Experiences in Simplifying Distributed Simulation: the HLA Development Kit Framework

A Falcone[1], A Garro[1], SJE Taylor[2], A Anagnostou[2], NR Chaudhry[2], O Salah[2]
[1] *University of Calabria, Rende, Italy;* [2]*Brunel University London, Uxbridge, United Kingdom*

Distributed simulation represents a solid discipline and an effective approach for handling the increasing complexity in the analysis and design of modern Systems and Systems of Systems (SoSs). The IEEE 1516-2010 - High Level Architecture (HLA) is one of the most mature and popular standards for distributed simulation and it is increasingly exploited in a great variety of application domains, ranging from aerospace to energy, due to its capabilities to enable the interoperability and reusability of distributed simulation components. However, the development of fully fledged simulation models, based on the IEEE 1516-2010 standard, is still a challenging task and requires considerable development effort that often results not only in an increase in development time but also in low reliability. In this context, the paper presents the HLA Development Kit Framework, a general-purpose, domain independent software framework that aims to ease the development of HLA-based simulations by letting the developers to focus on the specific aspects of their simulation rather than dealing with the common HLA functionalities. Moreover, the so obtained simulation code is independent of any specific HLA platform thus enabling its deployment and execution on any desired implementation of the HLA standard provided it is written in Java. The effectiveness of the proposed framework is shown in the context of the Simulation Exploration Experience (SEE), a project organized by SISO (Simulation Interoperability Standards Organization) and led by NASA that involves several U.S. and European Institutions.

Keywords: Distributed Simulation; High Level Architecture; Agent-based Simulation.

1. INTRODUCTION

Over the years, large-scale systems have increased in complexity and sophistication since, in general, they are composed of several components, which are often designed and developed by organizations belonging to different engineering domains, including mechanical, electrical, and software. As systems get increasingly complex, their design and development become more difficult and therefore new Modeling and Simulation (M&S) techniques, methods and tools are emerging also to benefit from distributed simulation environments (Banks et al, 2009; Falcone and Garro, 2015; Möller, 2013). In this context, the IEEE 1516-2010 - High Level Architecture (HLA) standard (IEEE Std. 1516-2010) attempts to handle this complexity by providing a specification of a distributed infrastructure in which simulation units can run on standalone computers and communicate with one another in a common simulation scenario.

HLA development was initiated and sponsored by the U.S. Department of Defense to facilitate the integration of distributed simulation models within a common architecture. Although it was initially developed to support military applications, it has been used in non-military industries for its many advantages related to the interoperability and reusability of distributed simulation components. In the HLA standard a distributed simulation is called a Federation, and it is composed of several HLA simulation entities, each called a Federate, which can interact among them by using a Run-Time Infrastructure (RTI). The RTI represents a backbone of a Federation execution that provides a set of services to manage the inter-Federates communication and data exchange. They interact with RTI through the standard services and interfaces to participate in the distributed simulation execution. Each Federation has a Federation Object Model (FOM) that is created in accordance with the Object Model Template (OMT) defined by the standard. A FOM includes the definition of Objects (*ObjectClass*) and Interactions (*InteractionClass*) 0. An *ObjectClass* is composed of a set of attributes whose values define the state of the object at any point during the simulation execution; whereas, an *InteractionClass* defines an event that a Federate can generate or react to during a simulation. It is composed of a set of parameters that define its characteristics. These two kinds of information are exchanged through a publish/subscribe model by using the services provided by the RTI. A Federate can register an Object, which is an instance of an *ObjectClass*, and then change the values of its attributes. Other Federates that are subscribed to that *ObjectClass* can discover the related instances and then receive attribute value updates. The Interactions work in a similar way, except that interactions have associated a set of parameters and are not persistent (an interaction is “destroyed” after being consumed).

Building complex and large distributed simulations systems using HLA is a challenging task and requires considerable development experience. Indeed, it requires expert engineers with knowledge and experience in distributed systems, simulation, middleware and software programming. The main problem is that the development and testing of HLA Federates are generally difficult, complex, and resource-intensive not only because of the complexity of the IEEE 1516 family standards but also due to the lack of proper documentations and ready-to-use examples. Moreover, developers have to spend a considerable effort to handle common HLA functionalities, such as the management of the simulation time, the connection on the RTI, and the management of common RTI

exceptions. As a result, they cannot fully focus on the specific aspects of their own simulations (the HLA Federates). In this context, the paper proposes an effective solution to enable the agile development of HLA-based simulation in which the common HLA aspects are clearly separated from the specific aspect of the Federates under development. Specifically, the paper presents a general-purpose and domain independent toolkit that consists of a Java-based development framework, called HLA Development Kit Framework (DKF), and an accompanying documentation comprehensive of a user guide and reference examples. Indeed, the DKF allows developers to focus on the specific aspects of their own HLA Federates rather than dealing with the common HLA functionalities that are managed by the DKF core components. Moreover, the DKF-based simulation code is independent of any specific RTI implementation and thus it can be executed on a desired RTI implementation, such as PITCH (Pitch Technologies, 2016), VT/MÄK (MÄK VR-Forces, 2016), PoRTIco (The PoRTIco project, 2016), CERTI (Certi Project, 2016). The above outlined capabilities demonstrated their benefits not only for expert HLA developers but also for HLA novice practitioners (e.g. undergraduates students) involved in the Simulation Exploration Experience (SEE), an educational distributed simulation project organized by SISO (Simulation Interoperability Standards Organization) and led by NASA that involves several U.S. and European Institutions (Simulation Exploration Experience (SEE) project, 2016).

The rest of the paper is organized as follows. Related works are discussed in Section 2. Section 3 presents the HLA Development Kit with particular focus on the architecture and main services provided by the HLA Development Kit software Framework (DKF). In Section 4, the development of a HLA Federate from scratch based on the DKF is exemplified in the context of the SEE project and compared with its development without using the DKF. Quantitative analysis of the benefits provided by the DKF is presented in Section 5. Finally, conclusions are drawn and future research directions are presented.

2. RELATED WORK

Several research efforts focused their attention on the creation of HLA simulation and development environments, mainly aiming at providing an integrated toolchain for creating and simulating complex systems by using specialized modeling tools and methodologies.

For MATLAB/Simulink users different packages and toolboxes are available for implementing HLA simulators such as the *Forwardsim HLA Toolbox for MATLAB* (The Forwardsim HLA Toolbox for MATLAB, 2016), which provides a user interface that allows developers to fully design and customize their HLA Federates. Another tool is the *HLA/DIS Toolbox for MATLAB and Simulink* (MÄK VR-Forces, 2016) that is based on the *Forwardsim HLA Toolbox for MATLAB* and it provides a library of Simulink blocks specifically designed for the integration of HLA services into Simulink models. It greatly simplifies Federation development and model reuse, as well as enabling organizations to more efficiently participate in multinational simulations or implement distributed simulation models locally.

Another tool that enables developers to effectively manage the structure and assets of an HLA Federate starting from a FOM file is the *PITCH Developer Studio* (Möller, 2013). This software allows programmers to reduce the HLA learning curve by providing functionalities for creating and exporting auto-generate C++/Java code classes based on the structure of the HLA Federate.

A domain specific HLA software framework was created by the Danish Maritime Institute (DMI) (Villimann, 1999). This framework defines a range of real-time simulation concepts to support the more informal concepts available at DMI with an HLA environment. The simulation framework provides mechanisms to simplify the development of real-time simulators. *FEDEF* is another domain specific framework developed by the Defence R&D Canada – Atlantic that defines a set of APIs to support both the DMSO 1.3 and IEEE HLA 1516-2000 standards. It also provides different capabilities to simplify many programming tasks that are normally required when developing a Federate in the military domain (Van Spengen, 2010).

Other HLA frameworks are based on GRID and Cloud computing infrastructures thus providing a way to model and study complex multi-actor systems by using the typical characteristics and capabilities provided by those infrastructures (Pitch Technologies, 2016; Xie et al, 2015). Moreover, both GRID and Cloud computing infrastructures offer to the users the possibility to access in a transparent way to the computing services remotely through the Internet, freeing them of the burdens associated with managing computing resources and facilities. These characteristics make HLA-based distributed simulations much more powerful and widely accessible to users who do not have ready access to high performance computing platforms (Fujimoto et al, 2010; Taylor, Turner et al, 2012).

The HLA DKF presented in this paper differ from the above mentioned approaches in several aspects. In particular, differently from a proprietary and commercial solution that requires tool-specific knowledge and training, the HLA Development Kit is an open source project released under the open source policy Lesser GNU Public License (LGPL) and can be freely and easily customized and/or extended to cover and deal with both domain independent and domain specific aspects (as was the case with the SEE-specific extension presented in Section 4). In addition, the DKF provides advanced facilities that allow keeping the code compact, readable and reliable (see Sections 3 and 5).

TABLE 1 contains a comparison of some general aspects of the above mentioned HLA frameworks. Only *Forwardsim HLA Toolbox for MATLAB* and *PITCH Developer Studio* supports all the versions of the HLA standard. The version 1.2.0 of the *HLA Development Kit (DKF)* currently supports both the IEEE HLA 1516.1-2000 and IEEE HLA 1516.1-2010, whereas it does not

provide compatibility with the oldest version of the standard, which is the HLA 1.3. The *Danish Maritime Institute (DMI) HLA framework* and *FEDEF* support both the HLA 1.3 and IEEE HLA 1516.1-2000 versions.

The code generated by all the frameworks with the exception of the *PITCH Developer Studio* can be executed on the main HLA/RTI platform implementation such as PITCH (Pitch Technologies, 2016), VT/MÄK (MÄK VR-Forces, 2016) and PoRTIco (The PoRTIco project, 2016). Moreover, the code is in the Java language for the *Danish Maritime Institute (DMI) HLA framework* and the *HLA Development Kit (DKF)*; whereas it is in C++ for *FEDEF*. The *PITCH Developer Studio*, unlike the others, is able to generate the code both in Java and C++.

