    }/

    importedInteractions:
    IC1 {
        sharing(subscribe);
        order(timestamp);
        triggerReceive /{
            std::cout << "Receiving : IC1 = "
            << thisInteraction->get_clockLong()
            << std::endl;
        }/
    }
};
```

The fifth step is to describe the federation itself composed of the two federates:

```cpp
Federation SmallBenchClock {
    SupportedFederates:
    FedClock;
    FedReflect;
};
```

While doing this, the FOM documentation is now available, with the federation configuration files used by the RTI.

The sixth and last step is to compile the two federates. Then, given the RTI, the compiler, includes and libraries paths in a generic makefile, the command 'make' launches compilation. After starting the RTI process, federates can be launched and results verified.

### 4. The Software Tool GENESIS

The GENESIS software is the only tool necessary to the user from the first steps of design through the end of the development of the code. It is thanks to the GENESIS tool that the user will (a) verify his work during design, (b) produce the FOM&SOM documentation, (c) produce all the code which he needs to compile the federates, and other appendices elements like the necessary configuration files for the RTIs.

#### 4.1. Structure of the software tool

The GENESIS tool consists of three parts. The first one is the parser, i.e. the lexical and syntactic analyzer. The parser is developed using ANTLR, ANother Tool for Language Recognition, (formerly PCCTS), a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions (here the GenDL grammar).

ANTLR is configured by a single element: the description of the grammar (BNF) of the GenDL language, which also contains the collect actions. This description is made in a single file, which facilitates the maintenance of the parser and the development of new features.

The parser reads input text files, analyzes them, verifies them and stores the results in internal data structures.

The second part is made up of those data structures. They are C++ classes whose purpose is to represent the elements collected by the parser. There are approximately 70 collecting classes in the GENESIS tool. Every notion present in standard HLA has its own class.

The third part includes all the generation actions that we can perform on the collected data. This includes production of documentations, generation of the C++ source code, and makefiles to compile the source code.

The following figure presents the tool GENESIS in its natural environment:
- inputs of text files written with the GenDL syntax,
- outputs which can be compiled to give object-code libraries or executable programs.

---

This bracket and the semi-colon are ending federate FedClock definition.
4.2. Production of documentation

The GENESIS tool produces the documentation of the federation and federates from the collected information. This can involve documentation produced in several formats:

- HTML format for easy-to-consult documentations. In this format, a set of pages is generated, namely one for each table of the HLA-OMT standard.
- OMT-DIF format respecting the DoD 1.3 standard. In a Lisp-like syntax, those files are useful for the certification of federates.
- FED format respecting the RTI NG 1.3 syntax. These configuration files are essential to the DMSO RTI and to other RTI which are compatible with it (as the CERTI of ONERA [6][7]).
- XML format according to the IEEE 1516 standard. With the coming into force of this standard, this format is now to be adopted to represent the documentation of a federation and federates.

The documentation can be produced by taking the information collected at three different levels:
- The first level is the model object, i.e. a "virtual" federation in which the federates are not described yet. No federate, no notion of publication/subscription, it is for example a reference object model (RPR-FOM).
- The second level concerns a single federate. This documentation is called a SOM.
- The third level is the federation seen as a collection of individual federates, with the union of the information collected within the federates.

4.3. C++ code generation

The first part of the C++ generation involves a library of types of data defined according to the HLA standard. The library will be common to all of the federates. It mainly implements the encoding and decoding of the type values through RTI, as well as the correspondence with the types defined by the user (EquivalentType).

The second part of the generation involves each federate. The organization of the code generated for a federate is as follows:
- The class of the federate, it is the communication point with the RTI.
- Two C++ classes for each HLA class of objects published or subscribed: one for storing the values and their management, the other for the actions defined by the user.
- Two classes likewise for each HLA interaction class and for each type of synchronization point.
- A standard main program is offered with a life cycle and change loops by default.

All of the classes generated inherit generic classes from the GENESIS library (see below).
Makefile files allow for the compiling of libraries and executables: one to produce the library of data types and one per federate.

5. The GENESIS Library

This library is composed of C++ codes not specific to a given federate, but common to all GENESIS federates (see [9]). This library is linked with generated code to produce federates executable programs.

