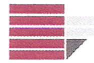
UNIVERSITÀ DELLA CALABRIA



UNIVERSITA' DELLA CALABRIA

Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica

Dottorato di Ricerca in

Information and Communication Engineering for Pervasive Intelligent Environments

CICLO

XXIX

DISTRIBUTION, REUSE AND INTEROPERABILITY OF SIMULATION MODELS IN HETEROGENEOUS DISTRIBUTED COMPUTING ENVIRONMENTS

Settore Scientifico Disciplinare ING-INF/05

Coordinatore: Ch.mo Prof. Felice Crupi

Firma ______________

Supervisore: Ch.mo Prof. Alfredo Garro

Firma ______________

Dottorando: Dott. Alberto Falcone

Firma ______________

Alberto Falcone

Distribution, Reuse and Interoperability of simulation models in heterogeneous distributed computing environments

Ph.D. Thesis

July 3, 2017

To my beloved girlfriend Rossana,
for her constant support and love.

Abstract (English)

Modeling and Simulation (M&S) is gaining a central role in several industrial domains such as automotive, e-science and aerospace, due to the increasing complexity of system requirements and thus of the related engineering problems. Specifically, M&S methods, tools, and techniques can effectively support the analysis and design of modern systems by enabling the evaluation and comparison of different design choices against requirements through virtual testing; this opportunity becomes even crucial when complete and actual tests are too expensive to be performed in terms of cost, time and other resources. Moreover, as systems result from the integration of components which are often designed and manufactured by different organizations belonging to different engineering domains (including mechanical, electrical, control, and software), great benefits can derive from the possibility to perform simulations which involve components independently developed and running on different and possibly geographically distributed machines. Indeed, distributed simulation promotes an effective cooperative, integrated and concurrent approach to complex systems analysis and design.

Although M&S offers many advantages related to the possibility of doing controlled experiments on an artificial representation of a system, its practical use requires to face with important issues such as, (i) difficulties to *reuse* simulation models already made; (ii) lack of rules and procedures by which to make *interoperable* models created with different simulation environments; and, (iii) lack of mechanisms for *executing* simulation models in *distributed and heterogeneous environments*.

Indeed, there are different simulation environments both commercial and noncommercial highly specialized that allow the design and implementation of simulation models in specific domains. However, a single simulation environment is not able to manage all the necessary aspects to model a system when it is composed of several components. Typically, the modeling and simulation of such systems, whose behavior cannot be straightforwardly defined, derived and easily analyzed starting from the behavior of their components, require to identify and face with some important research issues.

In this context, the Co-Simulation may be a viable solution, but currently there are some limitations to its applicability, that is why different research activities are focusing on the definitions of methods, models and architectures to enable the *distribution*, *reuse* and *interoperability* of simulation models made with different simulation environments. Although there are different approaches that differ in technology and application domain, they are not able to overcome in an integrated way the three issues mentioned above, but they offer solutions to a subset of them.

Two of the most mature and widely used solutions are the *IEEE 1516 - High Level Architecture (HLA)* standard, which allows the reuse of simulation modules and their execution in distributed environment, and the *Functional Mockup Interface (FMI)* standard, which permits to overcome the problem of interoperability among simulation models defined with different simulation environments. Moreover, great benefits can derive from the adoption of Model-Driven Engineering (MDE) approaches that allow to define and extend system models at different levels of abstraction, from the conceptual to the architectural level, up to their concrete realization.

From the analysis of these research and standardization efforts, emerges a scenario in which there are not any mature solutions able to address, in an integrated way, the above presented issues. The research presented in this Thesis aims to:

- define methods, models and techniques to address, in an integrated way, the issues of *distribution*, *reuse* and *interoperability* of heterogeneous simulation models through the integration of the international standards *IEEE 1516 - High Level Architecture (HLA)* and *Functional Mockup Interface (FMI)*.
- define a software framework capable of facilitating the development of distributed simulations whose modules are built with different simulation environments.
- define a Model-Driven method that allows to simulate system models on heterogeneous distributed simulation environments without dealing with platform specific issues.

The research activity has been carried out in collaboration with universities and international organizations such as the Simulation Interoperability Standards Organization (SISO) and the National Aeronautics and Space Administration (NASA).

Arcavacata di Rende (CS), July 2017 *Alberto Falcone*

Abstract (Italian)

La Modellazione e Simulazione (M&S) sta acquisendo un ruolo centrale in diversi settori industriali come quello automobilistico, aerospaziale, medico e farmaceutico, sotto la spinta della crescente complessità dei sistemi e dei relativi problemi ingegneristici. Gli attuali metodi, strumenti e tecniche di M&S consentono, attraverso l'applicazione di tecniche matematiche, statistiche ed informatiche, di riprodurre artificialmente un sistema sia esso esistente o in fase di realizzazione e di valutarne il comportamento qualitativo e quantitativo. Inoltre, è possibile condurre valutazioni e confronti tra le diverse scelte progettuali, sulla base dei requisiti che il sistema deve garantire, al fine di individuare la migliore soluzione. Questa opportunità diventa cruciale quando test completi sul sistema reale sono troppo costosi per essere eseguiti in termini di costi, tempo e/o altre risorse. Poichè gli attuali sistemi derivano dall'integrazione di componenti eterogenei spesso progettati e realizzati da diverse organizzazioni appartenenti a diversi domini come quello dell'ingegneria meccanica, energetica e del software, grandi benefici possono derivare dalla possibilità di effettuare simulazioni che coinvolgono componenti indipendenti in esecuzione su macchine distinte e, eventualmente, geograficamente distribuite. In effetti, la simulazione distribuita promuove un approccio cooperativo, integrato ed efficace per la progettazione e l'analisi di sistemi complessi. Sebbene la Modellazione e Simulazione (M&S) offra numerosi vantaggi legati alla possibilità di svolgere sperimentazioni controllate su una rappresentazione artificiale del sistema, il suo concreto utilizzo richiede la risoluzione di importanti problemi di ricerca quali, ad esempio: (i) difficoltà nel *riutilizzo* di modelli di simulazione; (ii) mancanza di regole e procedure attraverso il quale è possibile far *interoperare* modelli realizzati con differenti ambienti di simulazione; e, (iii) mancanza di meccanismi per l'*esecuzione* dei modelli di simulazione *in ambienti di calcolo distribuiti ed eterogenei.*

Oggigiorno, esistono diversi ambienti di simulazione sia commerciali che non commerciali altamente specializzati che consentono di progettare, realizzare ed eseguire modelli di simulazione in specifici domini applicativi. Tuttavia, un singolo ambiente di simulazione non consente di gestire tutti gli

aspetti necessari per definire un sistema quando esso è composto da diverse parti che coinvolgono domini differenti. Generalmente, la Modellazione e Simulazione di tali sistemi, il cui comportamento non può essere linearmente definito, derivato e facilmente analizzato partendo dal comportamento dei singoli componenti che lo definiscono, richiede l'identificazione e la risoluzione di alcuni importanti problemi di ricerca. In tale ambito, la Co-Simulation può essere una soluzione praticabile, ma attualmente esistono dei limiti nella sua applicabilità ed è per questo che uno dei principali aspetti su cui si stanno concentrando diverse attività di ricerca è quello riguardante la disponibilità di modelli e architetture in grado di consentire la *distribuzione*, il *riuso* e *l'interoperabilità* di modelli di simulazione realizzati con differenti ambienti di simulazione. Nonostante esistano diversi approcci che si differenziano per tecnologia e settore d'impiego, nessuno di essi è in grado di affrontare in modo integrato le tre problematiche sopra citate, ma offre soluzioni ad un loro sottoinsieme.

Tra le soluzioni più mature e diffuse troviamo lo standard *IEEE 1516 - High Level Architecture (HLA)*, che consente il riuso dei componenti/modelli e l'esecuzione in ambiente distribuito di scenari di simulazione, e lo standard *Functional Mockup Interface (FMI)* che permette di superare il problema dell'interoperabilità tra moduli realizzati con diversi ambienti di simulazione. Inoltre, grandi benefici possono derivare dell'adozione di formalismi e strumenti proposti nell'ambito dell'ingegneria del software e, in particolare, dell'ingegneria del software guidata dai modelli (*MDE, Model-Driven Engineering*). L'approccio Model-Driven consente di definire e/o estendere modelli del sistema a diversi livelli di astrazione, dal livello concettuale a quello architetturale, fino alla realizzazione concreta.

Dall'analisi dei suddetti sforzi di ricerca e standardizzazione, emerge uno scenario in cui non vi sono, di fatto, soluzioni mature in grado di risolvere in modo integrato le problematiche descritte. La ricerca presentata in questa tesi si propone di:

- Definire metodi, modelli e tecniche che consentano di affrontare in maniera integrata le problematiche di *distribuzione*, *riuso* ed *interoperabilità* tra ambienti di simulazione eterogenei attraverso l'integrazione delle soluzioni offerte dagli standard internazionali *High Level Architecture (HLA)* e *Functional Mockup Interface (FMI)*.
- Definire un software framework in grado di facilitare lo sviluppo di simulazioni distribuite i cui moduli sono realizzati con diversi ambienti di simulazione.
- Definire un metodo Model-Driven che consente di definire e simulare modelli di simulazione in ambienti di calcolo distribuiti ed eterogenei. In tal modo, lo specialista che realizza il modello di simulazione non dovrà esplicitamente occuparsi dai dettagli inerenti l'architettura di calcolo su cui il modello sarà eseguito.

L'attività di ricerca è stata condotta in collaborazione con importanti università e organizzazioni internazionali come la Simulation Interoperability Standards Organization (SISO) e la National Aeronautics and Space Administration (NASA).

Arcavacata di Rende (CS), luglio 2017 *Alberto Falcone*

Acknowledgements

I would like to express my sincere gratitude to my advisor Prof. Alfredo Garro for the continuous support of my Ph.D. study and related research, for his patience, motivation, and immense knowledge. I would like to thank him for encouraging my research and for allowing me to grow as a research scientist. I could not have imagined having a better advisor and mentor for my Ph.D. study.

A sincere thanks goes to the great people with whom I had the privilege to know and collaborate who became friends over the last three years. I would like to thank all the members of the Simulation and Graphics branch at NASA's Lyndon B. Johnson Space Center in Houston, Texas where I spent nine months of my Ph.D. program. Especially, my sincere gratitude goes to Dr. Edwin Zack Crues, Mike Red, Dan Dexter, and Judy Kimball for all the work done together and for their precious advices and inspiring discussions.

In addition, I would like to thank Prof. Andrea D'Ambrogio, Prof. Simon J. E. Taylor, Dr. Anastasia Anagnostou, Dr. Nauman Riaz Chaudhry and Dr. Andrea Giglio for all the work done together and for insightful conversations. I also want to thank my friend Dr. Andrea Tundis for his support during my Ph.D. study and for his advices in both research and life.

A note of thanks goes also to all the NASA staff involved in the Simulation Exploration Experience (SEE) project: Priscilla Elfrey, Stephen Paglialonga, Michael Conroy and Daniel Öneil, to Björn Möller and to all the members of SEE teams.

Furthermore, I am very grateful to my external reviewers, Dr. Umut Durak and Prof. Gregory Zacharewicz for their insightful comments, suggestions and engagement to making my Ph.D. thesis better.

I thank to all my colleagues and friends at the Institute for High Performance Computing and Networking (ICAR-CNR) and University of Calabria, for the stimulating discussions and for all the great time that we spent together.

A special thanks to my family: my parents Franco and Loredana for all of the sacrifices that they've made on my behalf, and to my brother Giovanni

Luca and sister Carmela for supporting me throughout writing this thesis and my life in general.

Last but not the least, I would like to express appreciation to my beloved girlfriend Rossana whose intelligence, simplicity and sweetness made me happier than ever before. Thank you very much for all her patience, encouragement and love.

July 2017 *Alberto Falcone*

Contents

1 Introduction . . . 1
1.1 Reference Context, Motivation and Objectives . . . 1
1.2 Main Results . . . 2
1.3 Thesis Overview . . . 4
1.4 Selected and Relevant Publications . . . 5

2 Background and Challenges . . . 9
2.1 Classical Methods and Techniques for Simulation . . . 9
2.2 Distributed Simulation . . . 12
2.2.1 Distributed Simulation Theory . . . 12
2.2.2 Distributed Simulation Standards . . . 15
2.3 High Level Architecture (HLA) . . . 18
2.4 Functional Mock-up Interface (FMI) . . . 21
2.4.1 FMI for Model Exchange . . . 24
2.4.2 FMI for Co-Simulation . . . 25
2.5 Conclusion . . . 26

3 Easing the Development of HLA-based Simulations . . . 27
3.1 Introduction . . . 27
3.2 Related work . . . 28
3.3 The HLA Development Kit Framework . . . 33
3.3.1 The HLA Development Kit Framework . . . 33
3.3.2 Architecture of the DKF . . . 34
3.3.3 Federate Behavioral Model . . . 38
3.4 Developing a Federate: Before and After . . . 41
3.4.1 The Excavator Agent-based Simulation . . . 42
3.4.2 Implementing a Federate without using the SEE-DKF . 43
3.4.3 The Development Process based on SEE-DKF . . . 45
3.4.4 Using the DKF to Develop the Excavator Federate . . . 46
3.5 DKF Quantitative Assessment . . . 47
3.6 Conclusion . . . 51

4 A model-driven approach to enable the simulation of complex systems on distributed architectures 53
4.1 Introduction 53
4.2 Model Driven Systems Engineering 55
4.3 MONADS: a model-driven method for distributed simulation . 56
4.4 MONADS: a running example 58
4.4.1 Reference scenario 58
4.4.2 Specify System Model.............................. 59
4.4.3 Define simulation environmental objectives and perform conceptual analysis 63
4.4.4 Design simulation environment 64
4.4.5 Develop and integrate simulation environment 69
4.4.6 Integrated Tool-Chain 72
4.5 Discussion and Related Work 75
4.6 Conclusion 76

5 On the integration of HLA and FMI 79
5.1 Introduction 79
5.2 Combining HLA and FMI 80
5.2.1 HLA for FMI 80
5.2.2 FMI for HLA 82
5.3 FMI for HLA: Integration Approaches 83
5.3.1 Adapter-based 83
5.3.2 Mediator-based 84
5.3.3 A case study...................................... 85
5.4 Conclusion 91

6 Further contribution on interoperability in distributed simulation 93
6.1 Introduction 93
6.2 SISO Space Reference FOM 94
6.2.1 The Space Reference FOM 95
6.2.2 Execution control of a compliant federation 103
6.2.3 Supporting Software 104
6.2.4 Conclusion 104
6.3 Java Space Dynamics Library (JSDL) 105
6.3.1 Introduction 105
6.3.2 Related works.....................................105
6.3.3 The JSDL project 107
6.3.4 Discussion and Results 117
6.4 Conclusion 118

7 Conclusions..119
7.1 Main contributions 119
7.2 Ongoing and Future Work 121

A Appendix . . . 123
A.1 The Simulation step of the Excavator agent in REPAST . . . 123
A.2 The *publishAndSubscribe()* method of the Excavator Federate . 123
A.3 The *updateAttributeValues()* method of the Exavator Federate 124
A.4 Receiving and decoding updates by the Excavator Federate Ambassador . . . 124
A.5 The FOM module of the Excavator Federate . . . 125
A.6 DataTypes in the FOM module of the Excavator Federate . . . 126
A.7 Time synchronization . . . 126

References . . . 129

List of Figures

2.1 Types of interdependencies between member applications. 13
2.2 Causality error scenario. 14
2.3 A generic HLA Federation. 19
2.4 The FMI for Model Exchange modality. 21
2.5 The FMI for Co-Simulation modality. 22
2.6 The FMI Model Description file [39]. 23
2.7 Piecewise-continuous variables of a FMU: continuous-time $x(t)$ and discrete-time $m(t)$. 24
2.8 An FMI event indicator that changes its domain. 25
2.9 An FMI Co-Simulation execution with one master that manages four simulation tools. 25

3.1 The architecture of a DKF-based Federation. 35
3.2 The architecture of the DKF. 37
3.3 The architecture of the SEE-DKF specific extension. 38
3.4 The architecture of a DKF-based Federate with the SEE Domain Extension. 39
3.5 The life cycle of a SEE Federate. 41
3.6 The phases of the IEEE 1730-2010 standard [55]. 46
3.7 The architecture of the Excavator Federate. 48

4.1 Overview of the MONADS method. 57
4.2 Border patrol simulation scenario. 58
4.3 SysML BDD diagram of the Border Patrol System. 61
4.4 SysML SD diagram of the border patrol simulation scenario. . . . 62
4.5 SysML SD Interaction2.reactiveBehavior. 62
4.6 SysML SD Interaction3.reactiveBehavior. 63
4.7 SysML-HLA annotated BDD diagram of the Border Patrol System. 65
4.8 Model-to-Model transformation inputs and outputs. 66
4.9 Model-to-Model transformation output: UML Class Diagram. . . 68

4.10 Model-to-Model transformation output: UML Sequence Diagram. 69
4.11 The architecture of the Border Patrol System simulation scenario based on the DKF framework. 73
4.12 The MONADS integrated tool-chain. 74

5.1 A FMU enriched with HLA features. 81
5.2 The Adapter-based integration approach. 84
5.3 The Mediator-based integration approach. 84
5.4 The reference Simulation Scenario. 86
5.5 Simulink model of the Lunar Lander. 87
5.6 Architecture of the HLA Hybrid Federate. 88
5.7 Lifecycle of the HLA Hybrid Federate. 89
5.8 Processing and updating sub-state of the Lander. 90
5.9 The HLA-FMI Federation. 90

6.1 Architecture of the SISO Space Reference FOM. 96
6.2 UML Class diagram of the *SISO_SpaceFOM_swithces* module. . . 96
6.3 UML Class diagram of the *SISO_SpaceFOM_datatypes* module. . 97
6.4 UML Class diagram of the *SISO_SpaceFOM_environment* module. 98
6.5 A Reference Frame Tree. 99
6.6 UML Class diagram of the *SISO_SpaceFOM_management* module. 101
6.7 UML Class diagram of the *SISO_SpaceFOM_entity* module. 102
6.8 The Architecture of the JSDL library. 108
6.9 The architecture of the Data Structure Service. 109
6.10 The architecture of the Frame Service. 110
6.11 Tree of ReferenceFrames. 111
6.12 The architecture of the Physical Entity Service. 113
6.13 The architecture of the Time Service. 114
6.14 The architecture of the Util Service. 116
6.15 The architecture of the Logging Service. 117

List of Tables

2.1 The IEEE 1278 standards. . . . 16
2.2 Other Distributed Simulation Standards. . . . 18
2.3 The IEEE 1516 standards. . . . 19

3.1 Comparison among HLA frameworks: General aspects. . . . 30
3.2 Comparison among HLA frameworks: HLA standard aspects. . . 31
3.3 Comparison among HLA frameworks: Functionalities. . . . 32
3.4 The @ObjectClass annotation. . . . 36
3.5 The @InteractionClass annotation. . . . 36
3.6 Comparison in building DKF and no-DKF based Federate. . . . 48
3.7 Cyclomatic complexity value ranges. . . . 49
3.8 Metrics at package level. . . . 50

4.1 Stereotypes of the *SysML4HLA* profile. . . . 64

5.1 HLA for FMI tags. . . . 81

1

Introduction

1.1 Reference Context, Motivation and Objectives

Modeling and Simulation (M&S) is gaining a central role in many application domains, ranging from energy to aerospace, due to the increasing complexity of system requirements and thus of the related engineering problems [12]. It represents one of the most important and effective methods for designing and studying both *Large-Scale System* and *System of Systems (SoS)*.

A *Large-Scale System* is defined in [70] as:

"A group of subsystems that are interconnected and organized so to form a whole system with clearly defined boundaries. Each subsystem is self-contained and independent from the other ones but it cannot work individually."

Whereas a *SoS* is defined in [62] as:

"A complex purposeful whole that is composed of complex, independent, self-organizing, component parts whose high levels of interoperability enable them to be recomposed into different configurations and even different systems of systems; is characterized by poorly-defined issues that significantly affect its behavior and make it difficult to understand; has ambiguous boundaries with critical contextual influences involving a mix of technical/non-technical factors; and exhibits emergent nonlinear properties. The complexity of a system of systems is a function of the number and diversity of its components and their linkages. System of systems linkages range from loosely to closely connected, but all systems of systems exhibit non-deterministic evolution and behavior and are cybernetically self-organizing."

In both the cases (*Large-Scale System* and *SoSs*), each component contributes to the functioning of the entire system but, in general, the behavior of the whole system cannot be straightforwardly derived from the behavior of its components [9].

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 [7]. 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 [35]. Despite the above sketched advantages, M&S has important challenges 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.*

To overcome these challenges, many research efforts are focusing on the definition of methods, models and techniques to support the *reuse* and *interoperability* of simulation models and their *execution on distributed computing environment.* Two of the most popular efforts going in these directions are *FMI (Functional Mock-up Interface)* [39] and *HLA (High Level Architecture)* [53]. However, each of the two mentioned proposals addresses part of the above issues and great benefits could derive from their combined exploitation [5, 6].

In this context, the research presented in this thesis has been focused on the definition of models, methods and techniques to address, in an integrated way, the issues of *reuse*, *distribution*, and *interoperability* among heterogeneous simulation models.

The research activity has been conducted in cooperation with the Software, Robotics, and Simulation Division (ER) - Simulation and Graphics Branch (ER7) of the NASA's Lyndon B. Johnson Space Center (JSC) in Houston (Texas, USA) [71] where I spent nine months of my Ph.D. program.

1.2 Main Results

Starting from the research objectives above described, the main contributions resulting from the research activity presented in this Thesis concern the definition of:

1. A software framework, which is called *HLA Development Kit*, that aims at facilitating the design and develop of distributed simulators compliant with the IEEE 1516.2010 - High Level Architecture (HLA) standard [53].

2. A Model-Driven method, which is called *MONADS*, that makes easier for Systems Engineers to design a Complex System and simulate it on a distributed simulation environment, without asking them to explicitly deal with the intricacies and difficulties of currently available standards and technologies.

3. Two methods, *HLA for FMI* and *FMI for HLA*, to address in an integrated way the issues of *reuse*, *distribution* and *interoperability* among heterogeneous simulation components through the integration of the functionalities offered by the HLA [53] and FMI [39] standards.

Concerning the first contribution, the *HLA Development Kit software Framework (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 [53, 99]. 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 to/from the HLA RTI; the publish, subscribe and updating of *ObjectClass* and *InteractionClass* elements [42, 30, 53]. 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) [71]. It has been successfully experimented in the SEE project since the 2015 edition [90]. In the 2016 edition, the Universities of Calabria (Italy), Bordeaux (France), Brunel (London, UK) and the Faculdade de Engenharia de Sorocaba, FACENS (Brazil) developed their SEE-Federates by using the Kit [29, 30, 94].

Concerning the second contribution, the result is a Model-Driven method called *MONADS (MOdel-driveN Architecture for Distributed Simulation)*. The MONADS method aims at facilitating the distributed simulation of complex systems, specified by using SysML [74], according to the Model-Driven Systems Engineering (MDSE) paradigm. Moreover, the HLA simulation code, generated starting from SysML models by a chain of *model-to-model* and *model-to-text* transformations, is based on the HLA Development Kit software Framework (DKF). This research activity is carried out working in cooperation with the Laboratory of Software Engineering, Department of Enterprise Engineering of the University of Tor Vergata (Rome).

In the end, regarding the third contribution. Although the HLA and FMI standards start from different objectives and are based on different techniques (see [39, 53, 56, 59]), they have several common features that can be jointly exploited so as to create a full-fledged solution to enable *reuse*, *interoperability* and *distributed execution* of simulation models. In this context, two approaches on how to fruitfully combine the HLA and FMI standards have been defined: (i) *HLA for FMI*, in which a FMU is enriched with HLA features and services; and, (ii) *FMI for HLA*, in which simulation modules that are available as FMUs are reused in a HLA simulation environment without modifying them.

