Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

AN INTRODUCTION TO DEVELOPING FEDERATIONS WITH THE HIGH LEVEL ARCHITECTURE (HLA)

Alberto Falcone
Alfredo Garro

Department of Informatics, Modeling, Electronics and Systems Engineering (DIMES)
University of Calabria
Via P. Bucci, 41/C
Rende, 87036, ITALY

Anastasia Anagnostou
Simon J.E. Taylor

Modelling & Simulation Group, Department of Computer Science
Brunel University London
Kingston Ln
London Uxbridge, UB8 3PH, UK

ABSTRACT

The IEEE 1516-2010 - High Level Architecture (HLA) for distributed simulation is growing in a variety of application domains due to its capabilities to enable the interoperability and reusability of distributed simulation components. However, the development of simulation models based on the HLA standard remains a challenging task that requires a considerable effort in terms of both time and cost. This paper provides an introduction tutorial on developing HLA-based simulations using the HLA Development Kit (DKF) framework. The tutorial guides developers through the necessary steps for defining and creating an HLA-based simulation, and explains how the HLA elements can be easily managed by using the DKF's services. The effectiveness of the DKF is proven by its concrete exploitation in the context of the Simulation Exploration Experience (SEE), a project led by NASA and which involves as partners several U.S. and European Institutions.

1 INTRODUCTION

In many areas of science and engineering, Modeling and Simulation (M&S) represents a pillar supporting the acquisition of knowledge so as to understand, predict and optimize the behavior of systems on which to perform experiments and theoretical analyses (Bocciarelli et al. 2014, Falcone et al. 2017). M&S represents an indispensable problem-solving methodology for the solution of many real-world problems. It can be considered as the ability to artificially reproduce, essentially by using the computer, the characteristics of a real process or system so as to describe, analyze and predict its behavior in presence of events subsequent to the imposition of conditions by the user (Falcone and Garro 2015) It is a very powerful analysis technique, used in many scientific and technological domains in which it is difficult or impossible to physically reproduce in the laboratory the whole system and the conditions to be analyzed (Fujimoto 2000).

By using M&S methods, tools and techniques, it is possible to reproduce the structure and behavior of systems over the time so as to observe and analyze them (Banks 1998). The use of M&S techniques offer many advantages, such as the possibility to study the behavior of a system without physically building it, and the evaluation and comparison of different design choices, policies, and operating procedures through experiments in a controlled (virtual) environment (Falcone et al 2016, Fujimoto 2000). Despite the above sketched advantages, M&S has important disadvantages many of those related to the significant efforts required for producing a full-fledged simulation model and analyzing simulation results. Moreover, it is often hard to *reuse* already available simulation models; indeed, there is a lack of mechanisms to make *interoperable* simulation models built on different simulation platforms and a scarce support to enable their *execution on distributed infrastructures*. Due to the increasing complexity of simulation systems, a Distributed Simulation (DS) approach has been introduced (Fujimoto 2000).

Distributed simulation refers to technologies that enable the execution of a simulation program on distributed computer systems containing multiple components such as processors that are geographically distributed and connected by a communication network. Initial research on distributed simulation has been conducted in the military domain where the main objective was on how to achieve model reuse via interoperation of heterogeneous simulation components.

The U.S. Department of Defence (DoD) made huge investments in the distributed simulation domain and played a key role in the developing of standards to facilitate interoperability of distributed simulation modules over a computer network. IEEE 1516-2010 - High Level Architecture is a well-known and accepted standard that provides a distributed infrastructure in which each simulation unit runs on an independent computer (in general, geographically distributed) and communicates with the others in a common simulation scenario (IEEE Std. 1516-2010). HLA was developed by the DoD Modeling and Simulation Coordination Office (M&S CO) in the period 1995-1996 (Modeling and Simulation Coordination Office 2017), as a general architecture to facilitate the integration of distributed simulation models within a common simulation environment. Although it was initially developed for purely military applications, it has been widely 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 interact among them by using a *Run-Time Infrastructure (RTI)* that provides standard protocols and services to manage the communications and data exchange among *Federates*. Each *Federation* has a *Federation Object Model (FOM)* that is created in accordance with the *Object Model Template (OMT)* defined by the standard (IEEE Std. 1516-2010, Kuhl et al. 1999). A FOM contains specifications of *Object classes* (objects are instances – or entities – of object classes that have attributes that can be updated), *Interaction classes* (a message sent among objects that has parameters) and *Data types* (the technical specifications and semantics of attributes and parameters).

Building complex and large distributed simulations, especially those based on the HLA standard, is usually a challenging task and requires considerable development experience in distributed systems, simulation, middleware, and software programming (Bocciarelli et al. 2015, Möller et al. 2016). The main issues are that the design and development of HLA Federates are generally 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 (Falcone et al. 2017, Möller et al. 2016). Moreover, developers have to spend a considerable effort to handle common HLA functionalities, such as the control of the simulation time; the synchronization process among Federates; the publish, subscribe and update of *Object class* and *Interaction class* elements along with related coders and decoders; the management of RTI callbacks and associated exceptions. As a result, they cannot fully focus on the specific aspects of their own simulation components (the HLA Federates).

Over the years, several research efforts focused their attention on the creation of software solutions to easing the development of HLA-based simulations mainly aiming at providing an integrated toolchain for creating and simulating complex systems. For MATLAB/Simulink users, different packages and toolboxes are available for implementing HLA simulators such as the F*orwardsim HLA Toolbox for MATLAB* (Forwardsim 2017), which provides a user interface that allows developers to fully design and customize their HLA Federates. 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* (Pitch Technologies 2017). This software allows programmers to add HLA or DIS functionalities quickly and easily to their simulators through a module that generates C++ or Java code starting from the structure of the HLA Federate. Other HLA tool that allows developer to easily build and test an HLA Federate is the *HlaListener* (HLAListener 2017). It was developed by the University of Alberta (Canada) and provides a simple Graphic User Interface (GUI) with all the common IEEE 1516-2010 functionalities such as:

connect/disconnect to/from an RTI, send/receive *InteractionClass* and *ObjectClass* updates and the definition of Synchronization Points. A domain-specific HLA software framework was created by the Danish Maritime Institute (DMI) with the aim to simplify the development of real-time simulators in the military domain (Villimann 1999). It defines a range of real-time simulation concepts to support functionalities available at DMI with an HLA environment. *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). In this context, the tutorial presents the *HLA Development Kit software Framework (DKF)* along with a step-by-step example on how to design, develop and run a complete HLA distributed simulation by using its functionalities. The *DKF* is a general-purpose, domain-independent framework, fully implemented in the Java language and released under the open source policy Lesser GNU Public License (LGPL), which facilitates the development of HLA Federates (Falcone et al. 2017, Falcone et al. 2015, IEEE Std. 1516-2010). The *DKF* allows developers to focus on the specific aspects of their own Federates rather than dealing with the common HLA aspects such as the management of the simulation time; the connection/disconnection on/from the HLA RTI; the publish, subscribe and updating of *ObjectClass* and *InteractionClass* elements (Falcone et al. 2015, Garro et al. 2015, IEEE Std. 1516-2010). The DKF has been designed and developed in the context of the research activities carried out within the SMASH-Lab (System Modeling And Simulation Hub - Laboratory) of the University of Calabria (Italy) working in cooperation with the Software, Robotics, and Simulation Division (ER) of the NASA's Lyndon B. Johnson Space Center (JSC) in Houston (Texas, USA) (National Aeronautics and Space Administration 2017). It has been successfully experimented in the Simulation Exploration Experience (SEE) project since the 2015 edition (Simulation Exploration Experience 2017). The SEE project, which has been organized, since 2011, by SISO in collaboration with NASA and other research and industrial partners, aims at providing 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. In the 2017 edition, the Universities of Calabria (Italy), Bordeaux (France), Brunel (London, UK), Genoa (Italy), the Faculdade de Engenharia de Sorocaba, FACENS (Brazil) and the team of the Bulgarian Modeling and Simulation Association (Bulgaria) developed their SEE-Federates by using the Kit (Falcone and Garro 2016b, Falcone et al. 2015, Taylor et al. 2014).

The rest of the tutorial is organized as follows. Section 2 presents the HLA Development Kit framework with particular focus on the architecture, main services provided and on a domain-specific extension, called *SEE HLA Starter Kit (SEE-SKF)*, defined in the context of the SEE project. In Section 3, a step-by-step example on how to develop a HLA Federate from scratch by using the functionalities provided by the *DKF* is exemplified in the context of the SEE project. Some considerations coming from the use of the *DKF* and its domain-specific extension *(SEE-SKF)* in the SEE project are presented in Section 4. Finally, conclusions are presented in Section 5.

2 THE HLA DEVELOPMENT KIT FRAMEWORK

The development of HLA distributed simulation is quite a complex task and there are few training resources for developers (Falcone et al. 2014, Falcone et al. 2015). The *HLA Development Kit Framework (DKF)* allows to easily conceptualize, define and build an HLA distributed simulation (Falcone et al. 2017). Developers can focus on domain-specific aspects of a simulation module since, the common HLA functionalities, which are generic across domain, are managed by the core components of the DKF.

The *HLA Development Kit Framework* does not represent another implementation of the HLA standard (IEEE Std. 1516-2010) but it is a software framework placed on the available HLA/RTI implementations such as, Pitch RTI (Pitch Technologies 2017) and VT MÄK (MÄK VR-Forces 2017) that provides an additional layer of abstraction suitable to hide the complexity of the standard and facilitate the development of HLA Federates. The Kit comes with the software framework, called *DKF* that provides all the services needed to develop HLA Federates; *technical documentation* that describes the architecture of the *DKF* along with its services; *user guide* to support developers in the use of the kit; a set of *reference examples* of HLA Federates created through the use of the *DKF*; and, some *video-tutorials* that show how to create both the structure and the behavior of an HLA Federate. In the next Subsections, the main objectives and benefits arising from its use are presented along with the architecture and services provided. Moreover, the *SEE HLA Starter Kit (SEE-SKF)* that is a domain-specific extension of the *DKF* is also introduced.

2.1 Objectives and expected benefits

The *HLA Development Kit Framework (DKF)* combines the objectives and benefits of the available HLA/RTI implementations with a new layer of functionalities exposed through a set of APIs (Application programming interfaces) making easier the design and develop of HLA Federations (Falcone et al. 2017). In this section, some benefits and drawbacks of *DKF* are presented.

The main benefit is that the *DKF* allows developers to operate under the paradigm "write one and run anywhere". Since the *DKF* is fully compliant with the IEEE 1516-2010 specifications (IEEE Std. 1516-2010), it is platform-independent and can be used with different HLA RTI implementations such as, Pitch RTI and VT MÄK. This characteristic gives developers the ability to concentrate on the problem and related behavior of their HLA distributed simulations without worrying on which platform the simulations will be executed. Another aspect to be highlighted concerns the learning curve that is an important concept that graphically combines the relationship between the increase of learning and the experience (Falcone et al 2016). The DKF allows programmers to reduce the HLA learning curve through a homogeneous set of APIs that hide the complexity of the standard.

Coming from the previous aspect, the use of the DKF leads to a significant reduction of the development time (both for full-fledged and Dummy/Tester HLA Federates) and overall costs. The code created by using the DKF is generally more compact and much easier to maintain and test compared with the same one produced through the use of other frameworks or tools (Falcone et al. 2017). Moreover, the reliability of a HLA Federate increases because the core functionalities such as the management of its structure and life cycle are managed directly by the DKF core components (see Subsections 2.2 and 2.3). Despite the above sketched benefits, the DKF has some drawbacks. In particular, the DKF supports partially the SyncronizationPoint mechanism, while does not provide support to event-based simulation execution and Data Distribution Management (DDM). These features, according to the DKF development roadmap (Falcone et al. 2017), will be added in the next release.

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-SKF: SEE HLA Starter Kit Framework*).

2.2 Architecture

As shown in Figure 1, the architecture of a DKF-based Federation is composed of two main layers: the *Federates Level* and *Core Level*.

Federates Level. It contains the application-specific services, which in turn use the services offered by the lower levels, to provide developers an environment to design and develop HLA Federates. A Federate can interact with both the *DKF* and the HLA RTI by using their APIs.

Core Level. It represents the core of the architecture and provides a set of APIs that are used to develop and manage a HLA Federate. This level is composed of two parts: (i) the *DKF*, which offers a set of domain-independent services that are used by programmers to access the *DKF* capabilities (see Subsection 2.3); and (ii) the HLA RTI, which represents the RTI implementation that host the Federation execution. Developer can choose at run-time the appropriate RTI implementation that fits their needs such as, Pitch RTI and VT MÄK without recompiling the simulation code. In the following Subsection, the *DKF* services with their UML Class diagrams are described in detail.

