An Approach to Build Source Code for HLA-based Distributed Simulations

Gabriel Cardoso dos Santos
Graduate Program in Computer Science
Federal University of Santa Maria
Santa Maria / RS, Brazil
Email: gcsantos@inf.ufsm.br

Raul Ceretta Nunes
Applied Computing Department
Federal University of Santa Maria
Santa Maria / RS, Brazil
Email: ceretta@inf.ufsm.br

***Abstract*—IEEE 1516-2010 High Level Architecture (HLA) is a standard used to build simulators that support interoperability. This standard requests a set of rules described in a Federation Object Model (FOM), which is an agreement for the simulation interoperability. In this context, developing a simulation with several simulators is a challenging task for developers due to the complexity of the HLA in handling the data provided in the FOM. Tools and techniques that seek to optimize the development process of simulators based on HLA have been emerging in recent years, bringing different types of approaches and ranging from the use of MDA to the source code, however, with little emphasis on the generation from the FOM file. In order to make the development process in the architecture more flexible, this article proposes an approach to HLA code generation from FOM file, hiding HLA specific functionalities and allowing developers to fully focus on the business rules of their simulators.**

***Index Terms*—simulation, High-Level Architecture, code generation**

I. Introduction

A computer simulation is a computer program developed to help the development of new technologies and to gain insight into the operation of systems and procedures. For example, operating procedures can be done through experiments in a controlled (virtual) environment. The distributed simulation of a system can satisfy the concurrent needs of multi-users and it is employed on systems of systems design. However, the challenge is the significant efforts required for producing a distributed simulation model and analyzing simulation results. One of the most popular efforts to mitigate this problem was the proposition of the IEEE 1516-2010 - High Level Architecture (HLA) standards [1], a general-purpose architecture for distributed computer simulation systems.

In HLA, a simulator is called $federate$ and a distributed simulation with a set of simulators is called $federation$ and are executed in a middleware called Run-Time Infrastructure (RTI). For the simulation development, it is mandatory the existence of a Federation Object Model (FOM), in which must be defined object classes, interactions, and data types. The FOM defines the agreement on how different simulators should operate in a federation. In this context, entities of a federate are represented by object classes and each object class has a set of attributes. Communication between federates is achieved by requests for objects and attributes manipulation. It is required that federates own an object or attribute instance before they can update its state or value. There is no global state and each federate is responsible for maintaining its own local information about objects simulated by other federates. Also, federates declare their interests via subscription and publication. Hence, developers must deal with distributed data and its serialization and deserialization procedures.

Coding and decoding complex data types can be a challenging task. Errors in coding and decoding information, data typing, and incorrect interpretation of the specification functioning are common problems in HLA simulation development [2] [3]. To help HLA developers, it was proposed the Encoding Helpers API [1], a common exchange model that covers all data that is produced and consumed by a HLA federation. Using Encoding Helpers on both sending and receiving side minimizes the risk of data mismatches, but no amount of help can handle a missing, incomplete or misinterpreted federation agreement. For that reason, it is desired a solution that reads a definition of a data type and generates HLA source code with correct encode and decode structure.

To help HLA developers, commercial solutions [4] [5] build source code from FOM definitions, but the dependence on proprietary external libraries makes it difficult to users to choose its preferable RTI. The Model-Drive Architecture (MDA) [6] has also been explored [7], but the management of encoding and decoding of complex data types is still a challenge.

This work proposes a new approach for building a HLA source code generator from FOM definitions. The approach explores text models to generate HLA source code that hides the complex coding of HLA API functionalities, facilitating the development/coding process, and the generated code can be executed in any RTI solution. It is fully compatible with the specification defined by the IEEE 1516 2010 OMT standard. By exploring pre-modeled text templates to create a more clean programming interface, our approach hides HLA functionalities programming needs and solve common problems [2] [3], such as the difficulty of coding and decoding information and incorrect interpretation of the specification. As result, the proposed solution generates flexibility for teams of developers who create federations in different contexts and domains. It enables them to use the tool or approach

they consider most appropriate for building the FOM file, safeguarding the ability to adopt a desired RTI solution.

The paper is organized as follows. Section II describes related works. Section III details our approach of building the HLA code generator. Section IV presents the execution of the code generator with an example FOM file, and finally, section V presents the conclusions of the work.