This research activity has been experimented in the MODRIO (Model-Driven Physical Systems Operation) ITEA3 project [57].

1.3 Thesis Overview

This thesis is organized as follows. In Chapter 2 an introduction to the essential concepts of Modeling and Simulation (M&S) is provided. In particular, the classical methods and techniques for simulation are presented in Section 2.1. The basic concepts from the theory of M&S and an historical perspective on Distributed Simulation (DS) with its related standards are presented in Section 2.2. Section 2.3 presents the IEEE 1516 - High Level Architecture (HLA) standard for DS, whereas the Functional Mock-up Interface (FMI) standard is described in Section 2.4. The chapter concludes, in Section 2.5, with some considerations.

Chapter 3 presents the *HLA Development Kit Framework (DKF)*, a general-purpose, domain independent software framework that aims to ease the development of HLA-based simulations by letting the developers to focus on the specific aspects of their simulation rather than dealing with the common HLA functionalities. Specifically, an introduction to the framework is provided in Section 3.1. Some related works are discussed in Section 3.2. Section 3.3 presents the framework with particular focus on its architecture and main services. In Section 3.4, the development of a HLA Federate from scratch based on the DKF is exemplified in the context of the Simulation Exploration Experience (SEE) project and compared with its development without using the DKF. Quantitative analysis of the benefits provided by the DKF is presented in Section 3.5. Finally, conclusions are drawn and future research directions are delineated in Section 3.6.

In Chapter 4, *MONADS (MOdel-driveN Architecture for Distributed Simulation)*, a Model-Driven method that allows the automated generation of a HLA distributed simulation starting from the definition of a complex system specified in UML/SysML is presented in terms of its steps and transformations. In particular, a brief introduction to the method is provided in Section 4.1. Section 4.2 presents the fundamental methods and techniques behind MONADS, whereas Section 4.3 describes the steps and transformations. MONADS is exemplified in Section 4.4 through a simulation scenario. Related proposals are discussed in Section 4.5 and in Section 4.6, conclusions are drawn and future works delineated.

Chapter 5 presents in detail two methods to address, in an integrated way, the issues of *reuse*, *distribution* and *interoperability* among heterogeneous simulation components through the integration of the functionalities offered by the HLA [53] and FMI [39] standards. Specifically, in Section 5.1 the main

opportunities from the integration of these standards are described. Sections 5.2 describes the principles behind a joint exploitation of the two standards by focusing on the benefits that HLA could offer to FMI and vice versa (*HLA for FMI* and *FMI for HLA* respectively). Section 5.3 presents two concrete approaches for realizing the *FMI for HLA* integration perspective. Finally, some considerations are introduced and discussed in Section 5.4.

In Chapter 6 further research contributions on interoperability in distributed simulation are presented. In particular, Section 6.2 presents the *SISO Space Reference FOM* ongoing standard that aims at supporting the development of interoperable simulations of complex space systems and missions. Section 6.3 presents the *Java Space Dynamics Library (JSDL)* project, a low level space dynamics library that provides high fidelity models and algorithms needed for defining space systems, such as space vehicles and satellites.

In Chapter 7 the contributions of this thesis are summarized and ongoing and future works delineated.

1.4 Selected and Relevant Publications

Contents of this thesis were published and presented in international Workshops and Conferences as well as in research Journals; some selected and relevant publications are the following:

- **Falcone, A.**, Garro, A., Taylor, S.J.E., Anagnostou, A., Chaudhry, N.R., Salah, O.: Experiences in simplifying distributed simulation: The HLA Development Kit framework. Journal of Simulation, 10(37), pp. 1-20, Palgrave Macmillan UK (2016). ISSN: 1747-7786, DOI: 10.1057/s41273-016-0039-4.

- **Falcone, A.**, Garro, A.: Using the HLA Standard in the context of an international simulation project: The experience of the "SMASHTeam". In Proceedings of the 15th International Conference on Modeling and Applied Simulation, MAS '16, Larnaca, Cyprus, September 26-28, 2016, pp. 121-129, Dime University of Genoa (2016). ISBN: 978-889799970-6.

- Möller, B., Garro, A., **Falcone, A.**, Edwin, Z.C., Dexter, D.E.: Promoting a-priori interoperability of HLA-based Simulations in the Space domain: the SISO Space Reference FOM initiative. In Proceedings of the 20th International Symposium on Distributed Simulation and Real Time Applications, ACM/IEEE DS-RT '16, London, United Kingdom, September 21-23, 2016, pp. 100-107, IEEE Computer Society (2016). ISBN: 978-1-5090-3505-2, DOI: 10.1109/DS-RT.2016.15.

- **Falcone, A.**, Garro, A.: 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), part of the 2016 Spring Simulation Multiconference, SpringSim '16, Pasadena, California, United States, April 3-6, 2016, pp. 1-8, Society for Computer Simulation International (2016). ISBN: 978-1-5108-2319-8.

- Garro, A., **Falcone, A.**: Enabling the rapid prototyping of distributed simulations in the space domain. In Proceedings of the 11th Conference of Italian Researchers in the World, Houston, Texas, USA, February 26-27, 2016.

- Bocciarelli, P., D'Ambrogio, A., **Falcone, A.**, Garro, A., Giglio, A.: A model-driven approach to enable the distributed simulation of complex systems. In Proceedings of the 6th Complex Systems Design & Management, CSD&M '15, Paris, France, November 23-25, 2015, pp. 171-183, Springer International Publishing (2016). ISBN: 978-3-319-26109-6, DOI: 10.1007/978-3-319-26109-6_13.

- Anagnostou, A., Chaudhry, N.R., **Falcone, A.**, Garro, A., Salah, O., Taylor, S.J.E.: Easing the development of HLA Federates: the HLA Development Kit and its exploitation in the SEE Project. In Proceedings of the 19th the International Symposium on Distributed Simulation and Real Time Applications, ACM/IEEE DS-RT '15, Chengdu, China, October 14-16, 2015, pp. 50-57, IEEE Computer Society (2015). ISBN: 978-1-4673-7822-2, DOI: 10.1109/DS-RT.2015.18.

- Anagnostou, A., Chaudhry, N.R., **Falcone, A.**, Garro, A., Salah, O., Taylor, S.J.E.: A Prototype HLA Development Kit: Results from the 2015 Simulation Exploration Experience. In Proceedings of the 3rd ACM Conference Principles of Advanced Discrete Simulation, ACM/SIGSIM-PADS '15, London, United Kingdom, June 10-12, 2015, pp. 45-46, Association for Computing Machinery, Inc (2015). ISBN: 978-1-4503-3583-6, DOI: 10.1145/2769458.2769489.

- **Falcone, A.**, Garro, A.: On the integration of HLA and FMI for supporting interoperability and reusability in distributed simulation. In Proceedings of the Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, DEVS 2015, part of the 2015 Spring Simulation Multi-Conference, SpringSim '15, Alexandria, Virginia, United States, April 12-15, 2015, pp. 9-16, Society for Computer Simulation International (2015). ISBN: 978-1-5108-0105-9.

- **Falcone, A.**, Garro, A., Longo, F., Spadafora, F.: Simulation Exploration Experience: A Communication System and a 3D Real Time Visualization for a Moon base simulated scenario. In Proceedings of the 18th International Symposium on Distributed Simulation and Real Time Applications, ACM/IEEE DS-RT '14, Toulouse, France, October 1-3, 2014, pp. 113-120, IEEE Computer Society (2014). ISBN: 978-1-4799-6143-6, DOI: 10.1109/DS-RT.2014.22.

- **Falcone, A.**, Garro, A., Tundis, A.: System Dependability Analysis through Platform-independent Simulation Models. In Proceedings of the International Workshop on Applied Modeling and Simulation, WAMS '14, jointly held with the NATO CAX FORUM, Istanbul, Turkey, September 16-19, 2014, pp. 27-36, Dime University of Genoa (2014). ISBN: 978-88-97999-46-1.

- **Falcone, A.**, Garro, A., Tundis, A.: Modeling and Simulation for the performance evaluation of the on-board communication system of a metro train. In Proceedings of the 13th International Conference on Modeling and Applied Simulation, MAS '14, Bordeaux, France, September 10-12, 2014, pp. 20-29, Dime University of Genoa (2014). ISBN: 978-889799934-8.

Publications accepted but not published yet:

- **Falcone, A.**, Garro, A., Anagnostou, A., Taylor, S.J.E.: An Introduction to Developing Federations with the High Level Architecture (HLA). Accepted at the 2017 Winter Simulation Conference, WSC '17, Las Vegas, Nevada, United States, December 3-6, 2017, IEEE Computer Society (2017).

- Möller, B., Garro, A., **Falcone, A.**, Edwin, Z.C., Dexter, D.E.: On the Execution Control of HLA Federations using the SISO Space Reference FOM. Accepted at the 21st International Symposium on Distributed Simulation and Real Time Applications, ACM/IEEE DS-RT '17, Rome, Italy, October 18-20, 2017, IEEE Computer Society (2017).

- **Falcone, A.**, Garro, A.: A Java Library for Easing the Distributed Simulation of Space Systems. Accepted at the 16th International Conference on Modeling and Applied Simulation, MAS '17, Barcelona, Spain, September 18-20, 2017, Dime University of Genoa (2017).

2

Background and Challenges

This chapter provides an introduction to the essential concepts of Modeling and Simulation (M&S) keeping Distributed Simulation (DS) in focus. The purpose of this chapter is to familiarize the reader with the terminology and concepts used frequently in subsequent chapters.

2.1 Classical Methods and Techniques for Simulation

In many areas in the sciences, in particular the natural and engineering sciences, simulation 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.

The *simulation* term refers to [7]:

"The imitation of the operation of a real-world process or system over the time."

Simulation 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 [61]. 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.

Simulation involves different core concepts. These include *system and model*, *state*, *entity and attribute*, *event*, and *resource* [7].

A *system* is an organized collection of components that interact with each other according to a pattern in order to achieve some purpose or functionality. A component is any entity that is capable of exhibiting an input-output behavior through a well-defined interface.

A *model* is a representation of a *system*. The model must be enough complex to cover the characteristics of the system under study, but not too complex to be designed, implemented and simulated on computers.

The *state* of the system is a collection of variables needed to define and describe what happening within it at a given point of time. The determination of these variable values provides information about the system's performance.

An *entity* can be either *dynamic* or *static*. The first one, is an object moving through the system that request one of the services provided by the system *resources*, whereas a *static entity* is an object that serves other entities (e.g., In a bank branch, customers are *dynamic entities*, whereas the bank teller is a *static entity*) [7]. Each entity has a set of *attributes* that describe its characteristic.

An *event* represents an instantaneous occurrence that may change the *state* of the system. An *endogenous event* is an event that occurs within the system, whereas an *exogenous event* is an event that occurs outside the system.

A *resource* is an *entity* that provides service to *dynamic* entities and can serve one or more *dynamic* entity at the same time. A *dynamic* entity can request one or more units of a resource. If denied, the requesting entity joins a waiting queue; otherwise, the entity acquires the resource, use it for a period of time and then releases it.

Simulation models can be classified along three different dimensions [61]:

- *Static vs. Dynamic Simulation Models.* A *static* simulation model is a representation of a system at a particular time, or one that may be used to represent a system in which time plays no role. A *dynamic* simulation model represents a system as it evolves over time.
- *Deterministic vs. Stochastic Simulation Models.* If a simulation model does not contain any probabilistic components, it is called *deterministic*. In *deterministic* models, the output is "determined" once the set of input quantities and relationships in the model have been specified, even though it might take a lot of computer time to evaluate what it is. Many systems, however, must be modeled as having at least some random input components, and these give rise to *stochastic* simulation models. *Stochastic* simulation models produce output that is itself random, and must therefore be treated as only an estimate of the true characteristics of the model.
- *Discrete vs. Continuous Simulation Models.* The system state variables in a *discrete* simulation model remain constant over intervals of time and change value only at certain well-defined points called event times (e.g., a bank is an example of a *discrete* simulation model, since state variables such as the number of customers in the bank change only when a customer arrives or when a customer finishes begin served and departs). *Continuous* simulation model have system state variables defined by differential or difference equations giving rise to variables that may change continuously over time (e.g., an airplane moving through the air is an example of a

continuous simulation model, since state variables such as position an velocity change continuously with respect to time).

The decision whether to use a *discrete* or a *continuous* model for a particular system depends on the specific objectives of the study [106].
Nowadays, the number of companies using simulation is increasing rapidly. Many managers are realizing the advantages of using simulation techniques in their daily operations. Some of these advantages are [7, 61]:

- *Practical feedback and explore alternatives.* One of the primary advantages of simulation is to provide users with practical feedback when designing real world systems. This allows the designer to explore different design alternatives without actually physically building the system so as to determine the correctness and efficiency of a design before building it.
- *Hierarchical decomposition.* Simulation permits system designers to study a problem at several different levels of abstraction. By approaching a system at a higher level of abstraction, the designer is better able to understand the behaviors and interactions of all the high level components within the system. The lower level components may then be designed and subsequently simulated for verification and performance evaluation. Moreover, working at different levels of abstraction also facilitates rapid prototyping in which preliminary systems are designed quickly for the purpose of studying the feasibility and practicality of the high-level design.
- *Compress and expand time.* By compressing or expanding time, simulation allows developers to speed up or slow down phenomena so that they can investigate them thoroughly.
- *Training.* Simulation models represent an effective means for teaching or demonstrating concepts to users. Such models dynamically show the behavior and relationship of all the simulated system's components, thereby providing the users with a meaningful understanding of the system's nature. This characteristic makes simulation less expensive and disruptive than on-the-job learning.

Despite the advantages of simulation presented above, it has important challenges. Some of these are [7, 61]:

- *Model building may take a long time.* Building complex simulation models is a challenging task and requires considerable development efforts. Indeed, it requires expert engineers with deep knowledge and experience in systems theory, mathematical modeling and software development.
- *Simulation results may be difficult to analyze.* Since many simulations use probability distribution functions to describe a phenomenon or the behavior of a system, sometimes may be hard to determine whether an observation is a valid result of the system/phenomenon under study or not. As a consequence, the analysis of the simulation results may be expensive and time consuming.

- *Simulation may be used inappropriately.* Simulation is used in some cases when an analytical solution is possible or preferable.

2.2 Distributed Simulation

This section presents existing solutions for distributed simulation and the theory behind it. In the first part, the distributed simulation theory along with the principal approaches and techniques are presented, whereas the second one gives an overview of the currently available standards.

2.2.1 Distributed Simulation Theory

Distributed simulation refers to technologies that enable the execution of a simulation program on distributed computer systems containing multiple resources such as processors that are linked together through a communication network [37].

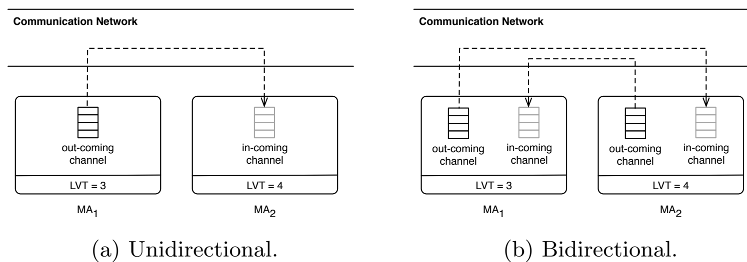
Initial research on distributed simulation has been conducted in the military domain where the main objective was on how to achieve model reuse via interoperation of heterogeneous simulation components. Interoperability is a key concept within distributed simulation, defined as *"the ability of two or more software components to act and cooperate together regardless of their programming languages, interfaces, operating systems and execution platforms"* [46]. There are different levels of interoperability between two components ranging from *no interoperability* to *full interoperability.* Concerning the technical point of view, many models for levels of interoperability have been successfully defined to find out the degree of interoperability between components [22]. Besides interoperability and reusability there are other benefits of distributed simulation, such as [37]:

- *Reduced execution time.* By splitting a large simulation computation into many sub-computations, and executing these concurrently across different computers can reduce the execution time.
- *Geographical distribution.* Executing the simulation program on a set of geographically distributed computers enables one to create virtual worlds with multiple participants that are physically located at different sites. Participants involved in the simulation scenario can interact with each other as they were located together at the same site.
- *Integrating simulators from different vendors.* Rather then porting heterogeneous programs from different vendors to a single computer, it may be more cost effective to connect them together, each one executing its task on a different computer.
- *Fault tolerance.* Utilizing multiple computers increase tolerance to failures, accordingly if a computer goes down, it may be possible for another one to pick up the work of the failing machine, allowing the simulation computation to proceed despite the failure.

Distributed simulations are composed of a number of member applications operating on different processors where each is responsible of a part of the whole model [110]. Typically, all interactions among member applications are based on time stamp order event messages and each Member Application (MA) contains input and output channels with associated FIFO (First in, First out) queues. These queues have functionally for sending messages by a MA and buffering messages received from other ones.

Each MA has a local simulation clock, called Local Virtual Time (LVT), that controls the simulation time during the simulation execution only for that. In addition, the theory of distributed simulation defines the concept of Channel Clock (CC) that represents the time of the last message received along a given channel. CC with value 0 means that no message has been received by means of that channel [110].

Two types of interdependencies between MAs can now be easily distinguished: *unidirectional* and *bidirectional*. In the *unidirectional* interdependency (see Figure 2.1a), the first MA produces and provides data to the second one that receives and consumes it. Since the second MA does not provide any data to the first one, it only have an in-coming channel, whereas the fist one only have an out-coming channel. There is no feedback between these two member applications and due to the absence of it they can be executed independently as a sequential process. In the *bidirectional* interdependency (see Figure 2.1b), MAs have an in-coming and out-coming channel for managing data because both produce and consume data to/from each other. In this case, MAs cannot be executed independently as a sequential process, because future events of each of them might depend on events generated by the another one. A feedback at any simulation time between MAs is required so as to avoid errors in the simulation.



(a) Unidirectional. (b) Bidirectional.

Fig. 2.1: Types of interdependencies between member applications.

Since events might occur at any time, it is necessary that a MA always selects and processes the most imminent event from the event queue (i.e., the event with the smallest timestamp) so as to avoid that the event under processing modifies state variables used by previous events leading in turn to

the situation in which a future event affects a past one [37]. An error resulting from out of order event processing is called *causality error*. Figure 2.2 presents a *causality error* scenario in which three MAs are processing events not in correct order.

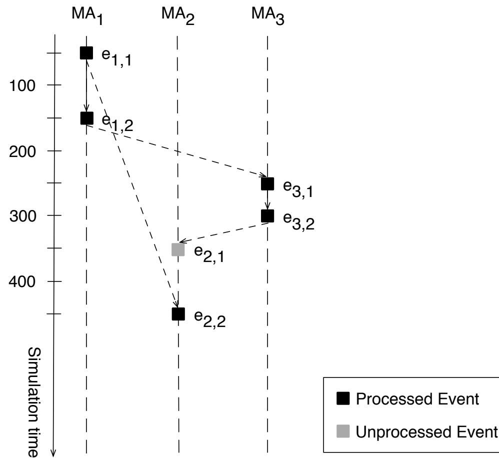


Fig. 2.2: Causality error scenario.

The scenario in Figure 2.2 shows several events processed and transmitted among the MAs during a distributed simulation. The member application MA_1 receives an event with timestamp 50 ($e_{1,1}$). When MA_1 executes $e_{1,1}$, it generates two new events: $e_{1,2}$ with timestamp 150 and $e_{2,2}$ with timestamp 450 that will be processed by MA_1 and MA_2, respectively. Upon receiving the event $e_{2,2}$, since MA_2 doesn't have events before simulation time 450, it processes the event so as to reduce the execution times of the whole simulation. The execution of the event $e_{1,2}$ by MA_1 generates another new event $e_{3,1}$ with timestamp 250 that will be processed by MA_3. When MA_3 receives and processes $e_{3,1}$, the event $e_{3,2}$ with timestamp 300 is generated, which in turn gives rise to a new event $e_{2,1}$ with timestamp 350 for MA_2. When MA_2 receives the event $e_{2,1}$, it has already processed the event $e_{2,2}$, this means that MA_2 received an event from the past in the future that might affect the results of the whole simulation.

To avoid causality errors, each MA must respect the *Local Causality Constraint* that guarantees *no causality errors occur if each MA involved in the distributed simulation handles events and messages in nondecreasing timestamp order* [110]. The general problem of ensuring that events are processed in the right order is referred in the distributed simulation theory to as the *Synchronization Problem* [37].

In order to execute the simulation on distributed computers in the right way, it is necessary that this kind of problem never happens. The simulation mechanisms must decide whether or not a given event or sequence of events can be executed concurrently with another event or events. To handle this dilemma two main synchronization approaches has been defined, *conservative* and *optimistic*.

Conservative approaches were introduced in the late 1970s by Chandy and Misra and Bryant [16, 18]. This approach satisfies the *Local Causality Constraint* through ensuring safe time stamp-ordered processing of simulation events within each MA. This means that the LVT of a MA can never exceed the CC of its incoming channel insuring that no causality errors occur. This approach requires that outgoing messages created by producers be transmitted in chronological order according to their timestamps. Then, If a MA contains at least one event message in each input channel, it can update its LVT to the minimum of all the timestamps and processes all the event messages with timestamp equal to LVT's value. Otherwise, if there is a channel that does not contain any event messages, the MA is blocked because it may receive an event message along that particular channel with timestamp smaller than its LVT. In this context, deadlock can occur when each MA is blocked. A solution to break the deadlock is to use *null* event messages. In particular, a *null* event message with timestamp t_1 from a MA_{sender} to a $MA_{receiver}$ guarantees that there will be no more messages from the sender with timestamp less than t_1 [110].

The *optimistic* approach doesn't avoid causality errors to occur, but fix them when detected. The LVT of a MA may run ahead of the CC of its incoming channel but if the MA receives an event with timestamp smaller than its LVC, a causality error is detected and the MA rolls back in simulated time and redoes its simulation to take into account the new event [110].

To integrate different simulation models in a distributed simulation execution it is necessary a common standard that defines distribution services. In the next Section the main standards are described in deep with particular focus on the IEEE 1516 - HLA standard [53].

2.2.2 Distributed Simulation Standards

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, such as *Distributed Interactive Simulation (DIS)* [109], *Aggregate Level Simulation Protocol (ALSP)* [111] and *High Level Architecture (HLA)* [53].

Distributed Interactive Simulation (DIS) is a standard for conducting Live, Virtual and Constructive (LVC) simulations across multiple computers [50]. It is used worldwide, especially by military organizations but also by other

agencies such as those involved in space exploration and medicine. DIS represents a message passing protocol that defines the messages and procedures for communication among the simulators. It defines a set of standard message packets, called *Protocol Data Units (PDUs)*, for publishing Entity State information, Fire and Detonate events, Emissions and Communications data, and Logistics information [110]. The development of DIS started in 1989 by the DoD with the aim to implement a system for military training, and became an IEEE Standard in 1993, under the official name IEEE 1278 [104]. The standard consists of five documents summarized in Table 2.1.

Table 2.1: The IEEE 1278 standards.