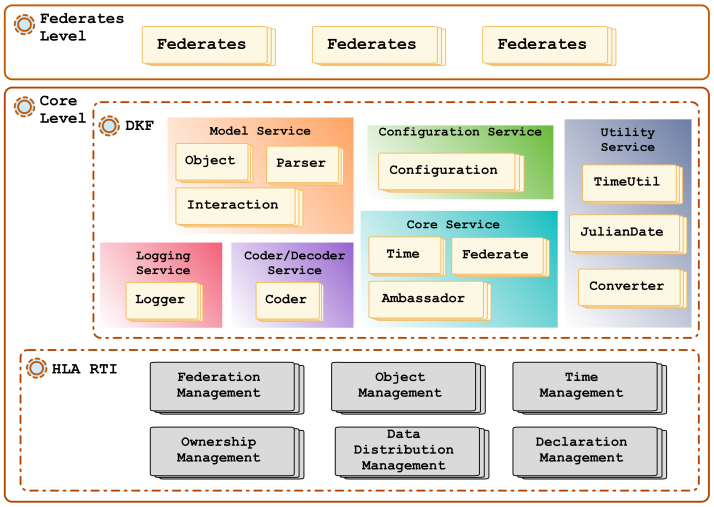


Figure 1: Architecture of a HLA Federation based on the DKF.

2.3 Services and Functionalities

As showed in Figure 1, the *DKF* layer provides a collection of function modules and services that are independent both of application domains and HLA RTI implementations. Each service offers some Java classes and interfaces that implement/define specific functionalities.

The *Logging Service* provides functionalities useful to both track down any problems and errors that occurred during the *DKF* use, and understand how the *DKF* core modules work. Also, this service logs messages and information coming from HLA Federates running in a Federation execution. Each of them uses the logging service API to interact with the logging modules so as to store information messages in the logstream. The Logging Service writes, at regular time intervals, the logstream into the *dkftrace.log* file. The structure of the *Logging Service* is shown in Figure 2 by using a UML Class Diagram.

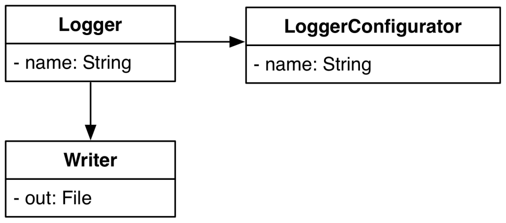


Figure 2: UML Class diagram of the Logging Service.

The *Model Service* offers functionalities to facilitate publishing, subscribing and data updating of both *ObjectClass* and *InteractionClass* instances (IEEE Std. 1516-2010). To manage *ObjectModels* (*ObjectClass* and *InteractionClass*), the DKF framework defines a set of Java annotations, each of which covers and handles a specific concept defined in the *HLA Object Model Specification* document (IEEE Std. 1516-2010). These annotations, which are applicable to a piece of the program code so as to guide the core components of the DKF in managing *ObjectModels*, are used by programmers to create an *ObjectModel* instance compatible with the kit. In particular, two annotation classes have been defined: *@ObjectClass* and *@InteractionClass*. The first one is defined in the Object module and provides attributes for the definition of *ObjectClass* instances, whereas the second annotation class is specified in the *Interaction* module and offers concepts to define and handle *InteractionClass* instances. These two classes are used at runtime by the *Parser* module to examine the structure of an *ObjectModel* instance.

The main Java classes and interfaces of the *Model Service* are shown in Figure 3 by using a UML Class Diagram.

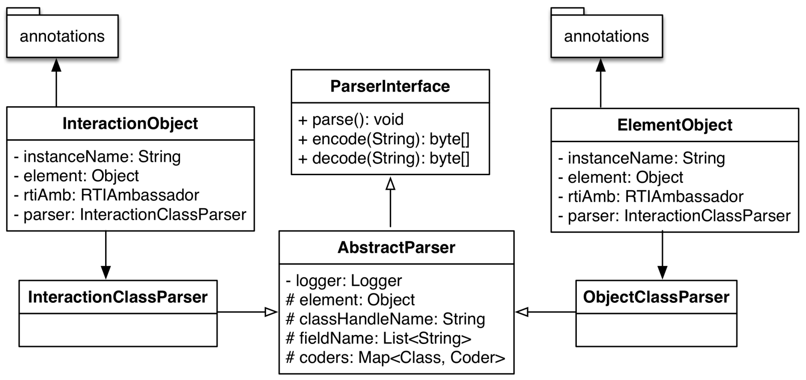
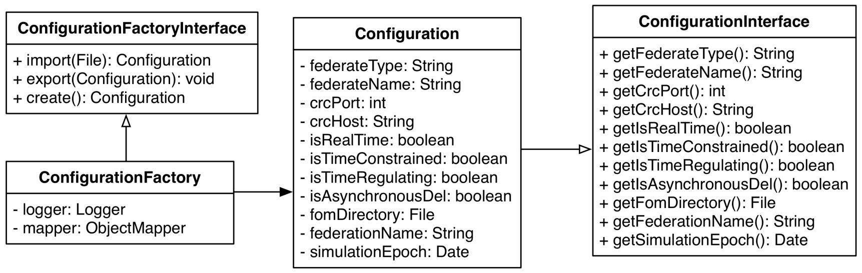


Figure 3: UML Class diagram of the Model Service.

The *Configuration Service* offers a set of services that manage the configuration parameters, which are provided through a **.json* file, make them consistent and use them to configure dynamically the other DKF services. Two kinds of parameters are managed by the *Configuration Service*: Federation and Federate parameters. The *Federation parameters* are used to properly load, configure and execute a HLA distributed simulation (HLA Federation); they include the name of the Federation Execution, the RTI connection details (e.g., IP address, listening port, etc.) along with data related to the simulation time (e.g. HLA time regulating, HLA time constrained and simulation ephoc). The *Federate parameters* represent the information related to the simulation module (HLA Federate) and include information such as, the type (*FederateType*) and name (*FederateName*) of the Federate that are used by the *DKF* Core Service to configure the Federate and its Federate Ambassador (IEEE Std. 1516-2010). Figure 4 shows the main Java classes and interfaces of the *Configuration Service* through a UML Class Diagram.