Concerning the user license, the *HLA Development Kit (DKF)* is the only framework to be released under the Open Source GNU Lesser General Public License (LGPL) license; whereas the *Forwardsim HLA Toolbox for MATLAB* and *PITCH Developer Studio* are commercial. The technical documentations of both the *Danish Maritime Institute (DMI) HLA framework* and *FEDEF* do not give information about their user licenses and terms of use. Moreover, these two frameworks are domain specific and can be used to ease the development of HLA Federates only in the military domain. The other ones are general-purpose and domain independent; this means that they can be exploited to create and manage HLA simulation components in different application domains.

All the frameworks provide technical documents in which all the details concerning its architecture and how to install and use it are well-explained. In addition, the *HLA Development Kit (DKF)* offers to developers a set of reference examples of HLA Federates created by using the DKF and some video tutorials, which show how to create both the structure and the behavior of a HLA Federate by using the Kit. Finally, the *PITCH Developer Studio* provides some ready-to-run examples through its website.

Since the *HLA Development Kit (DKF)* is an open source project it has a community of users that offer support to developers in using the DKF framework. Developers can post problems, concerns and solutions by using the official forum (The HLA Development Kit project, 2016). Some of the others framework such as the *Danish Maritime Institute (DMI) HLA framework* and *FEDEF* do not provide support because these are specific for military applications and thus arguably less popular in the scientific community; whereas the *PITCH Developer Studio* gives support to programmers through its customer support service (Pitch Technologies, 2016).

Table 1: Comparison among HLA frameworks: General aspects.

	Danish Maritime Institute (DMI) HLA framework	**Forwardsim HLA Toolbox for MATLAB**	**HLA Development Kit (DKF)**	**PITCH Developer Studio**	***FEDEF***
Version	-	-	1.2.0	5.2.0.1	-
HLA Standard	HLA 1.3/ IEEE HLA 1516-2000	HLA 1.3/ IEEE HLA 1516-2000/ IEEE HLA 1516-2010	IEEE HLA 1516-2000/ IEEE HLA 1516-2010	HLA 1.3/ IEEE HLA 1516-2000/ IEEE HLA 1516-2010	HLA 1.3/ IEEE HLA 1516-2000
HLA/RTI Supported Platform	PitchRTI, VT MAK, poRTIco	PitchRTI, VT MAK, poRTIco	PitchRTI, VT MAK, poRTIco	PitchRTI	PitchRTI, VT MAK, poRTIco
Programming Language	Java	Simulink	Java	Java/C++	C++
License	-	Commercial	LGPL	Commercial	-
Application domain	Military	General	General	General	Military
Documentation	Technical documents	Technical documents	Technical documents/ Ready-to-run examples/ Video Tutorials	Technical documents/ Ready-to-run examples	Technical document
Open Community Support	NO	NO	YES	NO	NO
Official website	-	http://www.forwardsim.com/	https://smash-lab.github.io/HLA-Development-Kit/	http://www.pitch.se/	-

A comparison of some main aspects related to the HLA standard of the introduced HLA frameworks are described in TABLE 2. The here presented *HLA DKF*, unlike the others, is the only one that provides and manages the life cycle of a HLA Federate. As a consequence, a developer has only to define the specific behavior of its HLA Federate without worrying about low-level implementation details since the DKF manages them. Concerning the Simulation model, the *Danish Maritime Institute (DMI) HLA*

framework, *FEDEF* and *HLA Development Kit (DKF)* provide functionalities to manage only time-stepped Federate; whereas the two commercial frameworks also support the event-driven model. Moving to the time management aspect, the two commercial HLA/RTI frameworks, which are the *Forwardsim HLA Toolbox for MATLAB* and *PITCH Developer Studio*, are able to manage Federates with and without HLA Time Management; whereas the *HLA Development Kit (DKF)*, *Danish Maritime Institute (DMI) HLA framework* and *FEDEF* provide support only to HLA Time Management mechanisms based on Time Advance Grant (TAG) and Time Advance Request (TAR) 0.

The HLA components created by using the capabilities provided by the *Danish Maritime Institute (DMI) HLA framework* and *FEDEF* are executed in a single-threaded mode because these frameworks do not handle multithreading mechanisms; this means that there may be performance issues as the Federation execution gets bigger. These multithreading aspects do not affect the others because they create Federate taking into account these characteristics during the generation of the source code.

Concerning the communication pattern, all the frameworks do not provide complex protocols over the basic Publish/Subscribe mechanism as defined in the HLA standard; but according to the roadmap of the *HLA Development Kit (DKF)* (The HLA Development Kit project, 2016), different communication patterns over Publish/Subscribe are under development and planned to be released in the future version of the DKF.

Table 2: Comparison among HLA frameworks: HLA standard aspects.

	Danish Maritime Institute (DMI) HLA framework	**Forwardsim HLA Toolbox for MATLAB**	**HLA Development Kit (DKF)**	**PITCH Developer Studio**	***FEDEF***
Federate lifecycle	NO	NO	YES	NO	NO
Simulation model	time-stepped	time-stepped/ event-driven	time-stepped	time-stepped/ event-driven	time-stepped
Federate execution model	Single-thread	Multi-threads	Multi-threads	Multi-threads	Single-thread
Time management	Based on Time Advance Grant (TAG) and Time Advance Request (TAR)	With and Without HLA Time Management	Based on Time Advance Grant (TAG) and Time Advance Request (TAR)	With and Without HLA Time Management	Based on Time Advance Grant (TAG) and Time Advance Request (TAR)
Communication pattern	Publish/Subscribe	Publish/Subscribe	Publish/Subscribe	Publish/Subscribe	Publish/Subscribe

In TABLE 3 the functionalities offered by the above considered HLA frameworks are delineated and compared. All the frameworks provide functionalities to manage the simulation time, mechanisms to handle the connection (set-up, hold-up and close-up) of an HLA Federate to the RTI and features to facilitate publishing and updating of *ObjectClasses* and *InteractionClasses* on the RTI.

However, concerning the latest point, it is worth noting that the *HLA Development Kit (DKF)* is the only framework that uses the Java annotation mechanism to directly inject the structure of an HLA Federate in the Java code. These metadata are also used by the DKF core components at run-time to inspect and check if an HLA Federate is compliant with the related definition in the FOM (see Section 3). Moreover, the *HLA Development Kit (DKF)* offers support in managing time standard conversions between the wall-clock and simulation time 0, and some functions to check the status of the MS Windows Firewall because in some distributed simulations it is necessary that all the computers that are participating in the simulation scenario have the firewall disabled (e.g. the Simulation Exploration Experience (SEE) project). In addition, the *HLA Development Kit (DKF)* provides a logging service useful to track down any problems or errors occurred during the execution of an HLA Federate; this information is stored into the *dkf.log* file.

Both the *Forwardsim HLA Toolbox for MATLAB* and *PITCH Developer Studio* are able to manage synchronization points, data distribution management (DDM) with logical regions services, and functionalities to transfer the ownership of an *ObjectClass* among Federates.

Despite the availability of different HLA frameworks; there are few training initiatives worldwide to promote their adoption. One of the most important is an annual event, formerly named “Smackdown” and now renamed Simulation Exploration Experience (SEE) that has been organized, since 2011, by SISO in collaboration with NASA and other research and industrial partners (Simulation Exploration Experience (SEE) project, 2016). The main objective of SEE is to provide undergraduate and postgraduate students with practical experience of participation in international projects related to M&S and, especially, to the HLA standard and compliant tools. The reference simulation scenario of the SEE Project concerns a human settlement called “Moonbase” composed of scientific equipment, storage buildings, rovers and other elements to allow astronauts to live and work on the Moon. The Modeling & Simulation Group (MSG) at Brunel University London has participated in the SEE Project since 2013. The group has

investigated issues concerning the development and standardization of distributed simulation for industry and healthcare (Taylor, Fishwick, et al, 2012; Taylor et al, 2014), as well as hybrid Federations consisting of real-time, discrete-event and agent-based simulations (Taylor, Turner, et al, 2012).

The main issue that arose from the SEE 2014 event was the complexity of the development. The students based their work on previous code developed by the group. However, the broad knowledge base of domain specific knowledge, distributed simulation (both Federate development and RTI interfacing) and the SEE event scenario still presented a major challenge due to the range of possible implementation approaches and the lack of clear development guidelines and tutorials. For these reasons the SEE project represented an excellent testbed for proving the effectiveness of the DKF and its SEE specific extension; this experience is discussed in the following Sections.

Table 3: Comparison among HLA frameworks: Functionalities.

Feature	***Danish Maritime Institute (DMI) HLA framework***	**Forwardsim HLA Toolbox for MATLAB**	**HLA Development Kit (DKF)**	**PITCH Developer Studio**	***FEDEF***
Mechanisms to manage the connection (set-up, hold-up and close-up) of a HLA Federate to the RTI.	YES	YES	YES	YES	YES
Mechanisms to facilitate the management and the publication of FOM modules.	NO	YES	YES	YES	NO
Mechanisms to facilitate the management of the configuration parameters.	NO	YES	YES	YES	NO
Mechanisms to facilitate publishing and updating of ObjectClasses and InteractionClasses on the RTI.	YES	YES	YES (Through Java annotations)	YES	YES
Mechanisms to manage the simulation time.	YES	YES	YES	YES	YES
Mechanisms for time standard conversions.	NO	NO	YES	NO	NO
Synch Points support.	NO	YES	NO	YES	NO
IP Configuration checker.	NO	YES	YES	YES	NO
MS Windows Firewall state checker.	NO	NO	YES	NO	NO
Logging	NO	NO	YES	NO	NO
Ownership transfer and data distribution management with regions.	NO	YES	NO	YES	NO

3. THE HLA DEVELOPMENT KIT FRAMEWORK

The *HLA Development Kit Framework* aims at easing the development of HLA Federates by providing the following resources: (i) a *software framework* (the *DKF*) for the development in Java of HLA Federates; (ii) a *technical documentation* that describes the *DKF*; (iii) a *user guide* to support developers in the use of the *DKF*; (iv) a set of *reference examples* of HLA Federates created by using the *DKF*; and, (v) *video-tutorials*, which show how to create both the structure and the behavior of a HLA Federate by using the *DKF*. In the following, the attention is focused on the DKF and, specifically, on its architecture and underlying Federate model-behavior.