Standard	Description
IEEE 1278-1993	IEEE Standard for Distributed Interactive Simulation - Application Protocols, approved by IEEE on March, 18th 1993.
IEEE 1278.1-1995	IEEE Standard for Distributed Interactive Simulation – Application Protocols, approved by IEEE on September, 21st 1995.
IEEE 1278.1a-1998	IEEE Standard for Distributed Interactive Simulation – Application Protocols. Supplement to IEEE Std 1278.1-1995, approved by IEEE on March, 19th 1998.
IEEE 1278.1-2012	IEEE Standard for Distributed Interactive Simulation – Application Protocols. Revision of IEEE Std 1278.1-1995, approved by IEEE on December, 19th 2012.
IEEE-1278.2-1995	IEEE Standard for Distributed Interactive Simulation – Communication Services and Profiles, approved by IEEE on September 21st, 1995.
IEEE-1278.2-2015	IEEE Standard for Distributed Interactive Simulation – Communication Services and Profiles. Revision of IEEE Std 1278.2-1995, approved by IEEE on September 3rd, 2015.
IEEE 1278.3-1996	IEEE Recommended Practice for Distributed Interactive Simulation – Exercise Management and Feedback, approved by IEEE on December 10th, 1996
IEEE 1278.4-1997	IEEE Trial-Use Recommended Practice for Distributed Interactive Simulation – Verification, Validation, and Accreditation, approved by IEEE on July 20th, 1998

These documents contain all of the information about the structure of the various PDUs, whereas the values for the DIS Enumerations, which are too dynamic and fast-changing to include in a slow-changing IEEE Standard, are maintained by the Simulation Interoperability Standards Organization (SISO) in a separate document called *Enumeration and Bit Encoded Values for Use with Protocols for Distributed Interactive Simulation Applications* [15, 104]. The main characteristics of DIS are [110]:

- *No central management.* The DIS standard doesn't define any central computer that controls the entire simulation execution.
- *Autonomous simulation modules.* Simulation modules that participate to the simulation execution remain autonomous and are interconnected by information exchange via PDUs.
- *No time management.* The DIS standard doesn't define any time management mechanism. A simulation module advances its local simulation time according to a real-time clock and can join or leave the simulation execution without disturbing the other modules.
- *No data distribution management.* In order to reduce network traffic, the simulation models only transmit changes in behavior via PDUs. PDUs are

transmitted in a ring or on a bus and each simulation module uses PDUs that are directed at one of his entities.

Although DIS has several advantages, it is mainly applicable for developing simulators in the military domain [110]. Starting from DIS, the *Aggregate Level Simulation Protocol (ALSP)* has been defined. ALSP is a protocol designed to allow multiple, pre-existing war game simulation models to interact with each other over LAN and WAN networks [110]. The ALSP project started in early 1990 when the Defence Advanced Research Projects Agency (DARPA) sponsored the Massachusetts Institute of Technology Research Establishment (MITRE) to investigate the design of a general interface between large, existing, aggregate level combat simulation models to be used for military training simulations. Unlike DIS, where the focus was on linking individual weapon simulators for military training not considering the aggregate-level combat simulations, ALSP manages this aspect defined as collections of military assets such as tank battalions and troops [112].

In the ALSP protocol a distributed simulation is called a *Confederation* and it is composed of several simulation models, each called a *Confederate* [112]. In contrast with DIS, ALSP provides: (i) a *time management service*, which provides time management services to confederation members. It handles the connection/disconnection of Confederates to/from the Confederation and manages the synchronization between the local simulation time of a confederate with confederation time; and (ii) a *data management service*, which allows confederations to share information in a commonly understood manner. It provides a mechanism for filtering incoming messages so that only those of interest are received by a confederate [104, 112].

As described above, DIS was designed to support the interoperability of entity level combat simulation models, whereas ALSP was developed to overcome the limitations of DIS and support the interoperability of aggregate level combat simulation models. Both DIS and ALSP are designed to develop simulators in the military domain.

In 1993, the DoD started the developing of a general purpose architecture for distributed computer simulation systems. The design and development of this general solution was conducted by the U.S. Modeling and Simulation Coordination Office (M&S CO) that ended in the 1995 with the definition of the *High Level Architecture (HLA)* standard [37].

Because of the huge diversity among the needs of the distributed simulation user community, there was necessary to avoid superfluous constraints on how distributed simulation environments were developed and executed, and generally to provide a high degree of flexibility with regard to how supporting processes are structured. During the years, several engineering processes for distributed simulations have been defined such as the *Recommended Practice for Distributed Interactive Simulation - Exercise Management and Feedback* defined in the DIS standard [52] and the *Recommended Practice for High Level Architecture (HLA) - Federation Development and Execution Process*

(FEDEP) defined in the HLA standard [54]. In contrast to these approaches, which are based to a specific simulation architecture, SISO decided to extend FEDEP so as to support the development and execution not only of HLA simulations but also other distributed simulation independently of the distributed simulation architecture, and in 2010 the *IEEE 1730-2010 - Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP)* was published [55, 104] (see Table 2.2).

Table 2.2: Other Distributed Simulation Standards.

Standard	Description
IEEE 1730-2010	IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP), approved by IEEE on September 30th, 2010.
IEEE 1730.1-2013	IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process Multi-Architecture Overlay (DMAO), approved by IEEE on August 23rd, 2013

DSEEP revised the seven steps of the IEEE 1516.3-2003 standard and defines an higher level framework for the development and execution of distributed simulations [55]. DSEEP specifies a set of process and procedure that should be followed by users for the definition, development and execution of distributed simulations (see [55] for more details).

Next Section presents and describes more accurately the *High Level Architecture (HLA)* standard.

2.3 High Level Architecture (HLA)

HLA is an IEEE standard, defined under the official name IEEE 1516, for distributed simulation systems [53]. It was developed by the M&S CO in the period 1995-1996 [66], 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 *Federation*. A *Federation* is composed of several HLA simulation applications, each called a *Federate*. A *Federate* is defined as an application that conforms to the HLA standard and interacts with other Federates using the services provided by the *Run-Time Infrastructure (RTI)*, which implements the HLA standards [32, 104].

Figure 2.3 shows a generic HLA Federation with the services provide by the RTI.

Federates use the methods provided by the *RTIAmbassador* to invoke RTI services, while the RTI uses the *FederateAmbassador* methods to deliver in-

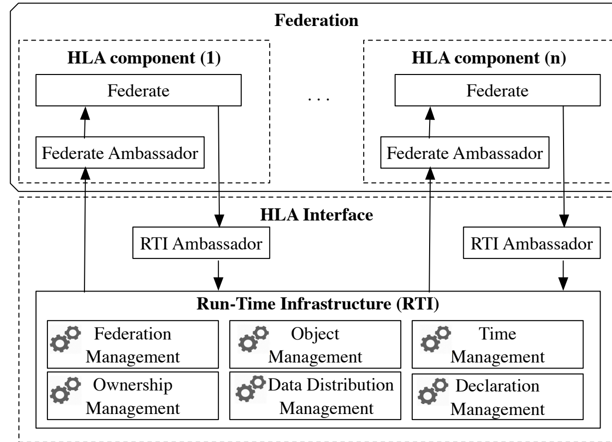


Fig. 2.3: A generic HLA Federation.

formation to a federate in a callback function mode. The IEEE 1516 standard is composed of five documents summarized in Table 2.3.

Table 2.3: The IEEE 1516 standards.

Standard	Description
IEEE 1516-2010	IEEE Standard for Modeling and Simulation (M&S) HLA Framework and Rules, approved by IEEE on August 18th, 2010.
IEEE 1516.1-2010	IEEE Standard for M&S HLA Federate Interface Specification, approved by IEEE on August 18th, 2010.
IEEE 1516.2-2010	IEEE Standard for M&S HLA Object Model Template (OMT) Specification, approved by IEEE on August 18th, 2010.
IEEE 1516.3-2003	IEEE Recommended Practice for HLA Federation Development and Execution Process, approved by IEEE on April 23rd, 2003.
IEEE 1516.4-2007	IEEE Recommended Practice for Verification, Validation, and Accreditation of a Federation: An Overlay to the High Level Architecture Federation, Development and Execution Process (HLA FEDEP), approved by IEEE on December 20th, 2007.

HLA standard is defined in three volumes [53, 104]:

- *HLA Framework and Rules Specification* (IEEE Std. 1516-2010). It outlines the elements of systems design and introduces rules that a Federate must follow in order to be compliant to the standard.
- *HLA Federate Interface Specification* (IEEE Std. 1516.1-2010). It introduces the functional interfaces that enable distributed simulation execu-

tion. This specification outlines the capabilities of the software infrastructure of HLA (i.e. RTI) and defines an interface specification between the RTI and Federates, which they use to exchange information to each other.
- *HLA Object Model Template (OMT)* (IEEE Std. 1516.2-2010). It presents the mechanism to specify the data model and defines the format of simulation data in terms of a hierarchy of object classes and interactions among the Federates running in a Federation execution.

Each Federate defines a so called *Simulation Object Model (SOM)* that is created in accordance with the Object Model Template (OMT). The SOM describes the shared information that a Federate can either provide or expect to/from other federates. The SOM is defined in terms of Objects (*ObjectClass*) and Interactions (*InteractionClass*) [53]. An *ObjectClass* is composed of a set of attributes whose values define the state of the object at any point during the simulation execution; whereas an *InteractionClass* defines an event that a Federate can generate or react to during the simulation. It is composed of a set of parameters that define its characteristics. These two kinds of information are exchanged through a publish/subscribe model by using the services provided by the RTI. A Federate can register an Object, which is an instance of an *ObjectClass*, and then change the values of its attributes. Other Federates that are subscribed to that *ObjectClass* can discover the related instances and then receive attribute value updates. The Interactions work in a similar way, except that them have associated a set of parameters and are not persistent (an interaction is "destroyed" after being consumed).

Furthermore, each Federation defines a so called *Federation Object Model (FOM)*, which describes the shared object, attributes, interactions and parameters for the whole federation. It represents a collection of Objects and Interactions coming from the individual SOMs of the federates that participate to the simulation execution.

The RTI represents a backbone of a Federation execution that provides a set of services to manage the inter-Federates communication and data exchange. They interact with RTI through the standard services and interfaces to participate in the distributed simulation execution. It supports the HLA rules and provides a set of services over the interfaces specified in the *Interface Specification*. More in details, the RTI services are grouped as follows:

- *Federation Management*. To this group belongs services to create and destroy a federation executions, and to keep track of the execution state of both Federation and the joined Federates that are running on an RTI.
- *Object Management*. This group provides services to manage how the running Federates can register, modify and delete object instances and to send and receive interactions once they have ownership of them.
- *Time Management*. This group includes services to manage the logical time of a Federation including delivery of time stamped data and advancing of the Federates time.

- *Ownership Management.* This group incorporates services useful to regulate acquiring and divesting ownership of registered objects;
- *Data Distribution Management.* To this group belongs services to manage the distribution of state updates and interaction information in a simulation executions. It limits and controls the volume of data exchanged during the simulation among Federates so as to relay data only to those requiring them.
- *Declaration Management.* This group gives services to manage data exchange among Federates according to a FOM by using a publish/subscribe scheme. For each Federate, this service keeps track of types of objects and interactions that it has both published and subscribed. In this way, the RTI is able to deliver data from publishers to subscribers.

2.4 Functional Mock-up Interface (FMI)

Functional Mock-up Interface (FMI) is a tool independent standard to support both Model Exchange (ME) and Co-Simulation (CO) of dynamic models using a combination of XML-files and compiled C-code (either compiled in DLL/shared libraries or in source code). The FMI development was initiated by Daimler AG within the MODELISAR ITEA2 project [56], with the goal to improve the exchange of simulation models between suppliers and OEMs (Original Equipment Manufacturers). The FMI standard defines two interfaces: *FMI for Model Exchange* and *FMI for Co-Simulation* [39].

The executable that implements the FMI interface is called a *Functional Mock-up Unit (FMU)*. The interface consists of 25 C-functions and type definitions (see [39] for more details) that are needed to instantiate, initialize and run the simulation of a FMU in a target simulator software such as Matlab Simulink [65] and OpenModellica [78].

The goal of the *FMI for Model Exchange* interface is to allow any modeling tool to generate C code or binaries representing a model in order to reuse/integrate it into another simulation environment. A model is described by differential, algebraic and discrete equations with time-, state- and step-events; these equations are specified in the C code and solved with the integrators of the environment where it is used.

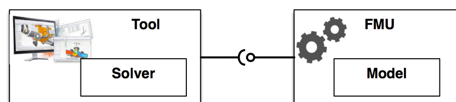


Fig. 2.4: The FMI for Model Exchange modality.

Figure 2.4 shows a diagram that exemplifies the FMI for Model Exchange modality.

The *FMI for Co-Simulation* is a standard interface for coupling two or more simulation tools in a Co-Simulation environment. It allows the solution of both time-continuous (models described by differential equations) and time-discrete (models described by discrete-time equations) systems.

Figure 2.5 shows a diagram that exemplifies the FMI for Co-Simulation modality.

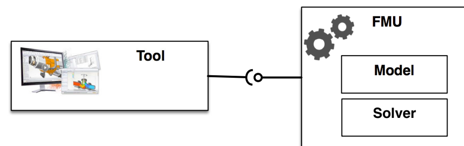


Fig. 2.5: The FMI for Co-Simulation modality.

The two interfaces are independent of the target environment, and have many parts in common with each other, this allows for the utilization of several instances of a Model and/or a Co-Simulation slave independently of the hosting environment.

Independently of the modality, a FMU is distributed in a compressed file with file name extension **.fmu* that contains:

- a *Model Description File*, which is an XML file containing the definition of all the exposed variables and other static information (see Figure 2.6).
- a *Dynamic Linked Library*, which is a shared library developed in C that implements the logic of the model. It represents a platform dependent executable binary file defined as a **.dll* file for Microsoft Windows operating systems; as a **.so* file for Linux operating systems and as a **.dylib* file for Mac OsX operating systems.
- *Further data*, such as icons, documentation files, source code, maps and tables, object libraries and dynamic link libraries that are used by the FMU during its execution.

The *Model Description* file defines both the input and output scalar variables with their attributes such as name, unit, type and initial value [39]. This information is used to set up and manage the structure and behavior of a FMU during a simulation execution. Its structure is shown in Figure 2.6.

If present the *ModelExchange* element, the FMU is based on FMI for Model Exchange and it includes the model or the communication to a tool that provides the simulation engine. Otherwise, the element *CoSimulation* is present and the FMU is based on FMI for Co-Simulation. In this case, the model and the simulation engine is included within it. At least one element

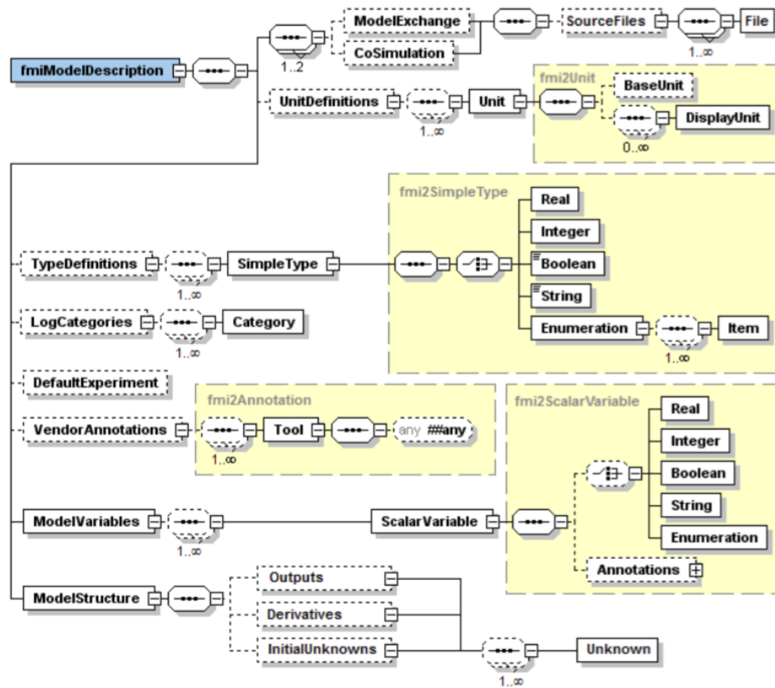


Fig. 2.6: The FMI Model Description file [39].

of *ModelExchange* or *CoSimulation* must be present to identify the type of the FMU. In the version 2.0 of the standard both elements may be defined; in this case, different types of models are included in the FMU.

The *UnitDefinitions* tag defines a global list of unit definitions and the *TypeDefinitions* attribute contains a list of type definitions.

The *LogCategories* tag, specifies a global list of log categories that can be set to define the level of the log information. In this way, a developer can check and track down both problems and errors occurred during the simulation of a FMU.

The *DefaultExperiment* element provides some default settings for the integrator, such as the stop time, step size and relative tolerance.

The *VendorAnnotations* tag defines vendor specific data but they can be ignored by other tools.

The *ModelVariables* element contains all the input/output variables of the model.

Finally, the *ModelStructure* attribute defines the structure of the model. Especially, the ordered lists of outputs, continuous-time states and initial unknowns.

2.4.1 FMI for Model Exchange

The main goal of the Model Exchange interface is to solve a system of hybrid Ordinary Differential Equations (ODEs) numerically. This kind of system is defined as a piecewise continuous system, which means that discontinuities can occur at different time instants $E = \{t_0, t_1, ..., t_n\}$ where $t_i < t_{i+1}$ and $i \in [0-|E|]$. These time instants are called *events* [39, 41]. If an event is known before hand, it is called a *time event*, otherwise if it is defined implicitly, it is called a *step event* or *state event* [39].

The “state” of a hybrid ODE is represented by:

- $x(t)$, which is a vector of time-continuous states defined as a continuous function inside every interval $t_i \leq t < t_{i+1}$;
- $m(t)$, which is a vector of time-discrete states that is constant inside each interval $t_i \leq t < t_{i+1}$ and only changes at events.

At every event instant t_i, variables might be discontinuous, and therefore two values at this time instant are defined, the “right” ($x(t_i)$, $m(t_i)$) and “left” ($x^-(t_i)$, $m^-(t_i)$) limit, see Figure 2.7.

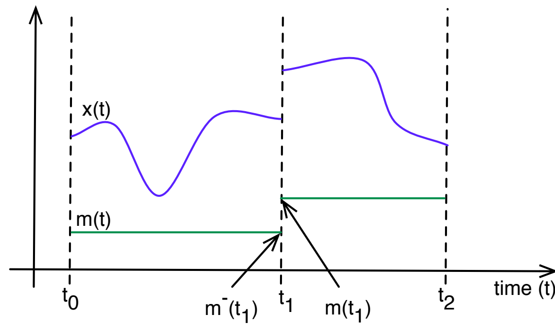


Fig. 2.7: Piecewise-continuous variables of a FMU: continuous-time $x(t)$ and discrete-time $m(t)$.

Events are defined by one of the following conditions:

- *External event.* The environment of the FMU triggers an event t_i because at least one discrete-time input changes its value, a continuous-time input has a discontinuous change, or a tunable parameter changes its value.
- *Time event.* t_i is defined in the previous event t_{i-1} either by the FMU or by the environment.
- *State event.* t_i occurs when an *event indicator* changes its domain from $z_i > 0$ to $z_i \leq 0$ or vice versa. All event indicators are piecewise continuous and are real numbers collected in a vector $z(t)$ (see Figure 2.8).

- *Step event.* t_i occurs when the *fmi2CompletedIntegratorStep* function returns $nextMode = EventMode$.

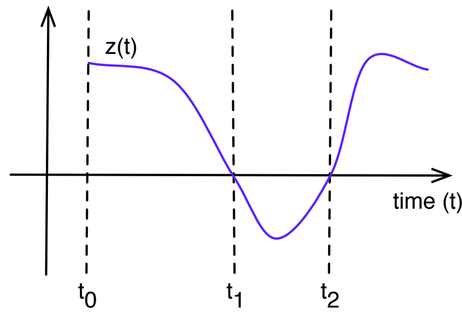


Fig. 2.8: An FMI event indicator that changes its domain.

An event is always triggered from the environment in which the FMU is running [39].

2.4.2 FMI for Co-Simulation

Co-simulation is a simulation approach for coupling time-continuous and time-discrete systems in a common simulation environment. These systems are generally developed by different organizations with different simulation tools and each subsystem manages a part of the whole system [39].

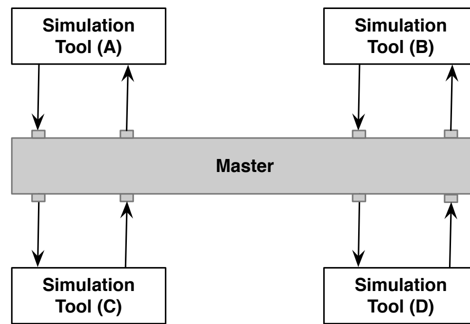


Fig. 2.9: An FMI Co-Simulation execution with one master that manages four simulation tools.

The FMI for Co-Simulation defines routines that manage communications between a *master* and *slaves*, each corresponding to one model or subsystem (see Figure 2.9). Each slave has a pre-defined set of inputs and outputs that are known by the master. The master is responsible for setting up, coordinating, and handling the slaves during their execution. The data exchange is limited to discrete communication points in time and the subsystems are solved independently between these communication points.

The FMI for Co-Simulation standard does not specify any FMI master algorithm, and thus its definition is in charge of the user.

2.5 Conclusion

This chapter provided an introduction to the essential concepts of Modeling and Simulation (M&S) keeping Distributed Simulation (DS) in focus. The classical methods and techniques for simulation have been described along with some core concepts, such as systems, models and simulation strategies.

After that, the chapter presented the notion of DS as an approach to solve integration of simulation models that are geographically distributed and connected to each other by a communication network [38]. In this context, it introduced the evolution of methods and techniques in the DS domain and described three advanced standards: *Distributed Interactive Simulation (DIS)* [109], *Aggregate Level Simulation Protocol (ALSP)* [111] and *High Level Architecture (HLA)* [53].

The *Functional Mock-up Interface (FMI)* as a tool independent standard to support both Model Exchange (ME) and Co-Simulation (CO) of dynamic models has been delineated along with its functionalities.

Nowadays, the increasing complexity and sophistication of modern systems makes their design, development and operation extremely challenging and therefore their analysis goes beyond the capabilities of the presented classical M&S methods, techniques and standards. So, the need of emerging M&S methods, models and techniques so to benefit from the available standards are nowadays even more required. In particular, many research efforts are focusing on the definition of new M&S methods, models and techniques to handle in an integrated way the *reuse*, *interoperability* and *execution on distributed computing environment* of heterogeneous simulation models. As a consequence, the present Thesis work has been conceived by starting from these considerations.

3

Easing the Development of HLA-based Simulations

Distributed simulation represents a solid discipline and an effective approach for handling the increasing complexity in the analysis and design of modern Systems and Systems of Systems (SoSs). The IEEE 1516 - High Level Architecture (HLA) is one of the most mature and popular standards for distributed simulation and it is increasingly exploited in a great variety of application domains due to its capabilities to enable the interoperability and reusability of distributed simulation components (see Section 2.3). However, the development of fully fledged simulation models, based on the HLA 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.

This chapter presents the *HLA Development Kit Framework*, a general-purpose, domain independent software framework that aims to ease the development of HLA-based simulations by letting the developers to focus on the specific aspects of their simulation rather than dealing with the common HLA functionalities.

The effectiveness of the proposed framework is shown in the context of the Simulation Exploration Experience (SEE), a project organized by SISO (Simulation Interoperability Standards Organization) and led by NASA that involves several U.S. and European Institutions.

3.1 Introduction

Over the years, large-scale systems have increased in complexity and sophistication since, in general, they are composed of several components, which are often designed and developed by organizations belonging to different engineering domains, including mechanical, electrical, and software. As systems get increasingly complex, their design and development become more difficult and therefore new Modeling and Simulation (M&S) techniques, methods and tools are emerging also to benefit from distributed simulation environments

[8, 41, 67, 1]. In this context, the HLA standard [53] attempts to handle this complexity by providing a specification of a distributed infrastructure in which simulation units can run on standalone computers and communicate with one another in a common simulation scenario.

Building complex and large distributed simulations systems using HLA is a challenging task and requires considerable development experience. Indeed, it requires expert engineers with knowledge and experience in distributed systems, simulation, middleware and software programming [32, 105]. The main problem is that the development and testing of HLA Federates are generally difficult, complex, and resource-intensive not only because of the complexity of the IEEE 1516 family standards but also due to the lack of proper documentations and ready-to-use examples. Moreover, developers have to spend a considerable effort to handle common HLA functionalities, such as the management of the simulation time, the connection on the RTI, and the management of common RTI exceptions. As a result, they cannot fully focus on the behavioral aspects of their own simulations.