II. Related Works

This section presents existing approaches in this research field, with the purpose of identifying the main shortcomings and provide a better understanding on how the proposed solution addresses the shortcomings.

In [7] was proposed the Model Driven Architecture for Distributed Simulation (MONADS), a model-oriented architecture focused on distributed simulation. Its general operation is to take a SysML specification as input to explore *model-to-model* and *model-to-code* transformations. Its source code generation tool is carried out by the HLA Development Kit (DKF) [8] that complies with the interface of the IEEE 1516-2010 standard to supports different RTI solutions. However, according to [9], DKF does not perform the encoding and decoding of complex data types.

Commercial tools [4] [5] also focused on generating source code for HLA developers. The tools create HLA source code hiding the HLA functionalities such as encoders and decoders, object updates, and RTI callbacks. Like our proposal, they work on FOM definitions to generate sketches for all defined classes and interactions, but the generated code depends on dynamic libraries and does not run over distinct RTIs. Their code generation approaches sounds good but are not open.

In [10] is presented the SimGe, a distributed simulation modeling and development environment focused on HLA. SimGe performs HLA code generation from the Federation Architecture Metamodel (FAMM) [11]. The SimGe environment allows the creation and modification of HLA object models and the import and export of HLA related files (configuration data and FOM). The SimGe code generation is oriented only for RACoN component, a wrapper for Microsoft .NET designed to make transparent the use of RTI interface. However, RACoN wrapper supports only a subset of federate interface specification of HLA 1.3 standard [12] [13] and SimGe did not comply with distinct RTI solutions.

III. Our Approach for HLA Code Generation

The proposed source code generation is designed and developed to assist developers to perform the transformation of a FOM file into HLA source code that complies with the HLA Evolved standard [1]. The goal is to achieve good abstractions for HLA functionality, contributing to reduce the time consuming on source code development, and to provide flexibility to run over distinct RTI.

Figure 1 summarizes the approach for generating generic source code in compliance to the IEEE 1516-2010 standard. The process starts with the XSD format specifications from HLA standard (ieee1516-OMT-2010.xsd), a XML Schema that is used to validate conformance with the HLA OMT specification. From this XML schema, and using the XML Schema Definition Tool (XSD Tool) [14], it is possible to generate classes that conform to OMT schema. These classes (called *ModelCode*) are base source code used on model-to-code process. The information presented on ModelCode is than used to verify if the structured information of a FOM file (FOM.xml) is well formed and if it complies with HLA specification. Finally, from a set of source code templates (Template Text), base classes and valid FOM definitions, the *Transformer* generates the source code (*Source*) that helps the user on its development task.

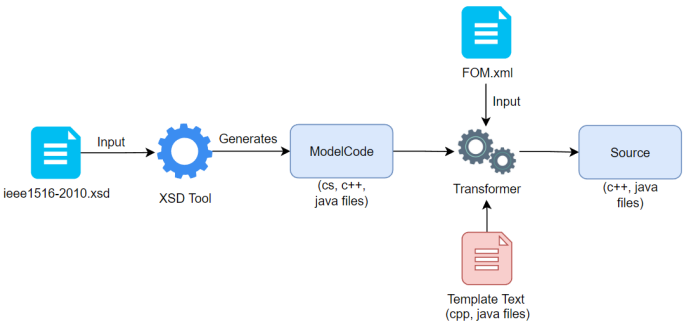

Fig. 1. Proposed Code-Generation Approach

Note that our approach corresponds to a model-to-code step in the MDA and its base elements are:

- XSD file that contains the standard rules for building a FOM file;
- COTS tool that reads the XSD file and transforms it into a model code, a file with strongly typed classes that complies with IEEE 1516-2010 standard;
- FOM file made for users or template adopted for them;
- Source code templates to turn the generated code robust, ease to use and portable.

To present details of the proposed approach, this section is structured as follows. Section III-A defines items referring to the XSD and the implementation of strongly typed classes, including the FOM validation. Section III-B specifies the construction of the templates files and the generation of the source code. Finally, Section III-C presents details of the generated source code.

A. Model Classes and FOM Validation

As the FOM is specified in XML (in one or more files), the IEEE 1516-2010 standard provides an XSD file that describes how the structure of a FOM should be defined. When modelling the distributed simulation, the designers should adopt a template FOM developed by a third part organization, like SISO, or develop a new template for its designed federation. The reuse of template FOMs is a very useful practice on HLA designs. However, when adopting a FOM, the designers frequently need to extend it for its simulation requirements. Nevertheless, the development or extension of a FOM is an error prone task and the designers must take care of a set of rules when formatting it [15].

