Easing the development of HLA Federates: the HLA Development Kit and its exploitation in the SEE Project

ABSTRACT

The Modeling & Simulation (M&S) of modern cyber-physical systems is presenting new challenges. New M&S techniques, methods and tools are emerging that take advantage of distributed simulation environments. One of the most mature and popular standard for distributed simulation is the IEEE 1516-2010 - High Level Architecture (HLA) that, although originally developed for military applications, is increasingly exploited in a great variety of application domains 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 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 a general-purpose, domain-independent framework that aims to ease the development of HLA-based simulations. Its effectiveness is exemplified in the context of the Simulation Exploration Experience (SEE) project lead by NASA and which involves several U.S. and European Institutions.

Keywords

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

1. INTRODUCTION

Modeling and Simulation (M&S) represents one of the most important and effective methods for designing and studying complex systems in a variety of industrial and scientific domains ranging from biology to space exploration [20]. M&S methods, tools, and techniques allow analyzing and evaluated design alternatives effectively and by avoiding the risk, costs and fails associated with extensive field experimentation; this opportunity becomes crucial when complete and actual tests are too expensive to be performed in terms of cost, time and other primary resources [2], [19].

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 M&S techniques, methods and tools are emerging also to benefit from distributed simulation environments [5]. In this context, the IEEE 1516-2010 - High Level Architecture (HLA) standard [9], [10] attempts to handle this complexity by providing a specification of a distributed infrastructure in which each simulation unit can run on an independent computer (in general, geographically distributed) and communicate with the others in a common simulation scenario.

The HLA was developed by the U.S. Modeling and Simulation Coordination Office (M&S CO) [12] to facilitate the integration of distributed simulation models within a common architecture. Although it was initially developed to support 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 can interact among them by using a Run-Time Infrastructure (RTI). The RTI represents a backbone of a Federation execution and provides a set of 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 [9], [10].

Building complex and large distributed simulations systems, based on the IEEE 1516 standard, is a challenging task and requires considerable development efforts. Indeed, it requires expert engineers with deep knowledge and experience in distributed systems, simulation, middleware and software programming. The main problem is that the development and testing of *HLA Federates* is generally difficult, complex, and resource-intensive not only because of the complexity of the IEEE 1516 standard [9] but also due to the lack of proper documentations and ready-to-use examples. Moreover, developers have to spend a considerable effort to solve common HLA issues, such as the management of the simulation time, the connection on the RTI, and the management of common RTI exceptions. As result, they cannot fully focus on the specific aspects of their own simulations (the *HLA Federates*). Thus, it would be desirable to separate the common HLA issues from those specific of a *HLA Federate* by providing a general-purpose, domain-independent framework that allows achieving these goals.

In this context, the paper presents the *HLA Development Kit* a general-purpose, domain-independent toolkit that eases the development of *HLA Federates* by providing a software framework, called the *DKF* (Development Kit software Framework), with related documentation, user guide and reference examples. Specifically, the *DKF* allows developers to focus on the specific aspects of their own *HLA Federates* rather than dealing with the common HLA issues which are managed by the *DKF* core components.

The rest of the paper is organized as follows. In Section 2, the main issues related to the development of HLA Federates are discussed. 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 Simulation Exploration Experience (SEE) project. The merits of this approach and main related works are discussed in Section 5 and 6, respectively. Finally, conclusions are drawn and future research directions are presented.

2. DEVELOPING HLA FEDERATES: MAIN ISSUES

As discussed in Bjorn Moller's introduction to the HLA [13], a HLA-based Federation (distributed simulation) consists of a number of Federates (usually simulations running on different computers) that interact with each other via software (middleware) called the Run Time Infrastructure (RTI). The Federates use the RTI to transfer information about each other and to coordinate and synchronize with each other.

The "contract" that describes how Federates interact is called the Federation Agreement. The major part of the agreement is a description of the information exchanged between Federates. This is called the Federation Object Model (FOM). A FOM can contain specifications of *Object* classes (objects are instances (entities) of object classes that have attributes that can be updated), *Interaction* classes (a message sent between objects that has parameters) and *Data types* (the technical specification and semantics of the attributes and parameters). Predefined HLA data types can be used to create typical complex data types. A FOM is divided into "sub-FOMs" or modules that effectively define the interface to a Federate. A Federate and its FOM can be therefore produced as a separate component of a Federation, ready to be composed into the full Federation on-demand. The XML-based HLA Object Model Template defines the syntax of a FOM and is part of the HLA standard. "Object-oriented" HLA allows object-like complex data types that allow an object to be shared between Federates (e.g. two Federates have local copies of an object instance and use attribute publishing to "synchronize" the values of the "shared" object).

