Building an HLA-based distributed simulation: a metadata approach

Agustín Santos
Institute IMDEA Networks
Avenida del Mar Mediterraneo 22
28918 Leganés, Madrid, Spain
agustin.santos@imdea.org

Katia Leal
Universidad Rey Juan Carlos
Dpto. de Sistemas Telemáticos y Computación
Escuela Superior de Ingeniería de Telecomunicación
Camino del Molino SN
28943 Fuenlabrada, Madrid, Spain
katia.leal@gmail.com

Luis F. Chiroque
Institute IMDEA Networks
Avenida del Mar Mediterraneo 22
28918 Leganés, Madrid, Spain
luisfelipe.nunez@imdea.org

***Abstract*—Building complex simulations is usually a challenge, mainly due to the broad spectrum of concepts that must be taken into account. Simulation middleware based on the IEEE 1516 standard tries to reduce this complexity, but there are still aspects that hinder the implementation of such simulations. The dependency between the source code and the corresponding formal description of the model could be hard to verify and maintain. Also, the code for the model simulation and the code needed for the integration with the HLA/RTI middleware coexist becoming complex and error prone. In this paper we propose a new framework based on *metadata* (Java annotations or C# attributes) that faces these problems and extends the IEEE 1516-2010 standard. In addition, we present an implementation of the IEEE 1516 standard in the C# language. Finally, we explain our experience in implementing the standard in C#.**

I. Introduction

Building complex and very large simulations is usually a challenge and requires a high effort, not only in its development but also for the costs of maintaining and updating such systems. Simulation middlewares based on the IEEE 1516 standard try to reduce this complexity, but there are still aspects that hinder the implementation of such simulations. Distributed simulations based on HLA/RTI are composed by three main elements that need to be developed and maintained in a coordinated way. The first one is the code of the model to be simulated. This code is accompanied by a formal description of the model (OMT specification). Thirdly, the developer must integrate all the necessary code for a distributed simulation, incorporating the usual mechanisms of HLA/RTI (such as the *federate ambassador* or the RTI calls).

The use of HLA/RTI concepts increases distributed simulation development costs in several ways. On the one hand, distributed simulations are very complex because it requires expert programmers with extensive knowledge and experience in distributed systems, simulation, middleware and other related fields. Also, in large models, is extremely difficult for developers to totally know all the model details, or even identify the best way to integrate it with the distributed framework. In addition, distributed simulation is often more difficult to tune and debug, mainly due to interactions between the simulation model and the HLA/RTI engine. Ideally, it would be desirable to be able to separately develop and test the simulation and the code that interacts with the middleware.

In addition, it is likely that the simulation model change with respect to its original conception, introducing an increasing effort to maintain and validate the model. As the simulation model change, it may appear the necessity to simulate new objects or to extend existing models. Also, it can happen that the objects have to communicate in a different way changing the protocol, message format or the internal representation. This creates inconsistencies between the three elements of the distributed simulation: model code, model specification and the code that interacts with the middleware. Of course, facing these problems supposes a great deal of time and a big effort to identify changes in the code that could violate the object model description. Thus, it is important to avoid code duplication and unnecessary verbosity on the model definition, aiming at a better code design, easier to understand and to modify.

With this in mind, in this paper we present an approach that will aid developers in implementing a distributed simulation by introducing metadata to an Object-Oriented simulation model. We propose adding semantic information to source code via metadata, expressing the intents of the annotated code and describing the HLA object model.

Source code metadata could be exploited in various ways. As descriptive information, it could be used *to automatically generate the HLA object model* and to check whether the implemented model is consistent and coherent with what is reported by external means (e.g. FOM or SOM files). But there are other uses that could be even more powerful for our purpose. We are referring to the ability to inject or generate dynamic code that extends and complements the code needed for distributed simulation. For example, it is possible to use this metainformation *to automatically generate proxies for remote objects*, or classes that represent interactions messages. As we will review in the following Section, similar techniques are implemented in some middlewares or frameworks.

In our investigation we explore the possibility of using these techniques and how to adapt them to the requirements of the IEEE 1516-2010, but at the same time keeping compatible with legacy implementations. Although Java already incorporates metadata, we wanted to achieve an additional goal by developing the first implementation known of the standard based on C#. Thus, we have developed an implementation of the HLA IEEE 1516-2010 standard in C# that could

be used as future reference for that programming language. Our framework will provide a (partial) implementation with support for Microsoft .NET 4.0/4.5 and Mono runtimes (this includes Windows and Linux platforms). Although, our first implementation is based on the C# language, our ideas can be used to create similar tools for other languages. In particular, the techniques proposed in our work can be easily translated and implemented in other languages that support metadata (e.g. Java).

All in all, our proposal consists of three major elements: the specification of the proposed annotations, the use of these annotations to generate the HLA Objet Model and a module to automatically generate code (proxies, interaction classes, etc).