In our process, to automate the generation of base code classes from HLA standard (XSD) and user definitions (FOM), we used two steps. The first step adopted the XML Schema Definition Tool [14] to extract a set of classes from the XSD file. These classes were then stored on a called *ModelCode*, a serial format for storage, manipulation or transport. The second step read the FOM file and mapped its definitions to the ModelCode classes, enabling us to check the FOM validation.

Note that once the ModelCode classes were built, the information could be loaded into memory for FOM validation. Without checking, the FOM definitions become susceptible to problems such as the miss specification of some object or data type. Thus, our building process checks if there is no duplicate information and no dependencies to be solved.

To deal with these dependencies, our process considers the ModelCode and FOM on building base code classes and it follows the below steps:

- Generate a set of classes that complies with the HLA standard and storage them as ModelCode;
- Load the FOM files and map it on ModelCode classes. This step performs the mapping of the information contained in the FOM for strongly typed classes;
- Analysis of FOM defined information. This step analyzes the FOM specifications and verifies if each object, interaction or data type is correctly defined. It ensures that all information provided by FOM files is consistent.

B. Transformer Process

The Transformer process is the core process of the model-to-code MDA transformation step. This process works on three sources of information: the strongly typed classes that complies with IEEE 1516-2010 standard (ModelCode), the validated FOM, and the source code templates. The process goal is to generate a HLA compliant source code that incorporates the FOM definitions and that offers to the user a friendly application programming interface (API) based on designed source code templates. Note that the templates must be previously build but they can change over time to improve the usability over the built API. Each elaborated source code template corresponds to a pattern for source code generation. In our approach we design templates to object classes, interaction classes, data types, and simulation information management.

The following subsections define how transformations are performed for object and interaction classes (Section III-B1) and for data types (Section III-B2).

1) Generating Object and Interaction Classes: In HLA two simulators interoperate only objects defined in FOM and the used FOM must be known for the RTI middleware. It means the generated code must be able to exchange each object or interaction class defined on the FOM. Hence, in a FOM object definition a designer defines the object name, its attributes and the object sharing format (publish and/or subscribe). However, an object or interaction class definition probably depends on different data types definitions and other classes that must be also defined in the FOM.

As result of our approach and considering all mentioned transformation tasks work in the same way for any object or iteration defined in the FOM, we have a pattern for the transformation process. It is always necessary: i) to instantiate the attributes or parameters; ii) to search for references in the RTI; iii) to insert the attributes or parameters into a dictionary; and iv) to decode the attributes or parameters from the dictionary.

2) Generating Data Types Manipulation Support: Following the IEEE 1516-2010 HLA standard, each data type has a rule for encoding and decoding. Standard-compliant RTI solutions offer the implementation of encoding and decoding functionalities by supporting Encoding Helpers API. However, the usability of this API is complex and it makes it difficult for developers to use it, increasing development time.

In order to generate a good abstraction for HLA encoding and decoding functionalities, we explore the use of data type definition templates, like illustrated on figures 2 and 3. Figure 2 shows templates for basic, simple and enumerated data types. On our template, basic data types can still be defined by setting data name, bit size, interpretation, type of endian and type of encoding, maintaining the ability to low-level declarations.

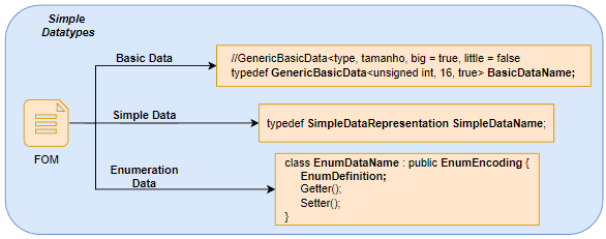

Fig. 2. Transforming simple data types to source code

For simple data types, their representations are defined by some basic type already defined, either by the standard itself or by the developer in the FOM file. Therefore, macros are used to define simple data types.