ConfigurationFactoryInterface
+ import(File): Configuration
+ export(Configuration): void
+ create(): Configuration
ConfigurationFactory
- logger: Logger
- mapper: ObjectMapper
Configuration
- federateType: String
- federateName: String
- crcPort: int
- crcHost: String
- isRealTime: boolean
- isTimeConstrained: boolean
- isTimeRegulating: boolean
- isAsynchronousDel: boolean
- fomDirectory: File
- federationName: String
- simulationEpoch: Date
ConfigurationInterface
+ getFederateType(): String
+ getFederateName(): String
+ getCrcPort(): int
+ getCrcHost(): String
+ getIsRealTime(): boolean
+ getIsTimeConstrained(): boolean
+ getIsTimeRegulating(): boolean
+ getIsAsynchronousDel(): boolean
+ getFomDirectory(): File
+ getFederationName(): String
+ getSimulationEpoch(): Date

Figure 4: UML Class diagram of the Configuration Service.

The *Core Service* represents the kernel of the DKF and offers a set of low level functionalities to manage a HLA Federate with its HLA Federate Ambassador (IEEE Std. 1516-2010). Figure 5 shows the main classes.

The *DKFAbstractFederate* class manages the life cycle of a HLA Federate. It offers functionalities to configure and connect/disconnect a HLA Federate to/from a HLA Federation execution. In particular, the *DKFAbstractFederate* class provides a concrete HLA Federate with the management of its life cycle (FLCM), as a consequence, developers have only to define the specific behavior of its HLA Federate (through the *proactive* and *reactive* states, see Figure 6). Thanks to the FLCM a developer can spend more time to improve its simulation module without worrying about low-level implementation details since the DKF manages them.

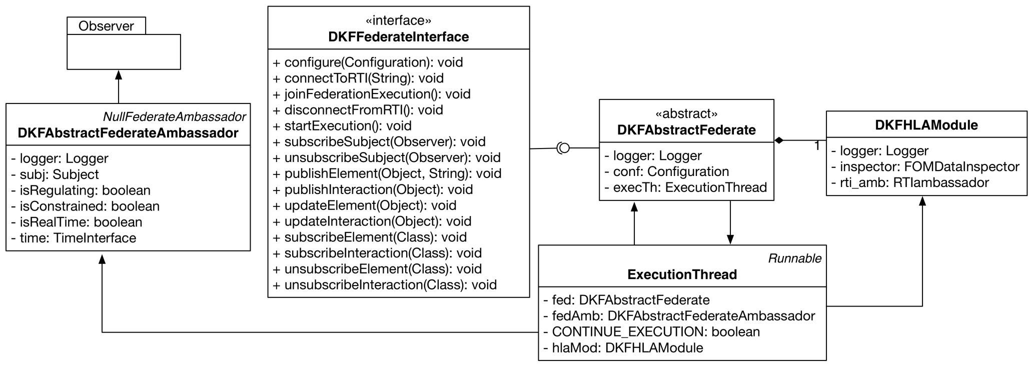

Observer
«interface»
DKFFederateInterface
+ configure(Configuration): void
+ connectToRTI(String): void
+ joinFederationExecution(): void
+ disconnectFromRTI(): void
+ startExecution(): void
+ subscribeSubject(Observer): void
+ unsubscribeSubject(Observer): void
+ publishElement(Object, String): void
+ publishInteraction(Object): void
+ updateElement(Object): void
+ updateInteraction(Object): void
+ subscribeElement(Class): void
+ subscribeInteraction(Class): void
+ unsubscribeElement(Class): void
+ unsubscribeInteraction(Class): void
NullFederateAmbassador
DKFAbstractFederateAmbassador
- logger: Logger
- subj: Subject
- isRegulating: boolean
- isConstrained: boolean
- isRealTime: boolean
- time: TimeInterface
«abstract»
DKFAbstractFederate
- logger: Logger
- conf: Configuration
- execTh: ExecutionThread
1
DKFHLAModule
- logger: Logger
- inspector: FOMDataInspector
- rti_amb: RTIambassador
Runnable
ExecutionThread
- fed: DKFAbstractFederate
- fedAmb: DKFAbstractFederateAmbassador
- CONTINUE_EXECUTION: boolean
- hlaMod: DKFHLAModule

Figure 5: UML Class diagram of the Core Service.

The *ExecutionThread* class handles the execution of a HLA Federate in the simulation environment, whereas the *DKFAbstractFederateAmbassador* class implements the methods that are called by the RTI for interacting with the Federate (RTI callback methods) along with the *NullFederateAmbassador* interface, which is used by a Federate to access the RTI services. With reference to the life cycle of a HLA Federate provided by the *DKFAbstractFederate*, in it is shown through a UML statechart diagram.

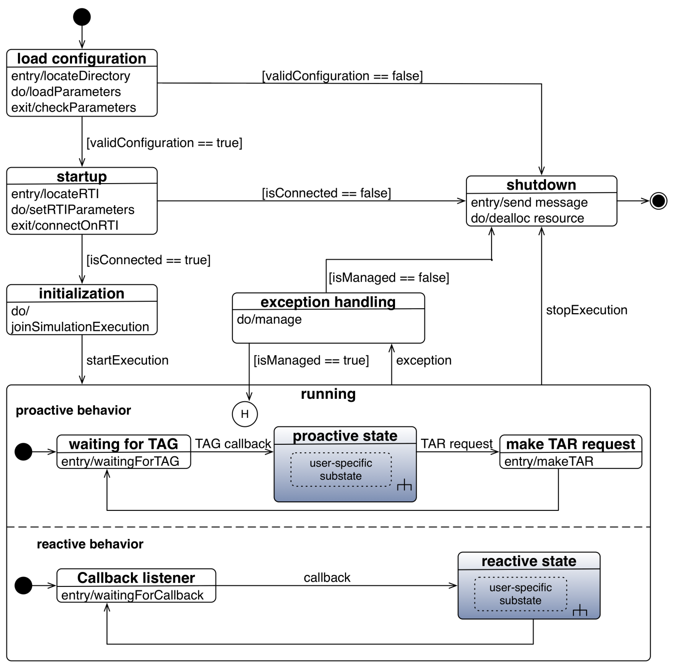


Figure 6: UML statechart diagram of a HLA Federate's life cycle.