The rest of the paper is organized as follows. In Section II we review the use of annotations and dynamic proxies in the context of distributed computation. Also, we will look at the most important implementations of the IEEE-1516 standard, both commercial and noncommercial, and their limitations. In Section III we present our implementation of the IEEE-1516 standard, the annotation design and the dynamic proxy generation. Also, Section III describes our experiences in programming the standard in the C# language compared to other languages like Java or C++. Finally, in Section IV we summarize the benefits of applying metadata to extend and enhance an IEEE 1516-2010 standard implementation.

II. State of the art

In this paper we propose the use of annotations to enhance the IEEE 1516-2010 standard so we could automatically generate OMT files, and thus, break the dependence between code and OMT files which difficulties portability, adaptability and maintenance. Next, we will review different solutions that apply marking techniques to middleware. Also, annotations allow us dynamic proxy generation. Thus, we can separate program logic and low level system calls which facilitates keeping classes and methods functionality, so we can write code to be less error prone. In this section we will also discuss the use of dynamic proxies in the literature. Finally, we will briefly review some of the most important, commercial and non commercial, implementations of the standard.

In last years, the use of metadata has been included in some object/component-oriented frameworks [1], [2]. Nowadays, as the amount of used metadata increases, some solutions have been studied for checking its consistency [3]. Basically, metadata is used to adapt or configure applications [4], [5], [6], to generate code [7], [8], or even to solve dependency problems [9].

As pointed out in [10], annotations are directly related to the notion of computational reflection, which was introduced in the context of procedural languages [11]. Reflection is defined in [12] as the mechanism by which a computer program can change its own structure and behaviour using self-representation metadata. Thus, a programming language is said to be reflective when it provides a reflective architecture which separates these matadata from the program itself [13]. Although, reflection is commonly used in high-level virtual machine programming languages like Smalltalk and in scripting languages, it is also used in typed programming languages such as Java, ML, Haskell and C#.

A. Metadata technique

A mark, an annotation [14] or an attribute [15] is a form of metadata that can be added to source code. Classes, methods, variables, parameters and packages may be marked. Marks can be reflective in that they can be embedded in class files generated by the compiler and may be retained to be made retrievable at run-time. Marks are often used by frameworks as a way of conveniently applying behavior to user-defined classes and methods that must otherwise be declared in an external source (such as an XML configuration file, like IEEE-1516 FOM) or programmatically. Marks are not methods calls nor comments and will not, by themselves, do anything. Compilers store annotations metadata in the class files. Later, other programs can look for the metadata to determine how to interact with the program elements or change their behavior.

The Java EE programming model uses the JDK 5.0 annotations [16] feature for Web containers, such as EJBs, servlets, Web applications, and JSPs [17]. Annotations simplify the application development process by allowing developers to specify within the Java class itself how the application component behaves in the container, requests for dependency injection, and so on.

The Spring Framework [18] is an open source application framework and inversion of control container for the Java platform. This framework provides a mechanism where it can automatically handle the injection of properties and referred objects without defining them in XML files. This is accomplished by using annotations. Spring provides different custom Java5.0+ annotations. For example, these annotations can be used in transactional demarcation, AOP, JMX, etc. There are core Spring Annotations, Spring MVC Annotations, AspectJ Annotations, JSR-250 Annotations, Testing Annotations, and so on.

The Windows Communication Foundation (WCF) [19] is a runtime and a set of APIs (application programming interface) in the .NET Framework for building connected, service-oriented applications. As Spring, WCF also provides users with annotations. For example, WCF annotations allow you to automatically validate WCF service operation arguments.

B. Dynamic proxy

As an extension of the well-known proxy pattern, a *dynamic proxy* is a normal proxy which is instantiated at run-time, rather than at compile-time. They can be defined as a meta-object [12] which, once instantiated, can intercept calls to the proxied object and change its behavior [20] without modifying the original object code. Due to its features, it is a helpful tool which are used in some programming paradigms, as AOP or dependency injection (DI) pattern [21].

Also, dynamic proxies are used in some solid frameworks for adding flexibility and reusability, as in the case of Spring or Hibernate [22].

C. Other IEEE 1516-2010 implementations

As we mentioned before, one of the contributions of this paper is the first implementation of the IEEE 1516-2010 in the C# language. In this Section we will briefly review

which other known implementations, commercial and non-commercial exist. As far as we know, none of these solutions are based in metadata.

In simulation, run-time infrastructure (RTI) is a middleware that is required when implementing the High Level Architecture (HLA). RTI is the fundamental component of HLA. It provides a set of software services that are necessary to support federates to coordinate their operations and data exchange during a runtime execution. By definition, all the approaches based on C++ or Ada do not use metadata or annotations. Only Java based solutions could use metadata, however, none of them apply this technique to their implementations.