There are many services in a HLA RTI. These are divided into seven service groups loosely split into information exchange services, synchronization services and coordination services. Two-way interaction between a Federate and an RTI uses RPC-like semantics, i.e. a Federate calls an RTI service method and receives a callback from that method. Information exchange services use a Publish/Subscribe approach to prevent every Federate just broadcasting its information to every other Federate regardless of whether or not that information is relevant to that Federate. A FOM describes the information that each Federate will publish and will expect to receive (will subscribe to). The RTI matches these publishing and subscription requirements to ensure efficient communication – information published by one Federate will only be received by those Federates that are subscribed to that information. Other data distribution management services build on this model to deliver more efficient and flexible communication schemes.

Synchronization services handle the synchronization of logical time across a Federation and the correct ordered delivery of time stamped data (time management), specification of synchronization points to allow Federates to coordinate when they have reached a given state, and state saving to checkpoint where a Federate has reached in its execution.

Coordination services facilitate the management of Federation execution and joining of Federates to a Federation, the transfer of "ownership" of an object from one Federate to another and the advanced inspection and management of a Federation (via the Management Object Model).

An RTI typically consists of central and local components. The central RTI component (CRC) is the middleware that provides requested services to the Federates (effectively a server). Federates access those services by interacting with the local RTI component (LRC) via some module developed for that purpose (effectively a client). Essentially to "convert" a simulation to a Federate, a developer must create the module (the Federate Ambassador) to connect to the local RTI component (the RTI Ambassador) and then implement the interaction of the simulation with the wider Federation (captured in the FOM). To join a Federation, the Federate calls its LRC to connect to the RTI. Interaction continues by calling the RTI via the LRC and by receiving callbacks – sometimes this is described as a Federate using its Federate Ambassador to calling the RTI and to receive callbacks from the LRC's RTI Ambassador.

As can be seen from the above, the development of a Federate and a Federation is extremely complex and there are few tutorial resources to help educate developers. In order to help new developers, the *HLA Development Kit*, which provides high level functionality to choreograph interaction between Federates and the RTI and among Federates, has been developed. This is described in the next section.

3. THE HLA DEVELOPMENT KIT

The *HLA Development Kit* 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. Moreover, a domain-specific extension of the DKF is also presented.

3.1 The HLA Development Kit Framework (DKF)

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 [9], [10]. Indeed, the DKF allows developers to focus on the specific aspects of their own Federates rather than dealing with the common HLA issues 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 [10]. The DKF is designed and developed by the SMASH-Lab (System Modeling And Simulation Hub - Laboratory) of the University of _______ (___) working in cooperation with the NASA JSC (Johnson Space Center), Houston (TX, USA).

The DKF is fully implemented in the Java language and is based on the following three principles: (i) *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 [15], VT/MÄK [11], PoRTIco [16] and CERTI [3]); (ii) *Portability and Uniformity*, 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; and (iii) *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 [6]. Furthermore, it has been developed according to the concept of *Object HLA* [9], [10], in this way, the development of HLA Federates could benefit also from the *Object HLA* features and functionalities provided by the Pitch Developer Studio [15] 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 Starter Kit* with the aim to ease the development of HLA Federates in the context of the Simulation Exploration Experience (SEE) project [17]. SEE is an event organized by the Simulation Interoperability Standards Organization (SISO), in collaboration with NASA and other research and industrial partners, with the objective to promote the adoption of the HLA standard and compliant tools by involving university teams in the distributed simulation of a Moonbase. The SEE-specific features introduced in both the DKF and the Development Kit (as an example, the implementation of 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 [17]. Moreover, the 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.

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 Starter Kit Framework).

3.1.1 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 [9], [10] (e.g. PITCH [15], VT/MÄK [11], PoRTIco [16] and CERTI [3]). Some application-specific extensions of the DKF can be also introduced (e.g. the SEE-specific ones).