This chapter describes an effective solution to enable the agile development of HLA-based simulation in which the common HLA aspects are clearly separated from the specific aspect of the Federates under development. Specifically, it presents a general-purpose and domain independent toolkit that consists of a Java-based development framework, called DKF, and an accompanying documentation comprehensive of a user guide and reference examples. Indeed, the DKF allows developers to focus on the specific aspects of their own HLA Federates rather than dealing with the common HLA functionalities that are managed by the DKF core components. Moreover, the DKF-based simulation code is independent of any specific RTI implementation and thus it can be executed on a desired RTI implementation, such as PITCH [82], VT/MÄK [64], PoRTIco [102], CeRTI [17]. The above outlined capabilities demonstrated their benefits not only for expert HLA developers but also for HLA novice practitioners (e.g., undergraduates students) involved in the Simulation Exploration Experience (SEE), an educational distributed simulation project organized by SISO (Simulation Interoperability Standards Organization) and led by NASA that involves several U.S. and European Institutions [90].

3.2 Related work

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

For Matlab/Simulink users different packages and toolboxes are available for implementing HLA simulators such as the *Forwardsim HLA Toolbox for Matlab* [98], 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* [64] that is based on the *Forwardsim HLA Toolbox for Matlab* and it provides a library of Simulink blocks specifically designed for the integration of HLA services into Simulink models. It greatly simplifies Federation development and model reuse, as well as enabling organizations to more efficiently participate in multinational simulations or implement distributed simulation models locally.

Another tool that enables developers to effectively manage the structure and assets of an HLA Federate starting from a FOM file is the *PITCH Developer Studio* [67]. This software allows developers 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)* [108]. This framework defines a range of real-time simulation concepts to support the more informal concepts available at DMI with an HLA environment. The simulation framework provides mechanisms to simplify the development of real-time simulators. *FEDEF* is another domain specific framework developed by the Defence R&D Canada-Atlantic that defines a set of APIs to support both the DMSO 1.3 and IEEE HLA 1516-2000 standards. It also provides different capabilities to simplify many programming tasks that are normally required when developing a Federate in the military domain [107].

Other HLA frameworks are based on GRID and Cloud computing infrastructures thus providing a way to model and study complex multi-actor systems by using the typical characteristics and capabilities provided by those infrastructures [82, 95, 113]. Moreover, both GRID and Cloud computing infrastructures offer to the users the possibility to access in a transparent way to the computing services remotely through the Internet, freeing them of the burdens associated with managing computing resources and facilities. These characteristics make HLA-based distributed simulations much more powerful and widely accessible to users who do not have ready access to high performance computing platforms [38, 96].

The HLA DKF differs from the above mentioned approaches in several aspects. In particular, differently from a proprietary and commercial solution that requires tool-specific knowledge and training, the HLA Development Kit is an open source project released under the open source policy Lesser GNU Public License (LGPL) and can be freely and easily customized and/or extended to cover and deal with both domain independent and domain specific aspects (as was the case with the SEE-specific extension presented in Section 3.4). In addition, the DKF provides advanced facilities that allow keeping the code compact, readable and reliable (see Sections 3.3 and 3.5).

Table 3.1 contains a comparison of some general aspects of the above mentioned HLA frameworks. Only *Forwardsim HLA Toolbox for Matlab* and *PITCH Developer Studio* supports all the versions of the HLA standard. The version 1.2.0 of the *HLA Development Kit (DKF)* currently supports both

the IEEE HLA 1516.1-2000 and IEEE HLA 1516.1-2010 whereas, it does not provide compatibility with the oldest version of the standard, which is the HLA 1.3. The *Danish Maritime Institute (DMI) HLA framework* and *FEDEF* support both the HLA 1.3 and IEEE HLA 1516.1-2000 versions.

Table 3.1: Comparison among HLA frameworks: General aspects.

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

The code generated by all the frameworks with the exception of the *PITCH Developer Studio* can be executed on the main HLA/RTI platform implementation such as PITCH [82], VT/MÄK [64], CeRTI [17] and PoRTIco [102]. Moreover, the code is in the Java language for the *Danish Maritime Institute (DMI) HLA framework* and *the HLA Development Kit (DKF)*; whereas it is in C++ for *FEDEF*. The *PITCH Developer Studio*, unlike the others, is able to generate the code both in Java and C++.

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

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

Since the *HLA Development Kit (DKF)* is an open source project it has a community of users that offer support to developers in using the DKF

framework. Developers can post problems, concerns and solutions by using the official forum [99]. Some of the others framework such as the *Danish Maritime Institute (DMI) HLA framework* and *FEDEF* do not provide support because these are specific for military applications and thus arguably less popular in the scientific community; whereas the *PITCH Developer Studio* gives support to programmers through its customer support service [82].

A comparison of some main aspects related to the HLA standard of the introduced HLA frameworks are described in Table 3.2. The here presented *HLA DKF*, unlike the others, is the only one that provides and manages the life cycle of a HLA Federate. As a consequence, a developer has only to define the specific behavior of its HLA Federate without worrying about low-level implementation details since the DKF manages them. Concerning the Simulation model, the *Danish Maritime Institute (DMI) HLA framework*, *FEDEF* and *HLA Development Kit (DKF)* provide functionalities to manage only time-stepped Federate; whereas the two commercial frameworks also support the event-driven model. Moving to the time management aspect, the two commercial HLA/RTI frameworks, which are the *Forwardsim HLA Toolbox for Matlab* and *PITCH Developer Studio*, are able to manage Federates with and without HLA Time Management; whereas the *HLA Development Kit (DKF)*, *Danish Maritime Institute (DMI) HLA framework* and *FEDEF* provide support only to HLA Time Management mechanisms based on Time Advance Grant (TAG) and Time Advance Request (TAR) [53].

Table 3.2: Comparison among HLA frameworks: HLA standard aspects.

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

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

Concerning the communication pattern, all the frameworks do not provide complex protocols over the basic Publish/Subscribe mechanism as defined in the HLA standard; but according to the roadmap of the *HLA Development Kit (DKF)* [99], different communication patterns over Publish/Subscribe are

under development and planned to be released in the future version of the DKF.

In Table 3.3 the functionalities offered by the above considered HLA frameworks are delineated and compared. All the frameworks provide functionalities to manage the simulation time, mechanisms to handle the connection (set-up, hold-up and close-up) of an HLA Federate to the RTI and features to facilitate publishing and updating of *ObjectClasses* and *InteractionClasses* on the RTI.

However, concerning the latest point, it is worth noting that the *HLA Development Kit (DKF)* is the only framework that uses the Java annotation mechanism to directly inject the structure of an HLA Federate in the Java code. These metadata are also used by the DKF core components at run-time to inspect and check if an HLA Federate is compliant with the related definition in the FOM (see Section 3.3). Moreover, the *HLA Development Kit (DKF)* offers support in managing time standard conversions between the wall-clock and simulation time [30], and some functions to check the status of the MS Windows Firewall because in some distributed simulations it is necessary that all the computers that are participating in the simulation scenario have the firewall disabled (e.g., the Simulation Exploration Experience (SEE) project). In addition, the *HLA Development Kit (DKF)* provides a logging service useful to track down any problems or errors occurred during the execution of an HLA Federate; this information is stored into the *dkf.log* file.

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

Table 3.3: Comparison among HLA frameworks: Functionalities.

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

3.3 The HLA Development Kit Framework

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

3.3.1 The HLA Development Kit Framework

The DKF is a general-purpose, domain independent framework, released under LGPL, which facilitates the development of HLA Federates [53]. Indeed, the DKF allows developers to focus on the specific aspects of their own Federates rather than dealing with the common HLA functionalities, such as the management of the simulation time; the connection/disconnection to/from the HLA RTI; the publishing, subscribing and updating of *ObjectClass* and *InteractionClass* elements. 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) [71]. It is fully implemented in the Java language and is based on the following three principles:

1. *Interoperability*, DKF is fully compliant with the IEEE 1516-2010 specifications; as a consequence, it is platform-independent and can interoperate with different HLA RTI implementations (e.g., PITCH [82], VT/MÄK [64], PoRTIco [102], CeRTI [17]).
2. *Portability*, DKF provides a homogeneous set of APIs that are independent from the underlying HLA RTI and Java version. In this way, developers could decide the HLA RTI and the Java run-time environment at development-time.
3. *Usability*, the complexity of the features provided by the DKF framework are hidden behind an intuitive set of APIs.

The DKF has been developed according to the concept of *Object HLA*, in this way, the development of HLA Federates could benefit also from the *Object HLA* features and functionalities provided by the Pitch Developer Studio [67] or similar IDEs.

Despite the availability of different HLA frameworks and IDEs; there are few training initiatives worldwide to promote their adoption. One of the most

important is an annual event, formerly named “Smackdown” and now renamed Simulation Exploration Experience (SEE) that has been organized, since 2011, by SISO in collaboration with NASA and other research and industrial partners [90]. The main objective of SEE is to provide undergraduate and postgraduate students with practical experience of participation in international projects related to M&S and, especially, to the HLA standard and compliant tools. The reference simulation scenario of the SEE Project concerns a human settlement called “Moonbase” composed of scientific equipment, storage buildings, rovers and other elements to allow astronauts to live and work on the Moon. The Modeling & Simulation Group (MSG) at Brunel University London has participated in the SEE Project since 2013. The group has investigated issues concerning the development and standardization of distributed simulation for industry and healthcare [93, 94], as well as hybrid Federations consisting of real-time, discrete-event and agent-based simulations [96].

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

To promote the adoption and experimentation of the HLA Development Kit and its DKF, it has been specialized in the *SEE HLA Development Kit* with the aim to ease the development of HLA Federates in the context of the SEE project [90]. The SEE-specific features (as an example, the possibility to easily implementation SEE Dummy and Tester Federates) aim not only at reducing the development efforts but also at improving the reliability of SEE Federates and thus reducing the problems arising during the final integration and testing phases of the SEE project [90]. Moreover, this SEE extension allows to prove how, starting from a domain independent core of the DKF, conceived for supporting the development of general-purpose HLA Federate, it is possible to easily add and integrate application-specific extensions for supporting the development of domain specific Federates [30, 42, 13].

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

3.3.2 Architecture of the DKF

The architecture of a DKF-based Federation is composed of three main layers (see Figure 3.1): (i) *Application Layer*, which contains the Federates that can interact with both the DKF and the HLA RTI by using their APIs; (ii) *DKF Layer*, which represents the core of the architecture and provides a set of domain independent APIs that are used to access the DKF capabilities;

and (iii) *HLA RTI Infrastructure*, which represents the RTI that host the Federation (e.g., PITCH [82], VT/MÄK [64], PoRTIco [102], CeRTI [17]). Some application-specific extensions of the DKF can be also introduced (e.g., the SEE-specific ones).

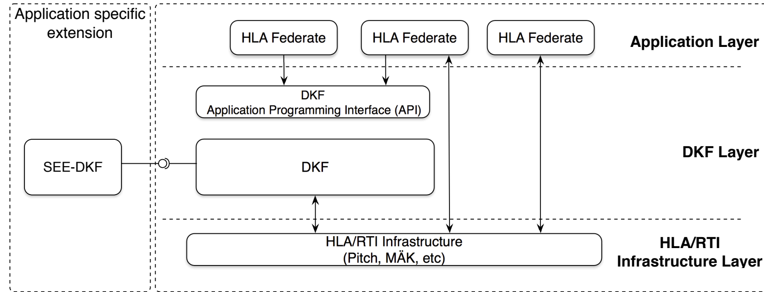


Fig. 3.1: The architecture of a DKF-based Federation.

The DKF provides a set of services that are independent both of application domains and HLA RTI implementations. Each service defines some Java classes and interfaces that implement specific functionalities. The DKF architecture is shown in Figure 3.2. The *Core Services* layer, represents the kernel of the DKF and provides a set of low level services to manage a DKF-based application. It is composed of eight services.

The *Data Management Service (DMS)* manages publishing, subscribing and the data updating of both an *ObjectClass* and an *InteractionClass* [53]. The DKF framework introduces a set of annotations to manage an *ObjectModel* (*ObjectClass* and *InteractionClass*), each of which covers a specific core concept involved in the HLA Object Model specification, and it is applicable to a piece of the program code so as to guide the core components of the DKF in managing *ObjectModels*. Annotations represent a form of metadata that provide data about a program that is not part of the program itself, thus they do not have direct effect on the operation of the code that they annotate [30, 42, 13]. Annotations have a number of uses, among them: (i) *Information for the compiler*, which can be used by the compiler to detect errors or suppress warnings; (ii) *Compile-time and deployment-time processing*, in which software tools can process annotation information to generate code, XML files, and so forth; and (iii) *Runtime processing*, which can be used to examine the structure of objects/classes at runtime.

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

Table 3.4: The @ObjectClass annotation.

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

InteractionClass instances. These two classes are used by the DKF core components at runtime to examine the structure of an *ObjectModel* instance. The structures of the above introduced annotation classes are summarized in Table 3.4 and Table 3.5, respectively.

Table 3.5: The @InteractionClass annotation.

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

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

The *Simulation Time Service (STS)* provides to developers some factory method that can be used to handle the two standard HLA logical time representations: *HLAinteger64Time* and *HLAfloat64Time* [53], and defines mechanisms for controlling the advancement of the time during the execution of a simulation. These mechanisms are coordinated with other components responsible for delivering information.

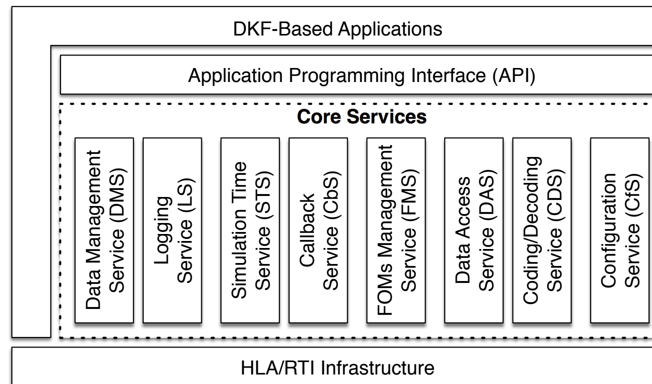


Fig. 3.2: The architecture of the DKF.

The *FOMs Management Service (FMS)* offers functionalities for retrieving and processing FOM files. More in detail, a set of components allow a DKF-based Federate to navigate the FOM tree and get the needed data by using a XPath expression [114].

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

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

The *Coding/Decoding Service (CDS)* defines all the standard HLA functionalities for coding and decoding both ObjectClass and InteractionClass instances [53].

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

Figure 3.3 shows the architecture of the *SEE-DKF*, a specific domain dependent extension of the DKF that provides some SEE domain specific services, which are used by the *core* components of the *DKF* to handle the main aspects related to a SEE Federation [90], such as transformations among *SEE Coordinate Reference Frames*, the publishing and subscribing of *PhysicalEntities*, and the management of Space FOMs [30, 31, 94]. The SEE-DKF architecture is organized in two main services sections.

The *Frame* section provides a set of services to manage basic space systems, and defines features for representing the position, geometry and characteristics of space objects such as planets and stars. Moreover, various algorithms to handle them are provided (conversions, propagations, etc.). It also defines data on the *International Celestial Reference Frames* [100] and includes algorithms and functionalities to manage them. Moreover, the *Frame* section has a factory

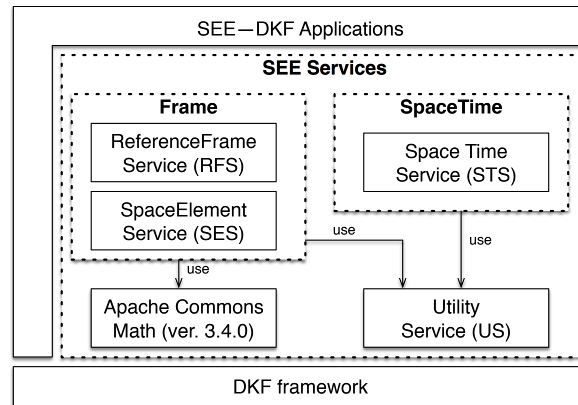


Fig. 3.3: The architecture of the SEE-DKF specific extension.

module that provides several predefined planet instances (e.g., Sun, Earth, Moon, etc.) with their specific characteristics (e.g., mass, volume, velocity, etc.) that developers can easily instantiate and use.

The *SpaceTime* section, defines mechanisms to handle epochs and dates that are commonly defined by specifying a point in a specific time scale. This section also provides many time standards such as Terrestrial Time (TT) and Universal Time Coordinate (UTC), and defines some epochs (e.g., Julian Epoch (JE), Modified Julian Epoch (MJE) and J2000 Epoch).

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

The *Apache Common Math library*, is a standard library of lightweight, self-contained mathematics and statistics components addressing the most common practical problems not immediately available in the Java programming language or commons-lang [97]. It is used by the Frame services to perform mathematics operations on arrays and matrices.

3.3.3 Federate Behavioral Model

The example architecture of a Federate created by using the capabilities of both the DKF and its SEE-specific extensions is shown in Figure 3.4 by using a UML Class Diagram; in the following its main classes are briefly described.

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

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

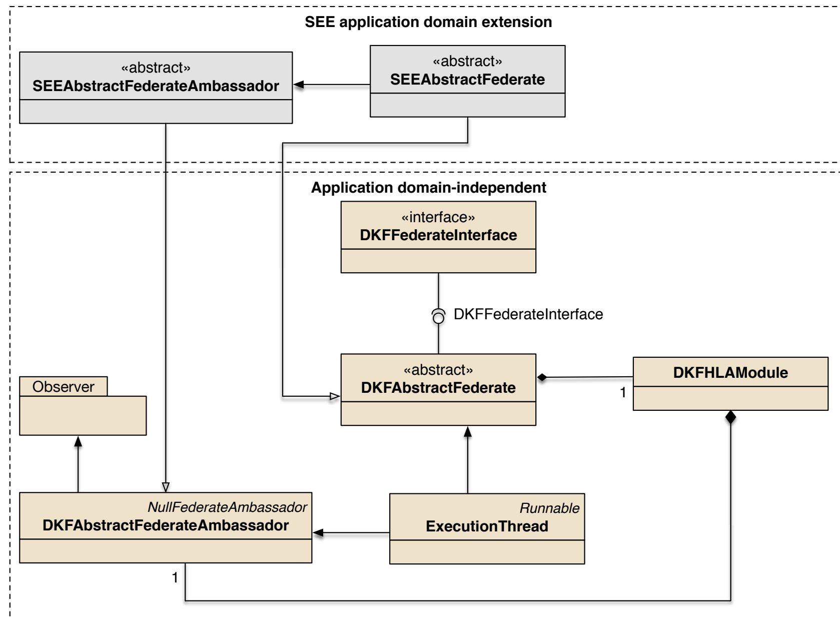


Fig. 3.4: The architecture of a DKF-based Federate with the SEE Domain Extension.

is worth noting that, in the SEE context, all the Federates are exclusively *time constrained* (can receive Time Stamp Order (TSO) messages) except the Environment Federate, provided by NASA and which leads the Federation execution, that is also *time regulating* (can send TSO messages) and acts as a Pacing/Clock Federate [37]; the DKF has been thus adapted to handle this situation.

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

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

The *DKFAbstractFederate* class also provides and manages the life cycle of a SEE Federate according to the behavioral model that is shown in Figure 3.5 through a UML Statechart diagram. As a consequence, a SEE working team has only to define the specific behavior of its SEE Federate without worrying about low-level implementation details since the DKF manages them. Specifically, the pro-active part of the behavior of a Federate is specified in the proactive composite state, which is accessed between a TAG (Time Advance Grant) and a TAR (Time Advance Request); whereas the re-active part of the behavior of a Federate is specified in the reactive composite state so as to indi-

cate how to handle the RTI callbacks about the interactions/objects that the Federate has subscribed. Please note that the current version of the DKF and its SEE specific extension only support the implementation of time-stepped Federates as this is the reference simulation model in the SEE project; however, ongoing efforts are geared to also supporting event-driven simulations [28].

With reference to the Federate life cycle depicted in Figure 3.5, in the *load configuration* state, the DKF loads the configuration parameters from a **.json* file. A transition to the *startup* state happens if the configuration parameters are valid and during the state transition a connection to the SEE Federation execution is performed. Otherwise, if the configuration parameters are invalid a state transition to the *shutdown* state is performed. In this latter state, all the resources engaged by the SEE-DKF classes are de-allocated through the dealloc resource operation, and then the life cycle terminates. In the *startup* state, the connection status is checked. If the connection is not established the lifecycle ends with a transition to the *shutdown* state, otherwise, three operations are done: (i) *locateRTI*, the parameters of the specific HLA/RTI implementation (e.g., PITCH [82], VT/MÄK [64], PoRTIco [102], CeRTI [17]) are located and loaded; (ii) *setRTIParameters*, the parameters loaded in the previous operation are set up according to the configuration parameters defined in a **.json*; and (iii) *connectOnRTI*, a connection to the Federation execution is performed.

A transition to the *initialization* state is performed if the connection has been properly established; in this state, the SEE Federate could perform additional operation for exchanging initialization objects before entering the *running* state (and thus the time advancement loop: waiting for a TAG → proactive state → make a TAR), as an example, the Federate could publish and subscribe some SEE information (e.g., *ReferenceFrames*, *InteractionClasses*, etc.). After that, the time management thread is activated and a transition to the *running* state is performed.

The *running* state is composed of two sub-states operating in an AND-decomposition fashion. The *proactive behavior sub-state* deals with the proactive part of the Federate behavior through three states: (i) *Waiting for TAG*: the DKF waits for the "TAG (Time Advance Grant) Callback" from the RTI. When the callback is received a transition to the proactive state is performed; (ii) *proactive state*: the "logical time" is updated, the pro-active behavior of the specific SEE Federate defined in the proactive composite state by the SEE working team is executed, and then a transition to the make TAR request state is performed; (iii) *make TAR request*: the DKF requests to the RTI the grant for the next "logical time". The *reactive behavior sub-state* deals with the reactive part of the behavior of the Federate: upon reception of RTI callbacks related to subscribed elements in the *Callback listener*, a transition to the *reactive* state is performed where the received information is handled through the execution of the reactive behavior of the specific SEE Federate defined in the *reactive composite state*. Note that, due to the AND decomposition

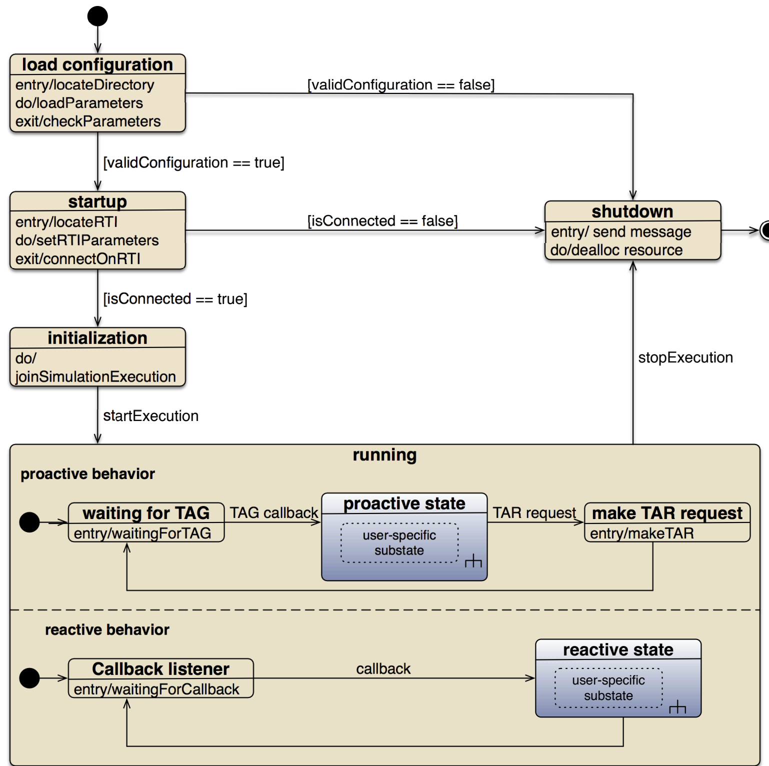


Fig. 3.5: The life cycle of a SEE Federate.

in the *running* state, its child states are parallel states; this implies that the *proactive behavior* and *reactive behavior* are concurrently executed. Since the current version of the DKF/SEE-DKF does not provide specific mechanisms to automatically handle the concurrency between the *reactive* and *proactive* tasks, it is in charge of developers manage this aspect through the use of the standard synchronization and concurrent mechanisms provided by the Java language and related JDK.