3.1 The HLA Development Kit Framework

The DKF is a general-purpose, domain independent framework, released under the open source policy Lesser GNU Public License (LGPL), which facilitates the development of HLA Federates 0. Indeed, the DKF allows developers to focus on the specific aspects of their own Federates rather than dealing with the common HLA functionalities, such as the management of the simulation time; the connection/disconnection on/from the HLA RTI; the publishing, subscribing and updating of *ObjectClass* and *InteractionClass* elements. The DKF is designed and developed by the SMASH-Lab (System Modeling And Simulation Hub - Laboratory) of the

University of Calabria (Italy) working in cooperation with the NASA JSC (Johnson Space Center), Houston (TX, USA). It is fully implemented in the Java language and is based on the following three principles:

1. *Interoperability*, DKF is fully compliant with the IEEE 1516-2010 specifications; as a consequence, it is platform-independent and can interoperate with different HLA RTI implementations (e.g. PITCH (Pitch Technologies, 2016), VT/MÄK (MÄK VR-Forces, 2016), PoRTIco (The PoRTIco project, 2016), CERTI (Certi Project, 2016));
2. *Portability*, DKF provides a homogeneous set of APIs that are independent from the underlying HLA RTI and Java version. In this way, developers could decide the HLA RTI and the Java run-time environment at development-time;
3. *Usability*, the complexity of the features provided by the DKF framework are hidden behind an intuitive set of APIs.

The design and implementation of the DKF has been centered on typical Software Engineering methods and, in particular, on an *agile* software development process. Furthermore, it has been developed according to the concept of *Object HLA*, in this way, the development of HLA Federates could benefit also from the *Object HLA* features and functionalities provided by the Pitch Developer Studio (Möller, 2013) or similar IDE.

To promote the adoption and experimentation of the HLA Development Kit and its DKF, the Kit has been specialized in the *SEE HLA Development Kit* with the aim to ease the development of HLA Federates in the context of the Simulation Exploration Experience (SEE) project (Simulation Exploration Experience (SEE) project, 2016). The SEE-specific features (as an example, the possibility to easily implementation SEE Dummy and Tester Federates) aim not only at reducing the development efforts but also at improving the reliability of SEE Federates and thus reducing the problems arising during the final integration and testing phases of the SEE project (Simulation Exploration Experience (SEE) project, 2016). Moreover, this SEE extension allows to prove how, starting from a domain independent core of the DKF, conceived for supporting the development of general-purpose HLA Federate, it is possible to easily add and integrate application-specific extensions for supporting the development of domain specific Federates (Anagnostou et al, 2015a; Anagnostou et al, 2015b; Bocciarelli et al, 2015).

The following subsections are devoted to present both the architectural and behavioral aspects of the DKF also with reference to its SEE-specific extension (the SEE-DKF).

3.2 Architecture of the DKF

The architecture of a DKF-based Federation is composed of three main layers (see Figure 1): (i) *Application Layer*, which contains the Federates that can interact with both the DKF and the HLA RTI by using their APIs; (ii) *DKF Layer*, which represents the core of the architecture and provides a set of domain independent APIs that are used to access the DKF capabilities; and (iii) *HLA RTI Infrastructure*, which represents the RTI that host the Federation (e.g. PITCH (Pitch Technologies, 2016), VT/MÄK (MÄK VR-Forces, 2016), PoRTIco (The PoRTIco project, 2016), CERTI (Certi Project, 2016)). Some application-specific extensions of the DKF can be also introduced (e.g. the SEE-specific ones).

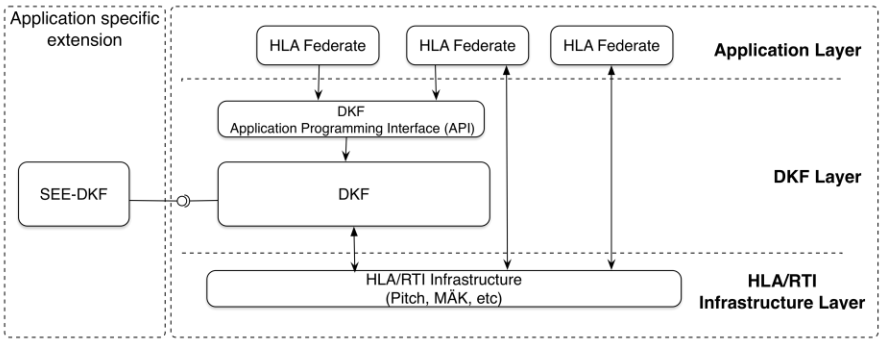

Figure 1. The architecture of a DKF-based Federation.

The *DKF* provides a set of services that are independent both of application domains and HLA RTI implementations. Each service defines some Java classes and interfaces that implement specific functionalities. The DKF architecture is shown in Figure 2. The *Core Services* layer, represents the kernel of the DKF and provides a set of low level services to manage a DKF-based application. It is composed of eight services.

The *Data Management Service (DMS)* manages publishing, subscribing and the data updating of both an *ObjectClass* and an *InteractionClass* 0. The DKF framework introduces a set of annotations to manage an *ObjectModel* (*ObjectClass* and *InteractionClass*), each of which covers a specific core concept involved in the HLA Object Model specification, and it is applicable to a piece of the program code so as to guide the core components of the DKF in managing *ObjectModel*s. Annotations represent a form of metadata that provide data about a program that is not part of the program itself, thus they do not have direct effect on the operation of the code that they annotate (Anagnostou et al, 2015a; Anagnostou et al, 2015b; Bocciarelli et al, 2015).

Annotations have a number of uses, among them: (i) *Information for the compiler*, which can be used by the compiler to detect errors or suppress warnings; (ii) *Compile-time and deployment-time processing*, in which software tools can process annotation information to generate code, XML files, and so forth; and (iii) *Runtime processing*, which can be used to examine the structure of objects/classes at runtime.

Table 4: The @ObjectClass annotation.

HLA Object Model Specification	*Annotation class name*	*Target Field in the code*	*Annotation field name*	*Description*
Object Class	@ObjectClass	Class, Interface	name	Manages the namespace of the object class and handles their relationships.
			sharing	Manages the sharing of the object class.
			semantic	Define the semantic of the object class.
Attribute	@Attribute	Class Attribute	name	Manages the attributes defined for the object class.
			coder	Defines the coder to be used to code and decode the attribute.
			sharing	Manages the sharing of the attribute.
			ownership	Handles the ownership of the attribute.
			updateType	Manages the update of the attribute.
			updateCondition	Stores the condition that defines how and when the attribute has to be updated on the RTI.
			order	Handles the order type of the attribute.
			transportation	Defines the transportation type for the attribute.
			semantic	Define the semantic of the attribute.

Table 5: The @InteractionClass annotation.

HLA Object Model Specification	*Annotation class name*	*Target Field in the code*	*Annotation field name*	*Description*
Interaction Class	@InteractionClass	Class, Interface	name	Manages the namespace of the interaction class.
			transportation	Defines the transportation type for the interaction class.
			sharing	Manages the sharing of the interaction class.
			order	Handles the order type of the interaction class.
			semantic	Define the semantic of the interaction class.
Parameter	@Parameter	Class Attribute	name	Manages the parameter defined for the interaction class.
			coder	Defines the coder to be used to code and decode the parameter.
			semantic	Define the semantic of the parameter.

In the DKF framework, two Java annotation classes, which have to be used by programmers so as to create an instance of an *ObjectModel* compatible with the DKF, have been defined: *@ObjectClass* and *@InteractionClass*. The first one provides annotations for the definition of *ObjectClass* instances; whereas the second annotation class specifies concepts to define and handle

InteractionClass instances. These two classes are used by the DKF core components at *runtime* to examine the structure of an *ObjectModel* instance. The structures of the above introduced annotation classes are summarized in Table 4 and Table 5, respectively.

The *Logging Service (LS)* allows data on the activity carried out by a simulation to be stored into the *dkf.log* file. It is very useful for finding out problems or errors occurred during the execution of a simulation, and for understanding how the DKF core services work.

The *Simulation Time Service (STS)* provides to developers some factory method that can be used to handle the two standard HLA logical time representations: *HLAinteger64Time and HLAfloat64Time* 0, and defines mechanisms for controlling the advancement of the time during the execution of a simulation. These mechanisms are coordinated with other components responsible for delivering information.

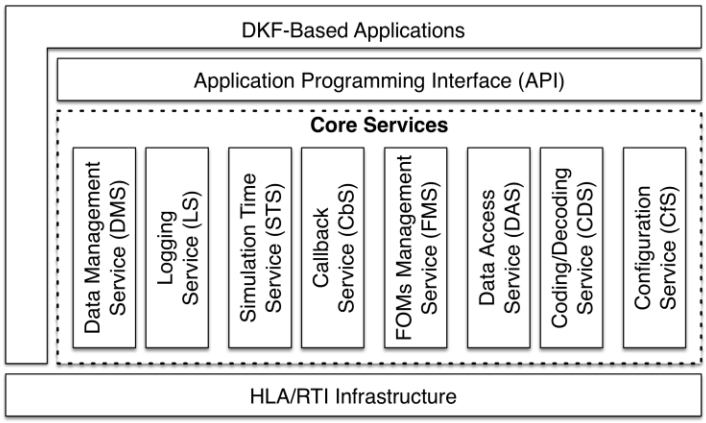

Figure 2. The architecture of the DKF.