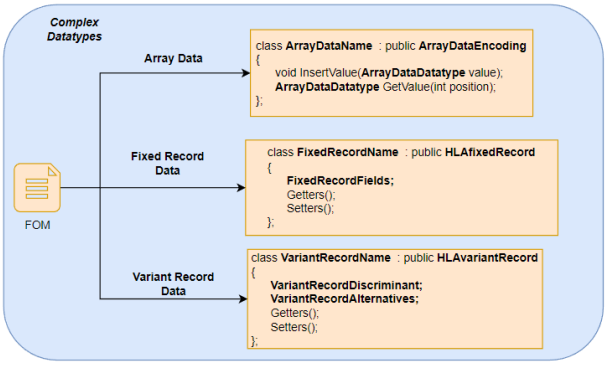

Fig. 3. Transforming complex data types to source code

For enumerators data type, it is generated a class that inherits the encoder type specified in the FOM, maintaining the FOM defined values. Based on the enumerators data type,

setters and getters methods work to validate the information as pre-defined in the FOM before sending it to RTI.

The transformation template for arrays data type defined on FOM is showed in Figure 3. The encoder offered by Encoder Helpers API is the `HLAfixedArray`, which is intended for fixed cardinality and works on encoding each element in order of occurrence. For dynamic arrays, it is necessary to use the approach proposed by `HLAvariableArray`, which has variable cardinality and the number of elements is encoded in `HLAinteger32BE`.

The solution for fixed record data type is also shown in Figure 3 and it is defined as follows: each field defined in the FOM within a `fixedRecordData` structure, in order of occurrence, is transformed into a variable with its FOM name, and its corresponding data type. When performing the data type construction, an address of each type added to the table is referred to an instance of this field in the table. Thus, methods are generated methods for: setters and getters, structure serialization and data management. Due to the inheritance of the `HLAfixedRecord` class, it is only necessary to implement the fields defined in the structure of interest.

Similar to what happens with the Fixed Record data type, the `VariantRecordData` type structures contain several fields with different types. The types must be defined in the HLA standard or in the FOM file, but they can be built from several type alternatives, where each alternative must be related to a previously defined enumerator from FOM. In our approach, the discriminant values are defined in `VariantRecordDiscriminant` and each of the alternatives are defined in `VariantRecordAlternatives` (see Figure 3).

C. Architecture of the Generated Code

This section presents the source code architecture that hides the HLA functionalities by creating methods from the Encoding Helpers routines, model classes and source code templates. This implies that the generated code must meet the following functionalities:

- Connect to a federation as a federate;
- Publish and subscribe objects and interactions specified in the FOM file;
- Encode any information using IEEE 1516-2010 OMT encoders;
- Decode information received from the RTI for use in a simulation;
- Intemperate with any RTI solution that complies to the IEEE 1516-2010 OMT standard.

Based on these functionalities, the architecture of the generated code is developed aiming to hide these HLA services. By exploring text models to orient the language interfaces we reduced the number of steps to build the code generator, as it allows isolate implementation to manage objects and interactions. Hereafter, we explain how the transformer process builds source code classes and we detail each generated class model and its functionalities. The architecture of the generated code is shown in Figure 4.

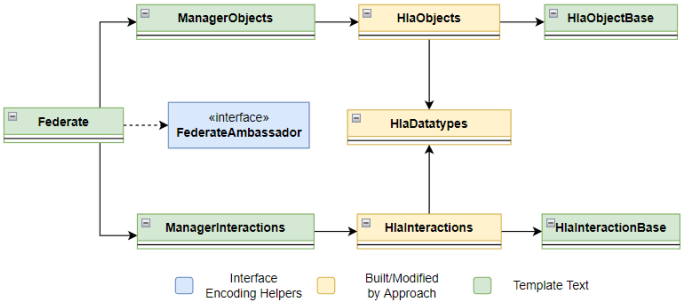

Fig. 4. Pre-modeled models and built models

The `Federate` class is used for the representation of Federate in a Federation. It hides the federation ambassador functionalities and inherits the features of the `FederateAmbassador` class specified on HLA standard. Its main task is to get reference to an `RtiAmbassador` object and to establish communication with RTI to connect the federate to federation and create, delete, update and discovery objects and interactions, hiding details for the developer.

The `ManagerObjects` class is the class designed to be responsible for managing the functionality related to objects. A dictionary of remote and local objects is defined in it, and also methods with the information of discovering, deleting, updating and reflecting attributes. We highlight these object related functionalities are represented on `Federate` class interface but in fact are redirected to implementation in this class. Methods for managing publishing and subscribing objects are also implemented in this class. Similarly, we designed the `ManagerInteractions` class to manage the functionality related to the interactions classes.