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.

3.4 Developing a Federate: Before and After

The previous section has shown that the DKF and its domain specific extensions can hide a significant amount of complexity related to the development

of HLA Federates. Based on this, to demonstrate that the DKF can be used to simplify Federate development a short case study is presented. The Brunel University team focused on the Excavator agent-based simulation, which was developed in REPAST SIMPHONY [72], as part of the SEE event in 2015 to show students how to create an agent-based simulation that can interoperate with other simulations in the lunar scenario. Students could explore how excavator "robots" could self-organize in the coordination of the extraction of lunar regolith materials and the degree to which REPAST could facilitate the study of these algorithms.

In this short case study, in order to focus on distributed simulation issues a single agent, with simplified input/output requirements, is presented. In particular, the simplified version of the Excavator simulation along with a discussion on how it could be implemented with and without the DKF are presented in the following Subsection.

3.4.1 The Excavator Agent-based Simulation

REPAST SIMPHONY is a free and open-source agent-based simulation environment [34, 72]. A REPAST agent-based simulation is created by using the *ContextBuilder* interface. In this class, the *environment* (i.e. the coordinate system that "places" the agents), the initial number of agents (and types/classes) that are located in the environment, and other basic settings are specified.

The attributes and methods of each agent are specified in an agents class. Each agent interacts with other agents and the environment via their methods. Time is managed in a REPAST simulation by the scheduler. A method can be annotated as being scheduled and will therefore include the frequency and priority that the method occurs. When a REPAST simulation runs, the simulation environment enters a cycle that calls the scheduler, the scheduler then runs the methods in priority order according to their frequency, and advances time at a specified time step until some terminating condition is met. In terms of distributed simulation, this REPAST simulation needs to be "plugged" into a Federate and synchronized with time advancement between the Federate and Federation.

A (simplified) single excavator agent explores its environment by coordinating with an Unmanned Aerial Vehicle (UAV). The UAV simulation was developed by Liverpool University [90]. The UAV slowly "flies" over the lunar surface detecting potentially interesting minerals. The UAV periodically broadcasts the results of its on-going survey to the excavator (in this case a single reading with the target coordinates), the excavator updates its local map and heads towards the target site. When the excavator reaches the site, it "mines" the mineral and adds it to its hopper that carries the excavated regolith. Once the hopper is full the excavator returns to its origin point and deposits the regolith material in a pile. The now empty excavator returns to where it left off and continues mining.

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

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

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

3.4.2 Implementing a Federate without using the SEE-DKF

To create a Federate of this agent-based simulation, the incoming and outgoing communication of the Excavator Federate (i.e., the information that the Federate will receive and send from/to other Federates) needs to be identified and specified in the Federate Object Model (FOM).

In the “normal” Federate implementation, the middleware was developed using poRTIco RTI implementation [102]. Generally, to create an HLA Federate from scratch, two classes need to be added to a model: (i) a concrete Federate class, here referred as Federate, that manages the life-cycle of the

Federate and defines the behavior of the model to be simulated. This class uses the mechanisms provided by the RTI Ambassador for sending information to the other Federates through the RTI [53]; and (ii) its Federate Ambassador class by implementing the *NullFederateAmbassador* interface, which defines a set of methods that define how the RTI sends information to the Federate in response to the changes in the state of the Federation execution [53]. A FOM XML schema needs also to be created. For our Excavator implementation, these two classes and the FOM file were based on the examples of the Federate and Federate Ambassador classes and modular FOMs that come with the poRTIco RTI [102].

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

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

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

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

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

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

Federate using the Time Advance Grant (TAG) method. The snapshot code in Appendix A.7 shows the above described methods.

An instance of the Excavator Federate, and subsequently an instance of Federate Ambassador too, is created when initializing the simulation in the REPAST Context Builder and is added in the same context as the agents. In this implementation, the update attributes method of the Federate must be modified to reflect the subscribed attributes. This method then can be called manually from the scheduled methods in the agent-based simulation (i.e. an update attribute method is called from the *moveExcavator()* method within the scheduled *step()* method in the Excavator class).

3.4.3 The Development Process based on SEE-DKF

As noted above, the SEE-DKF was developed as the space domain extension for the DKF. Rather than trying to follow examples from various RTI implementation, the DKF has a development process composed of four main steps that have a direct connection with the seven phases defined by the IEEE 1730-2010 (DSEEP) standard (see Figure 3.6) [55, 103]:

1. Build a *model* of the Federate that specifies: the *objects* that the Federate manages (as specified in the FOM), the *attributes* of these objects and the *coders* to handle such attributes. It is possible to use the basic coder set provided in the SEE-DKF or to implement new coders based on the SEE-DKF classes. This step can be traced back to the phases 1, 2 and 3 of the DSEEP standard.
2. Build a concrete Federate that specifies the *behavior* of the model defined at (1). It is required to extend the *SEEAbstractFederate* abstract class provided by the SEE-DKF and implement three methods according to the Federate life-cycle that is provided and completely managed by the SEE-DKF (see Figure 3.5). This step is linked to the fourth phase of DSEEP, specifically:
   (a) a method for initializing operations before entering the “running state” (the *configureAndStart()* method).
   (b) a method for specifying the active part of the behavior of the Federate (the *doAction()* method) executed between a TAR and a TAG.
   (c) a method (the *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 Ambassadors features are provided and managed by the SEE-DKF. The fifth phase of DSEEP covers this SEE-DKF step.

4. Implement a *main* class so as to instantiate and run the developed Federate. This last DKF-SKF step is associated with the phases 6 and 7 of the DSEEP standard.

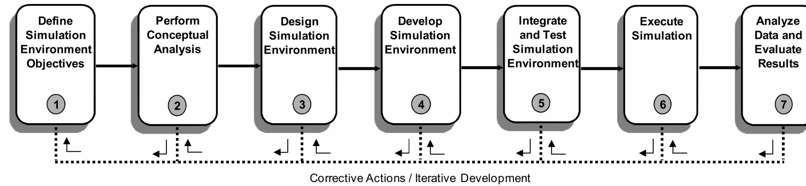


Fig. 3.6: The phases of the IEEE 1730-2010 standard [55].

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 [90].

3.4.4 Using the DKF to Develop the Excavator Federate

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

In step (1) a FOM that describes the input and output of the simulation was exploited. In this case the FOM represents the single Excavator object with UAVx, UAVy, EXCx and EXCy as noted above. All are *HLAinteger32BE* datatype. To begin the creation of the Federate, the Excavator class has been annotated to match the FOM as follows (see Section 3.3.2):

```
1 @ObjectClass(name = "PhysicalEntity.Excavator")
2 public class Excavator {
3     ...
4 }
```

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

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

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

```
setEXCx(getPointX());
setEXCy(getPointY());
```

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

In step (2), the *SEEAbstractFederate* class has been extended to create the *ExcavatorFederate* class. Within the *ExcavatorFederate* class the *configureAndStart()* method remained unchanged (i.e. it reaches the JSON config file and starts the Federation). The *doAction()* method is shown below. This method advances the agent-based simulation by first obtaining the current state (context) of the simulation, finding all agents (objects) and then “manually” running the *step()* method in the agents. In this example, the single excavator agent’s *step()* method is executed. It then calls updateElement(obj) to output the new state of the excavator Federate’s attributes.

```
protected void doAction() {
    for (Object obj : RunState.getInstance().getMasterContext()) {
        // update the excavator on RTI
        if(obj instanceof Excavator)
            ((Excavator) obj).step();
        super.updateElement(obj);
    }
}
```

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

The overall class diagram is shown in Figure 3.7.

Table 3.6 summarizes the main differences in implementing HLA Federations with and without the DKF. In SEE 2015, under guidance from the Brunel team, an undergraduate Computer Science student created the “distributed” side of the Excavator agent moderately quickly. This left more time for him to concentrate on the “simulation” aspects of the Excavator and its interactions with other simulations in the SEE event.

3.5 DKF Quantitative Assessment

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

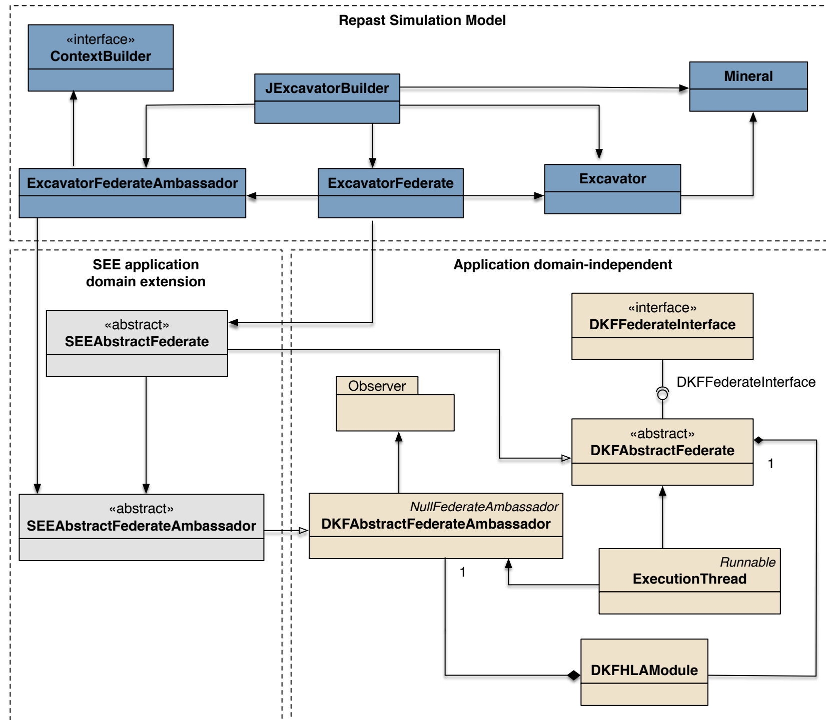


Fig. 3.7: The architecture of the Excavator Federate.

Table 3.6: Comparison in building DKF and no-DKF based Federate.

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

Software complexity is a primary topic in Software Engineering and has involved many researchers over the years. To analyze the quality of a software, it is necessary to measure the software source code in quantized form. Software metrics is one of the most traditional and effective way to measure the software system and they are related to various constructs like class, coupling, cohesion and inheritance. To evaluate the complexity of the source code of an SEE-DKF based HLA Federate, five standard metrics, which are proposed by various researchers, have been considered [116].

SLOC (Source Line of Code) is the most widely used metric for measuring the size of a software program. It is used to count the number of any line that is not a comment or blank line irrespective of the number of statements per line (also called *executable statements*). *SLOC* is easy to understand, fast to count, independent of the program language and it is a good metric to measure and evaluate the quantitative characteristics of a source code via the physics length. Typically, a method should be broken up if it has more than 50 lines of code; whereas a class should be split up and its functionalities delegates to other classes or sub-classes if it has over 750 lines of code. In this way, it is possible to increase both readability and maintainability of the software [116].

CCM (Cyclomatic Complexity Metric) is one of the most commonly used metric in many commercial and non-commercial tools for code complexity measurement. The *CCM* is based on graph theory and measures the complexity of a software module by analyzing its control flow structure. In particular, the control flow structure is represented as a graph *G(V, E)*, in which nodes (V) are used to represent decision or control statements; whereas edges (E) represent the control paths which define the program flow. The value of the CCM is the number of linearly independent paths and therefore, the minimum count of paths that should be tested, because any path can be expressed as a linear combination of some linearly independent paths. The *CCM* value gets an assessment of the complexity and indirectly of the maintainability of a software (see Table 3.7).

Table 3.7: Cyclomatic complexity value ranges.

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

HCM (Halstead Complexity Metric) measures the complexity of a software directly from source code analyzing its operators and operands. The operators are symbols used in expressions to specify the manipulations to be performed, whereas the operands are the basic logic unit to be operated. In particular,

HCM measures the logic volume of a software by using four numeric values: (i) the number of *non-repetitive operators (n1)*; (ii) the number of *non-repetitive operands (n2)*; (iii) the total *number of operators (N1)*; and, (iv) the total *number of operands (N2)*. This metric represents a strong indicator of code complexity and it is often used as maintenance metric and to evaluate development risk: higher values imply lower maintainability [116].

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

Finally, *NC (number of classes)* represents the number of concrete, abstract and interface classes. It provides an indicator of the extensibility of the software. Typically, the lower are the values of these metrics the lower is the complexity of the source code and thus higher should be the code compactness, readability and reliability [10, 116].

These six metrics are evaluated by considering the source codes of the *UNICOM* Federate [31].

Table 3.8: Metrics at package level.

Metric		**UNICOM Federate**	
		SEE-DKF	**Pitch Developer Studio**
NC		17	72
NF		94	784
SLOC		774	6186
CCM (average)		1,20	1,63
HCM	Number of distinct operators (n1)	22	38
	Number of distinct operands (n2)	312	1454
	Total number of operators (N1)	1337	4150
	Total number of operands (N2)	513	13217
	Software length (N)	1850	17367
	Software vocabulary (n)	334	1492
	Volume (V)	$1{,}584 \cdot 10^4$	$1{,}831 \cdot 10^5$
	Level (L)	0,1495	0,4784
	Difficulty (D)	47,13	172,71
	Programming Effort (E)	$7{,}469 \cdot 10^5$	$3{,}162 \cdot 10^7$
	Error Estimate (B)	5,28	61,03
	Programming Time (T)	$4{,}149 \cdot 10^4$	$1{,}756 \cdot 10^6$

More in detail, one source code is based on the SEE-DKF whereas the other one is that produced by the *Pitch Developer Studio* [67], which is a high quality IDE for HLA programming. The metrics have been calculated by using the *Google CodePro AnalytiX* tool, which is an application, developed by Google Inc., that allows developers to perform code measurement and com-

parison with user-defined programming standards and that is used by several large organizations, ranging from aerospace/defense to automotive/transport companies, to control their programming process [45].

Although the *DKF framework*, and its domain dependent extension SEE-DKF, does not cover all the IEEE 1516-2010 functionalities that are instead covered by the *Pitch Developer Studio*, the results reported in Table 3.8 show that the source code of an HLA Federate created by using the *DKF/SEE-DKF* is easy to manage and maintain even when compared to the same code produced by the *Pitch Developer Studio*.

Moreover, all the classes produced by using the *SEE-DKF* have *CCN* value less than twenty (see Table 3.8); as a consequence, these classes are easy to manage/extend by programmers (see [116] for a discussion).

3.6 Conclusion

HLA is undoubtedly one of the most mature and popular standard for distributed simulation. Due to its capabilities to enable the interoperability and reusability of distributed simulation components, it is increasingly exploited in a great variety of applications in both military and civil domains. However, the development of full-fledged simulation models, based on HLA, is still a challenging task.

In this context, the chapter has discussed an effective solution to enable the agile development of HLA-based simulations based on the *HLA Development Kit*, a general-purpose, domain independent toolkit that provides a software framework (the *DKF*), with related documentation, user guide and reference examples. The effectiveness of the DKF has been exemplified in the context of the Simulation Exploration Experience (SEE), an international project organized by SISO and led by NASA that involves several U.S. and European Institutions in the distributed simulation of a “Moonbase”. In terms of developing educational resources for HLA development, the *DKF* presents a solid foundation for future expansion. The SEE event is exciting in that students can create a wide variety of simulations and take part in an international project.

4

A model-driven approach to enable the simulation of complex systems on distributed architectures

The increasing complexity of modern systems makes their design, development and operation extremely challenging and therefore new Systems Engineering and Modeling and Simulation (M&S) methods, techniques and tools are emerging, also to benefit from distributed simulation environments. In this context, building and maintaining distributed simulations components, based on the IEEE 1516-2010 standard [53], is still a challenging and effort-consuming task.

To ease the development of full-fledged HLA-based simulations, the chapter proposes the MONADS method that, according to the Model-Driven Systems Engineering (MDSE) paradigm, allows system designers to generate the HLA-based simulation code starting from SysML models through a chain of *model-to-model* and *model-to-text* transformations. More in detail, the generated code is based on the *HLA Development Kit software Framework* (see Chapter 3). The effectiveness of the method is shown through a case study that concerns a military patrol operation, in which a set of drones are engaged to patrol the border of a military area, in order to prevent both ground and flight attacks from entering the area.

4.1 Introduction

Systems are constantly increasing in complexity and sophistication involving several heterogeneous components that are often designed and developed by organizations belonging to different engineering domains, including mechanical, electrical, and software. Moreover, moving from large-scale systems to Systems of Systems (SoSs), the involved several components that can be regarded as systems themselves. In particular a SoS is identified by five properties [63]: (i) *Independent components*, a SoS is composed of components that are independent and able to perform operations independently of one another; (ii) *Operational independence of the individual components*, components are typically individually gathered and integrated so as to create the whole SoS.

Each component preserves its characteristics and operates independently to achieve their own aims as well as the purpose of the whole SoS; (iii) *Geographic distribution*, the involved components are often geographically distributed and interact with each other so as to exchange information and knowledge by using a common network; (iv) *Emergent behavior*, each component contributes to the functioning of the entire SoS but, in general, the behavior of the whole SoS cannot be straightforwardly derived from the behavior of its components; (v) A SoS is continuously growing, changing and evolving. The development of this kind of systems is evolutionary over the time, and then its structure, functionalities, and objectives can be modified. In addition, during the life of a SoS, as new systems may join the SoS and other dynamically may leave it, its components and their relationships typically change. This increasing level of complexity makes the design, development and operation of modern systems extremely challenging. As a consequence, new Systems Engineering methods and techniques are emerging also to benefit from Modeling and Simulation (M&S) distributed simulation environments [37].

Building complex and large distributed simulations components, based on the IEEE 1516 standard, is usually a challenging task and requires considerable effort, not only in their development, but also for the cost of maintaining such components. On the development side, the building and testing of HLA Federates, is generally difficult, complex, and resource-intensive because of the complexity of the IEEE 1516 standard [53, 59], the lack of proper documentation, and the unavailability of ready-to-use examples. Moreover, developers have to spend a considerable effort to face with common HLA aspects, such as the management of the simulation time, the connection on the HLA/RTI, and the management of common RTI exceptions.

To ease the development of full-fledged HLA-based simulations, Model-Driven Software Engineering approaches, tools and techniques could be effectively exploited. It represents an approach to software design and implementation that addresses the rising complexity of execution platforms by focusing on the use of formal models [3, 49]. According to this paradigm, a software system is initially specified by the use of high-level models. Such models are then used to generate other models at a lower level of abstraction, which in turn are used to generate other models, until stepwise refined models can be made executable. One of the most important initiatives in implementing the Model-Driven Software Engineering principles is the Model-Driven Architecture (MDA) [75].

This chapter presents the fundamental methods and techniques behind the MONADS (MOdel-driveN Architecture for Distributed Simulation) method and introduces a Model-Driven approach to support the automated generation of a DKF-based HLA distributed simulation (see Chapter 3) starting from the definition of a complex system specified in UML/SysML [13].

MONADS aims at facilitating the distributed simulation of complex systems, which are specified by using SysML, according to the Model-Driven Software Engineering paradigm. Moreover, the HLA simulation code, generated

starting from SysML models by a chain of *model-to-model* and *model-to-text* transformations, is based on the *HLA Development Kit software Framework (DKF)* [30, 42]. The method is exemplified by considering the reference scenario that concerns a situation in which a set of drones are engaged to patrol the border of a military area, in order to prevent both ground and flight attacks from entering the area.

4.2 Model Driven Systems Engineering

For a long time, the design of a complex system has been based on systems engineering processes that exploited engineering data and text documents in different formats. Such document-based manual approach presents natural limitations, which have been addressed by the *Model Based Systems Engineering (MBSE)* approach, endorsed by the International Council on Systems Engineering (INCOSE), which defines MBSE as *"the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle"* [49].

In this context, the need for a language that provides the modeling capability required in the systems engineering domain emerges. To this purpose, SysML (Systems Modeling Language) is a UML-based language that is now considered the standard modeling notation adopted in the MBSE context [74].

The fundamental concept behind MBSE is that a model evolves over the system development lifecycle, until it becomes the build-to baseline. In the early phases of the lifecycle, the models have high levels of abstraction and are mainly used for decision making. As the system is developed, the level of detail increases until the models can be used for design. Finally, models are transformed yet again into the build-to baseline.

Therefore, the adoption of the MBSE approach results in many significant advantages, in terms of improved quality, enhanced communications, increased productivity, enhanced knowledge transfer, and reduced development risks. These improvements can be further enhanced by the use of novel approaches that increase the level of automation throughout the system lifecycle by focusing on models as the primary artifacts of development. The use of such an approach, which is denoted hereafter as *Model-Driven Systems Engineering (MDSE)*, enables a radical shift in terms of modeling activities, from a strictly contemplative use of models to a more productive and powerful model use. Metamodeling techniques and automated model transformations are key enabling principles introduced in the broader field of Model-Driven Engineering [3, 89]. MDSE includes these principles into the systems engineering domain, thus enhancing the aforementioned advantages of the MBSE approach. Various incarnations of Model-Driven Engineering principles have proposed different standards and tools claiming to support MDE. Among these, the MDA (Model-Driven Architecture) effort of the Object Management Group

has gained a special consideration in the software engineering community [75]. MDA-based software development is founded on the principle that a software system can be built by specifying a set of model transformations, which allow to obtain models at lower abstraction levels starting from models at higher abstraction levels.

To achieve such an objective, MDA has introduced a language for specifying technology neutral metamodels (or models that define modeling languages, i.e., models that describe other models), referred to as the Meta Object Facility (MOF) [76], and a standard for specifying model transformations, i.e., the Query/View/Transformation (QVT) standard [77]. A model transformation specified in QVT allows to automatically generate a target model, instance of a given MOF-based metamodel, from a source model, instance of the same or of a different MOF-based metamodel. In case the target model is of text type (e.g., code written in a given programming language), the MOFM2T (MOF Model To Text) standard [73] can be used to specify the relevant transformation.

4.3 MONADS: a model-driven method for distributed simulation

According to the context outlined in Section 4.2, the system development process is concerned with two different engineering domains. On the one hand, it is related to the system development domain, in which systems engineers deal with design and implementation issues. On the other hand, it addresses the simulation development domain, in which simulation engineers deal with system verification and validation issues by introducing distributed simulation-based analysis techniques. In this respect, the proposed method supports both system and simulation engineers, as depicted in Figure 4.1.

At the beginning, the system under study is specified in terms of a SysML model (e.g., block definition diagrams, sequence diagrams, etc.). According to the MDA terminology [75], such a model is referred to as the *platform-independent model (PIM)* of the system. At the system development level, the system engineer in charge of producing the system model is not concerned with any details regarding the simulation model and is strictly focused on the specification of a SysML-based system design model, starting from the system requirements.