Among commercial solutions we find MÄK RTI [23] and Pitch pRTI [24]. Both of them provide C++ bindings and implement current version of HLA, known as "HLA Evolved". On the other hand, Open HLA [25] is the only non-commercial solution that provides an open-source implementation of the HLA RTI spec 1.3, IEEE 1516 and IEEE 1516 Evolved. As many other Java implementations, Open HLA provides a framework to wrap the standard RTI classes and FOM to code generation to make life simpler. In contrast, we propose the use of annotations both, to automatically generate the HLA object model and the proxies for remote objects. Thus, with our solution developers could avoid code duplication and unnecessary verbosity on the model definition, aiming a better code design, easier to understand and to modify.

III. Our approach

Figure 1 shows the proposed architecture and how metadata fits in such proposal. Our goal is to enhance federates functionality while keeping compatibility with the HLA IEEE 1516-2010 standard. Thus, distributed simulation code could remain functional and federates will use FOM/SOM files as usual. However, with our approach developers can introduce annotations to mark important parts of the code. Then, the system included in the federate engine can explore the code, by using reflection, to gather information in order to fill up files with the HLA Object Model. Also, we could use this information to generate web services description (WSDL) if required.

In addition, our solution can even be more powerful if developers activate the automatic code generation. In this case, the system creates several C# files with classes, interaction messages, serializers and other code that can be useful to generate the simulator. As discussed below, we can generate these classes having the *partial class* characteristic of C# language. Thus, programmers can generate partial code, which is very common when part of the code is generated by automatic tools. In C#, it is possible to split the definition of a class over two or more source files. Thereby, each source file will contain a section of the definition, and all parts will be combined when the application is compiled.

Finally, the developer can activate a more advanced feature of our system by introducing (injecting) dynamic proxies in the federate code. This characteristic (very close to the concept of AOP) enables the interception of methods calls allowing the framework to take control of all the functions related to distributed simulation. For example, the system detects when a federate alters the state of some object or instance and marks

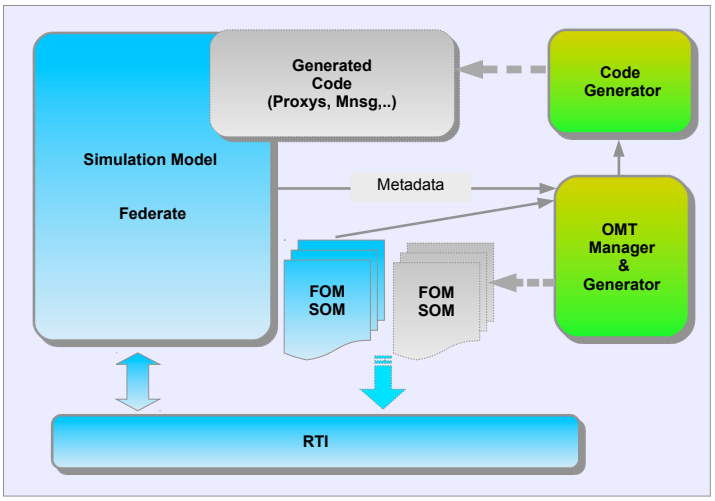

Fig. 1. **Proposed Architecture.** In our proposal we use reflection for metadata exploration and automatic OMT model description. Additionally, our system generates code that could help to maintain consistency between code and the formal description.

it as pending to be propagated to the others federates. The distributed actualization (through the RTI) can be automatic (when a state modification is detected, the dynamic proxy invokes the method updateAttributeValues) or delayed (in which case the dynamic proxy delayed the actualization until the programmer flush the information). Our interception system performs a great service for the programmer: developers can avoid to write and maintain the code specifically associated to HLA/RTI.

In this section, we will review three major functional blocks of our proposal (mark or annotation system, code generation and interception of methods calls). We end this section by describing some of the design decisions that have arisen while studying the mapping of HLA IEEE 1516-2010 standard to the C# language.

A. Metadata design

This section introduces our annotation framework composed by a set of annotations. Each core concept involved in HLA Object Model is defined as an annotation applicable to a piece of the program code. The resulting annotations are summarized in Table I.

Our first step in designing our metadata framework was to decide how to match each HLA Object Model concept with an specific metadata. In C# custom metadata (attributes) are essentially traditional classes that extend directly or indirectly the Attribute class (a class defined in the .Net platform). Just like traditional classes, custom attributes contain methods that store and retrieve data. Each Attribute is defined by three properties: the kind of element on which it is valid to apply an attribute, whether inheritance retains metadata and if the same item can be marked multiple times.