The `HlaObjects` class implements the functionalities of federation objects. It implements each object class found in the FOM and it defines the attributes and methods for updating each one. This classes inherits the `HlaObjectBase` class to deal with HLA functionalities defined on IEEE 1516-2010 OMT. The `HlaInteraction` is similar to `HlaObjects` class. It defines each of the interaction class defined in the FOM and is responsible to implement interaction functionalities.

The `HlaObjectBase` class is unique and defines common declarations independently of the object class specification in the FOM. HLA functionalities are defined in it, such as the attribute dictionary, reference to `RtiAmbassador`, location in the FOM file and references to object instance in RTI.

The `HlaInteractionBase` class is similar to the `HlaObjectBase` class. It is a single class that defines common declarations independently of the interaction class specification in the FOM. It deals with the reference to `RtiAmbassador` and with references to interactions instances in RTI.

Finally, the `HlaDatatypes` class is defined for modelling each data type according to the code model rules described in

the Section III-B2.

IV. Experiments

This section shows a case study with a set of experiments considering the SpaceFOM [16], a FOM modelled for distributed simulation of aerospace industry.

The algorithm in Figure 5 illustrates the code of this case study, generated from SpaceFOM files. The developer works on the PhysicalEntity class, which contains a `AccelerationVector` attribute defined by a `AccelerationVector` basic data type referenced to a `HLAfixedArray`. The first step of the code is to define a name and federation type for the federated. Right after the `Connect` method is called, the connection is made with the RTI in the federation named `Federation`. In line 6, the connection status is awaited, and if the connection is not successful, it returns in the condition of line 8. It is clear to note that this code hides federation connection functionalities for the developer.

```
1  wstring nameFederate;
2  wcin >> nameFederate;
3  Federate _federate(nameFederate);
4  _federate.Connect(L"Federation");
5
6  while (!_federate.isConnected());
7
8  if (!_federate.isConnectedSuccess())
9      return -1;
10
11 _federate.managerObjects.SubscribePhysicalEntity();
12 _federate.managerObjects.PublishPhysicalEntity();
13
14 cout << "start" << endl;
15
16 HlaPhysicalEntity physicalEntity =
17         _federate.managerObjects.CreateObject<HlaPhysicalEntity>();
18
19 HlaModeTransitionRequestInteraction interaction =
20         _federate.managerInteractions.CreateInteraction<HlaModeTransitionRequestInteraction>();
21
22 float value = 10.0f;
23 for (int i = 0; i < 5; i++) {
24     AccelerationVector vec();
25     vec.InsertValue(Acceleration(30.0 + value));
26     vec.InsertValue(Acceleration(10.0 + value));
27     vec.InsertValue(Acceleration(40.0 + value));
28
29     physicalEntity.Setacceleration(vec);
30     physicalEntity.UpdateValues();
31
32     value += 1.2f;
33
34     MTRMode mtrmode();
35     if (i % 2 == 0) {
36         mtrmode->set(MTRMode::MTRModeEnum::MTR_GOTO_RUN);
37     }
38     else {
39         mtrmode->set(MTRMode::MTRModeEnum::MTR_GOTO_SHUTDOWN);
40     }
41     interaction.Setexecution_mode(mtrmode);
42     interaction.sendInteraction();
43 }
44
45 auto all = _federate.managerObjects.GetAllRemoteObjectsOfType<HlaPhysicalEntity>();
46
47 for (auto e : all) {
48     wcout << "PhysicalEntity: Name =" << e->Getname()
49     << L" - AccelerationVector = " << e->Getacceleration().GetValue(0)->get()
50     << L" - " << e->Getacceleration().GetValue(1)->get()
51     << L" - " << e->Getacceleration().GetValue(2)->get() << endl;
52 }
53 _federate->Disconnect();
```

Fig. 5. Algorithm with example of implementation using SpaceFOM

Lines 11 and 12 request to subscribe and publish `PhysicalEntity` object class. These requests hide the search for attributes and object class references on RTI, like detailed in the Section III-B1. Once the interest of publishing and subscribing is concluded, the allocation of resources corresponding to the `PhysicalEntity` class is carried out (lines 16-17). The allocation process hides the construction of the `PhysicalEntity` attributes, encoders, object location information and attribute dictionary. The same occurs with the interaction class `ModeTransitionRequestInteraction` in line 19.