Specifically, 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 Federation execution platform is performed. 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 *DKF* framework are de-allocated and the lifecycle terminates. In the *startup* state, the core components of the DKF check the connection status. If the connection is not established the lifecycle ends with a transition to the *shutdown* state. Otherwise, a transition to the initialization state is performed; in this state, the Federate could perform additional operation for exchanging initialization objects before entering the *running* state, as an example, the Federate could publish and subscribe some information (e.g. Object and/or Interactions). After that, the *DKF* activates the time management thread and a transition to the running state is performed. The *running* state is composed of two sub-states operating in an AND-decomposition modality. The *proactive behavior sub-state* deals with the *pro-active* part of the Federate behavior through three states: (i) *Waiting for TAG*: the *DKFAbstarctFederate* 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 behavior of the specific Federate defined by the developers are executed, and then a transition to the make TAR request state is performed; (iii) make TAR (Time Advance Request) request: the *DKFAbstractFederateAmbassador* requests to 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 Federate defined in the *reactive* composite state.

If unexpected events or exceptions occur during the life cycle, the execution flow of the Federate' behavior stops running and a transition to the *exception handling state* is performed. This latter state is responsible for managing the event/exception; if it is correctly handled, the running state is entered at the history state meaning that the execution flow is resumed where it last left off, otherwise a transition to the *shutdown* state is performed and, during the state transition, the HLA Federate is disconnected from the RTI. Figure 7 shows the structure of the *Utility Service* through a UML Class Diagram. This service provides support functionalities to the simulation components (HLA Federates) using the framework.

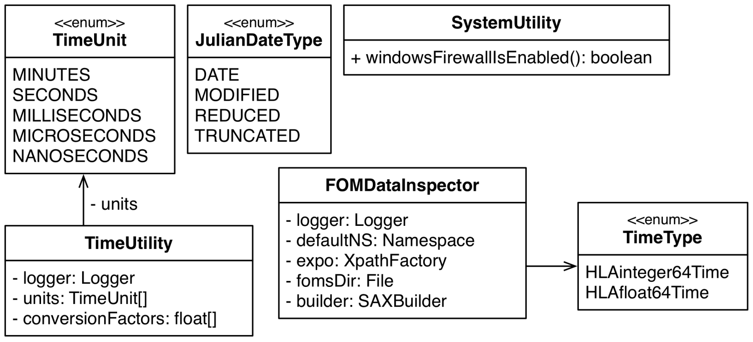


Figure 7: UML Class diagram of the Utility Service.

The *FOMDataInspector* class handles *FOM (Federation Object Model)* modules (IEEE Std. 1516-2010) with their attributes and defines functionalities to automatically generate a java object representation of the *FOM* modules. *TimeUtility* class offers mechanisms for time standard conversions whereas, the *SystemUtility* class provides a method to check the status of the *MSWindows Firewall*.

Finally, the *Coder/Decoder Service* defines all the standard HLA functionalities for coding and decoding both *ObjectClass* and *InteractionClass* instances during the phases of publication and updating. (IEEE Std. 1516-2010). Its UML Class Diagram is shown in Figure 8.

The generic *Coder* interface defines three methods that must be implemented by a subclass that inherits from it, but the provided interface does not dictate how the subclass implements these required methods. This aspect provides flexibility on the developers' side that may implement the *Coder* interface defining the required methods in different ways.

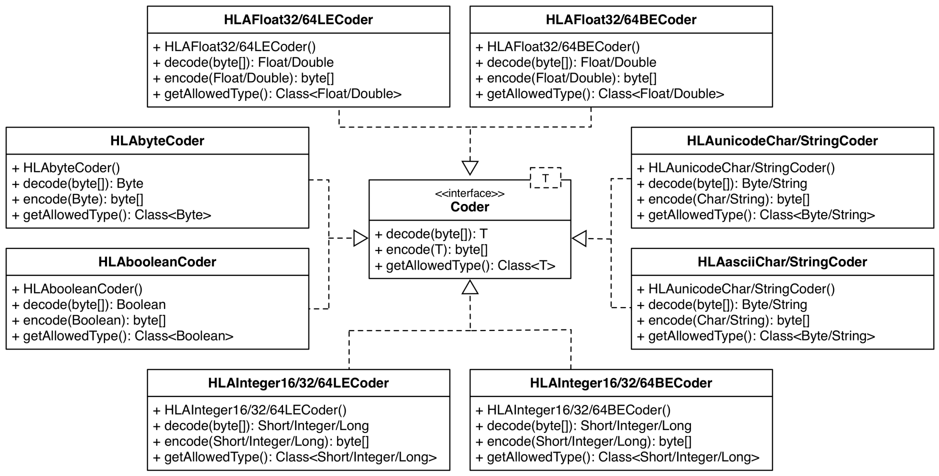

HLAFloat32/64LECoder
+ HLAFloat32/64LECoder()
+ decode(byte[]): Float/Double
+ encode(Float/Double): byte[]
+ getAllowedType(): Class<Float/Double>
HLAFloat32/64BECoder
+ HLAFloat32/64BECoder()
+ decode(byte[]): Float/Double
+ encode(Float/Double): byte[]
+ getAllowedType(): Class<Float/Double>
HLAbyteCoder
+ HLAbyteCoder()
+ decode(byte[]): Byte
+ encode(Byte): byte[]
+ getAllowedType(): Class<Byte>
<<interface>>
Coder
T
+ decode(byte[]): T
+ encode(T): byte[]
+ getAllowedType(): Class<T>
HLAunicodeChar/StringCoder
+ HLAunicodeChar/StringCoder()
+ decode(byte[]): Byte/String
+ encode(Char/String): byte[]
+ getAllowedType(): Class<Byte/String>
HLAbooleanCoder
+ HLAbooleanCoder()
+ decode(byte[]): Boolean
+ encode(Boolean): byte[]
+ getAllowedType(): Class<Boolean>
HLAasciiChar/StringCoder
+ HLAunicodeChar/StringCoder()
+ decode(byte[]): Byte/String
+ encode(Char/String): byte[]
+ getAllowedType(): Class<Byte/String>
HLAInteger16/32/64LECoder
+ HLAInteger16/32/64LECoder()
+ decode(byte[]): Short/Integer/Long
+ encode(Short/Integer/Long): byte[]
+ getAllowedType(): Class<Short/Integer/Long>
HLAInteger16/32/64BECoder
+ HLAInteger16/32/64BECoder()
+ decode(byte[]): Short/Integer/Long
+ encode(Short/Integer/Long): byte[]
+ getAllowedType(): Class<Short/Integer/Long>

Figure 8: UML Class diagram of the Coder/Decoder Service.