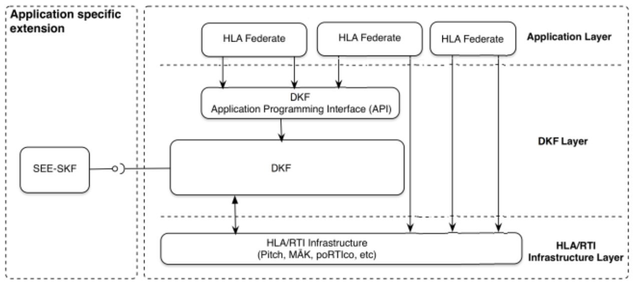
Application specific extension
HLA Federate
HLA Federate
HLA Federate
Application Layer
DKF
Application Programming Interface (API)
DKF Layer
SEE-SKF
DKF
HLA/RTI Infrastructure
(Pitch, MÄK, poRTIco, etc)
HLA/RTI Infrastructure Layer

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

The *DKF* is organized into a hierarchy of packages and sub-packages; each of which contains a set of Java classes and interfaces that implement specific functionalities; the main packages are shown in Figure 2 by using a UML package diagram. In particular, the *DKF* is composed of seven main packages (in yellow) that are independent both of application domains and HLA RTI implementations.

The *core* package implements the kernel of the DKF. It includes the fundamental *DKFAbstractFederate* and *DKFAbstractFederateAmbassador* classes (see Figure 3) that provide the basic functionalities to manage a Federate. The *config* package contains the collection of classes 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. The *utility* package contains several miscellaneous utility classes, such as time standard conversions (e.g. JulianDate, RJulianDate, etc.) and Windows Firewall Check. The *logging* package defines a set of classes used to track down any problems or error occurred during the execution of SEE Federates; this information is stored into the *dkf.log* file. The *exception* package contains some definitions of exceptions that are used for handling dysfunctional events throughout the DKF framework.

The *model* package contains some classes to facilitate publishing, subscribing and the data updating of both an *ObjectClass* and an *InteractionClass* through Java annotations [9], [10]. Two Java annotation classes have been created in order to manage an *ObjectModel* (*ObjectClass* and *InteractionClass*) instance: (i) *ObjectClassAnnotation*, which defines the annotations that have to be used by the programmer so as to create an *ObjectClass* instance compatible with the DKF; and (ii) *InteractionClassAnnotation*, which specifies the annotations to create and handle *InteractionClass* instances.

The *coder* package contains the classes used to coding and decoding both *ObjectClass* and *InteractionClass* instances.

Finally, the application domain extension *see.smackdown* package (in gray), contains some SEE domain-specific classes, which are used by the *core* components of the *DKF* to handle some specific aspects related to the SEE Federation [17] such as transformations among *SEE Coordinate Reference Frames*, the publish and subscribe of *PhysicalEntities*, and the management of the *SISO Space FOMs* [4], [21].

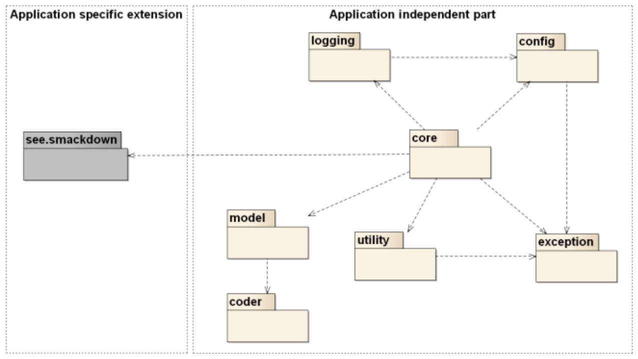

Figure 2: The DKF architecture.

3.1.2 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 3 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 2). 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 *time constrained* except the *Environment* Federate provided by NASA and which lead the Federation execution that is *time regulating* [5]; the DKF has been thus adapted to handle this situation.