The *FOMs Management Service (FMS)* offers functionalities for retrieving and processing FOM files. More in detail, a set of components allow a DKF-based Federate to navigate the FOM tree and get the needed data by using a XPath expression (XQuery 1.0 and XPath 2.0 Functions and Operators, 2016).

The *Caching Service (CS)* represents a caching system used during the execution of a DKF-based application for optimizing access to data.

The *Data Access Service (DAS)* defines some low level services to retrieve resources in a file system.

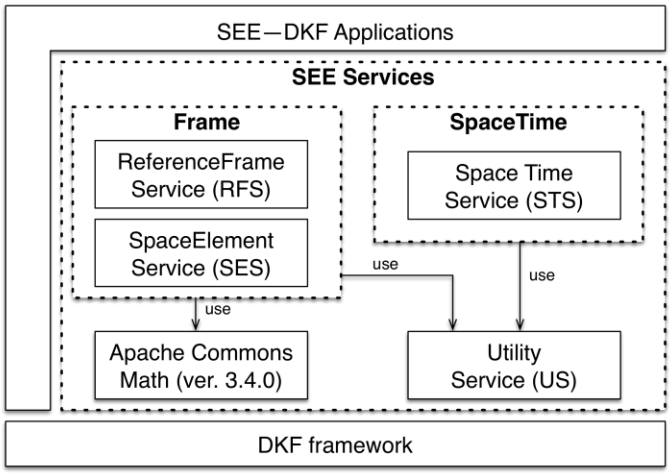

Figure 3. The architecture of the SEE-DKF specific extension.

The *Coding/Decoding Service (CDS)* defines all the standard HLA functionalities for coding and decoding both *ObjectClass* and *InteractionClass* instances (IEEE Std. 1516-2010).

Finally, the *Configuration Service (CfS)* defines a collection of services that manage the configuration parameters provided by a *.json* file. These parameters include the name of the Federation Execution, the RTI connection details (e.g. IP address, port, etc.), and details about the simulation time.

Figure 3 shows the architecture of the *SEE-DKF*, a specific domain dependent extension of the DKF that provides some SEE domain specific services, which are used by the *core* components of the *DKF* to handle the main aspects related to a SEE Federation (Simulation Exploration Experience (SEE) project, 2016), such as transformations among *SEE Coordinate Reference Frames*, the publishing and subscribing of *PhysicalEntities*, and the management of *Space FOMs* (Anagnostou et al, 2015a; Falcone et al, 2014; Taylor et al, 2014). The SEE-DKF architecture is organized in two main services sections.

The *Frame* section provides a set of services to manage basic space elements, and defines features for representing the position, geometry and characteristics of space objects such as planets and stars. Moreover, various algorithms to handle them are provided (conversions, propagations, etc.). It also defines data on the *International Celestial Reference Frames* (The International Celestial Reference Frames, 2016) and includes algorithms and functionalities to manage them. Moreover, the *Frame* section has a factory module that provides several predefined planet instances (e.g. Sun, Earth, Moon, etc.) with their specific characteristics (e.g. mass, volume, velocity, etc.) that developers can easily instantiate and use.

The *SpaceTime* section, defines mechanisms to handle epochs and dates that are commonly defined by specifying a point in a specific time scale. This section also provides many time standards such as *Terrestrial Time (TT)* and *Universal Time Coordinate (UTC)*, and defines some epochs (e.g. *Julian Epoch (JE)*, *Modified Julian Epoch (MJE)* and *j2000 Epoch*).

The *Utility Service (US)* provides several miscellaneous functions to manage both space elements and the space simulation time.

The *Apache Common Math library*, is a standard library of lightweight, self-contained mathematics and statistics components addressing the most common practical problems not immediately available in the Java programming language or commons-lang (The Apache Commons Mathematics Library, 2016). It is used by the *Frame* services to perform mathematics operations on arrays and matrices.

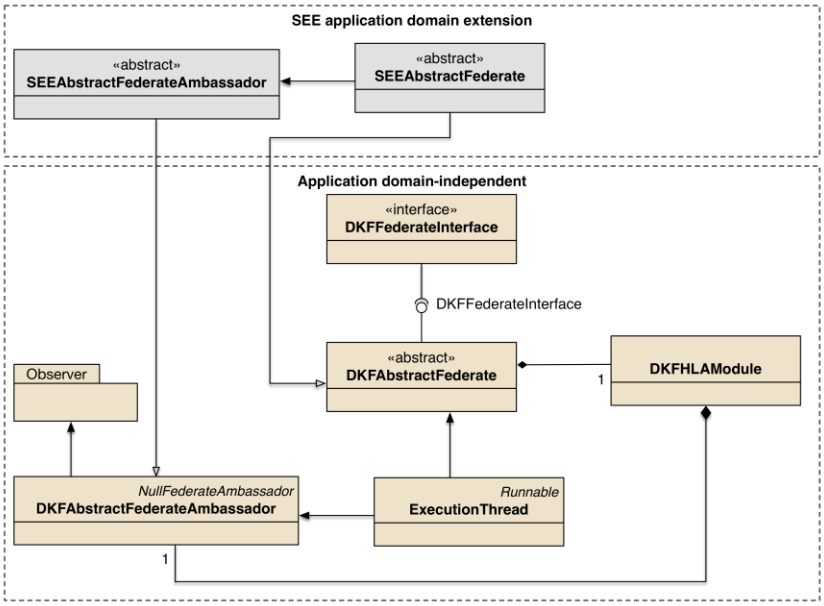

Figure 4. The architecture of a DKF-based Federate with the SEE Domain Extension.

3.3 Federate Behavioral Model

The example architecture of a Federate created by using the capabilities of both the DKF and its SEE-specific extensions is shown in Figure 4 by using a UML Class Diagram; in the following its main classes are briefly described.

The classes *SEEAbstractFederate* and *SEEAbstractAmbassador*, which are in grey, define the behavior of a SEE Federate, while the classes in yellow belong to the DKF application independent part (see Figure 4).

The *SEEAbstractFederate* class implements the methods of the *DKFAbstractFederate* class. This latter class provides functionalities to configure and connect/disconnect a Federate to/from a Federation Execution. Moreover, it is worth noting that, in the SEE context, all the Federates are exclusively *time constrained* (can receive Time Stamp Order (TSO) messages) except the

Environment Federate, provided by NASA and which leads the Federation execution, that is also *time regulating* (can send Time Stamp Order (TSO) messages) and acts as a Pacing/Clock Federate (Fujimoto, 2010); the DKF has been thus adapted to handle this situation.

The *SEEAbstractAmbassador* class implements the *DKFAbstractFederateAmbassador* class in order to interact with the RTI services.

Finally, the *ExecutionThread* class handles the execution of a HLA Federate in the simulation environment.

The *DKFAbstractFederate* class also provides and manages the life cycle of a SEE Federate according to the behavioral model that is shown in Figure 5 through a UML Statechart diagram. As a consequence, a SEE working team has only to define the specific behavior of its SEE Federate without worrying about low-level implementation details since the DKF manages them. Specifically, the pro-active part of the behavior of a Federate is specified in the *proactive* composite state, which is accessed between a TAG (Time Advance Grant) and a TAR (Time Advance Request); whereas, the re-active part of the behavior of a Federate is specified in the *reactive* composite state so as to indicate how to handle the RTI callbacks about the interactions/objects that the Federate has subscribed. Please note that the current version of the DKF and its SEE specific extension only support the implementation of time-stepped Federates as it is the reference simulation model in the SEE project; however, ongoing efforts are geared to also supporting event-driven simulations (see Falcone and Garro 2016).

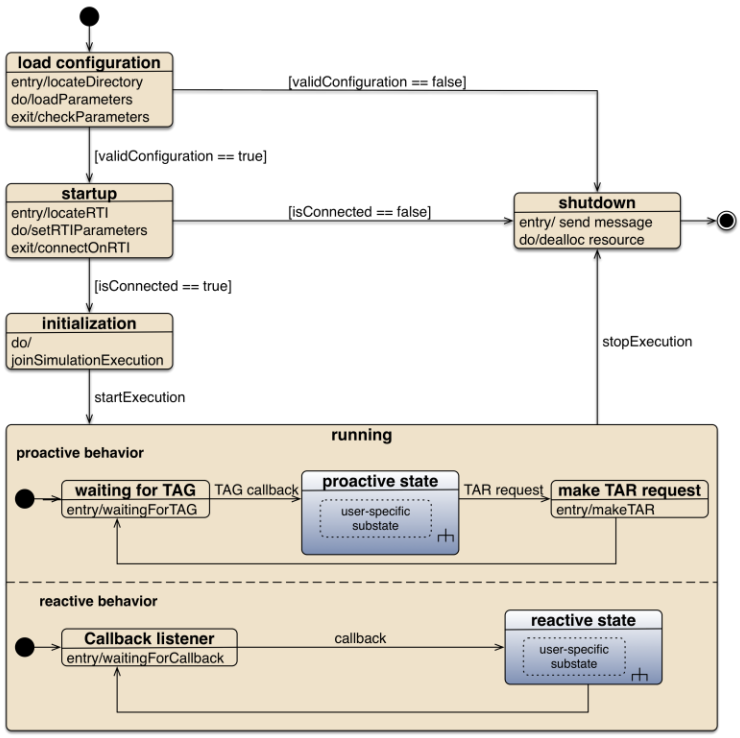

Figure 5. The life cycle of a SEE Federate.