The SysML model identifies the input of the sub-process that is related to the development of the distributed simulation. In this respect, according to the DSEEP (Distributed Simulation Engineering and Execution Process) standard [55], simulation engineers carry out a *conceptual analysis* of the required simulation and use the defined *SysML4HLA* profile to annotate the PIM in order to enrich such a model with the information required to derive the HLA-based simulation model (see Table 4.1). Specifically, the *SysML4HLA* profile allows to specify both how the system has to be partitioned in terms

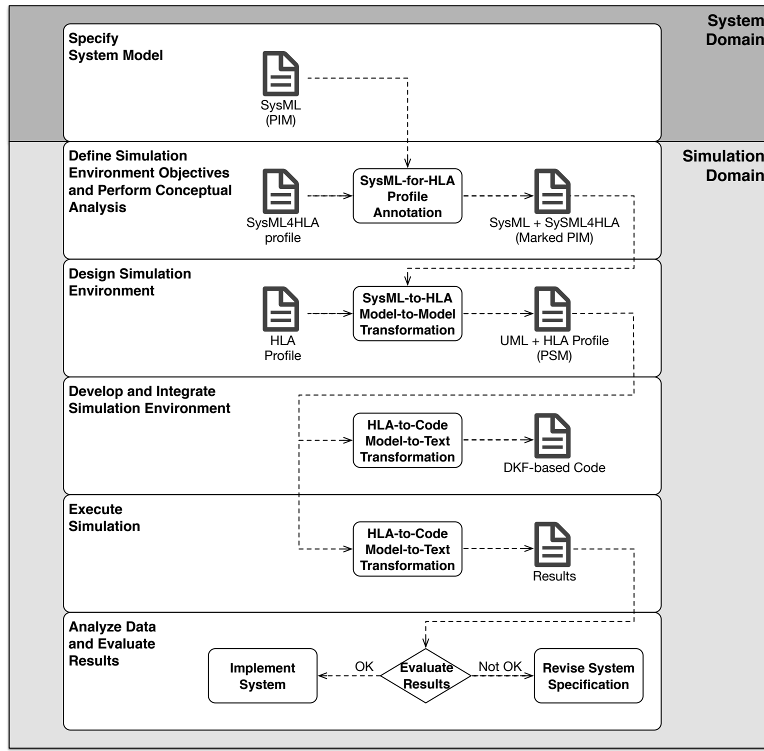


Fig. 4.1: Overview of the MONADS method.

of Federation/Federates and how system model elements have to be mapped to HLA model elements such as *Object classes* and *Interaction classes*.

Then, the *design simulation environment* step is executed. This step takes as input the marked PIM and executes the *SysML-to-HLA model-to-model* transformation in order to automatically obtain a UML model that represents the HLA application model. Such a model is annotated with the stereotypes provided by the *HLA profile* and, according to the MDA terminology, is referred to as the *platform specific model (PSM)*. The design simulation environment step also concerns with the discovery of existing Federates to be integrated in the distributed simulation.

The *develop and integrate simulation environment* step is then performed to execute the distributed simulation. The simulation code, specified by the use of the *HLA Development Kit software Framework (DKF)* (see Section 3.3), is generated through the execution of the *HLA-to-Code model-to-text* transformation [30]. This step also includes the coding activities needed to integrate the existing Federates identified in the previous step.

Finally, the distributed simulation is executed and the results are evaluated to check whether or not the predicted system behavior satisfies the user requirements and constraints. In the positive case, the validated SysML-based system specification can be used to drive the possible design and implementation of the system. Alternatively, the system specification has to be revised.

The next section introduces a running example that is used hereinafter to illustrate the details of the method steps. It shows the steps that go from the initial system specification down to the development of the distributed simulation code.

4.4 MONADS: a running example

This section describes the various steps needed to carry out the proposed Model-Driven method and thus generate the distributed simulation code with reference to a *border patrol system*.

The following subsections describe the different steps in more detail, according to the method overview illustrated in Section 4.3.

4.4.1 Reference scenario

The simulation scenario deals with a *border patrol system* and concerns a situation in which a set of drones are engaged to patrol the border of a military area, in order to prevent both ground and flight attacks from entering the area.

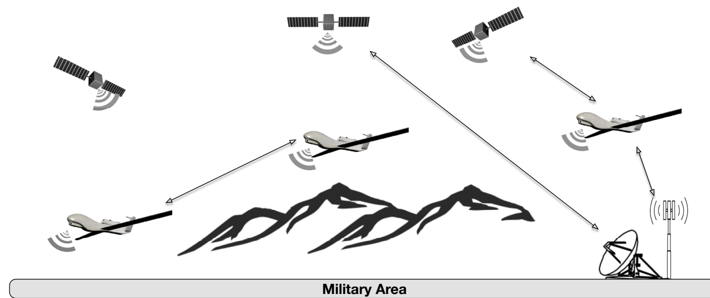


Fig. 4.2: Border patrol simulation scenario.

The border patrol simulation scenario (see Figure 4.2) is composed of (i) *drones*, which represent flying robots that can be remotely controlled or can fly autonomously through software-controlled flight plans in their embedded systems working in conjunction with a GSP receiver. Each drone is equipped

with a number of different sensors such as: way-point sensors, which indicates the orientation and distance of a selected GPS way-point; patrol boundary sensors with a range of 50 m; Boolean boundary sensors, which indicates whether the drone is inside the patrol zone or not; and intruder sensors, which detects intruders by using a high-resolution camera; (ii) *satellites*, which are organized as a constellation and provide communication services to the drones; (iii) *ground station*, which is provided with an omnidirectional and a parabolic antenna that are used to communicate with drones and satellites. More in details, the first one is an all-purpose antenna that is used to communicate with drones; whereas the second one uses a parabolic reflector to send/receive radio waves to/from satellites. The ground station collects and displays real-time data about the performance of both satellites and drones (e.g., velocity, position, status, etc.), as well as ground/flight attacks.

When a drone detects a target moving in its patrol area, it makes a tactical decision based on the type of the target. It determines whether it should gain altitude, in order to use a larger field of vision to track multiple targets that are moving in different directions, or to lose attitude so to follow a single target. After detecting a target, the drone sends a *Target Detect* message to the base station by using either its communication module or a satellite, which operating as a bridge, delivers the drone's message to the base station.

4.4.2 Specify System Model

The proposed method is carried out through several steps, the first of which includes the definition of the system model by using the the SysML notation. For the purposes of this discussion, the study is limited to those diagrams that are necessary to obtain the simulation model and the code of the distributed simulation. Specifically, the SysML model of the border patrol system is composed of the following diagrams: (i) a *block definition diagram (BDD)*, which specifies the structure of the system; (ii) a set of *sequence diagrams (SDs)*, which specify the behavioral view of the system. Such diagrams depict the ordered set of interactions between different system components.

Figure 4.3 shows the complex system under study, called *Border Patrol System*, through a SysML BDD diagram that highlights the main subsystems (i.e., Satellite, Drone and Base Station).

A **satellite** is composed of seven main blocks: (i) *Engine*, which provides continuous finite thrust; (ii) *Flight Computer*, which represents the on-board computer that manages the data flow from both subsystems and navigation sensors, so as to handle the satellites activities; (iii) *Communication and Data Handling System*, which provides the services that handle the satellite-drones and satellite-base station communication; (iv) *Environmental Control System*, which is composed of a set of sensors that are used to monitor environmental factors; (v) *Orbit Control System*, which is a system that manages the orbit trajectory of the satellite; (vi) *Attitude Control System*, which controls the orientation of the satellite with respect to an inertial frame of reference

or another entity such as the celestial sphere or nearby objects. It is composed of sensors to measure the satellite orientation, actuators to apply the torques needed to re-orient the satellite to a desired attitude, and algorithms to command the actuators based on sensor measurements of the current attitude and the target attitude; (vii) *Electrical Power System*, which provides electrical power to the satellite's components.

A **base station**, is a terrestrial radio station designed for handling radio communication with spacecraft, satellite or aircraft. The Base station is located on the surface of the Earth and communicate with spacecraft, satellite or aircraft by transmitting and receiving radio waves in the super high frequency or extremely high frequency bands (e.g., microwaves). When the base station successfully transmits radio waves to a spacecraft, satellite or aircraft (or vice versa), it establishes a telecommunications link. More in detail, a base station is composed of four main blocks: (i) *Engine*, which allows to change the position and orientation of the parabolic antenna; (ii) *Computer*, which represents the computer that manages the data flow from both satellites and drones; (iii) *Communication and Data Handling System*, which provides the services that handle the satellite-drones and satellite-base station communication; (iv) *Electrical Power System*, which provides electrical power to the base station's components.

A **drone** is an aircraft without a human pilot aboard. The flight of drones may be controlled either autonomously by on-board computers or by the remote control of a pilot on the ground or in another vehicle. In this context, drones work in cooperation under a shared control so that they overlap well to provide a full coverage of the patrol area. A drone is composed of six main blocks: (i) *Engine*, which offers continuous finite thrust; (ii) *Fight Computer*, which represents the on-board computer that manages the data flow from both subsystems and navigation sensors so to handle the drones activities; (iii) *Communication and Data Handling System*, which provides the services that handle the satellite-drones and satellite-base station communication; (iv) *Environmental Control System*, which is composed by a set of sensors that are used to monitor environmental factors; it is also responsible for monitoring the surrounding area; (v) *Control System*, which is a system that manages both the trajectory and the orientation of the drone. It is composed of sensors to measure the drone orientation, actuators to apply the torque needed to re-orient the drone, and algorithms to command the actuators based on sensor measurements of the current position and the target position; (vi) *Power System*, which is the component used to supply and transfer the electricity to power both the radio and controllers.

The three aforementioned subsystems (*Drone*, *Satellite* and *Base Station*), constitute the *Border Patrol System* under study, as shown in Figure 4.3, which illustrates the *structural* view of the PIM in terms of a SySML Block Definition Diagram.

The *behavioral* definition of the system is represented by a set of Sequence Diagrams describing the interactions between the different components of the

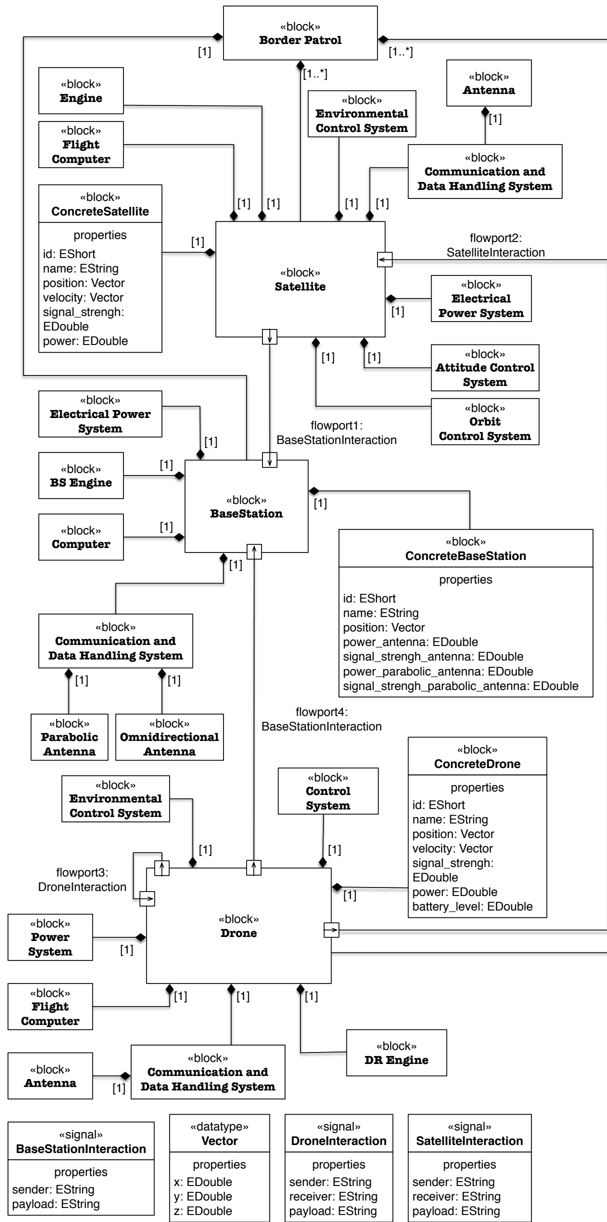


Fig. 4.3: SysML BDD diagram of the Border Patrol System.

system. The sequence diagrams will subsequently be taken as input by the *model-to-model* transformation without being further annotated.

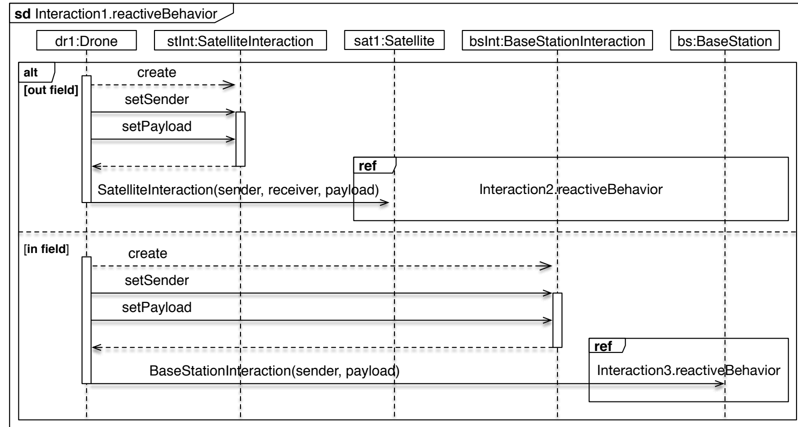


Fig. 4.4: SysML SD diagram of the border patrol simulation scenario.

Figure 4.4, describes the main SD, which illustrates the behavior of a drone during its patrol activity. According to this behavior, if the base station is out of range and not able to receive a signal from the drone, the latter composes a signal of type *SatelliteInteraction*, (i.e., setting *sender* and *payload*), and sends it to a satellite. Otherwise, the drone sets the *sender* and the *payload* of a signal of type *BaseStationInteraction*, and sends it directly to the Base Station.

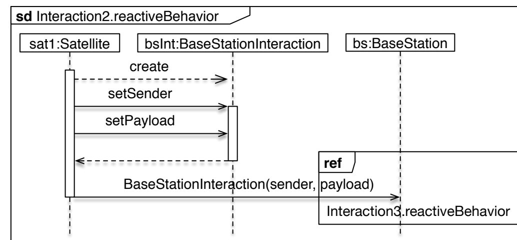


Fig. 4.5: SysML SD Interaction2.reactiveBehavior.

The aforementioned signals, *SatelliteInteraction* and *BaseStationInteraction*, are also defined in the BDD depicted in Figure 4.3. The *Ref* box labeled

as *Interaction2.reactiveBehavior* is a SysML *InteractionUse*, i.e., a placeholder that refers to the SD illustrating the behavior of a satellite receiving a message from a drone, which is shown in Figure 4.5.

As represented in the diagram, once received a message from a drone, the satellite sends a message to the Base Station after setting a sender field and an appropriate payload. The InteractionUse in Figure 4.5, i.e., the *Ref* box labeled as *Interaction3.reactiveBehavior*, refers to the SD in Figure 4.6, which describes the behavior of the Base Station when receiving a message. As represented in the diagram, after receiving a message, the Base Station extracts the sender and the payload, and finally stores the content.

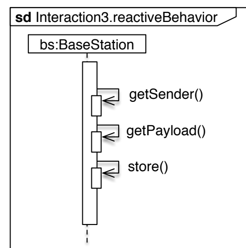


Fig. 4.6: SysML SD Interaction3.reactiveBehavior.

The full set of SDs provides the *behavioral* view of the PIM.

4.4.3 Define simulation environmental objectives and perform conceptual analysis

At the second step of the method, the objectives for the simulation environment are defined and transformed into a set of specific choices that will be used during the subsequent steps.

As aforementioned, the intrinsic complexity, heterogeneity and distribution properties of the system under study naturally lead to the choice of a distributed simulation based on the HLA standard. The simulation execution allows system designers to experiment with various execution scenarios and evaluate the effectiveness of the corresponding solutions in terms of a set of key performance measures (e.g., coverage of the patrol area, detection efficiency and accuracy, power consumption, etc.), as well as to analyze the operational impact of various design choices (e.g., autonomous or remotely controlled management of drones, characteristics of the drone and satellite constellations, etc.). Once the objectives have been defined, the relevant entities within the domain of interest are to identified, in order to provide the implementation level guidance needed to design and develop the HLA-based distributed simulation.

Table 4.1: Stereotypes of the *SysML4HLA* profile.

Stereotype	**Extension**
`<<federation>>`	`Block` (UML `Class`)
`<<federate>>`	`Block` (UML `Class`)
`<<objectClass>>`	`Block` (UML `Class`)
`<<interactionClass>>`	`Block` (UML `Class`)

In this respect, the *SysML4HLA* is used to annotate the SysML model built at the first step, so as to drive the mapping of SysML domain elements into the corresponding HLA-based UML domain elements.

The *SysML4HLA* profile provides a set of *stereotypes* (or *metaclass extensions*) that extend the *Block* element of SysML, which in turn is an extension of UML *Class* metaclass. The introduced stereotypes model the basic elements of an HLA simulation: Federation, Federates, object classes and interaction classes [53, 67], as shown in Table 4.1.

Specifically, the stereotypes of the *SysML4HLA* profile allow to specify both how the system has to be partitioned in terms of Federation/Federates and how system model elements have to be mapped to HLA model elements such as *object classes* and *interaction classes*.

Figure 4.7 shows, as an example, how the structural definition of the system under study (i.e., the BDD in Figure 4.3), is annotated applying the *SysML4HLA* profile. The so-annotated model is referred to as the *Marked PIM* in Figure 4.1.

4.4.4 Design simulation environment

The resulting *SysML4HLA* annotated models defined in the previous step of the MONADS method (marked PIM) are taken as input by the automated *SysML-to-HLA model-to-model* transformation, which yields as output the UML model of the corresponding HLA-based distributed simulation.

The *SysML-to-HLA model-to-model* transformation has been specified by use of the QVT/Operational Mappings language [77], the standard language for defining operational transformations consisting of a set of mapping functions, which specify the mapping rules by use of conventional imperative primitives.

The resulting UML model is composed of the following diagrams: (i) a set of *class diagrams*, which describes the structural view of the model with the publish/subscribe associations between Federates, ObjectClass and InteractionClass; and (ii) a set of *sequence diagrams*, which specify the behavioral view of the HLA simulation application.

A more detailed vision of the inputs and outputs of the *model-to-model* transformation is given in Figure 4.8.

For each block stereotyped as *Federate*, a sequence diagram describing the DKF corresponding class is created. This class is responsible for creating

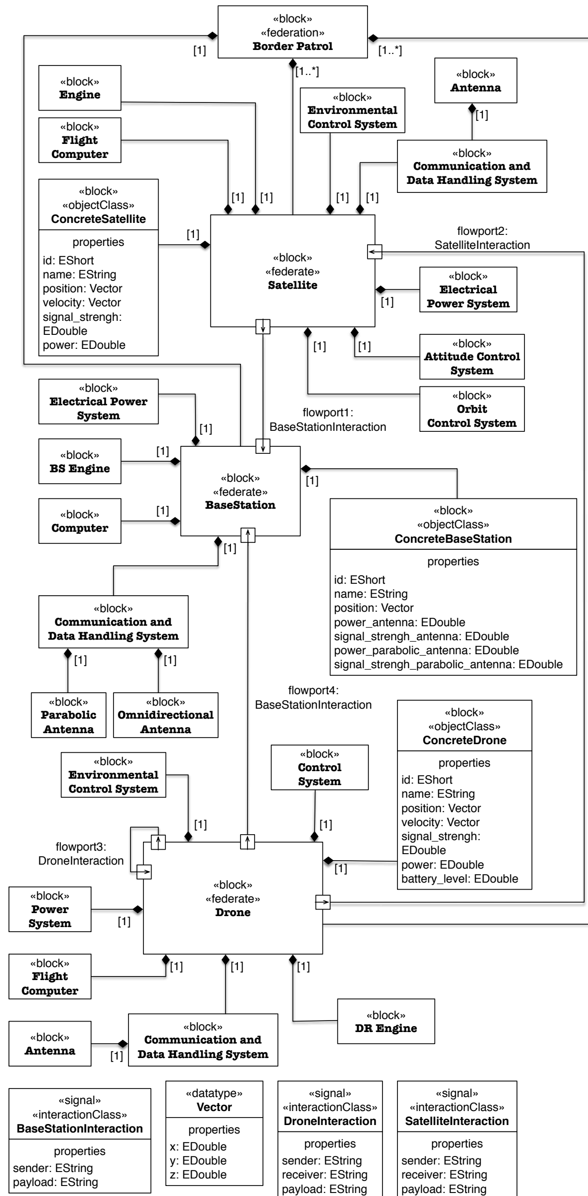


Fig. 4.7: SysML-HLA annotated BDD diagram of the Border Patrol System.

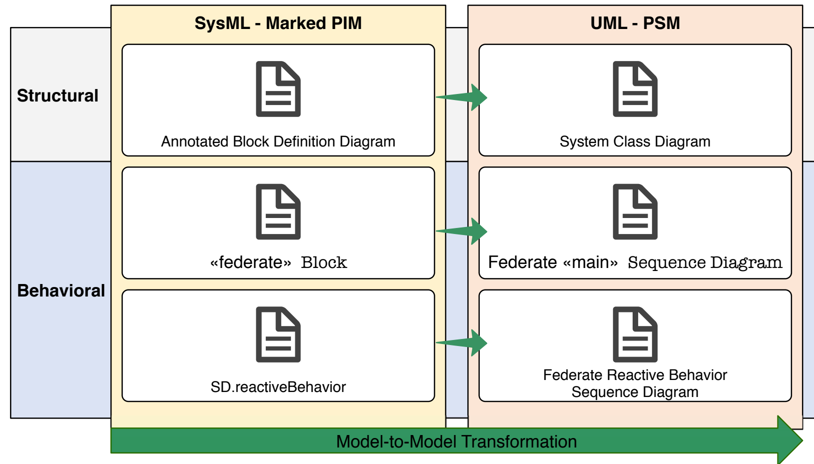


Fig. 4.8: Model-to-Model transformation inputs and outputs.

the Federate which carries out the publish and subscribe operations with the RTI, and the corresponding Federate ambassador. In addition, the SysML Sequence Diagrams named with the *.reactiveBehavior* extension, are transformed as they represent the reactive part of the Federate. The *model-to-model* transformation takes as input the SysML model denoted as *Marked PIM* and produces as output the UML model denoted as *PSM*. The obtained UML is extended by use of the *HLAProfile*, which has been defined to model concepts, elements and relationship of an HLA Federation.

The HLA profile includes several stereotypes organized in two packages, namely the *HLADatatypes* package and the *OMTKernel* package. The former specifies the datatypes of the various attributes used to define the stereotypes of the latter, which has been defined according to the HLA Object Model Template Specification [53] and includes the following stereotypes:

- `<<federation>>` and `<<federate>>`: the UML elements representing the whole Federation and associated Federates, respectively (both extensions of the UML `Class` metaclass).
- `<<objectClass>>`: an object class (extension of the UML `Class` metaclass).
- `<<interactionClass>>`: an interaction class (extension of the UML `Class` metaclass).
- `<<objectAttribute>>`: an object class attribute (extension of the UML `Property` metaclass).
- `<<interactionParameter>>`: an interaction class attribute (extension of the UML `Property` metaclass).

- <<publish>>: the association between a Federate and a published element (extension of the UML `Association` metaclass).
- <<subscribe>>: the association between a Federate and a subscribed element (extension of the UML `Association` metaclass).