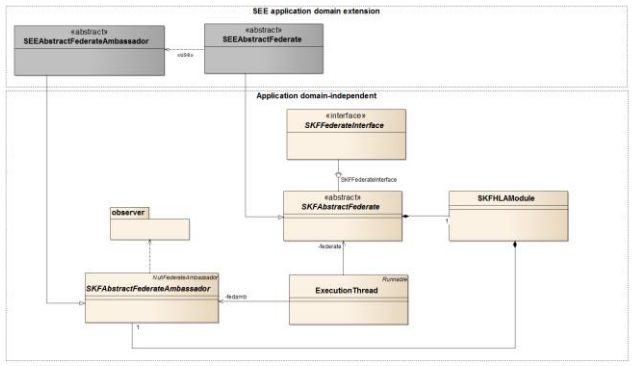

Figure 3: The example architecture of a DKF-based Federate.

In particular, the *DKFAbstractFederate* class provides a concrete SEE Federate with the management of its life cycle (FLCM), 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 "processing and update data" composite state, which is accessed between a TAG and TAR request; whereas, the re-active part of the behavior of a Federate is specified in the "processing interaction" composite state so as to indicate how to handle the RTI callbacks about the interactions/objects that the Federate has subscribed (see Figure 4).

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.

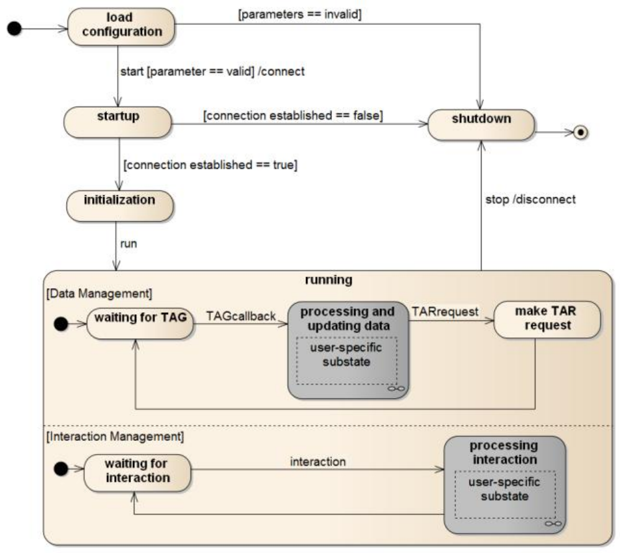

Figure 4: The example life cycle of a SEE Federate.

The lifecycle of a SEE Federate provided by the *DKFAbstractFederate*, consists of four phases as shown in Figure 4 through a UML Statechart diagram. 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 SEE Federation execution 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 SEE-SKF classes are de-allocated and the lifecycle 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, a transition to the *initialization* state is performed; 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 TAG* → *processing and update data* → *make TAR request*), 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 by two sub-states operating in an AND-decomposition fashion. The *Data Management* 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 *processing and update data* state is performed; (ii) *processing and update data*: the "logical time" is updated, the pro-active behavior of the specific SEE Federate defined by the SEE working team is executed, and then a transition to the *make TAR request* state is performed; (iii) *make TAR (Time Advance Request) request*: the DKF requests to the RTI the grant for the next "logical time". The *Interaction Management* sub-state deals with the re-active part of the behavior of the Federate: upon reception of RTI callback related to subscribed elements, a transition to the *processing* state is performed where the received information is handled.

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. EXPLOITING THE (SEE) HLA STARTER KIT

This section presents a case study concerning the development of a HLA Federate by using the SEE HLA Starter Kit in the context of the SEE 2015 project. The architecture and behavior of the developed and experimented SEE HLA Federate are described along with the feedback coming from the experimentation.

4.1 The development process based on SEE-SKF

The process to build a Federate from scratch by using the SEE-SKF is composed by the following 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 *coder* to handle such attributes. It is possible to use the set of basic coder provided by the SEE-SKF or simply implement new coders by using the SEE-SKF classes; other available coders can be also exploited;

2. Build a concrete Federate that specifies the *behavior* of the *model* defined at step 1. It is required to extend the *SEEAbstractFederate* abstract class provided by the SEE-SKF and implement three methods according to the Federate life-cycle that is provided and completely managed by the SEE-SKF (see Figure 4), specifically: (i) a method for initialization operations before entering in the "running state" (a *configureAndStart* method); (ii) a method for specifying the pro-active part of the behavior of the Federate (*doAction* method) and that is executed between a TAG and TAR request; (iii) 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-SKF.