Many HLA concepts have a direct relationship with a object-orientated concept (i.e. a HLA Object class is a C# class). This relationship has greatly simplified the design of

TABLE I. OUR METADATA PROPOSAL

HLA OMT Concept	Targets	Allow Multiple	Inherited
Object Model Ident	Assembly	FALSE	FALSE
Keyword	Assembly	TRUE	FALSE
POC	Assembly	TRUE	FALSE
References	Assembly	TRUE	FALSE
Glyph	Assembly	FALSE	FALSE
Object Class Structure	Class, Interface, Struct	FALSE	TRUE
Interaction Class Structure	Class, Interface	FALSE	TRUE
Attribute	Property	FALSE	TRUE
Parameter	Field, Property	FALSE	TRUE
Dimension	Assembly	TRUE	FALSE
Time Representation	Assembly	TRUE	FALSE
User-Supplied Tag	Assembly	TRUE	FALSE
Synchronization	Assembly	TRUE	FALSE
Transportation Type	Assembly	TRUE	FALSE
Update Rate	Assembly	TRUE	FALSE
Switches	Assembly	TRUE	FALSE
Basic Data Rep.	Assembly	TRUE	FALSE
Simple Datatype	Assembly, Class, Struct	TRUE	FALSE
Enumerated Datatype	Enum	TRUE	FALSE
Array Datatype	Assembly, Class, Struct	TRUE	FALSE
Fixed Record Datatype	Class, Struct	TRUE	FALSE
Record field	Field, Property	FALSE	FALSE
Notes	All	TRUE	TRUE
OnInteraction	Event, Method	TRUE	TRUE

our framework. However, other concepts of simulation model information, like point of contact (POC), have no obvious correspondence. A model is a combination of classes, objects and interactions that, together, offer a simulation or part of it. In the course of this section we will see some of the main concepts and how they are translated into metadata.

In our framework, a typical federate could be composed by one or more compiled code libraries (*assemblies*, in the Common Language Interface) that are loaded dynamically using a plug-in system. The definition of the plug-in provides information about the assemblies to be loaded (file location and load order) and if the metadata must take precedence over information from existing files. Once the assemblies are loaded, our framework explores them extracting all the metadata information.

a) Object model identification table: The purpose of this table is to document certain key identifying information within the object model description including name of the model, purpose, key dates, POC and so on. All this information is global to the HLA object model and thus, Object Model identification table information is stored as metadata at the *assembly* level.

In our framework, this concept has been mapped to the following C# attributes: ObjectModelIdentification, POC, Keyword, References and Glyph. All theses attributes must be defined at assembly level. Following the standard, some of them may be defined multiple times (i.e POC or Keywords). Obviously, our framework shall validate and verify that the metadata included in the assemblies are valid and conform to the standard.

b) Object Class Structure and Attributes: This information records the namespace of all federate or federation object classes and describes their class-subclass relationships. Given the Object-Oriented approach of C#, a HLA Object Class can be directly associated with class-level annotations. There are two schools of thought on how to best extend, enhance, and reuse code in an object-oriented system: Inheritance (extend the functionality of a class by creating a subclass) and Aggregation (create new functionality by taking other classes and combining them into a new class). In this version of our framework, the class-subclass relationships are defined using inheritance. For this reason, the Object Class Structure is defined using the C# attribute ObjectClassAttribute at *class* level. We plan to extend our framework in order to include also aggregation as a mechanism to describe class-subclass relationships. For this the reason we also allow the definition of the metadata at *interface* and *struct* levels.

Attributes table is used to specify features of object attributes in a federate or federation. At this point, care is required with the terminology used because the word *attribute* means different things depending if we are talking about HLA Attributes (a element of information inside a object) or about a C# attribute (a metadata or annotation). HLA attributes refers to fields or properties in a C# class. Therefore, this information is annotated as C# attributes at *field* or *property* level [1]. We don't allow applying them multiple time.

c) Interaction Class Structure and Parameters: This table is used to record the namespace of all federate or federation interaction classes and to describe their class-subclass relationships.

As before, this information corresponds directly to common concepts of object-orientation in C#. Interaction classes are defined using the C# attribute InteractionClassAttribute at *class* or *interface* level. On the other hand, Parameters are defined using ParameterAttribute.

Additionally, we have defined an attribute that can be applied to C# methods or events. This attribute is called OnInteractionAttribute and defines the method or an event that is called when an iteration arrives. This information is available for the federate ambassador and is optional to the developer.

An example is provided in code figure 2 both in C# and in Java.

d) Data types: Several of the OMT tables provide information for datatype specifications. These tables describe types that are used to specify others characteristics like attributes or parameters.

Basic data representation are predefined as primitive data types at *assembly* level. The C# attribute includes information about the native type. An example is presented at the code snippet 3

Simple data type information is defined at several levels. In addition to class or struct, we have also allowed to define this attribute at assembly level as some of the simple data types have primitive implementation in C# (i.e char or byte). A similar situation occurs when dealing with Array data types. It can be defined at class, struct or assembly (unicode string) levels. Enumerated data types are mapped directly to C# enums. Fixed and Variant record data types are defined using annotations at the class or struct level. Unlike Java, C#