With reference to the Federate life cycle depicted in Figure 5, in the *load configuration* state, the DKF loads the configuration parameters from a *.json* file. A transition to the *startup* state happens if the configuration parameters are valid and during the state transition a connection to the SEE Federation execution is performed. Otherwise, if the configuration parameters are invalid a state transition to the *shutdown* state is performed. In this latter state, all the resources engaged by the SEE-DKF classes are de-allocated through the *dealloc resource* operation, and then the life cycle terminates. In the *startup* state, the connection status is checked. If the connection is not established the lifecycle ends with a transition to the *shutdown* state, otherwise, three operations are done: (i) *locateRTI*, the parameters of the specific HLA/RTI implementation (e.g. PITCH (Pitch Technologies, 2016), VT/MÄK (MÄK VR-Forces, 2016), PoRTIco (The PoRTIco project, 2016), CERTI (Certi Project, 2016)) are located and loaded; (ii) *setRTIParameters*, the parameters loaded in the previous operation are set up according to the configuration parameters defined in a *.json*; and (iii) *connectOnRTI*, a connection to the Federation execution is performed.

A transition to the *initialization* state is performed if the connection has been properly established; in this state, the SEE Federate could perform additional operation for exchanging initialization objects before entering the *running* state (and thus the time advancement loop: *waiting for a TAG* → *proactive state* → *make a TAR*), as an example, the Federate could publish and subscribe

some SEE information (e.g. *ReferenceFrames*, *InteractionClasses*, etc.). After that, the time management thread is activated and a transition to the *running* state is performed. The *running* state is composed of two sub-states operating in an AND-decomposition fashion. The *proactive behavior* sub-state deals with the pro-active part of the Federate behavior through three states: (i) *Waiting for TAG*: the DKF waits for the "TAG (Time Advance Grant) Callback" from the RTI. When the callback is received a transition to the *proactive* state is performed; (ii) *proactive state*: the "logical time" is updated, the pro-active behavior of the specific SEE Federate defined in the *proactive composite state* by the SEE working team is executed, and then a transition to the *make TAR request* state is performed; (iii) *make TAR request*: the DKF requests to the RTI the grant for the next "logical time". The *reactive behavior* sub-state deals with the re-active part of the behavior of the Federate: upon reception of RTI callbacks related to subscribed elements in the *Callback listener*, a transition to the *reactive* state is performed where the received information is handled through the execution of the reactive behavior of the specific SEE Federate defined in the *reactive composite state*. Note that, due to the AND decomposition in the *running* state, its child states are parallel states; this implies that the *proactive behavior* and *reactive behavior* are concurrently executed. As a consequence, the concurrency between the *reactive* and *proactive* tasks of a Federate has to be properly managed by the developer; indeed, the current version of the DKF/SEE-DKF does not provide specific mechanisms to handle these concurrent aspects and thus it relies on the standard synchronization mechanisms provided by the Java language and related SDK.

When the simulation ends a transition from the *running* state to the *shutdown* state is performed and, during the state transition, the HLA Federate is disconnected from the RTI.

4. DEVELOPING A FEDERATE: BEFORE AND AFTER

The previous section has shown that the DKF and its domain specific extensions can hide a significant amount of complexity related to the development of HLA Federates. Based on this, to demonstrate that the DKF can be used to simplify Federate development we now present a short case study. We focus on the Excavator agent-based simulation we developed in REPAST SIMPHONY as part of the SEE event in 2015 to show students how to create an agent-based simulation that can interoperate with other simulations in the lunar scenario. Students could explore how excavator "robots" could self-organize in the coordination of the extraction of lunar regolith materials and the degree to which *REPAST* could facilitate the study of these algorithms. In this short case study, in order to focus on distributed simulation issues we present a single agent with simplified input/output requirements. We therefore first introduce this simplified version of the excavator simulation and then discuss how this could be implemented without and with the DKF.

4.1 The Excavator Agent-based Simulation

REPAST SIMPHONY is a free and open-source agent-based simulation environment (Fortino et al, 2004; North, et al 2013). A *REPAST* agent-based simulation is created by using the *ContextBuilder* interface. In this class, the *environment* (i.e. the coordinate system that "places" the agents), the initial number of agents (and types/classes) that are located in the environment, and other basic settings are specified. The attributes and methods of each agent are specified in an agent's class. Each agent interacts with other agents and the environment via their methods. Time is managed in a *REPAST* simulation by the scheduler. A method can be annotated as being scheduled and will therefore include the frequency and priority that the method occurs. When a *REPAST* simulation runs, the simulation environment enters a cycle that calls the scheduler, the scheduler then runs the methods in priority order according to their frequency, and advances time at a specified time step until some terminating condition is met. In terms of distributed simulation, we need to be able to "plug" this simulation into a Federate, synchronize time advancement between the Federate (and therefore the Federation), and send and receive information to and from the simulation to represent the Excavator's interaction with other simulations.

A (simplified) single excavator agent explores its environment by coordinating with a UAV. The UAV simulation was developed by Liverpool University (Simulation Exploration Experience (SEE) project, 2016). The UAV slowly "flies" over the lunar surface detecting potentially interesting minerals. The UAV periodically broadcasts the results of its on-going survey to the excavator (in this case a single reading with the target coordinates), the excavator updates its local map and heads towards the target site. When the excavator reaches the site, it "mines" the mineral and adds it to its hopper that carries the excavated regolith. Once the hopper is full the excavator returns to its origin point and deposits the regolith material in a pile. The now empty excavator returns to where it left off and continues mining.

The agent-based simulation consists of three main classes: the *JExcavatorsBuilder*, *Excavator* and *Mineral*. *JExcavatorsBuilder* implements the *REPAST ContextBuilder* interface to create the simulation environment; a continuous space with a superimposed grid in which the excavator(s) move around. An Excavator has several internal variables that specify where it is currently located on the grid, its origin point, the amount of cargo it carries, and a map with the current target coordinate from the UAV. For the distributed simulation, the agent-based simulation needs to be able to receive the Cartesian coordinates of the target location from the UAV (UAVx, UAVy) and to send the Cartesian coordinates of its own current location to other Federates that need to coordinate with it (EXCx, EXCy) (including the visualization Federate that shows the entire scenario during the execution of the SEE distributed simulation).

When the REPAST agent-based simulation starts, the initialization of the environment happens in the *JExcavatorBuilder* class. At this stage, the grid is populated with an *Excavator* agent at location 0,0. At the first simulation time unit, the model calls the scheduler and executes all the scheduled methods with the modeler-defined frequency and priority configurations (if not defined by the modeler the schedule would follow the REPAST default configurations). In the case study, each agent has a *step()* method where all agent actions are implemented. This method is annotated as scheduled and therefore it is added to the scheduler. In this example, the Excavator's *step()* method is called. The first action in the step method reflects the communication with the UAV by receiving the next location (if any) and updating the map by calling *updateMap()*.

The excavator then checks to see if it is full and needs to return to origin. If it does, it moves towards the origin. If not, it moves towards the target location. Arriving at the origin point it will unload its cargo and then move towards the next target location. Arriving at a target location it will "mine" for a period of time and update its load. In this simplified scenario the agent therefore needs to receive a target location from the UAV and to send its current location to the other simulations (Federates) in the distributed simulation. The scheduled *step*() method in REPAST is reported in Appendix A.1.

4.2 Implementing a Federate without using the SEE-DKF

To create a Federate of this agent-based simulation we first identify the incoming and outgoing communication of the Excavator Federate, i.e. the information that the Federate will receive and send from/to other Federates. This is specified in the Federate Object Model (FOM).
In the "normal" Federate implementation, the middleware was developed using poRTIco RTI implementation (The PoRTIco project, 2016). Generally, to create an HLA Federate from scratch, two classes need to be added to a model: (i) a concrete Federate class, here referred as Federate, that manages the life-cycle of the Federate and defines the behavior of the model to be simulated. This class uses the mechanisms provided by the RTI Ambassador for sending information to the other Federates through the RTI (IEEE Std. 1516-2010); and (ii) its Federate Ambassador class by implementing the *NullFederateAmbassador* interface, which defines a set of methods that define how the RTI sends information to the Federate in response to the changes in the state of the Federation execution 0. A FOM XML schema needs also to be created. For our Excavator implementation, these two classes and the FOM file were based on the examples of the Federate and Federate Ambassador classes and modular FOMs that come with the poRTIco RTI (The PoRTIco project, 2016). The examples were helpful but assumed a certain level of HLA expertise. Learning how to implement a distributed simulation was very demanding. We implemented two HLA functionalities, data exchange and time synchronization.

The HLA specification supports several forms of communication. For example, every Object and its attributes and every Interaction and its parameters can be published by a Federate. Other Federates subscribe to these. Both publish and subscribe mechanisms are declared manually in the *Federate* class. Data exchange in poRTIco is achieved by calling *ObjectClassHandle* and *InteractionClassHandle* for every instance that requires data exchange. In the *Federate* class, handle variables for all Object attributes and all Interaction parameters that need to be communicated must be declared. To do this, the modeler must create these handle variables. Then these handle variables must be added to the respective *Collections*, different for Objects and Interactions. These *Collections* must be then registered for updates, publish and/or subscribe. The method that does that is the *publishAndSubscribe()* method in the *Federate* class (see Appendix A.2).

The final step is to do the actual updates. This involves updating the handle variables values and encode them. An example of the update method for updating the Excavator Cartesian coordinates is shown in Appendix A.3.

The *Federate Ambassador* is responsible for receiving attribute values and decoding them (the Object attributes that Federate has subscribed). An example code for receiving updates from the UAV Object is shown in Appendix A.4 along with the implemented decode method.

As mentioned earlier, together with the *Federate* and its *Federate Ambassador* classes, the FOM XML schema needs to be created too. A portion of the FOM module is reported in Appendix A.5.

Generally, the FOM in both the "normal" and DKF implementations is the same for specifying the publish/subscribe *ObjectClasses* and *InteractionClasses*. However, in the "normal" implementation all data types must be explicitly stated in the FOM. If the Federation exchanges many different data types, this part of FOM can be substantial (see Appendix A.6).