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 [17].

4.2 The reference simulation scenario

The reference simulation scenario of the SEE Project (see [4], [21]) 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 Modelling & Simulation Group (MSG) at _______ 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 [18], [22] as well as hybrid federations consisting of real-time, discrete-event and agent-based simulations [1]. The 2013 "entry" involved the development of a lunar factory that 3D printed building materials using processed lunar regolith. This was in fact a discrete-event simulation model developed in the Simul8 simulation software. The 2014 entry expanded the scope of the factory by developing a new factory in Simul8, a lunar mining operation in REPAST SIMPHONY (the Recursive Porous Agent Simulation Toolkit - an agent-based simulation system from Argonne National Laboratories [14]) and a real-time simulation of an astronaut. The agent excavators of the mine dug out the regolith materials and returned these to a stockpile. The astronaut transported the materials to the factory for processing.

Most (if not all) of the teams in SEE are postgraduate. The aim of the work at ______ was to see if a small group of Undergraduate B.Sc. (HONS) Computer Science students could take on the challenge of developing in hybrid distributed simulation. This formed their Final Year Projects undertaken during their final year studies and represents a major part of their degree classification. All the students had two years of a general computer science degree. The students had support from the staff and postgraduate students of the MSG. However, it was the students' responsibility to deliver.

The main issue that arose from the 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.

In this year's entry participating was restricted to one undergraduate student. His task was to develop a new agent-based mining simulation that simulates one or more small excavators working across the lunar surface. The simulation is designed to work with an astronaut federate and a UAV federate. 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. This section now gives a brief overview of agent-based simulation with REPAST, the excavator Federate and how the SEE-SKF was used to simplify implementation.

4.3 Agent-based Modeling & Simulation

Agent-Based Modeling and Simulation (ABMS) historically originated from Complex Adaptive Systems (CAS), where the principal area of study is the complex behaviors among individual and autonomous agents. ABMS is used mainly to model decentralized, complex systems that consist of many inter-dependencies [7]. The main components of ABMS are agents - heterogeneous, adaptive and goal-directed autonomous entities that have a sort of intelligence in that they can recognize their environment and other agents and interact with them. Agents have attributes and methods. Agents represent static or dynamic elements that describe the current state of the agent. Methods describe behaviors, the ability to modify these behaviors, and the ability to update rules and dynamic attributes. Agents have four essential characteristics:

- Agents are distinguished, independent individuals with rules that administer their behavior and decision-making capability. Their nature is discrete, which means that they have clear boundaries and it can be easily determined whether a characteristic belongs to a specific agent or is shared among agents.
- Agents are active components of an environment and coexist with other agents, and, therefore, can be characterized as social components. Usually, communication protocols enable agents to interact with one another and their environment. Agents can recognize the behavior of other agents.
- Agents are autonomous and self-directed. They have their own set of behavioral rules that dictate their decisions and actions. The degree of sophistication of these behavioral rules indicates the intelligence of the agent which is decided according to the scope of the model.
- Agents have a state that varies over the simulation time. The state of an agent is dictated by its state variables and can be a set or a subset of its attributes.

The way that agents are connected to each other constitutes the topology of an agent-based model. Agents can move in a number of different topologies. Agents also have neighborhoods. Each agent can hold information about its local neighborhood and the neighboring agents and communicate with them. For example, a very common spatial topology for agents is a grid and can be von Neumann (five cell neighborhood) or Moore (nine cell neighborhood).

As introduced above REPAST SIMPHONY is a free and open-source agent-based simulation environment. A REPAST agent-based simulation is created by using the *ContextBuilder* interface. In this class the environment, the initial number of agents (and types/classes) that are located in the environment, etc. are specified. The attributes and methods of each agent is 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. This will include the frequency and priority that the method occurs. When a REPAST simulation runs, the simulation environment enter a cycle that calls the scheduler. The scheduler then runs the methods in priority order according to their frequency, so advancing the simulation until it reaches some terminating condition.