[1] C# Properties get and set values. It is a convenient way to simplify syntax.

```
// a C# example
public class CountrySimulator
{
    [OnInteraction(Interaction="startSimulation")]
    public void OnReceivedStart() {}


    [OnInteraction(Interaction="stopSimulation")]
    public event OnInteraction OnReceivedStop;

    // Other fields and methods go there ...
}

// The same example in Java
public class CountrySimulator
{
    @OnInteraction(Interaction="startSimulation")
    public void OnReceivedStart() {}

    // Other fields and methods go there ...
}
```

Fig. 2. **Interactions Metadata.** This code snippet represents how metadata could be used to identify methods or events that are called when an interaction arrives.

```
[assembly: BasicData(Name = "HLAinteger16BE",
                     Size = 16,
                     Interpretation = "Integer in the range
                          [-2^15, 2^15 - 1]",
                     Endian = EndianType.Big,
                     NativeType = typeof(System.Int16),
                     Encoding = "16-bit two's complement
                          signed integer. The most
                          significant bit contains the
                          sign.")]
```

Fig. 3. **BasicData Metadata.** This example shows how an assembly level annotation could be used to define basic data or other general information.

allows to define value types (as opposite to reference types). These types are called *struct*, and it is a concept very close to the concept of record defined at the standard. In the record definition has been necessary to incorporate other annotations identifying fields (normal, discriminant and alternative fields).

e) Other information: Other information like Transportation Types, Dimensions, Switches, Synchronization, and so on are defined at assembly level. For instance, the transportation type *HLAreliable* is defined at code 4

```
[assembly: TransportationType (Name = "HLAreliable",
                    Reliability = true,
                    Semantics = @"Provide reliable delivery
                         of data in the sense that TCP/IP
                         delivers its data reliably")]
```

Fig. 4. **Transportation Type Metadata.** This metadata has no real correspondence in functional code but provides useful information that could be retrieve at runtime.

f) Notes: Every HLA concept may be annotated with additional descriptive information. The standard defines this information as notes that could be included as a separate table. In our work we have enabled the option to include notes on each source code element accompanying other metadata elements.

B. Code generation

OMT generator is relatively simple. The framework internally has data structures corresponding to one or several HLA Object model. These structures can be serialized to XML using the format specified by the IEEE 1516-2010. The model definition can be derived from one or more sources. The system can be configured to consider and mix both OMT files and information from the metadata.

Combining information from the OMT files and metadata, our system can automatically generate code. The system has multiple levels of usage. On the one hand, it can be configured to generate code C# in files. These files are composed of several elements that are described below. The purpose of generating this code is to facilitate the work of the developer providing a base upon which to expand the simulation model. The generated code has two specific features that make it easy to maintain and extend. In the first place, the generated code includes appropriate markings or annotations that could be used to regenerate OMT tables if necessary in the future. This avoids having to keep both versions of the model (OMT file and code). In addition, the generated code may *partial*. Partial classes and partial methods are two programming language features of .NET programming languages that make it possible for developers to extend and enhance auto-generated code. In a nutshell, partial classes allow for a single class's members to be divided among multiple source code files. At compile-time these multiple files get combined into a single class as if the class's members had all been specified in a single file. If the developer using the auto-generated code wants to extend the functionality of the class by adding new methods or properties she can do so by creating a new partial class file and putting her additions there. By having these additions in a separate file there's no risk of the tool overwriting the developer's changes when regenerating the code. The following code snippet 5 represents this idea.

```
// This code is generated into some file, for instance
     CountryGenerated.cs
[ObjectClass(Name = "Country", Sharing =
     HLAsharingType.PublishSubscribe)]
public partial class Country : HLAObjectRoot
{
    // this code could be automatically generated
    [Attribute(Name="Population")]
    public float Population { get; set; }

    // Other fields and methods go there ...
}

// This code is in another file, for instance Country.cs
public partial class Country
{
    // Code generated by a programmer..
    public void UpdateCountry()
    { }

    // More fields and methods go there ...
}
```

Fig. 5. **Object Class definition.** Using a set of metadata, programmers could define the structure and properties of the Object Class model. Note the use of partial classes.

Moreover, the generated code can be used as a basis for proxies injection and utilization. When our framework detects that a federated intends to use a particular class defined in the SOM, the tool automatically generates (in memory) and dynamically compiles the proxy that manages that object class.

Another element of the generated code is composed by classes that correspond to the objects class of the OMT. These classes can be used as proxies (in the usual sense, sometimes also called *stub*). By this, we mean a class that represents a remote real object. These classes are used as substitutes

for actual objects being simulated and updated by a remote federate. As they change the state of an object, the system propagates the changes and updates a local replica of the object. The advantage of using proxies is that it is possible to avoid the distribution of the original source code. Additionally, proxies could produce a significant reduction in memory usage by the simulation. Reproducing the full state of a remote object could be very expensive in memory usage. If in addition, the number of objects is high, the impact on memory consumption can be huge and adversely affect performance. But, in some scenarios, federates only access to a partial view of simulated objects. In these scenarios, the simulator does not require to reproduce the complete state of the remote objects and thus, proxies of this objects will help to reduce the memory impact. As developers could change the composition of objects by enabling or disabling *Subscribe* in the model definition, manual proxies implementation is complex and error prone. With our proposal, this code is generated automatically. These proxies can be generated prior to compilation (generating files that are included in the project as additional items) or automatically and transparently to the programmer (in this case, the code is generated in memory and compiled at runtime).