Time synchronization is achieved by using HLA time services and REPAST scheduler. REPAST is a time-driven simulator, therefore the time strategy that is implemented in the Excavator Federate is based on Time Advance Requests (TARs). The Excavator Federate after updating EXCx and EXCy attributes requests time advancement through the RTI Ambassador. Then, the Federate Ambassador, after receiving the updated UAVx and UAVy attributes from the UAV Federate, grants time advancement to the Excavator Federate using the Time Advance Grant (TAG) method. The snapshot code in Appendix A.7 shows the above described methods.

An instance of the Excavator Federate, and subsequently an instance of Federate Ambassador too, is created when initializing the simulation in the REPAST Context Builder and is added in the same context as the agents. In this implementation, the update attributes method of the Federate must be modified to reflect the subscribed attributes. This method then can be called manually

from the scheduled methods in the agent-based simulation (i.e. an update attribute method is called from the *moveExcavator()* method within the scheduled *step()* method in the Excavator class).

4.3 The Development Process based on SEE-DKF

As noted above, the SEE-DKF was developed as the "lunar" domain extension for the DKF. Rather than trying to follow examples from various RTI implementation, the DKF has a development process. This has four main steps:

1. Build a *model* of the Federate that specifies: the *objects* that the Federate manages (as specified in the FOM), the *attributes* of these objects and the *coders* to handle such attributes. It is possible to use the basic coder set provided in the SEE-DKF or to implement new coders based on the SEE-DKF classes;
2. Build a concrete Federate that specifies the *behaviour* of the *model* defined at (1). It is required to extend the *SEEAbstractFederate* abstract class provided by the SEE-DKF and implement three methods according to the Federate life-cycle that is provided and completely managed by the SEE-DKF (see Figure 5), specifically: (a) a method for initializing operations before entering the "running state" (a *configureAndStart()* method); (b) a method for specifying the active part of the behavior of the Federate (*doAction()* method) executed between a TAR and a TAG; and (c) a method (*update()* method) that specifies the re-active part of the behavior of the Federate, i.e. how to handle the RTI callbacks about the interactions/objects that the Federate has subscribed;
3. Implement the Federate Ambassador. This step requires extending the *SEEAbstractFederateAmbassador*; typically, since no specific implementation is required, the child class has only to define its constructor which in turn calls the parent one: all the typical Ambassador's features are provided and managed by the SEE-DKF;
4. Implement a *main* class so as to instantiate and run the developed Federate.

In the following, after presenting the reference simulation scenario, the above sketched process will be exemplified with respect to the development of a Federate in the context of the SEE Project (Simulation Exploration Experience (SEE) project, 2016).

4.4 Using the DKF to Develop the Excavator Federate

The above description of the simple excavator focuses on a single excavator agent. The mining operation may be also of interest to other simulations (e.g. an astronaut who takes away mined materials for processing). To create a Federate based on the above introduced agent-based simulation, the SEE-DKF main steps have been followed.

In step (1) a FOM that describes the input and output of the simulation was exploited. In this case the FOM represents the single Excavator object with UAVx, UAVy, EXCx and EXCy as noted above. All are *HLAinteger32BE* datatype. To begin the creation of the Federate, the Excavator class has been annotated to match the FOM as follows (see Section 3.2):

```
1.  @ObjectClass(name = "PhysicalEntity.Excavator")
2.  public class Excavator { …
```

To create the I/O from the simulation to the rest of the Federation, the Excavator class was augmented with attributes and coders. For example, to enable the sharing of the X, Y coordinates of the excavator the following attributes and coders have been added to the declarations:

```
1.  @Attribute(name = "EXCx", coder = HLAinteger32BECoder.class)
2.  private Integer EXCx;
3.  @Attribute(name = "EXCy", coder = HLAinteger32BECoder.class)
4.  private Integer EXCy;
```

At the end of the *step()* method described above, the two calls

```
1.  setEXCx(getPointX());
2.  setEXCy(getPointY());
```

have been added to update the current position of the excavator. Similar attributes and coders for the other attributes described in the FOM have been added.

In step (2), the *SEEAbstractFederate* class has been extended to create the *ExcavatorFederate* class. Within the *ExcavatorFederate* class the *configureAndStart()* method remained unchanged (i.e. it reaches the JSON config file and starts the Federation). The *doAction()* method is shown below.

```
1. protected void doAction()
2.   {
3.     for (Object obj : RunState.getInstance().getMasterContext())
4.       {
5.        if(obj instanceof Excavator) // update the excavator on RTI
6.           ((Excavator) obj).step();
```

```
7.          super.updateElement(obj);
8.        }
9.    }
```

This method advances the agent-based simulation by first obtaining the current state (context) of the simulation, finding all agents (objects) and then "manually" running the *step()* method in the agents. In this example, the single excavator agent's *step()* method is executed. It then calls *updateElement(obj)* to output the new state of the excavator Federate's attributes.

Step (3) simply extended the *SEEAbstractFederateAmbassador* class with the *ExcavatorFederateAmbassador*. Step (4) was unnecessary, as the simulation had already been developed. The only addition to these steps was that of the *ExcavatorFederate* and *ExcavatorFederateAmbassador* to the context (*JExcavatorsBuilder*) to include them in the scope of the agent-based simulation. The overall class diagram is shown in Figure 6.

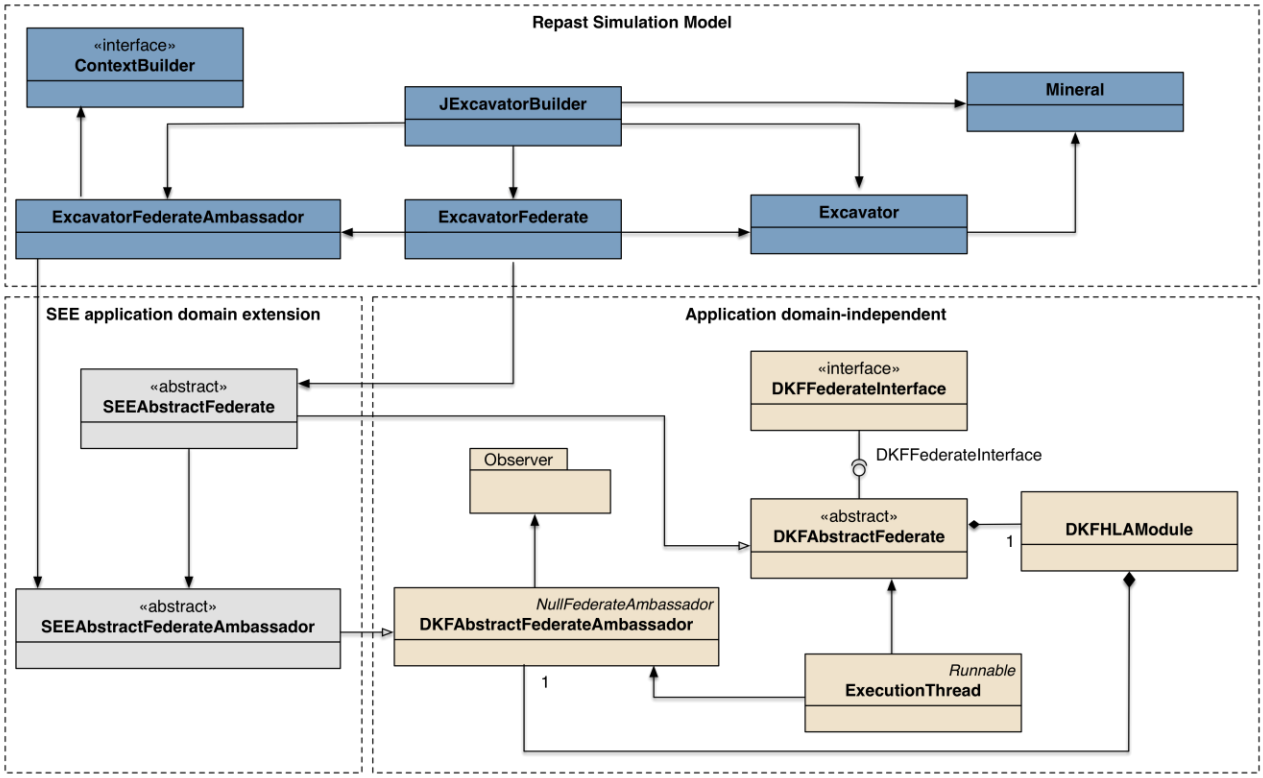

Figure 6. The architecture of the Excavator Federate.

TABLE 6 summarizes the main differences in implementing HLA Federations with and without the DKF. In SEE 2015, under guidance from the Brunel team, an undergraduate Computer Science student created the "distributed" side of the Excavator agent moderately quickly. This left more time for him to concentrate on the "simulation" aspects of the Excavator and its interactions with other simulations in the SEE event.

Table 6: Comparison in building DKF and no-DKF based Federate.

	Without DKF	***With DFK***
Object / attribute declaration	Manually declared in Federate Class	Annotated in Object Class
Interaction / parameter declaration	Manually declared in Federate Class	Annotated in Interaction Class
Attribute / parameter update	Manually for each element	Collectively for each Object/Interaction
Data Types Coders	Explicitly stated in FOM	Using DKF coder package
Time advance	Scheduled and managed in Repast	Managed by HLA/RTI via DKF

5. DKF QUANTITATIVE ASSESSMENT

This section presents a quantitative analysis of the quality of the code produced by using the DKF and aims at highlighting the benefits provided by its exploitation in the SEE project.