In lines 23 to 43, it is declared a loop to make changes in the values of the `AccelerationVector` attribute of the `PhysicalEntity` class and in the `MTRMode` parameter of the `ModeTransitionRequestInteraction` interaction class. The value of each field `AccelerationVector` is changed every loop cycle by 1.2.

Finally, when completing the changes, in lines 45 to 53, the search for remote information from `PhysicalEntity` classes is performed and their values are displayed in the console window. The `GetAllRemoteObjectsOfType` method hides the search for instances referring to the `HlaPhysicalEntity` class in the dictionary of remote instances. The values related to the `ModeTransitionRequestInteraction` interaction class are displayed when the `receiveInteraction` method of Encoding Helpers is called. The algorithm related to this event searches for the name of the interaction in the RTI, allocates the resources of the interaction class using the class constructors dictionary and ends up displaying the information if it is a `ModeTransitionRequestInteraction`.

When running the algorithm with two federates, it can be seen (see Figure 6) that the connection was made successfully in the Pitch RTI. The two federates (FederateOne and FederateTwo) join the federation named Federation that agree with SpaceFOM definitions.

Fig. 6. Connection of 2 federates with SpaceFOM in Pitch RTI

Figure 7 illustrates traces that explain what happens on encoding and decoding of attributes and parameters in Pitch RTI. The figure shows the prompt command window for each federate after the federates join the connection to the federation and start the interest on PhysicalEntity's publish and subscribe events. At the beginning, each federate starts by registering its objects on RTI and sending updates. It is noticed that the execution runs correctly and objects were created in both federates. As each federate executes asynchronously in the federation, the process of discovery, update and provide information for each object is requested by `Federate` and each information is displayed in the window. We can observe that the algorithm performs 5 times the changes in object attributes and interaction parameters.

After submission and changes are complete, the value assigned to the `AccelerationVector` attribute of the PhysicalEntity object class that was sent from FederateTwo to RTI is displayed at the end of figure 7. Note that in

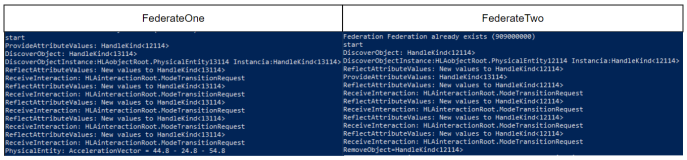

Fig. 7. Encoding and decoding attributes and parameters in Pitch RTI

FederateTwo, no value referring to FederateOne is displayed on the `PayRate` attribute of the Cashier object class, because as the federate had already finished its execution, the objects related to FederateOne are removed from its list of remote objects.

We repeated the experiment using the same algorithm but updating the RTI libraries to MAK RTI. Figures 8 and 9 show the results. It can be noted that the execution was also carried out with success on MAK RTI. The federation connection and the attribute updates were performed correctly. The values were displayed in the console window as expected.

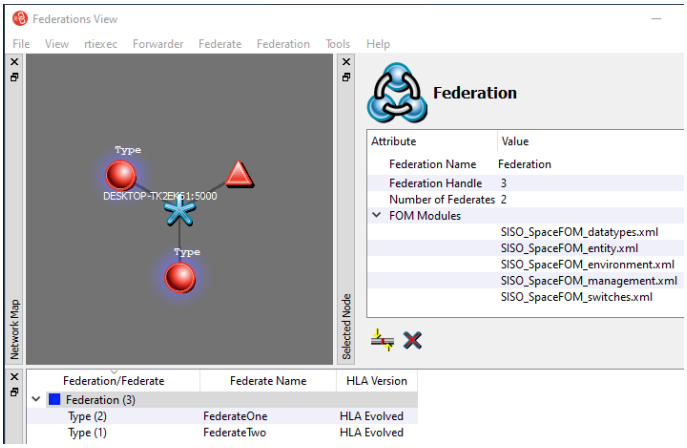

Fig. 8. Connection of 2 federates with SpaceFOM on MAK RTI

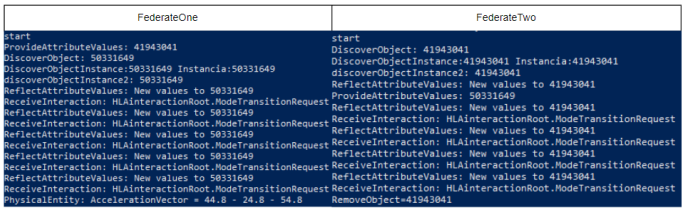

Fig. 9. Encoding and decoding attributes and parameters in MAK RTI

As experiments have showed, our approach hides a set of HLA functionalities and contributes to reduce the time needed to develop HLA based distributed simulations. The experiments have also shown that our implemented tool generates source code that is able to run on different RTIs.