4.4 The Excavator Agent-based Simulation

The ultimate goal of the excavator agent-based simulation was to 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. As this paper focuses on how the SKF was used to simplify the implementation of a Federate, a simple version of the agent-based simulation is presented. In this example there is a single excavator that explores its environment in a simple "scanning" pattern. It uses a map populated by a UAV with a magnetometer. The UAV slows "flies" overhead detecting potentially interesting minerals (modelled by a reading of 0 for nothing, 1 for something). The UAV periodically broadcasts the results of its on-going survey to the excavator and the excavator updates its local map (at an arbitrary ten readings an update). When the excavator reaches a mineral, it mines it 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 the *JExcavatorsBuilder*, Excavator and Mineral classes. *JExcavatorsBuilder* implements the REPAST *ContextBuilder* interface to create the simulation environment. It does this by first creating a continuous space and then superimposing a grid for the excavators to move around. The grid is then populated by an Excavator at 0,0. An Excavator has several variables that specify where it is currently located on the grid (pt), the amount of regolith carried (cargo), and if it has decided to return to the origin point to offload its regolith (returnOrigin). When the simulation begins it calls all *step*() methods in its agents. In this example, the single excavator *step*() method is called. This first simulates the interaction with the UAV by generating the next ten magnetometer readings and updating the map by calling *updateMap*(). This populates the grid with zero to ten new Minerals at x,y points ([UavX1, UavY1, MagReading], etc.) MagReading is either 0 or 1 to indicate the presence of a mineral deposit. The excavator then reads its own location on the grid and checks to see if its cargo limit has been reached. If it has, it sets a Boolean *returnOrigin* to TRUE. If *returnOrigin* is FALSE, the *moveLinearly*(pt) method moves the excavator along to the next point, checks the map to determine if there is anything to mine and, if there is, mines it (adds 1 to the cargo and deletes the mineral from the map). If *returnOrigin* is TRUE, then this method excavator first moves the excavator one point at a time along its X axis to 0 and then its Y axis to 0 to reach the origin. The excavator then "dumps" its cargo and retraces its steps back to where it left off, again one point at a time.

```
//@ScheduledMethod(start=1, interval = 1)
public void step() {
  updateMap();
  GridPoint pt = grid.getLocation(this);
  checkCargoLimit(cargo);
  moveLinearly(pt);
}
```

4.5 Using the SKF 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-SKF main steps have been followed.

In step (1) a FOM that describes the input and output of the simulation was defined. In this case the FOM represents the single Excavator object with *ExcavatorX* and *ExcavatorY* representing the current coordinates of the excavator, and *PileNumber*, representing the number of minerals in the regolith pile. All are *HLAinteger32BE* datatype. To begin the creation of the Federate, first annotate the Excavator class to match the FOM has been annotated as follows:

```
@ObjectClass(name =
  "PhysicalEntity.Excavator")
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 coder to the declarations have been added:

```
@Attribute(name = "ExcavatorX", coder =
 HLAinteger32BECoder.class)
private Integer ExcavatorX;
@Attribute(name = "ExcavatorY", coder =
 HLAinteger32BECoder.class)
private Integer ExcavatorY;
```

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

```
setExcavatorX(getPointX());
setExcavatorY(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* classes. 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.

```
@Override
protected void doAction() {
try {
 Context<Object> con =
  RunState.getInstance().getMasterContext();
    for (Object obj : con) {
      if(obj instanceof Excavator){
        // update the RTI on the excavator
        ((Excavator) obj).step();
        }
      super.updateElement(obj);
    }
}
```

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*() to output the new state of the excavator Federate's attributes.

Step (3) simply extended the *SEEAbstractFederateAmbassador* class with as the *ExcavatorFederateAmbassador*. Step (4) was unnecessary as the simulation had already been developed. The only addition to these steps was the addition 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 5.