Software complexity is a primary topic in Software Engineering and has involved many researchers over the years. To analyze the quality of a software, it is necessary to measure the software source code in quantized form. Software metrics is one of the most traditional and effective way to measure the software system and they are related to various constructs like class, coupling, cohesion and inheritance. To evaluate the complexity of the source code of an SEE-DKF based HLA Federate, five standard metrics, which are proposed by various researchers, have been considered (Yu and Zhou, 2010).

SLOC (Source Line of Code) is the most widely used metric for measuring the size of a software program. It is used to count the number of any line that is not a comment or blank line irrespective of the number of statements per line (also called *executable statements*). *SLOC* is easy to understand, fast to count, independent of the program language and it is a good metric to measure and evaluate the quantitative characteristics of a source code via the physics length. Typically, a method should be broken up if it has more than 50 lines of code; whereas a class should be split up and its functionalities delegates to other classes or sub-classes if it has over 750 lines of code. In this way, it is possible to increase both readability and maintainability of the software (Yu and Zhou, 2010).

CCM (Cyclomatic Complexity Metric) is one of the most commonly used metric in many commercial and non-commercial tools for code complexity measurement. The CCM is based on graph theory and measures the complexity of a software module by analyzing its control flow structure. In particular, the control flow structure is represented as a graph *G(V, E)*, in which nodes *(V)* are used to represent decision or control statements; whereas edges *(E)* represent the control paths which define the program flow. The value of the CCM is the number of linearly independent paths and therefore, the minimum count of paths that should be tested, because any path can be expressed as a linear combination of some linearly independent paths. The CCM value gets an assessment of the complexity and indirectly of the maintainability of a software (see Table 7).

HCM (Halstead Complexity Metric) measures the complexity of a software directly from source code analyzing its operators and operands. The operators are symbols used in expressions to specify the manipulations to be performed, whereas the operands are the basic logic unit to be operated. In particular, HCM measures the logic volume of a software by using four numeric values: (i) the number of *non-repetitive operators* (*n1*); (ii) the number of *non-repetitive operands* (*n2*); (iii) the total *number of operators* (*N1*); and, (iv) the total *number of operands* (*N2*). This metric represents a strong indicator of code complexity and it is often used as maintenance metric and to evaluate development risk: higher values imply lower maintainability (Yu and Zhou, 2010).

NF (number of function) represents the total number of functionalities that are present in a software. This metric can be used to estimate the limits of code readability. In this context, a function that has a large number of code lines (e.g. greater than 800) should be decomposed, thus ensuring better clarity of individual code segments.

Table 7: Cyclomatic complexity value ranges.

Cyclomatic Complexity	*Code Evaluation*	*Risk Evaluation*
1 - 10	The software code is considered simple and easy to understand and test.	No much risk
11 - 20	The software code is quite complex but still be comprehensible; however testing becomes more difficult due to the greater number of possible branches.	Moderate risk
21 - 50	The software code is complex and has got a very large number of potential execution paths that and can only be fully understandable and tested with difficulty and effort.	High risk
> 50	The software code is extremely complex and unmaintainable.	Very high risk

Finally, *NC (number of classes)* represents the number of concrete, abstract and interface classes. It provides an indicator of the extensibility of the software. Typically, the lower are the values of these metrics the lower is the complexity of the source code and thus higher should be the code compactness, readability and reliability (Basili and Perricone,1984; Yu and Zhou, 2010).

These six metrics are evaluated by considering the source codes of the *UNICOM* Federate, another simulation in the SEE event that provides communication services to the other entities populating the Moon base scenario and is substantially more complex than the Excavator agent (Falcone et al, 2014).

Table 8: METRICS AT PACKAGE LEVEL.

Metric		*UNICOM Federate*	
		SEE-DKF	***Pitch Developer Studio***
NC		17	72
NF		94	784
SLOC		744	6186
CCM (average)		1,20	1,63
HCM	Number of distinct operators (n_1)	22	38
	Number of distinct operands (n_2)	312	1454
	Total number of operators (N_1)	1337	4150
	Total number of operands (N_2)	513	13217
	Software length (N)	1850	17367
	Software vocabulary (n)	334	1492
	Volume (V)	$1,584*10^4$	$1,831*10^5$
	Level (L)	0,1495	0,4784
	Difficulty (D)	47,13	172,71
	Programming Effort (E)	$7,469*10^5$	$3,162*10^7$
	Error Estimate (B)	5,28	61,03
	Programming Time (T)	$4,149*10^4$	$1,756*10^6$

More in detail, one source code is based on the SEE-DKF whereas the other one is that produced by the *Pitch Developer Studio* (Möller, 2013), which is a high quality IDE for HLA programming. The metrics have been calculated by using the *Google CodePro AnalytiX* tool, which is an application, developed by Google Inc., that allows developers to perform code measurement and comparison with user-defined programming standards and that is used by several large organizations, ranging from aerospace/defense to automotive/transport companies, to control their programming process.

Although the *DKF* framework, and its domain dependent extension SEE-DKF, does not cover all the IEEE 1516-2010 functionalities that are instead covered by the *Pitch Developer Studio*, the results reported in Table 8 show that the source code of an HLA Federate created by using the *DKF*/*SEE-DKF* is easy to manage and maintain even when compared to the same code produced by the *Pitch Developer Studio*.

Moreover, all the classes produced by using the SEE–DKF have *CCN* value less than twenty (see Table 8); as a consequence, these classes are easy to manage/extend by programmers (see (Yu and Zhou, 2010) for a discussion).

6. CONCLUSION

HLA is undoubtedly one of the most mature and popular standard for distributed simulation. Due to its capabilities to enable the interoperability and reusability of distributed simulation components, it is increasingly exploited in a great variety of applications in both military and civil domains. However, the development of full-fledged simulation models, based on HLA, is still a challenging task. In this context, the paper has proposed an effective solution to enable the agile development of HLA-based simulations based on the *HLA Development Kit*, a general-purpose, domain independent toolkit that provides a software framework (the *DKF*), with related documentation, user guide and reference examples. The effectiveness of the *DKF* has been exemplified in the context of the Simulation Exploration Experience (SEE), an international project organized by SISO and led by NASA that involves several U.S. and European Institutions in the distributed simulation of a "Moonbase". In terms of developing educational resources for HLA development, the *DKF* presents a solid foundation for future expansion. The SEE event is exciting in that students can create a wide variety of simulations and take part in an international project.

The *SEE-DKF* is therefore an example of how the *DKF* can be extended to be domain specific. Future work on *DKF* includes the further development, testing and evaluation of the *DKF* and its domain specific extensions so as to also provide interesting education and research resources to easily develop distributed simulations in various application domains.

ACKNOWLEDGMENT

The authors would like to thank Edwin Z. Crues (NASA JCS) for his precious advice and suggestions in the development of the *HLA Development Kit*. A special note of thanks goes also to all the NASA staff involved in the SEE Project: Priscilla Elfrey, Stephen Paglialonga, Michael Conroy, Dan Dexter, Daniel Oneil, to Björn Möller (PITCH Technologies), and to all the members of SEE teams.

REFERENCES

Anagnostou A, Chaudhry N R, Falcone A, Garro A, Salah O and Taylor S J E (2015a). Easing the development of HLA Federates: the HLA Development Kit and its exploitation in the SEE Project. In: *Proceedings of the 19th IEEE/ACM International Symposium on Distributed Simulation and RealTime Applications (ACM/IEEE DS-RT 2015)*, IEEE Computer Society: Chengdu (CN), pp. 50-57.

Anagnostou A, Chaudhry N R, Falcone A, Garro A, Salah O, and Taylor S J E (2015b). A Prototype HLA Development Kit: Results from the 2015 Simulation Exploration Experience. In: *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (ACM SIGSIM PADS 2015)*, London (UK), pp. 45-46.

Banks J, Carson J S, Nelson B L, and Nicol D M (2009). Discrete-Event System Simulation, 5th Ed., Prentice Hall.

Basili V R and Perricone B T (1984). Software errors and complexity: an empirical investigation. Communications of the ACM 27(1): 42-52.

Bocciarelli P, D'Ambrogio A, Falcone A and Garro A (2015). A model-driven approach to enable the distributed simulation of complex systems. In: *Proceedings of the 6th International Conference Complex Systems Design & Management (CSD&M)*, Springer International Publishing: Paris (FR), pp 171-183.

Certi Project (2016). The simulation toolkit home page. http://savannah.nongnu.org/projects/certi.

Falcone A, Garro A, Longo F, and Spadafora F (2014). Simulation Exploration Experience: A Communication System and a 3D Real Time Visualization for a Moon base simulated scenario. In: *Proceedings of the 18th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (ACM/IEEE DS-RT)*, IEEE Computer Society: Toulouse (FR), pp 113-120.

Falcone A and Garro A (2015). On the integration of HLA and FMI for supporting interoperability and reusability in distributed simulation. In: *Proceedings of the Symposium on Theory of Modeling and Simulation - DEVS Integrative M and S Symposium, DEVS 2015, Part of the 2015 Spring Simulation Multi-Conference (SpringSim 2015)*, SCS Press: Alexandria (VA, USA), pp 9-16.

Falcone A and Garro A (2016). The SEE HLA Starter Kit: enabling the rapid prototyping of HLA-based simulations for space exploration. In: *Proceedings of the Simulation for Planetary Space Exploration (SpringSim-SPACE), Part of the 2016 Spring Simulation Multi-Conference (SpringSim 2016)*, SCS Press: Pasadena (CA, USA).

Fortino G, Garro A and Russo W (2004). From Modeling to Simulation of Multi-Agent Systems: an integrated approach and a case study. *In Proceedings of the 2nd Multiagent System Technologies*, Springer Berlin Heidelberg: Erfurt (D), pp 213-227.

Fujimoto R M (2010). Parallel and distributed simulation systems, John Wiley & Sons.

Fujimoto R M, Malik A W, and Park A (2010). Parallel and distributed simulation in the cloud. SCS M&S Magazine, Vol. 3, pp 1-10.