V. Conclusion

The development of HLA based distributed simulation is a challenging task, since errors in coding and decoding information, data typing, and incorrect interpretation of the specification functioning are common mistakes that increase the risk and cost for the development. This work proposed an approach for building a tool that automatically generates a source code from federation object model (FOM) definitions and that hides complex codes of HLA functionalities.

Our approach explores source code templates (text models) to create an friendly programming interface and HLA standard definitions. Hence, the developer does not need to worry about basic HLA functionalities, which are handled automatically, nor with the adopted RTI. Moreover, since FOM definitions are validated, any FOM file can be used. As result, with our approach developers can focus on the simulator's business rules, optimizing their development time. A case study showed the approach works well in practice.

Acknowledgements

We thank the Brazilian Army and its Army Strategic Program ASTROS for the financial support through the SIS-ASTROS GMF project (TED 20-EME-003-00).

References

[1] IEEE, "Ieee standard for modeling and simulation (m&s) high level architecture (hla)– federate interface specification," *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)*, pp. 1–378, Aug 2010.

[2] B. Möller, M. Karlsson, and B. Löfstrand, "Reducing integration time and risk with the hla evolved encoding helpers," in *2006 Spring Simulation Interoperability Workshop*, Apr 2006.

[3] J. Graham, "Creating an hla 1516 data encoding library using c++ template metaprogramming techniques," 2007.

[4] Pitch Technologies, "Pitch developer studio," 2022. [Online]. Available: https://pitchtechnologies.com/developer-studio/

[5] MAK Technologies, "Vr link," 2022. [Online]. Available: https://www.mak.com/mak-one/infrastructure/vr-link

[6] Object Management Group, *Model Driven Architecture (MDA): The MDA Guide*, 2nd ed., Jun 2014, OMG Document ormsc/2014-06-01.

[7] P. Bocciarelli, A. D'Ambrogio, A. Falcone, A. Garro, and A. Giglio, "A model-driven approach to enable the simulation of complex systems on distributed architectures," *SIMULATION: Transactions of The Society for Modeling and Simulation International*, Fev 2019.

[8] A. Falcone, A. Garro, S. Taylor, A. Anagnostou, N. Riaz Chaudhry, and O. Salah, "Experiences in simplifying distributed simulation: The hla development kit framework," *Journal of Simulation*, vol. 11, Oct 2016.

[9] A. Falcone, A. Garro, A. Anagnostou, and S. J. E. Taylor, "An introduction to developing federations with the high level architecture (hla)," in *2017 Winter Simulation Conference (WSC)*, 2017, pp. 617–631.

[10] H. Oğuztüzün and O. Topçu, *Guide to Distributed Simulation with HLA*. Springer, Nov 2017.

[11] O. Topçu, M. Adak, and H. Oğuztüzün, "A metamodel for federation architectures," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 18, p. 10, Jul 2008.

[12] O. Topçu, "RACoN," 2022, access on July 2022. [Online]. Available: https://user.ceng.metu.edu.tr/ otopcu/racon/

[13] O. Topçu and H. Oğuztüzün, "Layered simulation architecture: A practical approach," *Simulation Modelling Practice and Theory*, vol. 32, pp. 1–14, 2013.

[14] Microsoft, "The XML Schema Definition Tool and XML Serialization," 2022. [Online]. Available: https://docs.microsoft.com/pt-br/dotnet/standard/serialization/the-xml-schema-definition-tool-and-xml-serialization

[15] W. Wenguang, X. Yongpinq, C. Xin, L. Qun, and W. Weiping, "High level architecture evolved modular federation object model," *Journal of Systems Engineering and Electronics*, vol. 20, no. 3, pp. 625–635, 2009.

[16] SISO, *SISO-STD-018-2020 - Standard for Space Reference Federation Object Model (SpaceFOM)*, 2020.