Code generator is part of the tool that covers generation of object classes, interaction messages and other source code elements. Specifically, our code generator generates the following elements,

1) *Object class.* For each object class in the model definition we generate the source code representing a class with the same inheritance, structure and fields described by the HLA model. This code could be used in several ways. For a developer, this functionality can serve as a starting point to extend the class. Taking advantage of the ability to define C# partial classes, the programmer can work on different files other than those generated automatically by the tool.
2) *Interactions.* For each interaction, the framework could generate a class representing the message.
3) *Serializers.* Using information described in the OMT Model, we generate a class with the serializer and deserializer of each object class and iteration. This characteristic is specially useful given that message representations are prone to modifications. For instance, a parameter or an attribute could be transmitted using big-endian or little-endian just changing the OMT description. Generating the appropriate code at runtime, developers could concentrate their efforts in other areas of interest.
4) *Data types.* For each data type, our framework generates a C# code that support the description given in the FOM/SOM file. C# has some programming characteristics that help in the definition of these data types. For instance, C# has value types (*struct*) that could be used for the definition of FixedRecord. Or the utilization of *[StructLayout(LayoutKind.Explicit)], [FieldOffset(-)]* for VariantRecord definitions.

C. Dynamic proxy generation

The dynamic proxy functionalities allow us to automatically maintain and complete the simulator code. We use the CastleProject's DynamicProxy C# library [26] for our purpose. It works using the proposed *metadata* for adding new and transparent functionality into the code. When an object is created by a federate, our framework intercepts the creation process and produces on the fly at runtime a dynamic proxy around the object. Proxy objects allow calls to members of an object to be intercepted without modifying the code of the class. Both classes and interfaces can be proxied, however only virtual members can be intercepted.

Our solution has proved to be fast and lightweight. This is due mainly to the design of the CastleProject's DynamicProxy. It has some powerful capabilities that allow the customization of the proxy type being created. In our proposal, this feature has been a key factor so we could decide which methods must intercept the dynamic proxy and avoid innecessary methods interceptions.

The alternative to this proposal is to use MarshalByRefObject. Extending MashalByRefObject to proxy an object can be too intrusive because it does not allow the class to extend another class and it does not allow transparent proxying of classes. In addition, MarshalByRefObject are sensitive to context and therefore, there is no interception of method calls when produced inside the object.

Using *interceptors* we were able to inject behaviors into the proxy and hide all the middleware functionality inside. When a simulation calls a method on the simulated object and before the method call reaches the target object it goes through a pipeline of interceptors. Each interceptor gets its chance to inspect and change those values before the actual method on the target object is called.

We use those interceptors in order to add the following functionalities,

1) *Automatic generation of handlers.* Using the metadata, the system knows the elements to be described by the handlers. Specifically, in the objects construction, the RTI can be automatically requested for handler assignment.
2) *Property update interception.* When a property is set, intercepting the method call, the system looks for any object status change. Internally, the dynamic proxy marks this object as *dirty* and uses this information for further state propagation.
3) *Property update propagation.* The dynamic proxy observes the object status and keep track of changes. When needed it propagates this information calling the RTI (for instance, calling updateAttributeValues).
4) *Remote property update.* If the RTI notify the system about any change, the object is transparently updated by the dynamic proxy. The programmer does not need to update or take care of this process. She only have a view of the remote objects that are updated as the RTI provides actual information.

One aspect that remains unclear is the destruction of the object. In C# (like Java) there is no destructor method and a more deep study is needed if we want to detect automatically when an object is destroyed. Although, C# has the *IDispose* interface, in our first implementation we do not use it for object destruction and we rely on an explicit call that must code the developer.

Considering all of these, we provide an example of how our approach could improve the implementation of an object

```
public class Country
{
    public int Population {get; set;}

    public void UpdateCountry()
    {
        this.Population += 10;

        AttributeHandleValueMap attributeMap = _ambassador.
              getAttributeHandleValueMapFactory().
              create(1);
        attributeMap.Add(_attributePopulationHandle,
             populationCoder.encode(this.Population));
        _ambassador.updateAttributeValues(thisHandle,
             attributeMap, null);
    }
}
```

Fig. 6. **Usual simulation object.** This snippet of code represents the usual code for an object class where the code of the simulation coexists with the HLA/RTI code.

class. In the usual simulation approach we mix the code about the simulation with the code that calls RTI or other middleware elements. A simplified example, could be observed in figure 6. However, using our approach, we just mark the object class and its fields with attributes (metadata). Developers only have to concern about the simulation code (i.e. how the object is compute or simulated). In the code at figure 7, we show how it could be implemented.

```
[ObjectClass(Name = "Country", Sharing =
    HLAsharingType.PublishSubscribe)]
public class Country : HLAObjectRoot
{
    [Attribute(Name="Population",
         Transportation="HLAreliable")]
    public int Population {get; set;}

    public void UpdateCountry()
    {
        this.Population += 10;
    }
}
```

Fig. 7. **Enhanced simulation object.** This new version of the same object class has annotations and without HLA/RTI code.