The first part of the library is composed of super-classes from which automatically generated classes are inheriting. These classes implement the functional mechanism of the federate using a model equivalent to the well-known Model-View-Control (MVC).

The second part of the library includes the "virtual RTI" which assumes the independence of implementations towards the RTI.

5.1. A model-view-control

The following figure shows the functional mechanism of a GENESIS federate. The main benefit of the MVC model is the very clear distinction between the data (Model), the behaviours (View), the events (Control).

The C++ classes generated for each object/interaction classes and synchronisation points are the model. They store the state (data) of instances present in the federate.

Some other classes generated by GENESIS are composing the view, storing the behaviour defined by the user, one for each object/interaction classes and synchronisation points. So, an HLA class is represented by two C++ classes: one for the model part, another for the behaviour part.

In each part of the MVC model, variables and methods are available to access to the two other parts. For example, in the receive trigger of an interaction (see §3: federate FedReflect receiving an interaction called "IC1"):

- this returns the address of the behaviour class of interaction (View);
- thisInteraction is the address of the interaction class where are stored the parameters (Model);
- getFederate() returns the federate (Controller).

The functioning of the MVC model can be illustrated using an example: imagine a federate that is capable of calculating the values of attributes, and another that needs to receive them for its calculations.

The transmitting federate calculates a new attribute value in its non-HLA simulation part. It calls up a method of the model part of the MVC to change the value in question. The controller (federate) is informed of this modification and puts the object instance in a pile of instances to be updated (in the HLA sense: updateAttributeValues). As soon as it is possible, the update goes through via the RTI.

Upon reception, a method (callback) is called up in the control part (receiving federate). This method decodes the message to determine the instance involved, requests that this instance be updated (modification of the model part), and then calls up the behavioral part (view) defined by the user for updating an instance. In this latter part we can assume that the user makes the link with his own needs (i.e. the "business" part of the application).

5.2. Virtual RTI: independence towards the RTI

The Virtual RTI library is the only part of a GENESIS federate which depends on the RTI used. At present, federates can operate with two RTI: DMSO RTI NG1.3v6 and ONERA CERTI [6][7]. It is intended to be compatible with a compliant IEEE 1516 RTI [8].
6. Recent Uses of GENESIS

To date, we have designed and realized several federations:
- Some benchmarks federations to show the ability of a user to produce quickly a federation;
- A more important application (in the domain of extended air defence). Two federations have been developed to evaluate several C3 (command, control & communication) architectures.

In the test federations (benchmarks) we show that it is possible to create federates using only the GENESIS tool. There is no "business" class; the user directly uses the classes produced from the HLA conceptual model to introduce the variables and methods – non-HLA – that he needs.

An application in the broader field of air defense was developed according to a different principle. A classic, non-distributed simulation was reused. In the federates, an object is represented by both the code automatically produced by GENESIS and by classes of objects presented in the reused simulator. The work of the federation designer essentially involves in making the link between the HLA objects and the pre-existing "business" objects.

We evaluated the productivity of the GENESIS tool for all of these federations: about 30 lines of code or documentation are produced for one line written in GenLD.

7. Future Works

In 2005, the GENESIS project carries on for its third and last year. The main objective is now to put the tool into practice. For that a teaching effort is currently made: instruction manuals, days of formation and training works are in preparation.

It is also planned to prove the confidence in the GENESIS produced federates while passing an HLA Federate Compliance Test, such as it was defined by NATO MSG-025 group. This test is set up by France under the DGA authority.

In conclusion, we can say that after two years of work, the initial objectives were widely achieved: we have a tool functioning according to HLA principles, without limitations or a priori. It easily produces functional federates; in which users do not have any more but to be concerned with the internal logic, the simulation part of their federates. The tool produces all that is useful for the user, from the project manager which initiates the federation design, to the C++ specialist who actually codes the federates.

8. References

[5] Base Object Models (BOMs): Reusable Component Objects for Federation Development, Paul L. Gustavson, John P. Hancock, Mark McAuliffe, SISO-SIW 1998 Fall, 98F-SIW-034