For the sake of clarity, the coder/decoder classes related to a specific HLA data type have been grouped in a single UML Class element. Specifically, the *HLAInterger16/32/64BECoder* represents the three classes: *HLAInterger16BECoder, HLAInterger32BECoder* and *HLAInterger64BECoder* that manage the coding and decoding of an *HLAIntegerBE* attribute (Falcone et al. 2017, IEEE Std. 1516-2010). In the same way, the *HLAInterger16/32/64LECoder* characterizes the *HLAInterger16LECoder*, *HLAInterger32LECoder* and *HLAInterger64LECoder* classes for handling a *HLAIntegerLE* attribute (Falcone et al. 2017, IEEE Std. 1516-2010). Analogous consideration can be done for the rest of the classes showed in Figure 8.

2.4 Domain-Specific Extensions: The SEE HLA Starter Kit

To encourage the use of the *HLA Development Kit* by the scientific community, a domain-specific extension of it, named the *SEE HLA Starter Kit (SEE-SKF),* has been created (Falcone et al. 2015, Falcone et al. 2017, Garro et al. 2015).

The *SEE-SKF* aims at easing the development of space systems in the context of the *Simulation Exploration Experience (SEE)* project (Simulation Exploration Experience 2017) by providing a software framework, called the *SEE-SKF*, with related documentation, user guide and reference examples. Specifically, the *SEE-SKF* combines the benefits coming from the *DKF* with high fidelity models and algorithms needed for defining space systems that are as accurate and robust as those provided by existing commercial and government software. The *SEE-SKF* is fully implemented in the Java programming language and offers a consistent set of functionalities for defining, developing and running complex HLA simulation systems in space such as, time scales, reference frames, planets, orbital parameters and propagation, physical entities and attitude (Garro et al. 2015, Möller 2016). The *SEE-SKF* characteristics (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 HLA Federates and thus reducing the problems arising during the final integration and testing phases of the SEE project (Simulation Exploration Experience 2017). Furthermore, it prove how, starting from a domain-independent core of the *DKF*, defined for supporting the development of general-purpose HLA Federations, it is possible to easily add and/or integrate application-specific extensions for supporting the development of domain-specific HLA Federations (Falcone and Garro 2016a).

3 HOW TO DEVELOP A FEDERATE USING THE DKF

This section provides a simple and quick introduction to the *DKF* and its domain-specific extension (the *SEE-SKF*) by walking through the creation of a simple HLA Federate that simulates a *Lander* module. The lander represents a spacecraft that descends and comes to rest on the surface of an astronomical body. During the landing phase, the lander may use either parachutes to slow down and to maintain a low terminal velocity or small landing rockets, which are fired just before impact to reduce the impact velocity. The equations that regulate the vertical motion of the lander during the landing operation are defined in (Garro and Falcone 2015). Once developers are done with this section, they will have a general knowledge of how to create and run a simple HLA Federate through the use of the *DKF*/*SEE-SKF* framework. To develop an HLA Federate, the following software and resources are needed: (i) a Java Integrated Development Environment (IDE) such us Eclipse (Eclipse Foundation 2017) and NetBeans (NetBeans 2017); (ii) a Java Development Kit (JDK) version 1.7.0 or above; (iii) an HLA/RTI implementation, such us Pitch RTI and VT MÄK; and finally, (iv) the DKF/SEE-SKF framework.

The first step in the developing process of the *Lander Federate* is to build a Java project in the IDE environment and create the *HLA ObjectClass* classes with their attributes and coders (as specified in the *FOM*) by using the annotation mechanism provided by the *DKF* (Falcone et al. 2017). It is possible to use the set of basic coder or simply define new coders by implementing the *Coder<T> interface* (see Figure 8). The code snippet below shows the *Lander* class (*HLA ObjectClass*).

```
@ObjectClass(name = "HLAobjectRoot.Lander")
public class Lander {

    @Attribute(name = "name", coder = HLAunicodeStringCoder.class)
    private String name = null;
    @Attribute(name = "velocity", coder = Vector3Coder.class)
    private double[] velocity= null;
    @Attribute(name = "attitude", coder = AttitudeCoder.class)
    private double[] attitude= null;
    @Attribute(name = "acceleration", coder = Vector3Coder.class)
    private double[] acceleration= null;
    @Attribute(name = "position", coder = Vector3Coder.class)
    private double[] position= null;

    public Lander(){}
    // getter and setter methods (not shown)
}
```

In order to be processed by the *DKF* core components, an *ObjectClass* class must be defined according to the JavaBeans API specifications (Oracle 2017) and it must use the provided Java annotation classes (see Section 2.3). The main characteristics that distinguish a JavaBean from other Java classes are: (i) a JavaBean provides a default, no-argument constructor; (ii) it should be serializable and implement the Serializable interface, but this characteristic is not necessary for the *DKF*; (iii) it may have a number of properties which can be read or write; and finally, (iv) it may have a get and set method for each attribute. The *DKF* comes with four main Java annotation classes that are used by the *DKF* core services at runtime to inspect and manage the structure of an object. The *@ObjectClass* and *@Attribute* classes are used for managing *HLA ObjectClass* instances, whereas the *@InteractionClass* and *@Parameters* classes for handling *HLA IntercationClass* objects (Falcone et al. 2017, IEEE Std. 1516-2010). These classes enable a developer to specify metadata information related to a given object both at class and field level. In particular, the *@ObjectClass* and *@InteractionClass* are class-level annotations contain all the HLA metadata information such as *name*, *sharing* and *semantic* used to build a whole object at runtime. The *@Attribute* and *@Parameter* classes are field-level annotations applied to a java attribute for specifying finer-grained HLA information such as *name*, *coder* and *semantic* that are used to publish, subscribe and update the corresponding HLA attribute or parameter, respectively (Falcone et al. 2017, IEEE Std. 1516-2010). For example, with reference to the code above reported, the value of the *name* field matches the name of the corresponding HLA attribute in the FOM module whereas, the *coder* field defines the specific class to code/decode the attribute.