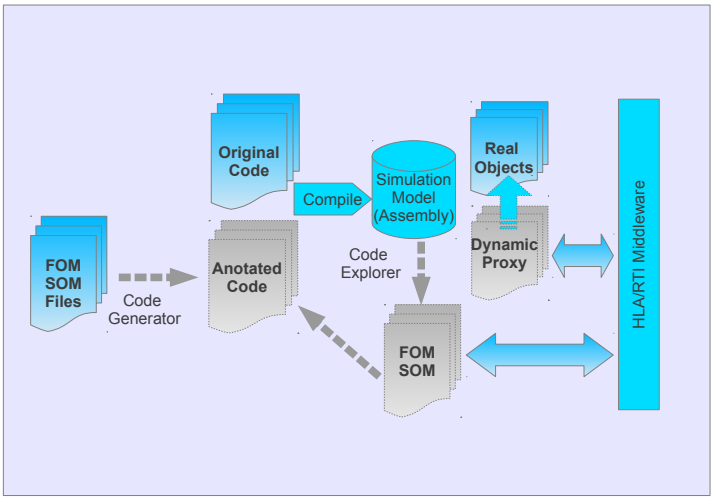

Fig. 8. **Code interactions.** Simulation modules could include both manual and automatic code. Dynamic Proxies intercept method calls at runtime.

D. C# implementation experience

While our work is focused on the application of metadata for the OMT model description, it has also generated parallel lines of work that have been of interest. One of the most interesting has emerged when we faced the porting of the standard to C#. This section briefly describes the work performed and the main conclusions.

In general, the mapping of the standard to C# was relatively straightforward. Based on the existing C++ or Java proposals, the mapping of the standard to C# was in general easy to implement. Most of the modifications were changes in notation mainly due to the usual style guide in C#. Only a small number of aspects required a more detailed study.

One of these aspects was the specification of *logical time* and *logical time interval*. In Java, the implementation of these concepts uses generics although methods in the RTI interface use the non-generic versions. This is possible due to the Java compatibility between generic and non-generic classes. In C# this is not possible and we have changed the implementation of *LogicalTime* and *LogicalTimeInterval* as follows. First, we have defined non-generic interfaces (i.e. *ILogicalTime*). Then, we have defined a generic interface that extends the non-generic version, and finally we have implemented concrete classes where operators could be defined. C# allows the definition of mathematical operator like $+$, $-$, etc., but these operators are static and thus, interfaces can not include operators' definitions. We think that our proposal can combine both C++ and Java proposals. A code excerpt of all of this is included in the code snippet 9.

```
public interface ILogicalTime
{
  // Methods like IsInitial, IsFinal, Add, etc.
}

public interface LogicalTime<T, U> : IComparable<T>,
     ILogicalTime where T : LogicalTime<T, U> where U :
     LogicalTimeInterval<U>
{
  // Methods go there
}

public struct HLAinteger64Time  :
     LogicalTime<HLAinteger64Time, HLAinteger64Interval>
{
 // Methods go there
    public static HLAinteger64Time operator
         +(HLAinteger64Time time, HLAinteger64Interval
         interval) {...}
}
```

Fig. 9. **Logical Time Proposal.** A combination of interface and generic is used to define the logical time concept. For the concrete implementation of the HLAinteger64Time a *struct* is used.

There was another issue that needed a minor change in the C# implementation. A large number of Java classes extend the Serializable interface. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable. But in C# this semantic can be expressed by using the attribute *[Serializable()]* or extending the interface ISerializable. None of these solutions is applicable to our case. The former is a property that is not inherited by extending a class and therefore meaningless in an interface. The second one requires implementing serialization methods with no use in the context of HLA/RTI. In our implementation we have chosen to remove this semantics from the standard.

Other differences of our implementation in C# are due to the use of the .Net libraries. For example, C# provides libraries for generic collections that are very similar to Java. For proper operation of these collections is necessary to implement the Equals and GetHashCode methods. We opted for the same solution proposed by Java defining these methods in the interface, even if this solution does not force programmers to override these methods.

IV. Conclusions and future work

In this paper we have proposed the use of annotations for model description and dynamic code generation in order to provide developers with a mechanism that will aid them to write clean and easy to maintain distributed simulations. As far as we know, current implementations of the IEEE 1516-2010 standard do not support this technique, not even Java ones. In addition, we report the real experience of applying this idea to the first implementation of RTI in the C# language. Thus, we explain decisions taken when mapping HLA Object Model concepts to specific annotations. Also, we describe the use of these annotations to automatically generate the HLA Objet Model and other important elements such us proxies, interactions, serializers and data types. As we have seen, code examples provided show how attributes are a convenient way to simplify syntax. So, future changes in the simulation model will be easier to maintain and validate, saving developers time.