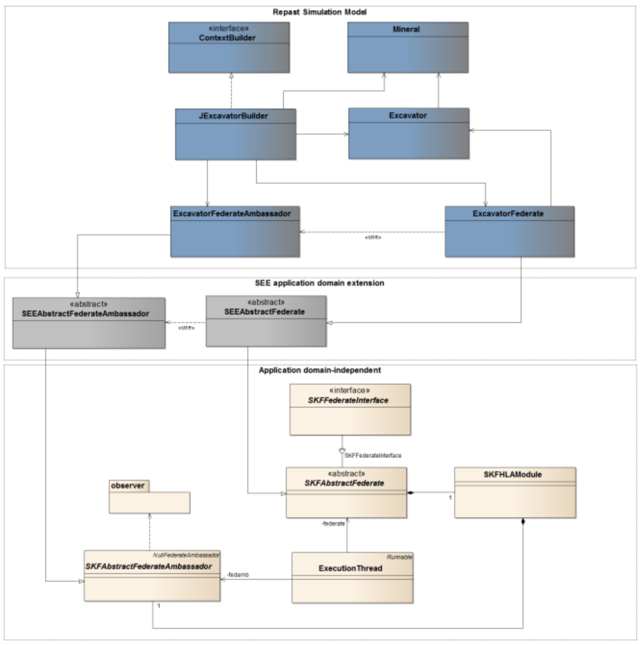

Figure 5: The architecture of the Excavator Federate.

5. DISCUSSION

The challenge presented to the ______ undergraduate team was how to create a simulation of a set of self-organizing excavators and their mining operation. The first task was to build the simulation in REPAST. The team had never done any kind of simulation before but had some experience in Java programming. However, REPAST has reference examples and documentation that could be used to support the teams' development. The second task was to then implement a Federate based on the agent-based simulation. In 2014's SEE event this proved to be an extremely challenging task for the team of that year, despite the support of the MSG team. The main problem was the lack of support material that the team could use. Very little existed at the time apart from Fujimoto's text book [6], Moller's introduction to the HLA [13], and "hints" in key articles on HLA issues (e.g. time management [5]). The experiences of using the DKF and its associated process had a great impact on the development time of the Federate as much of the HLA complexity was hidden away. The first attempt to understand the DKF and SEE-SKF was to produce a simple astronaut Federate. This essentially allowed a user to move a point across a space using a keyboard. The simulation just produced the coordinates of the astronaut. Following the SEE-SKF process, the team analyzed the data produced and required by the simulation to create the FOM (coordinates) and then used the SEE-SKF attributes and coders to map the FOM to the input/output in the simulation. The *SEEAbstractFederate* class was then implemented with its methods: The *configureAndStart* method to run, the *doAction* method to move the astronaut, and the *update* method to send the current coordinates of the astronaut. Finally, the Federate Ambassador was implemented and the main class was made available.

Developing the REPAST Federate proved more challenging as it was not clear at first how the above mapped to the environment and agent classes of the simulation. The previous section presented how this was achieved. The only unsatisfactory aspect of the implementation was the delegation of REPAST time management to the Federate Ambassador. REPAST has excellent time management facilities and this would have been better if the Federate Ambassador had been coordinated with REPAST time management. We expect this issue to be resolved in future work. To test the simplicity of using the SEE-SKF, the Excavator was linked to the UAV simulation developed by Liverpool University that hovers over the excavator grid. The UAV is equipped with a magnetometer that takes readings from the surface to identify interesting areas for excavation. To add this to the Excavator we assume that the UAV will produce ten sets of readings each update. *UavX1*, *UavY1*, *MagRead1*, etc. were then added to the FOM under a UAV class to allow the Excavator to subscribe to the attributes. Attributes and coders were added to the Excavator agent along with a *GetMap()* method that takes the current UAV updates and adds these to the excavator's map. Once the FOM had been agreed with Liverpool it took the team a remarkably short time update the agent and get the Excavator/UAV distributed simulation up and running.

6. RELATED WORK

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

For MATLAB/Simulink different packages and toolboxes are available for implementing HLA simulators such as the *Forwardsim HLA Toolbox for MATLAB* [23], 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* [11] that provides a library of Simulink blocks specifically designed for the integration of High Level Architecture services into Simulink models. It greatly simplifies Federation development and model reuse, as well as enables 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 a HLA Federate starting from a FOM file is the *PITCH Developer Studio* [15]. 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) [24]. This framework defines a universe of real-time simulation concepts to support the more informal concepts available at DMI with a HLA environment. The simulation framework provides mechanisms to simplify the development of real-time simulators.

Other HLA frameworks are based on GRID-computing infrastructure and they have become in recent times a popular way to model and study complex multi-actor systems by using the typical characteristics and capabilities of the GRID [25].