The following listing gives a concrete example of a transformation rule that is applied to generate the UML class diagram from an annotated SysML BDD. Specifically the rule is applied to blocks stereotyped as <<Federate>>.

```
1  case(self.getAppliedStereotypes()->exists(g|g.name='Federate')) {
2      // Federate class construction
3      name := self.name;
4      package := p;
5
6      // stereotype <<Federate>> application
7      result.applyStereotype(omtKernel.ownedStereotype->any(name = '
       Federate'));
8
9      // federateAmbassador class construction
10     var amb:UML::Class:=null;
11         object amb:UML::Class {
12         name:=self.name + 'Ambassador';
13         package:=p;
14         };
15
16     amb.applyStereotype(omtKernel.ownedStereotype->any(name = '
       FederateAmbassador'));
17
18     var as:UML::Association:=null;
19     object as:UML::Association{
20     name:=result.name +'_'+ amb.name;
21     package:=p;
22     };
23
24     var fedpr:uml::Property:=null;
25     result.ownedAttribute:=object fedpr:UML::Property{
26         fedpr.type:=amb;
27         fedpr.association:=as;
28         fedpr.aggregation:=uml::AggregationKind::composite;
29         };
30
31     var ambpr:uml::Property:=null;
32     amb.ownedAttribute:=object ambpr:UML::Property{
33         ambpr.type:=result;
34         ambpr.association:=as;
35     };
36 as.memberEnd:=fedpr;
37 as.memberEnd+=ambpr;
38 }
```

The rule scans all blocks contained within the BDD and for each block stereotyped as *Federate* applies the following rules:

- create a UML class with the same name of the SysML block and apply the stereotype <<Federate>> of the *HLAProfile*;
- create a UML class for the Federate ambassador with a name that results from the concatenation of the block name and the *Ambassador* string, and apply the stereotype <<FederateAmbassador>> of the *HLAProfile*;
- create an association of type *composition* between the classes that represent a Federate and its ambassador, respectively.

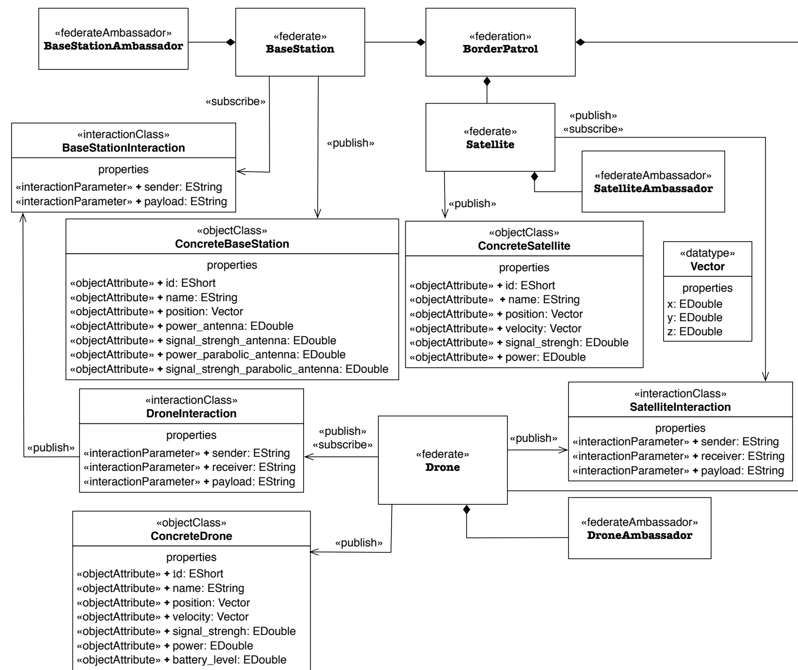


Fig. 4.9: Model-to-Model transformation output: UML Class Diagram.

Figure 4.9 illustrates the *UML Class Diagram* that results from the execution of the aforementioned transformation rules. Such a diagram gives the structural view of the PSM.

The *SysML-to-HLA model-to-model transformation* also generates the behavioral view of the PSM. Specifically, for each Federate in the SysML BDD (i.e., the *marked PIM*), a sequence diagram is created, describing the behavior of the relevant DKF main class, responsible for managing the simulation of that Federate.

As an example, Figure 4.10 gives the UML Sequence Diagram obtained by use of the *model-to-model* transformation for the Satellite Federate.

The sequence diagram in Figure 4.10 represents the behavior to be coded in the corresponding class of the *Satellite Federate*.

Similarly for the SDs that describe the reactive behavior of the Federate, obtained from the corresponding SysML diagrams. The exchanged messages are defined according to the DKF framework, which provides an intuitive set of APIs to deal with both the initialization and the Federate execution starting.

The complete *model-to-model* transformation produces as output a set of UML Sequence Diagrams for each Federate, plus a generic one that is obtained from the SD reported in Figure 4.4, thus gathering the behavioral part of the PSM.

The so obtained UML model, denoted as the PSM in Figure 4.1, is ready to be taken as input by the next step of the MONADS method, as described in the next section.

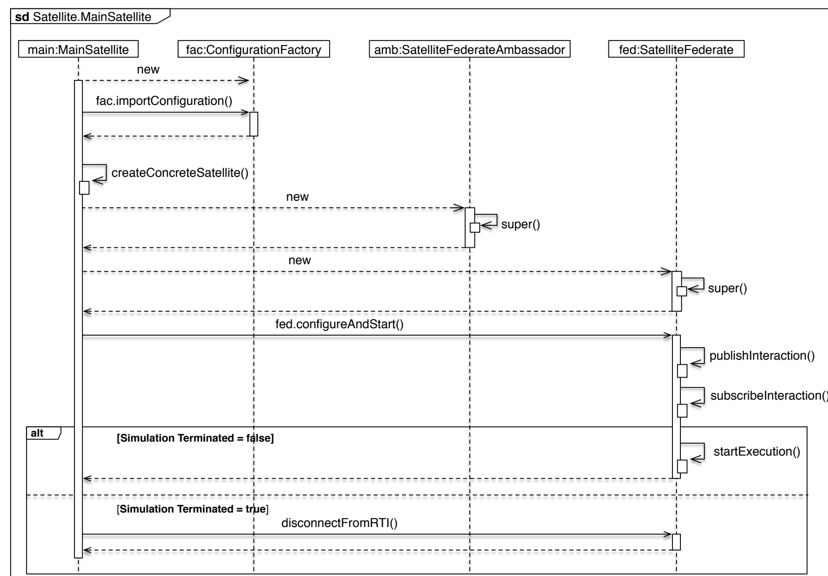


Fig. 4.10: Model-to-Model transformation output: UML Sequence Diagram.

4.4.5 Develop and integrate simulation environment

In this step, the HLA-based UML model produced in the previous step is taken as input by the automated *HLA-to-Code model-to-text* transformation,

which yields as output (a considerable part of) the distributed simulation code.

The *HLA-to-Code* transformation has been specified in the *MOFM2T* language, which adopts a template-based approach, wherein a code template specifies placeholders for data to be extracted from models [76].

Specifically, the *HLA-to-code* transformation generates a Java class template that contains the class structure, including constructors, methods and attributes, declarations and exception management, and most of the required HLA-related code, such as data type definitions and RTI interaction methods. The only code that has to be manually added is the one implementing the Federate simulation logic.

The following listening shows a portion of the *model-to-text transformation*, as implemented by the use of the Eclipse *Acceleo* plugin.

```
1  [comment ObjectClass definition /]
2
3  [template public genetareFederateObjectClass(aClass : Class,aModel:
       Model)]
4      [if (aClass.getAppliedStereotypes()->exists(stereotype|stereotype.
       name = 'ObjectClass'))]
5
6      [file (detectFederateOfObjectClass(aClass, aModel).toString().
       toLower()
7      +'.model/'+aClass.name.toUpperFirst()+'.java', false, 'UTF-8')]
8  package [detectFederateOfObjectClass(aClass, aModel).toString().toLower
       ()/].model;
9
10 import dkf.coder.HLAfloat64LECoder;
11 import dkf.coder.HLAinteger16BECoder;
12 import dkf.coder.HLAunicodeStringCoder;
13 import dkf.model.object.annotations.Attribute;
14 import dkf.model.object.annotations.ObjectClass;
15
16 @ObjectClass(name = "[aClass.name/]")
17
18 public class [aClass.name.toUpperFirst()/]{
19 [for (pr : Property | aClass.attribute)]
20     [if (pr.name<>null)]
21     @Attribute(name = "[pr.name/]", coder=[mappingType(pr.type)/].class
       )
22     private [detectType(pr.type)/] [pr.name/]= null;
23     [/if]
24
25 [/for]
26
27 [comment getter and setter methods/]
28 [for (pr : Property | aClass.attribute)]
29     [if (pr.name<>null)]
```

```
30      public [detectType(pr.type)/] get[pr.name.toUpperFirst()/](){
31      return [pr.name/];
32      }
33
34      public void set[pr.name.toUpperFirst()/]([pr.type.name/] [pr.name
        /]){
35      this.[pr.name/]=[pr.name/];
36      }
37
38      [/if]
39  [/for]
40
41  public [aClass.name.toUpperFirst()/](){}
42  }
43          [/file]
44      [/if]
45  [/template]
```

This code scans all the classes of the model and, for each class stereotyped as `<<objectClass>>`, creates a package with the same name and extension *.model*. Within this package, a Java file is also created with the name of the class and the *.java* extension. Then, for each attribute of the class, an annotation and an attribute qualified as *private* are created.

Finally, for each attribute, the *getter* and *setter* methods are created. As an example, the following listing gives the output of the aforementioned transformation as applied to the *ConcreteBaseStation* class.

```
1   package basestation.model;
2
3   import dkf.coder.HLAfloat64LECoder;
4   import dkf.coder.HLAinteger16BECoder;
5   import dkf.coder.HLAunicodeStringCoder;
6   import dkf.model.object.annotations.Attribute;
7   import dkf.model.object.annotations.ObjectClass;
8
9   // class definition
10  @ObjectClass(name = "ConcreteBaseStation")
11  public class ConcreteBaseStation {
12
13   // class attributes
14   @Attribute(name = "position", coder = VectorCoder.class)
15   private Vector position = null;
16
17   @Attribute(name = "power_parabolic_antenna", coder = HLAfloat64LE.
        class)
18   private Double power_parabolic_antenna = null;
19
20   @Attribute(name = "signal_stenght_parabolic_antenna", coder =
        HLAfloat64LE.class)
```

```
21  private Double signal_stenght_parabolic_antenna = null;
22
23  @Attribute(name = "name", coder = HLAUnicodeString.class)
24  private String name = null;
25
26  @Attribute(name = "id", coder = HLAinteger16LE.class)
27  private Short id = null;
28
29  @Attribute(name = "signal_stengh_antenna", coder = HLAfloat64LE.class)
30  private Double signal_stengh_antenna = null;
31
32  @Attribute(name = "power_antenna", coder = HLAfloat64LE.class)
33  private Double power_antenna = null;
34
35 // costructor
36 public ConcreteBaseStation() {}
37
38  public Vector getPosition() {
39   return position;
40  }
41
42 // class methods
43  public void setPosition(Vector position) {
44   this.position = position;
45  }
46
47  public Double getPower_parabolic_antenna() {
48   return power_parabolic_antenna;
49  }
50
51 // additional methods (not shown)
52
53 }
```

Figure 4.11 illustrates the UML Class Diagram that gives an overall view of the set of main DKF classes that are generated for the border patrol scenario.

The so obtained distributed simulation code is ready to be executed, in order to address the main objectives of the simulation effort, as mentioned in Subsection 4.4.3.

The simulation execution and results evaluation steps, shown in Figure 4.1, are out of this chapter scope and thus are not further detailed.

4.4.6 Integrated Tool-Chain

An integrated tool-chain has been set up to enact the MONADS method for distributed simulation of complex systems. The model definitions and transformations introduced in previous sections have been implemented by use of the various tools integrated into the *Eclipse platform* [24]. Specifically, the

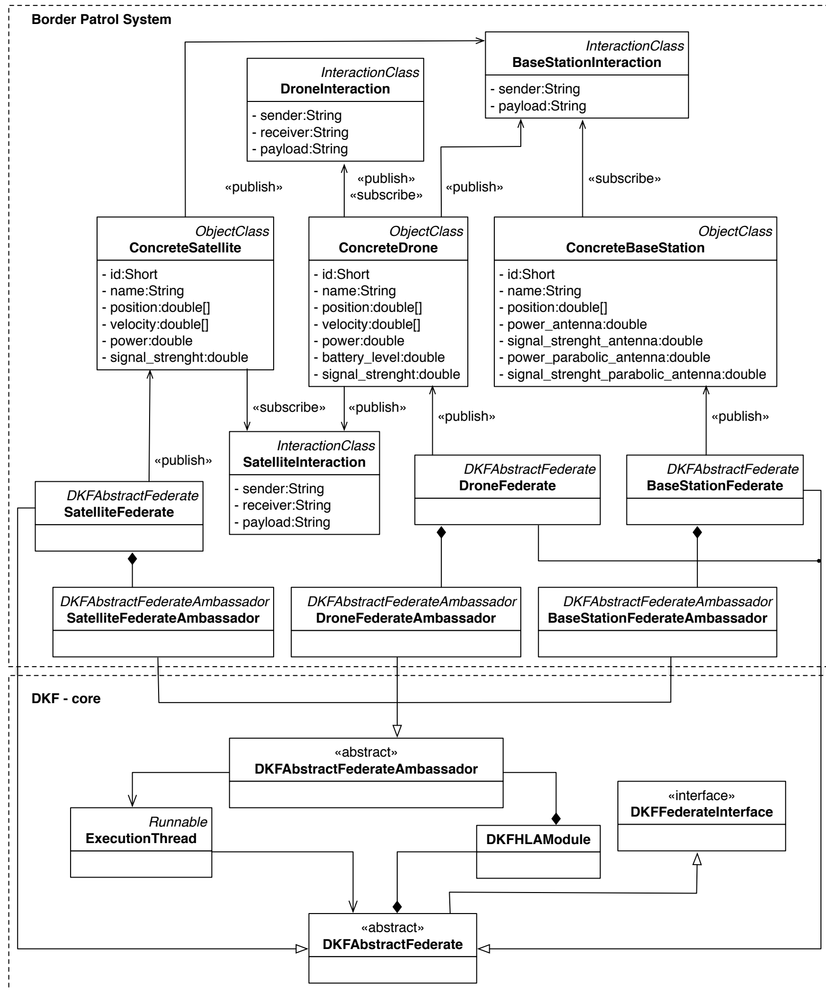


Fig. 4.11: The architecture of the Border Patrol System simulation scenario based on the DKF framework.

Papyrus Modeling Environment project has been adopted as the reference modeling tool [25].

Papyrus provides a full-fledged graphical modeling tool that allows creation and editing of SysML and UML diagrams. The *SysML4HLA* profile and the HLA profile have been implemented into Papyrus, as well.

The QVT Operational component [26], a partial implementation of the Operational Mappings Language defined by the OMG [77] and the Acceleo

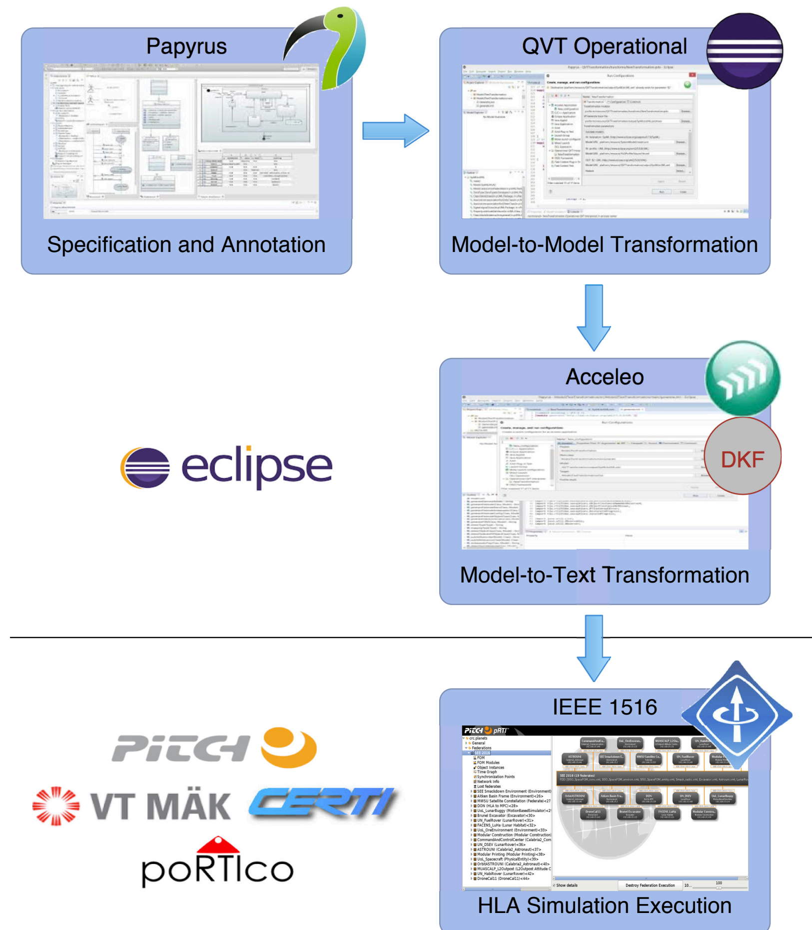


Fig. 4.12: The MONADS integrated tool-chain.

component [23], an open source code generator implementing the OMGs MOF Model to Text Language (MOFM2T) standard [73], have been used as the tools that execute *model-to-model* and *model-to-text* transformations, respectively.

Figure 4.12 illustrates the aforementioned integrated tool-chain. Papyrus is initially used to specify a SysML model and annotate it by use of the *SysML4HLA* profile.

Next, the *model-to-model* transformation from the annotated SysML model to the HLA-based UML model (i.e., the UML model annotated by use of the HLA profile) is carried out by use of the QVT Operational component.

Finally, the *model-to-text* transformation that yields as output the DKF-based distributed simulation code is performed by the use of the Acceleo component. The resulting code, after a refinement process, is then executed on a specific HLA RTI.

The fully integrated tool-chain provides a flexible and easy-to-use implementation platform for the proposed method, thus bridging the gap between the SysML-based system specification and the HLA-based distributed simulation implementation.

4.5 Discussion and Related Work

This section reviews the existing literature dealing with both the use of SysML in the Modeling & Simulation (M&S) domain and the modeling/development of HLA-based distributed simulation systems.

As regards the use of SysML in the M&S context, significant contributions that specifically address the generation of Java/HLA code from SysML specifications can be found in [13, 14, 20]. This work extends and improves such contributions both on the method side, which is now designed according to the DSEEP, and on the model transformation side, which now exploits the advantages of using the *HLA Development Kit software Framework (DKF)* rather than a conventional HLA implementation.

More generally, several contributions are available that propose the use of SysML as a notation suitable not only for defining systems specification but also for supporting system simulation activities, such as [80] and [79] in which SysML is used as a notation to support the simulation-based design of systems, in order to derive executable parametric models and simulation-specific languages, respectively.

Differently from the aforementioned contributions, this chapter describes a Model-Driven method to automate the generation of an HLA-based implementation of a distributed simulation software, starting from a SysML specification.

As regards the issue of supporting the implementation of simulation systems, contributions that apply a Model-Driven paradigm in the M&S domain can be found in [20] and [48], which propose a method to generate a Java/HLA-based implementation of a distributed simulation software from a UML system model and the main theoretical concepts behind the application of MDA to HLA, respectively.

Differently, this chapter illustrates the design and implementation of a model driven method to reduce the gap between the SysML-based system specification and the HLA-based distributed simulation implementation.

As regards the modeling/development of HLA-based distributed simulation systems, several commercial and research efforts aim at providing integrated toolchains for creating and simulating complex systems by using specialized modeling tools and methodologies. Packages and toolboxes have been developed for implementing HLA simulators in Matlab/Simulink, such as the Forwardsim HLA Toolbox for Matlab [98].

Another tool that enables developers to effectively manage the structure and assets of a HLA Federate starting from a FOM (Federation Object Model) file is the PITCH Developer Studio [82]. A domain-specific HLA software framework was created by the Danish Maritime Institute (DMI) [108] to provide mechanisms that simplify the development of real-time simulators. Other HLA frameworks are based on GRID-computing infrastructure [113].

The HLA Development Kit and its software framework (DKF), used in the proposed MONADS method for generating the HLA-based simulation code, 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 LGPL license and can be freely and easily customized and/or extended to cover and deal with both domain independent and domain-specific aspects. 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 HLA objects according to its definition in the FOM. The above-sketched capabilities showed a great benefit not only for expert HLA developers but also for HLA novice practitioners as were the undergraduate students involved in the Simulation Exploration Experience (SEE) project led by NASA and which involves several U.S. and European Institutions [30].

The current DKF implementation targets HLA Federation executions based on Federates using a specific time flow mechanism, namely *timestepped* [38], in which each time advance of the Federate is of a fixed simulation time duration and time does not advance to next time step until all simulation activities for current time step are completed. This choice does not limit the validity of the MONADS method, which can be easily extended to address other mechanisms, such as the *event-driven* one, in which each Federate processes local and external events in time stamp order and the time typically advances to the time stamp of the event currently being processed.

4.6 Conclusion

Modern large-scale systems or systems of systems require the adoption of distributed simulation approaches to properly take into account inherent complexity.

This chapter discussed an innovative and automated method (denoted as MONADS) that makes easier for systems engineers the use of distributed simulation techniques, without asking them to explicitly deal with the intricacies and difficulties of currently available standards and technologies.

It introduces a Model-Driven method based on the execution of model transformations that automatically map the abstract representation of a system, specified in SysML, into an intermediate HLA-based software model, specified in UML, down to the final code of the HLA-based distributed simulation. Specifically, the generated code is based on the *HLA Development Kit software Framework* (see Chapter 3) that allows a developer to appropriately handle common HLA issues thus making it easier to develop a distributed simulation.

The proposed approach allows to automatically obtain a significant portion of the final HLA-based code, by limiting the manual activity to the implementation of the Federate simulation logic, and can be effectively used even by systems engineers who are not familiar with the HLA standard.

5

On the integration of HLA and FMI for supporting interoperability and reusability in distributed simulation

Many research efforts are focusing on the definition of methods, models and techniques to support the reuse and interoperability of simulation models and their execution on distributed computing environments. In this context, great benefits derive from the joint exploitation of two popular standards: FMI (Functional Mock-up Interface) [39] and HLA (High Level Architecture) [53].

The chapter presents how to combine HLA and FMI from two different perspectives: (i) *HLA for FMI* and (ii) *FMI for HLA*. With reference to the *HLA for FMI* perspective, some possible extensions to the FMI standard to include HLA features are proposed. With respect to the *FMI for HLA* perspective, two concrete approaches, based on well-known design patterns, for integrating and (re)using FMUs (Functional Mock-up Unit) in HLA-based simulations are described. Then, to demonstrate the effectiveness of a presented solution, a case study concerning a Moon base simulated scenario is presented.

5.1 Introduction

As described in Chapter 2, the use of M&S has 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 environment [5, 35, 43, 92]. 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 scarce support to enable their execution on *distributed infrastructures*.

To overcome these disadvantages, many research efforts are focusing on the definition of methods, models and techniques to support the reuse and

interoperability of simulation models and their execution on distributed computing environment. Two of the most popular efforts going in these directions are FMI (Functional Mock-up Interface) [39] and HLA (High Level Architecture) [53, 59]. However, each of the two mentioned proposals addresses part of the above issues and great benefits derive from their combined exploitation [4, 6].

The chapter discusses in detail the main issues and opportunities from the integration of FMI and HLA and describes the principles behind their joint exploitation by focusing on the benefits that HLA offers to FMI and vice versa (*HLA for FMI* and *FMI for HLA* respectively). Two concrete approaches for realizing the *FMI for HLA* integration perspective are presented and one of them is exemplified by a case study concerning the integration of a FMU (Functional Mock-up Unit) in a HLA Federation.

5.2 Combining HLA and FMI

Although the HLA and FMI standards start from different objectives and are based on different techniques (see [39, 53, 56, 59]), they have several common features that can be jointly exploited so as to create a full-fledged solution to enable reuse, interoperability and distributed execution of simulation models.

To investigate how to fruitfully combine HLA and FMI standards, two different integration perspectives should be considered:

- *HLA for FMI*, i.e. how to support the definition of a FMU able to include and exploit HLA features and services;
- *FMI for HLA*, i.e. how to include, in a HLA simulation, applications that are available as FMUs so as to enable their reuse in a HLA context.