We are currently researching several alternatives to provide a decentralized communication scheme for distributed simulation systems. Of course, we are looking for a solution that accomplishs the IEEE 1516-2010.

ACKNOWLEDGMENT

This work is supported in part by Comunidad de Madrid grant S2009TIC-1692, and Factory Holding Company 25, S.L. grant SOCAM.

References

[1] V. Cepa, *Attribute Enabled Software Development*. Saarbrücken, Germany, Germany: VDM Verlag, 2007.

[2] R. Rouvoy and P. Merle, "Leveraging Component-Oriented Programming with Attribute-Oriented Programming," Tech. Rep., 2006.

[3] P. Vaclavik, "Verification of metadata specified in various forms," *Journal of Information, Control and Management Systems*, vol. 10, no. 1, 2012. [Online]. Available: http://kifri.fri.uniza.sk/ojs/index.php/JICMS/article/view/1342

[4] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/352029.352035

[5] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005. [Online]. Available: http://doi.acm.org/10.1145/1118890.1118892

[6] J. Newkirk and A. Vorontsov, "How .net's custom attributes affect design," *Software, IEEE*, vol. 19, no. 5, pp. 18–20, 2002.

[7] J. Kollr and M. Forg, "Combined approach to program and language evolution," *COMPUTING AND INFORMATICS*, vol. 29, no. 6+, 2012. [Online]. Available: http://www.cai.sk/ojs/index.php/cai/article/view/134/111

[8] J. Porubän, M. Forgac, M. Sabo, and M. Behalek, "Annotation based parser generator," *Comput. Sci. Inf. Syst.*, vol. 7, no. 2, pp. 291–307, 2010.

[9] E. B. Passos, J. W. S. Sousa, E. W. G. Clua, A. Montenegro, and L. Murta, "Smart composition of game objects using dependency injection," *Comput. Entertain.*, vol. 7, no. 4, pp. 53:1–53:15, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1658866.1658872

[10] E. Guerra, F. Silveira, and C. Fernandes, "Classmock: A testing tool for reflective classes which consume code annotations," in *Proceedings of the Brazilian Workshop for Agile Methods (WBMA 2010)*, 2010, pp. 1–14. [Online]. Available: http://www.agilebrazil.com/2010/pt/wbma2010.pdf

[11] B. C. Smith, "Reflection and semantics in lisp," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '84. New York, NY, USA: ACM, 1984, pp. 23–35. [Online]. Available: http://doi.acm.org/10.1145/800017.800513

[12] P. Maes, "Concepts and experiments in computational reflection," *SIGPLAN Not.*, vol. 22, no. 12, pp. 147–155, Dec. 1987. [Online]. Available: http://doi.acm.org/10.1145/38807.38821

[13] F. Doucet, S. Shukla, and R. Gupta, "Introspection in system-level language frameworks: meta-level vs. integrated," in *Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 382–387.

[14] J. Gosling, B. Joy, G. L. S. Jr., G. Bracha, and A. Buckley, *"The Java Language Specification, Java SE 7 Edition"*. Addison-Wesley Professional, February 2013.

[15] J. Albahari and B. Albahari, *"C# 5.0 in a Nutshell: The Definitive Reference"*. O'Reilly Media, June 2012.

[16] "Java 5.0 Annotations," http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html, 2011.

[17] "Using Java EE Annotations and Dependency Injection," http://docs.oracle.com/cd/E11035_01/wls100/programming/annotate_dependency.html, 2013.

[18] "Spring is the most popular application development framework for enterprise Java," http://www.springsource.org/, 2013.

[19] "MSDN Windows Communication Foundation," http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html, 2011.

[20] Y. Hassoun, R. Johnson, and S. Counsell, "Reusability, open implementation and java's dynamic proxies," in *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, ser. PPPJ '03. New York, NY, USA: Computer Science Press, Inc., 2003, pp. 3–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=957289.957293

[21] "Aspect-Oriented Programming, Interception and Unity 2.0," http://msdn.microsoft.com/en-us/magazine/gg490353.aspx, 2010.

[22] "An open source Java persistence framework project," http://www.hibernate.org, 2013.

[23] "HLA RTI Run Time Infrastructure MK RTI," http://www.mak.com/products/link-simulation-interoperability/hla-rti-run-time-infrastructure.html, 2013.

[24] "A Runtime Infrastructure for the Latest Standard," http://www.pitch.se/products/prti, 2013.

[25] "Open HLA (oh-la)," http://ohla.sourceforge.net, 2013.

[26] "Library for generating lightweight .NET proxies on the fly at runtime," http://www.castleproject.org/projects/dynamicproxy/, 2013.