As second step, the *LanderFederate* class that specifies the behavior of the *Lander* model has to be created. It is required to extend the *DKFAbstractFederate* abstract class and implement two methods according to the Federate life cycle that is provided and completely managed by the DKF core components (see Figure 6). More in detail, a developer has to define the *doAction()* method for specifying the pro-active part of the behavior of the Federate that is executed between a TAG and a TAR (IEEE Std. 1516-2010); and the *update(Observable ob, Object arg)* method that represents the re-active part of the Federate's behavior, i.e. how to handle the RTI callbacks about the interactions/objects that the Federate has subscribed.

```
public class LanderFederate extends DKFAbstractFederate implements Observer {
    private Lander lander = null;

    public LanderFederate (DKFAbstractFederateAmbassador fedamb) {
        super(fedamb);
        this.lander = new Lander("Lander", FrameType.MoonCentricFixed, new
        Position (100, 200, 300));
    }
    public void configureAndStart(Configuration conf) throws Exception {
        // 1. configure the DKF framework
        super.configure(conf);
        // 2. Connect on RTI
        super.connectOnRTI("crcHost="+conf.getCrcHost()+
                           "\ncrcPort="+conf.getCrcPort());
        // 3. join the SEE Federation execution
        super.joinIntoFederationExecution();
        // 4. Subscribe the Subject to receive HLA/RTI callbacks
        super.subscribeSubject(this);
        // 5. publish the lander object
        super.publishElement(lander, lander.getName());
        // 6. Execution-loop
        super.startExecution();
    }
    @Override
    protected void doAction() throws Exception {
        lander.step();
        super.updateElement(lander);
    }
    @Override
    public void update(Observable arg0, Object arg1) {
        System.out.println(arg1);
    }
}
```

The third phase of the developing process of the *Lander Federate* consists in implementing its Federate Ambassador as defined in the HLA standard (IEEE Std. 1516-2010). To define and build the *LanderFederateAmbassador* class, it is necessary that it extends the core functionalities provided by the *DKFAbstractFederateAmbassador* class. Generally, since no specific implementation is required, the child class has only to define its constructor which in turn calls the parent one; in this way, all the typical Ambassador's features are provided and managed by the DKF core services. The code snippet below shows the *LanderFederateAmbassador* class.

```
public class LanderFederateAmbassador extends DKFAbstractFederateAmbassador {
    public LanderFederateAmbassador(){
        super();
    }
}
```

Finally, as last step, the *Main* class needs to be created. This class manages the setup of the HLA simulation infrastructure by reading a *.json* configuration file (see Figure 4) that contains all the parameters necessary to create and manage the HLA Federation execution and the *LanderFederate* with its *LanderFederateAmbassador*. The code snippet below reports the Main class.

```
public class Main {
    private static final File conf = new File("config/conf.json");
    public static void main(String[] args) throws Exception {
        LanderFederate fed = new LanderFederate(new LanderFederateAmbassador());
        // start execution
        federate.configureAndStart(new ConfigurationFactory().
                                importConfiguration(conf));
    }
}
```

Below the *.json* configuration file with all the parameters.

```
{
  "asynchronousDelivery" : true,
  "crcHost" : "localhost",
  "crcPort" : 8989,
  "federateName" : "Lander",
  "federateType" : "Spacecraft",
  "federationName" : "Simulation Execution Platform",
  "fomDirectory" : "C:\fom_mudules",
  "realtime" : false,
  "simulationEphoc" : "2015-04-12T20:00:00.000Z",
  "timeConstrained" : true,
  "timeRegulating" : false
}
```

4 CONSIDERATION FROM THE USE OF THE DKF/SEE-DKF IN THE SEE PROJECT

The *DKF* and its domain-specific extension for the SEE project, the *SEE-SKF*, was used by a team of Universities across the world to develop a Federation within the SEE project first time in the SEE 2015. Brunel University has participated in SEE projects since 2013 developing Federates using different simulation software such as SIMUL8 and REPAST, and fully programmed in Java (Taylor et al. 2014). Pitch and MÄK RTI implementations were used for the HLA interface. The project was undertaken by undergraduate Computer Science students as part of their final year project supported by PhD candidates and research staff. Admittedly, it presented a big challenge for the students having to familiarize themselves with domain-specific and distributed simulation technologies. Developing a Federate involves the development of the simulation model and creation of the Federate and Federate Ambassador classes for RTI interfacing. Also, the *FOM* and *SOM XML schemas* should be created. In the first two SEE events the development was done manually, whereas in the 2015 edition the DKF was used. The experiences of creating the Federate for SEE with and without the *DKF* and a summary comparison is listed in Table 1.

The Federate was an agent-based mining simulation that simulates one or more small excavators working across the lunar surface. The Federate was developed in REPAST implementing the following scenario. Within the Federation, the simulation was designed to interact with the astronaut Federate and Unmanned Aerial Vehicle (UAV) Federate developed by different Institutions. It also interacts with the visualization Federate provided by NASA. The UAV Federate moved over the area to be mined and takes magnetometer readings. These update a map of the area and show the coordinates of areas of interest. The excavator systematically works across its area and digs out regolith materials from the surveyed points of interest. Once the excavator is full it returns the materials to its origin point and deposits them there for the astronaut to pick up. The excavator then returns to its excavation task. 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).

Table 1: Comparison in building DKF and no-DKF based Federate.

	Without DKF	With DKF
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

In HLA, each Federate must declare the published/subscribed Objects and their attributes and Interactions and their parameters. These attributes and parameters are then updated during the Federation execution. Without the *DKF*, these attributes and parameters should be manually declared in the Federate Class one-by-one and then manually update each element during runtime. With the *DKF*, using the provided Java annotations, these published/subscribed classes are annotated as *@ObjectClass* or *@InteractionClass* accordingly. Also, the elements are annotated as *@Attributes* or *@Parameters* in their respective classes. Another consideration when developing a distributed simulation is the data types and how they are encoded. These should be explicitly stated in the *FOM*, individually for each published/subscribed element data type. The *DKF* provides a coder package and the specific coder can be added where the element is annotated (see Section 2.3). Another difference is the time advance management in the Federate. Without the *DKF*, the *REPAST* Federate uses the scheduler provided by the simulator to manage TAR and TAG actions, while via the *DKF* the time advance is managed by the RTI.