IEEE Std. 1516-2010. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA): 1516-2010 (Framework and Rules); 1516.1-2010 (Federate Interface Specification); 1516.2-2010 (Object Model Template (OMT) Specification).

MÄK VR-Forces (2016). MÄKhome page. http://www.mak.com/.

Möller B (2013). The HLA tutorial v1.0, Pitch Technologies, Sweden.

North M J, Collier N T, Ozik J, Tatara E R, Macal C M, Bragen M and Sydelko P, (2013). Complex Adaptive Systems Modeling with Repast Simphony. 1(1): 1-26, Springer.

Pitch Technologies (2016). The simulation toolkit home page. http://www.pitch.se.

Simulation Exploration Experience (SEE) project (2016). Simulation Exploration Experience home page. http://www.exploresim.com/.

Taylor S J E, Turner S J, Janahan T, Tan G and Ladbrook J (2002). GRIDS-SCF: An infrastructure for distributed supply chain simulation. Simulation, 78(5), 312-320.

Taylor S J E, Fishwick P, Fujimoto R, Page E, Urhmacher A and Wainer G (2012). Panel on Modeling & Simulation Grand Challenges. In: *Proceedings of the 2012 Winter Simulation Conference (WSC)*, Association for Computing Machinery Press: New York (USA), pp 1-15.

Taylor S J E, Turner S J, Mustafee N, and Strassburger S (2012). Bridging the gap: a standards-based approach to OR/MS distributed simulation. ACM Transactions on Modeling and Computer Simulation (TOMACS), 22(4), art. 18.

Taylor S J E, Revagar N, Chambers J, Yero M, Anagnostou A, Nouman A, and Chaudhry N R (2014). Simulation Exploration Experience: A Distributed Hybrid Simulation of a Lunar Mining Operation. *In Proceedings of the 18th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (ACM/IEEE DS-RT)*, IEEE Computer Society: Toulouse (FR), pp 107-112.

The Apache Commons Mathematics Library (2016). Apache Commons Mathematics home page. https://commons.apache.org/proper/commons-math/.

The Forwardsim HLA Toolbox for MATLAB (2016). Forwardsim home page. http://www.forwardsim.com/products/hla-toolbox/.

The HLA Development Kit project (2016). HLA Development Kit home page. https://smash-lab.github.io/HLA-Development-Kit/.

The International Celestial Reference Frames (2016). International Celestial Reference Frames home page. http://www.iers.org.

The PoRTIco project (2016). PoRTIco home page. http://www.porticoproject.org/.

Van Spengen J W (2010). FEDEF: A High Level Architecture Federate Development Framework. No. DRDC-ATLANTIC-TM-2010-105. Defence Research and Development - Atlantic Dartmouth (CANADA).

Villimann O (1999). HLA Framework. Danish Maritime Institute. CTO Project, Documentation.

Xie Y, Teo Y M, Cai W and Turner S J (2015). Towards grid-wide modeling and simulation.

XQuery 1.0 and XPath 2.0 Functions and Operators (2016). XQuery 1.0 and XPath 2.0 home page. http://www.w3.org/TR/xpath-functions/.

Yu S and Zhou S (2010). A survey on metric of software complexity. *In Proceedings of the 2nd IEEE International Conference on Information Management and Engineering (ICIME)*, IEEE Computer Society: Chengdu (CN), pp 352-356.

APPENDIX

This appendix reports the code that has been developed for creating an HLA Federate starting from a REPAST-based agent without using the functionalities provided by the SEE-DKF.

A.1 The Simulation step of the Excavator agent in REPAST

```
1. @ScheduledMethod(start=1, interval=1)
```

```
2. public void step(){
3.     updateMap(ExcavatorMap, UAVx, UAVy); //This updates its map with the next target from the UAV
4.     checkCargoLimit(returnOrigin);
5.     moveExcavator(grid.getLocation(this), returnOrigin, ExcavatorMap, EXCx, EXCy); //This moves the Excavator X,Y
6.   }
```

A.2 The publishAndSubscribe() method of the Excavator Federate

```
1. private void publishAndSubscribe() throws RTIexception
2.    {  // get all the handle information for the attributes of ObjectRoot.Excavator
3.       this.classHandle = rtiamb.getObjectClassHandle("HLAobjectRoot.Excavator");
4.       this.exHandle = rtiamb.getAttributeHandle(classHandle, "ex");
5.       this.eyHandle = rtiamb.getAttributeHandle(classHandle, "ey");
6.       this.uxHandle = rtiamb.getAttributeHandle(classHandle, "ux");
7.       this.uyHandle = rtiamb.getAttributeHandle(classHandle, "uy");
8.       AttributeHandleSet attributes = rtiamb.getAttributeHandleSetFactory().create();
9.       attributes.add(exHandle);
10.      attributes.add(eyHandle);
11.      attributes.add(uxHandle);
12.      attributes.add(uyHandle);
13.      rtiamb.publishObjectClassAttributes(classHandle, attributes);
14.      rtiamb.subscribeObjectClassAttributes(classHandle, attributes);
15.      ...
16.   }
```

A.3 The updateAttributeValues() method of the Exavator Federate

The code reported below updates the handle variables with the encoded Excavator's coordinates, put them in the attributes *Collection*, and send them to the RTI with a timestamp.

```
1. public void updateAttributeValues(int EXCx, int EXCy) throws RTIexception
2.    {
3.        AttributeHandleValueMap attributes = rtiamb.getAttributeHandleValueMapFactory().create(2);
4.        // 2 is the initial capacity of the newly created map
5.
6.        HLAinteger32BE exValue = encoderFactory.createHLAinteger32BE(EXCx);
7.        HLAinteger32BE eyValue = encoderFactory.createHLAinteger32BE(EXCy);
8.
9.        attributes.put(exHandle, exValue.toByteArray());
10.       attributes.put(eyHandle, eyValue.toByteArray());
11.
12.       HLAfloat64Time time = timeFactory.makeTime(fedamb.federateTime+fedamb.federateLookahead);
13.       rtiamb.updateAttributeValues(objectHandle, attributes, generateTag(), time);
14.   }
```

A.4 Receiving and decoding updates by the Excavator Federate Ambassador

The following code is defined in the Excavator Federate Ambassador for receiving updates from the UAV Object:

```
1. for(AttributeHandle attributeHandle : theAttributes.keySet())
2. {
3.        // uxHandle and uyHandle hold the UAV Cartesian coordinates and are updated in the UAV Federate
4.        if(attributeHandle.equals(federate.uxHandle)){
5.              UAVx=decodeInt(theAttributes.get(attributeHandle));
6.          }
7.        if(attributeHandle.equals(federate.uyHandle)){
8.              UAVy=decodeInt(theAttributes.get(attributeHandle));
9.          }
10. }
```

The decoder for the above received data is reported below:

```
1.  private int decodeInt(byte[] bytes)
2.  {
3.       HLAinteger32BE value = federate.encoderFactory.createHLAinteger32BE();
4.       try
5.         {
6.           value.decode(bytes);
7.         }
8.       catch(DecoderException de)
```

```
9.         {
10.          log("Decoder Exception: "+de.getMessage());
11.        }
12.      return value.getValue();
13. }
```

A.5 The FOM module of the Excavator Federate

The snapshot of code that describes the published *EXCx* coordinate, which is an attribute of the *Excavator* class, is shown below:

```
1. <objects>
2.   <objectClass>
3.      <name>HLAobjectRoot</name>
4.      <sharing>Neither</sharing>
5.         <objectClass>
6.            <name>Excavator</name>
7.            <sharing>PublishSubscribe</sharing>
8.            <semantics>NA</semantics>
9.               <attribute>
10.                 <name>ex</name>
11.                 <dataType>HLAinteger32BE</dataType>
12.                 <updateType>Conditional</updateType>
13.                 <updateCondition>NA</updateCondition>
14.                 <ownership>NoTransfer</ownership>
15.                 <sharing>PublishSubscribe</sharing>
16.                 <dimensions>NA</dimensions>
17.                 <transportation>HLAreliable</transportation>
18.                 <order>TimeStamp</order>
19.                 <semantics>NA</semantics>
20.              </attribute>
21.              ...
22.  </objectClass>
23. </objects>
```

A.6 DataTypes in the FOM module of the Excavator Federate

In the Excavator FOM only an Integer data type has been defined. The snapshot of code in XML is shown below:

```
1. <dataTypes>
2.   <basicDataRepresentations>
3.     <basicData>
4.       <name>HLAinteger32BE</name>
5.       <size>32</size>
6.       <interpretation>Integer in the range [-2^15, 2^15 - 1] </interpretation>
7.       <endian>Big</endian>
8.       <encoding>32-bit two's complement signed integer. The most significant bit contains the sign</encoding>
9.     </basicData>
10.     ...
11.  </basicDataRepresentations>
12. </dataTypes>
```

A.7 Time synchronization

The method below implements TAR requests and belongs to the Excavator Federate class. This method is annotated as scheduled and therefore it is added to the REPAST scheduler.

```
1.  @ScheduledMethod(start=1,interval=1, priority = ScheduleParameters.LAST_PRIORITY)
2.  public void advanceTime() throws RTIexception
3.    {
4.      fedamb.isAdvancing = true;
5.      HLAfloat64Time time = timeFactory.makeTime(fedamb.federateTime + timestep);
6.      rtiamb.timeAdvanceRequest(time);
7.
8.     while(fedamb.isAdvancing)
9.       {
10.          rtiamb.evokeMultipleCallbacks(0.1, 0.2);
11.       }
12.   }
```

The method below handles TAGs and belongs to the Federate Ambassador class.

```
1.  @Override
2.  public void timeAdvanceGrant(LogicalTime time)
3.    {
4.       this.federateTime = ((HLAfloat64Time)time).getValue();
5.       this.isAdvancing = false;
6.    }
```