The HLA Development Kit and its software framework (DKF) differ from the above mentioned solutions 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). In addition, the DKF provides advanced facilities that allow keeping the code compact, readable and reliable. As an example, Java annotations are used to directly inject the structure of a HLA Federate in the Java code. These metadata are used by the core components of the DKF at run-time to inspect and check a HLA Federate according to its definition in the FOM. The above sketched capabilities showed their great benefits not only for expert HLA developers but also for HLA novice practitioners as were the undergraduates students involved in the SEE project.

7. CONCLUSION

The IEEE 1516–2010 – Standard for Modeling and Simulation High Level Architecture (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 the IEEE 1516 standard, is still a challenging task that requires both considerable development efforts and advanced skills and experience in distributed systems, simulation, middleware and software programming. This is due not only to the complexity of the IEEE 1516 standard but also to the lack of proper documentations and easy-to-use development framework. Indeed, developers have to spend a considerable amount of time to face with common HLA issues (such as the management of the simulation time, the connection on the RTI, and the management of common RTI exceptions), rather than to focus on the specific aspects of their own HLA Federates. This often results not only in an increase of the development time but also in a low reliability of the produced simulators. In this context, the paper has presented the HLA Development Kit, a general-purpose, domain-independent toolkit that aims at easing the development of HLA-based simulations by providing 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 lead by NASA and which involves several U.S. and European Institutions in the distributed simulation of a Moonbase. In particular, the HLA Development Kit and its DKF, has been specialized for the SEE application domain and exploited in the development of an Excavator Federate.

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 collaboration to create the Moonbase scenario. The SEE-SKF is therefore an example of how the DKF can be extended to be domain specific.

Future work on DKF includes the development of extensions to link DKF to REPAST. Previous work by _______ has included the development of a hybrid distributed Emergency Medical Simulation that consisted of different simulation systems that modelled Emergency Rooms and an ambulance service [1]. The expansion of DKF into this domain may provide an interesting education and research resource to easily develop distributed simulations of civil applications.

8. 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.

9. REFERENCES

[1] .

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

[3] Certi Project. The simulation toolkit, available from: http://savannah.nongnu.org/projects/certi, access February 2015.

[4] .

[5] Fujimoto, R.M. 1998. *Time Management in the High Level Architecture*. Simulation, Vol. 71, No. 6, pp. 388-400, 1998.

[6] Fujimoto, R.M. 2010. *Parallel and distributed simulation systems*. John Wiley & Sons.

[7] .

[8] Highsmith, J. 2002. *Agile Software Development Ecosystems.* The Agile Software Development Series, ed. A. Cockburn and J. Highsmith, Addison-Wesley.

[9] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification, IEEE Standard 1516-2010.

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

[11] MÄK, VR-Forces. The complete simulation toolkit, available from: http://www.mak.com/products/simulate/vr-forces.html, accessed February 2015.

[12] Modeling and simulation coordination office, available from: http://www.msco.mil/, accessed February 2015.

[13] Möller, B. 2013. THE HLA TUTORIAL v1.0. Pitch Technologies, Sweden.

[14] North, M.J., Collier, N.T., Ozik, J., Tatara, E., Altaweel, M., Macal, C.M., Bragen, M. and Sydelko, P. 2013. *Complex Adaptive Systems Modeling with Repast Simphony*. Springer, Heidelberg.

[15] Pitch Technologies. The simulation toolkit, available from: http://www.pitch.se, accessed February 2015.

[16] Portico Project. The simulation toolkit, available from: http://www.porticoproject.org/, accessed February 2015.

[17] Simulation Exploration Experience (SEE) project, available from: http://www.exploresim.com/, accessed February 2015.

[18] Simulation Interoperability Standards Organization 2010. *SISO-STD-006-2010 Standard for COTS Simulation Package Interoperability Reference Models*.

[19] Sokolowski, J.A., Banks, C.M. 2011. *Principles of modeling and simulation: a multidisciplinary approach*. John Wiley & Sons.

[20] .

[21] .

[22] .

[23] The Forwardsim HLA Toolbox for MATLAB, available from: http://www.forwardsim.com/products/hla-toolbox/ accessed February 2015.

[24] Villimann, O. 1999. CTO Project, Documentation, HLA Framework. Danish Maritime Institute.

[25] Xie, Y., Teo, Y.M., Cai, W., Turner, S.J. 2005. *Towards grid-wide modeling and simulation*.