5 CONCLUSIONS

This paper provided a tutorial on developing HLA federates using the *DKF*. The framework and its architecture has been explained in detail as well as its services and functionalities. The tutorial guides the reader through the necessary steps for a federate creation and explains how the HLA elements are declared using annotations. Finally, a comparison on developing HLA federates with and without the DKF is represented through experiences on developing federates for the SEE project.

ACKNOWLEDGMENTS

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

Banks, J. 1998. *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. John Wiley & Sons.

Bocciarelli, P., A. D'Ambrogio, A. Giglio, and E. Paglia. 2015. "A Model-Driven Framework for Distributed Simulation of Autonomous Systems." In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, 213-220. The Society for Computer Simulation International.

Bocciarelli, P., A. D'Ambrogio, A. Giglio, and E. Paglia. 2014. "Simulation-Based Performance and Reliability Analysis of Business Processes." In *Proceedings of the 2014 Winter Simulation Conference*, edited by S. J. Buckley, and J. A. Miller, 3012-3023. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Eclipse Foundation. 2017. Eclipse IDE for Java developers home page. https://eclipse.org/.

Falcone, A., and A. Garro. 2016a. "The SEE HLA Starter Kit: Enabling the Rapid Prototyping of HLA-Based Simulations for Space Exploration." In *Proceedings of the Modeling and Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems 2016 (MSCIAAS 2016) and Space Simulation for Planetary Space Exploration (SPACE 2016),* 1-8. The Society for Computer Simulation International.

Falcone, A., and A. Garro. 2016b. "Using the HLA Standard in the Context of an International Simulation Project: The Experience of the "Smashteam"." In *Proceedings of the 15 International Conference on Modeling and Applied Simulation (MAS '16)*, 121–129. Dime University of Genoa.

Falcone, A., A. Garro, S. J. E. Taylor, A. Anagnostou, N. R. Chaudhry, and O. A. Salah. 2017. "Experiences in Simplifying Distributed Simulation: The HLA Development Kit Framework." In *Journal of Simulation* 11(3): 208-227. Palgrave Macmillan, UK.

Falcone, A., A. Garro, A. Anagnostou, N. R. Chaudhry, O. A. Salah, and S. J. E. Taylor. 2015. "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 Real Time Applications (DS-RT '15)*, 50-57. Institute of Electrical and Electronics Engineers, Inc.

Falcone, A., A. Garro, F. Longo, and F. Spadafora. 2014. "Simulation Exploration Experience: A Communication System and a 3D Real Time Visualization for a Moon Base Simulated Scenario." In *Proceeding of the 18th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT '14)*, 113-120. Institute of Electrical and Electronics Engineers, Inc.

Forwardsim. 2017. The Forwardsim HLA Toolbox for MATLAB/Simulink home page. http://www.forwardsim.com/products/hla-toolbox/.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*. Vol. 300. New York: Wiley.

Garro, A., and A. Falcone. 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 & Simulation: DEVS Integrative M&S Symposium*, 9-16. The Society for Computer Simulation International.

Garro, A., A. Falcone, N. R. Chaudhry, O. A. Salah, A. Anagnostou, and S. J. E. Taylor. 2015. "A Prototype HLA Development Kit: Results from the 2015 Simulation Exploration Experience." In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (ACM/SIGSIM-PADS '15)*, 45-46. Association for Computing Machinery, Inc.

HLAListener. 2017. An HLA Software to Develop and Test IEEE 1516-2010 Federates home page. https://github.com/EMostafaAli/HlaListener.

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).

Kuhl, F., R. Weatherly, and J. Dahmann. 1999. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice-Hall, Inc.

MÄK VR-Forces. 2017. MÄK software home page. http://www.mak.com/.

Modeling and Simulation Coordination Office. 2017. The M&S CO home page. http://www.msco.mil/.

Möller, B., A. Garro, A. Falcone, E. Z. Crues, and D. E. Dexter. 2016. "Promoting A-Priori Interoperability of HLA-Based Simulations in the Space Domain: The SISO Space Reference FOM Initiative." In *Proceedings of the 20th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT '16)*, 100–107. Institute of Electrical and Electronics Engineers, Inc.

National Aeronautics and Space Administration. 2017. The Simulation and Graphics Branch Division (ER7) home page. http://er.jsc.nasa.gov/ER7/.

NetBeans. 2017. NetBeans for Java home page. https://netbeans.org/.

Oracle. 2017. JavaBeans Specification 1.01 Final Release. http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/.

Pitch Technologies. 2017. The Simulation Toolkit home page. http://www.pitch.se.

Simulation Exploration Experience. 2017. The Simulation Exploration Experience (SEE) project home page. http://www.exploresim.com/.

Taylor, S. J. E., N. Revagar, J. Chambers, M. Yero, A. Anagnostou, A. Nouman, N. R. Chaudhry, and P. R. Elfrey. 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 (DS-RT '14)*, 107–112. Institute of Electrical and Electronics Engineers, Inc.

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.

AUTHOR BIOGRAPHIES

ALBERTO FALCONE is a PhD candidate in Information and Communication Engineering for Pervasive Intelligent Environments at University of Calabria (Italy). In 2016, he was Visiting Researcher at NASA Johnson Space Center (JSC), working with the Software, Robotics, and Simulation Division (ER7). His e-mail address is alberto.falcone@dimes.unical.it.

ALFREDO GARRO, PhD is an Associate Professor of Computer and Systems Engineering at the Department of Informatics, Modeling, Electronics and Systems Engineering (DIMES) of the University of Calabria (Italy). In 2016, he was Visiting Professor at NASA Johnson Space Center (JSC). He is vice chair of the Space Reference Federation Object Model (SRFOM) Product Development Group of SISO. He is Technical Director of the Italian Chapter of INCOSE. His e-mail address is alfredo.garro@unical.it.

ANASTASIA ANAGNOSTOU, PhD is a Research Fellow at the Department of Computer Science, Brunel University London. She holds a PhD in Hybrid Distributed Simulation. Her research interests are related to the application of M&S techniques in the Healthcare and Industry. Her e-mail address is anastasia.anagnostou@brunel.ac.uk.

SIMON J. E. TAYLOR, PhD is the convener of the new phase of Grant Challenge activities. He is the Founder and Chair of the COTS Simulation Package Interoperability Standards Group under SISO. He is Reader in the Department of Computer Science at Brunel University and leads the M&S Group. His e-mail address is simon.taylor@brunel.ac.uk.