In the following, an extension of the model description XML file [39] that could be used to support the first integration perspective (*HLA for FMI*), and two possible integration approaches to address the second one (*FMI for HLA*) are described.

5.2.1 HLA for FMI

The FMI standard does not have mechanisms that allow a FMU to interact with others heterogeneous components in a distributed simulation environment, such as those supported by HLA; indeed, currently, they can only be used as external components in a stand-alone simulation tool.

To overcome this limitation, it is necessary to define concepts and mechanisms in order to enrich a FMU, during the modeling phase, with HLA features.

As described in Section 2.4, a FMU contains both a model-description XML file and a C code that implements the model. Starting from these two

elements, the integration process can consist of two phases: (i) extend the *model description* XML-Schema file by defining new tags that cover specific HLA functionalities; and, (ii) create a *C shared library* that defines the related HLA commands. A set of possible tags, which could be added to the XML file that describes a FMU so as to make it compatible with HLA specifications, are described in Table 5.1.

Table 5.1: HLA for FMI tags.

XML tag	**Description**
HlaTime	Provides settings for managing the federation time, synchronization points and the timestamps of the events.
HlaDeclarationManagement	Defines functions to manage the operations of publishing and subscribing of events and attributes.
HlaFederation	Defines functions to manage a HLA federation.
HlaOwnership	Provides settings for handling the ownership of an object.
HlaAttribute	Provides settings for managing the attributes of a HLA *ElementObject*.
HLAInteraction	Provides settings for managing HLA interactions.
HLATypeDefinition	A global list of the HLA type definitions.

Figure 5.1 shows the combined solution, defined according to the proposed integration approach, in which the FMU model contains some HLA features defined during the modeling phase. More in detail, the *Model* defines the simulation logic of the FMU according to the FMI specification and uses the *HLA Core features* component to integrate HLA characteristics in order to make the FMU compatible with a HLA-based simulation environment. The *FMU/HLA Solver* enriches the FMU solver with specific features to manage the HLA commands.

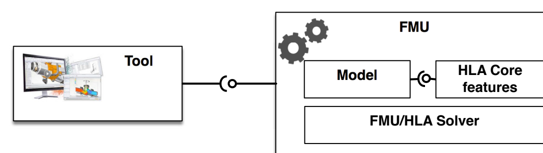


Fig. 5.1: A FMU enriched with HLA features.

The advantages arising from the proposed integration approach could be: (i) greater integration and execution control of FMUs in a HLA Federation,

since a FMU can natively support HLA services; (ii) reduction of development costs and time, since a FMU can be reused in HLA context; and (iii) better performances, since FMUs that share the same HLA simulation platform can be executed on geographically distributed computers. Despite the above sketched advantages that could derive from the native support of HLA features, the proposed solution requires not only to extend the current FMI standard, but also to add to the current FMU solvers the capabilities to interpret and execute HLA-based operations.

5.2.2 FMI for HLA

The integration of a FMU in a HLA Federation is a highly challenging process since it is necessary to solve different issues. In the last few years, several research efforts have been devoted to support this integration perspectives (*FMI for HLA*); some of them aim at providing an integration mechanism based on a master algorithm for Co-Simulation using FMI [11], others are based on using the HLA RTI as a master for FMUs [5, 6] or on the definition of wrappers that allows to connect a FMU to a HLA Federation [115].

The key difference between FMI and HLA is that HLA provides specific mechanisms for data exchange and time management that enable the integration in a distributed computing environment of heterogeneous simulation models created according to the HLA standard. As in the FMI for Model Exchange modality the solver module is not part of the FMU (see Section 2.4), it is not practicable to integrate such a kind of FMU in a HLA simulation; as a result, only FMU generated according to the *FMI for Co-Simulation modality* can be taken into consideration for the inclusion in a HLA simulation.

To achieve this kind of integration, a HLA component has to act as a master for the FMUs (that thus act as slaves) in order to manage their lifecycle during the HLA simulation. In particular, the master has the responsibility to orchestrate the steps of Co-Simulation through the execution of two tasks: (i) track and control the data exchange between the Federation and the controlled FMUs; and, (ii) synchronize the simulation time between the HLA Federation and the FMUs, so to control its advancement. This approach allows combining in a HLA simulation both FMUs (controlled by HLA components acting as masters) and standard HLA modules [5]; this integration has many advantages:

- Heterogeneous FMUs can be reused in a HLA simulation environment without making structural or behavioral changes on them;
- Greater interoperability among FMUs because they can interact with one another in a distributed computing environment through HLA;
- FMUs can be created and tested independently through well-established simulation environments compliant to the FMI standard;
- Better performances because the FMUs share only the RTI infrastructure, while their execution takes place on different computers, which can be also geographically distributed.

Nevertheless, the integration of a FMU into a HLA simulator is a complex process, since the FMI standard defines some features that make a FMU not fully compatible with HLA. In particular, the FMI standard does not support all the *HLA DataTypes*. It defines only five data types: *fmiReal*, *fmiBoolean*, *fmiInteger*, *fmiString*, and *fmiEnumeration* [39]; furthermore, the FMI for Co-Simulation modality does not support timestamps events, therefore it is hard to execute event-driven simulations. Moreover, the FMI standard does not define the concepts of publication and subscription of attributes and interactions, and resources ownership.

In the following, two possible solutions to realize the above described integration perspective are presented.

5.3 FMI for HLA: Integration Approaches

In this section two approaches, *Adapter-based* and *Mediator-based*, that allow integrating a FMU into a HLA simulation, are presented. Both the approaches, which are related to the proposal presented in [115], use the concept of Hybrid Federate to manage the lifecycle of a FMU and overcome the FMI for HLA integration issues.

5.3.1 Adapter-based

In the Adapter-based approach the Hybrid Federate is composed by two elements: (i) a FMU, which contains the behavior of the component to simulate and its solver; and (ii) the FMI-HLA Adapter that manages all the interactions between the RTI infrastructure and the FMU (e.g., publish/subscribe of the attributes that are produced/used by the FMU, Object discovery, Datatypes mapping), as well as the lifecycle of the FMU. In the software engineering practice, an adapter is a software design pattern that allows the interface of an existing class to be used from another interface [3, 40]. It is often used to make existing classes work with others without modifying their source code.

The *FMI-HLA Adapter* allows the two heterogeneous elements to work together in a distributed simulation environment through a software layer that interprets the application-specific services. The software layer is composed of three distinct parts: (i) *Communication*; (ii) *FMI-HLA Logic*; (iii) *Monitoring*. The first component, handles the two-way communication and data exchange, as well as the connection to the RTI. The second one contains the logic that allows the two standards (FMI and HLA) to work together. Finally, the *Monitoring* component provides a set of services to monitor the simulation execution of the FMU module and the status updates of the whole *Hybrid Federate*. Figure 5.2 shows the *Adapter-based* approach.

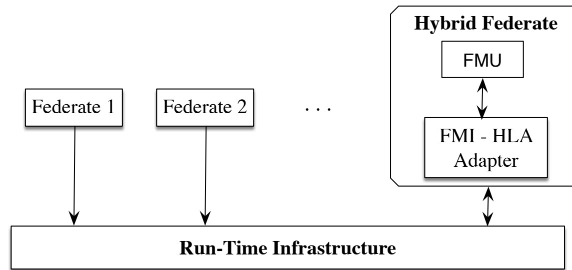


Fig. 5.2: The Adapter-based integration approach.

5.3.2 Mediator-based

In the *Mediator-based* approach, the structure of the Hybrid Federate changes with respect to the previous approach. Indeed, it is composed of two elements: (i) a set of *FMUs*, each of which defines a dynamic model according to the FMI standard; and (ii) a *HLA Federate*, which is not a simple adapter, as in the adapter-based approach, but contains its own simulation logic and uses the FMUs to simulate specific components. For example, a HLA Federate that simulates a car can use specific FMUs to handle the simulation of specific car components (e.g., engine, transmission, driveline, external sensors, brakes). During the simulation execution the HLA Federate uses the FMU modules by using the mediator layer to coordinate/orchestrate the behavior of the whole *Hybrid Federate*.

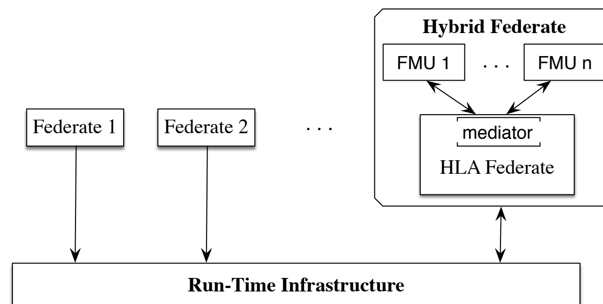


Fig. 5.3: The Mediator-based integration approach.

The *mediator* is a software layer that encapsulates the modalities with which the HLA Federate interacts with the FMUs. In particular, the HLA Federate cannot communicate directly with the FMUs, but it has to use the

mediator layer; this mechanism reduces the dependencies between communicating parts, thereby lowering the coupling [40]. In Figure 5.3 the described *Mediator-based* approach is exemplified.

5.3.3 A case study

This section presents a case study concerning the integration of a FMU module into a HLA simulation. The simulation scenario, chosen to demonstrate the proposed approaches, concerns a human settlement called "Moon base" composed of scientific equipment, storage buildings, rovers and other elements to allow astronauts to live and work on the moon. This scenario is based on that adopted in the context of the 2014 edition of the Simulation Exploration Experience (SEE) project (see Chapter 3).

Reference Scenario

The simulation scenario takes place on the lunar surface and concerns a situation in which a landing is taking place on the platform located in the Moon base. To allow the lander to land safely, all the entities operating within the Moon base have to be informed in order to face this situation. All the communications are managed by the UNICOM radio communication system that provides flexible communication functionalities to the other entities populating the "Moon base". UNICOM works as a mediator and to provide its services is equipped with an antenna mounted on a tower [31].

When the lander begins the landing operations, it sends a "Landing started" message to UNICOM, which forwards the incoming message to be broadcast to all the entities that populate the human settlement and which are located within the coverage area of the UNICOM's antenna. Each receiving entity may react to the situation in different ways; e.g., the astronauts react by moving away from the landing site. During the landing phase, the lander constantly sends information about its acceleration, velocity and altitude to UNICOM, which forwards this information by using a "Landing data" message to all the subscribed entities. Finally, a "Landing completed" message is sent from the lander to the UNICOM communication module, when the lander touches down on the platform. UNICOM receives the "Landing completed" message and sends it to all the entities that operate in the Moon base, so to notify that the landing operations has been completed.

Figure 5.4 shows the scenario with the three main elements that define the simulation scenario: UNICOM, the Lunar Rover, and the Lander. The first two elements (UNICOM and the Lunar Rover) are developed as HLA Federates, whereas the Lander is developed as a FMU.

The Lander FMU

The Lander module is a spacecraft that descends and comes to rest on the surface of an astronomical body. During the landing phase, the lander may

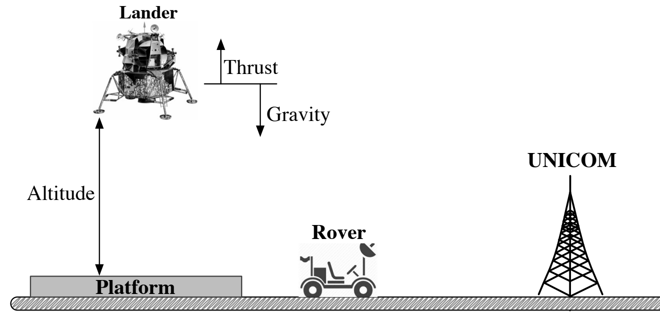


Fig. 5.4: The reference Simulation Scenario.

use either parachutes to slow down and to maintain a low terminal velocity (for planetary bodies with significant atmospheres) or landing rockets, which are fired just before impact to reduce the impact velocity. The lander is simulated by a FMU developed by using Matlab Simulink [65] and generated by using the Modelon FMI Toolbox for Matlab/Simulink [33]. This latter toolbox provides functionalities for the import/export and simulation of FMUs, compliant with both the FMI for Model Exchange and FMI for Co-Simulation modalities, into/from Matlab/Simulink. Figure 5.5 shows the Simulink model of the Lander module.

The equations that regulate the vertical motion of the lander during the landing operation are [36]:

$$\begin{aligned}
&Acceleration = \frac{G \cdot Mass_{moon}}{(Altitude_{lander} + Radius_{moon})^2} \\
&Mass' = -MassLossRate \cdot |Thrust| \\
&Altitude' = Velocity \\
&Velocity' = Acceleration
\end{aligned} \tag{5.1}$$

All the reported equations have been developed, by using Simulink components, in the Lander block. In order to import and use the lander module in a HLA Federation, it has been exported as a FMU (according to the FMI for Co-Simulation modality) by using the Modelon FMI Toolbox for Matlab/Simulink tool [33].

Integration of the Lander FMU in the HLA Federation

To integrate the Lander FMU into a HLA Federation, a HLA Hybrid Federate based on the *Adapter-based* approach has been developed by using the functionalities offered by the SEE HLA Development Kit (see Chapter 3).

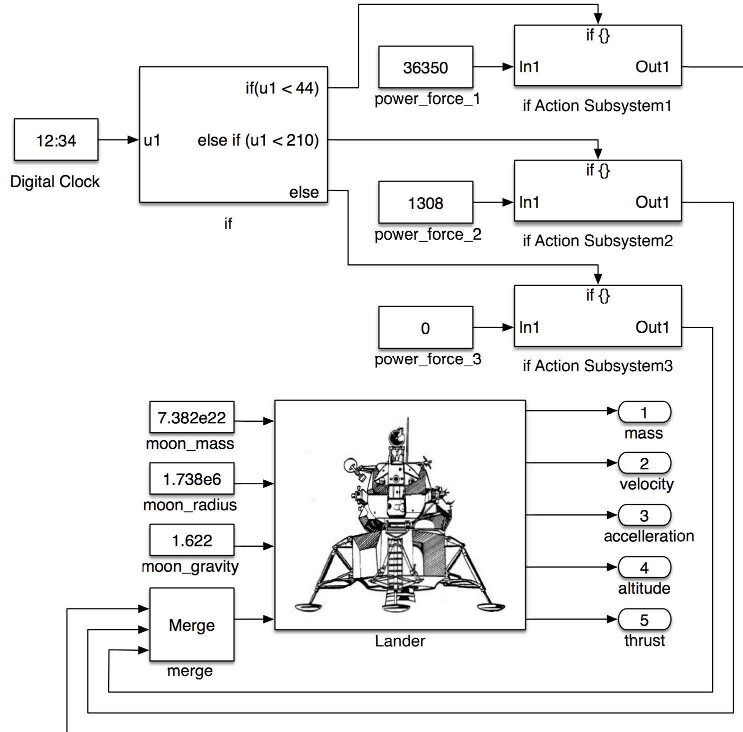


Fig. 5.5: Simulink model of the Lunar Lander.

Figure 5.6 shows the architecture of the Hybrid Federate by using a UML Class Diagram.

The *AbstractAdapter* class manages the lifecycle of the Hybrid Federate; it includes the management of the simulation time and the synchronization among the objects connected to the RTI. This abstract class is extended by the *Adapter* class that implements the functionalities to manage the FMU during the simulation execution.

The *AbstractFederateAmbassador* class, which extends the *SEEAbstractFederateAmbassador* class defined in the SEE HLA Development Kit, implements the methods that are called by the RTI for interacting with the Federate (RTI callback methods); along with the RTI Ambassador interface, which is used by the Federate to access the RTI services. Moreover, it handles the interactions with the other Federates through the use of the *Observer* package.

The *SimulationConfig* class is used to load the configuration parameters of the *Adapter* class, which are stored in a property file. These parameters include the name of the Federate, the IP address of the Federation and its

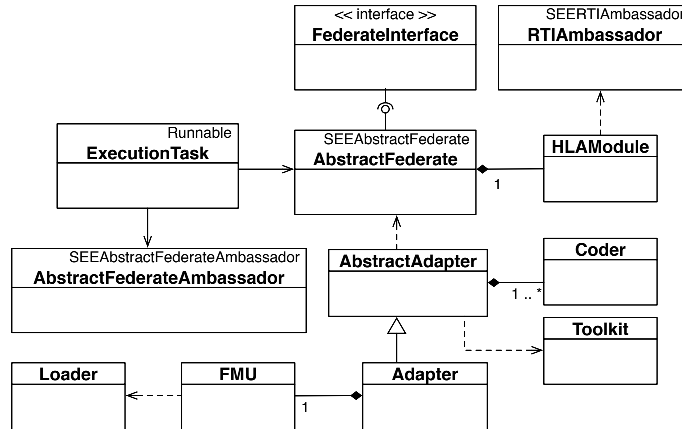


Fig. 5.6: Architecture of the HLA Hybrid Federate.

port number, and other parameter concerning the simulation scenario (e.g., Moon gravity, radius and mass).

The *LanderFMU* class handles the lander module with its properties such as: name, altitude and acceleration. The *Loader* class is used by the *LanderFMU* class to load the .zip file and parse the XML model description of the Lander FMU. The loading operations are managed through the *javaFMI* library [58], which allows interfacing Java applications with FMUs for Co-Simulation or Model Exchange.

The *Toolkit* class contains several miscellaneous methods, such as simulation time standard conversions and Windows Firewall Check.

The lifecycle of the application consists of four phases as shown in Figure 5.7.

In the *load FMU module* state, the configuration parameters of the *Hybrid Federate* and the Lander FMU are loaded, from a property file, by using the *SimulationConfig* and *Loader* classes. A transition to the *startup* state happens if the configuration parameters and the FMU are valid, and during the state transition a connection to the federation execution platform (the HLA RTI) is performed. If the configuration parameters or the FMU module are invalid, a state transition to the *shutdown* state is performed. In this latter state, all the resources engaged by the application are de-allocated and the lifecycle terminates.

In the *startup* state, the Federate checks the connection status. If the connection is not established the lifecycle ends with a transition to the shutdown state. Otherwise, the parameters of the FMU are registered on the RTI platform, and a transition to the *running* state is performed.The running state contains three sub-states: (i) *waiting for TAG*: the Adapter module waits for the TAG (Time Advance Grant) Callback from the RTI; (ii) *processing and*

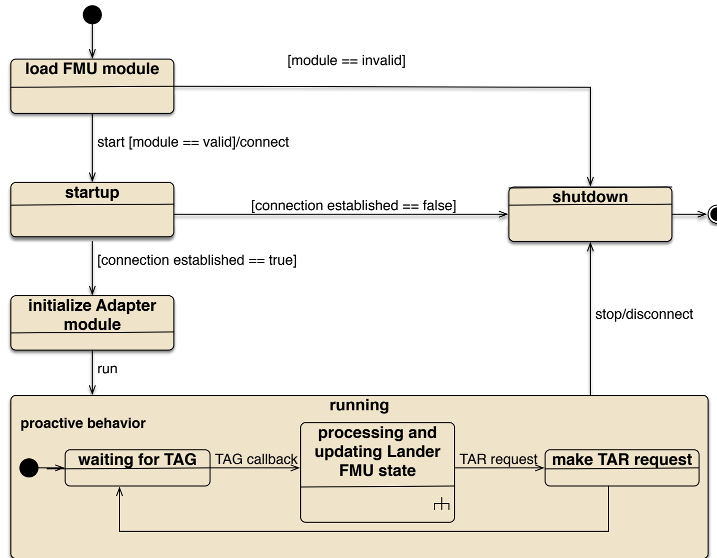


Fig. 5.7: Lifecycle of the HLA Hybrid Federate.

updating Lander FMU state: when the *TAGcallback* is received, the Adapter calls the Lander FMU to make a simulation step; after that, the output values (altitude, acceleration, force, etc.) of the Lander are returned to the Adapter, which deals with their publication on the RTI; and (iii) *make TAR request*: the *Adapter* requests to the RTI the grant for the next logical time.

The *processing and updating Lander FMU state* is a sub-state that contains four states (see Figure 5.8): (i) *initialize*: if the parameters of the Lander FMU are NULL (first simulation step) a transition to the *set Input* state is performed; otherwise a transition to the *make simulation step* state is done; (ii) *set Input*: the parameters of the Lander are set, and then a transition to the *make simulation step* is performed; (iii) *make simulation step*: the Lander module makes one simulation step ahead; and (iv) *get values*: the output values of the simulated step are retrieved and sent back to the *Adapter*.

The Resulting HLA Federation

The reference simulation scenario has been developed and executed by using the Pitch RTI [82] that provides a complete implementation of the IEEE 1516 interface specifications [53]. Figure 5.9 shows the HLA-FMI Federation with the four elements that compose the scenario (Environment, Lander, Rover, and UNICOM). The Federation FOM (Federation Object Model) file is composed of six parts [53, 59]: (i) *Core*, which defines the DataTypes used during

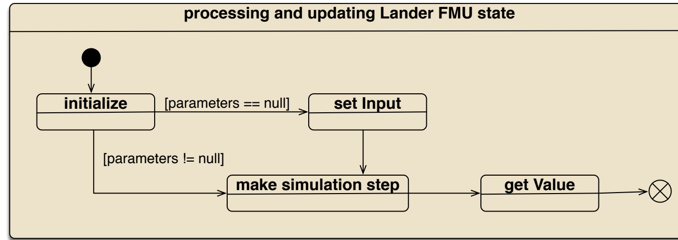


Fig. 5.8: Processing and updating sub-state of the Lander.

a Federation execution; (ii) *Environment*, which specifies the structure of the Reference Coordinate Frames (e.g., *SolarSystemBarycentricInertial, Earth-Centric Inertial*); (iii) *Entity*, which defines the structure of a space entity (e.g., space vehicles, lunar rovers); (iv) *UNICOM*, which defines data and interactions of the UNICOM Communication module [31]; (v) *Lander*, which defines the properties of the Lander as described below; and, (vi) *Rover*, that contains the characteristics of a Rover. The first three parts of the FOM module (Core, Environment, Entity) have been defined by the NASA team for the SEE 2014 project [90] in order to provide a baseline for the simulation scenario.

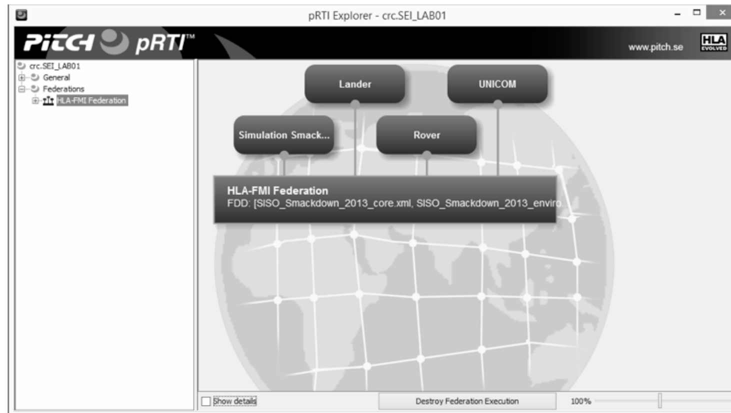


Fig. 5.9: The HLA-FMI Federation.

The FOM module of the *Lander* describes the Lander *Hybrid Federate* in terms of the categories of *ObjectClasses*, *Attributes*, and *InteractionClasses* that it can offer to the Federation [53]. Specifically, the Lander FOM is composed of one *ObjectClass* called *Lander* that inherits from the *PhysicalEn-*

tity class [31, 90], and one *InteractionClass* called *LandingStatusInteraction* to interact with the UNICOM Federate during the landing operations. This interaction provides a single parameter (called *payload*) containing the payload of the message; specifically, it can be "Landing started" or "Landing completed". The Lander *ObjectClass* defines five attributes: *Mass, Altitude, Velocity, Acceleration,* and *Thrust*, which are published and updated on the RTI platform by the *FMI-HLA Adapter* component (see Figure 5.2). Whenever the UNICOM Communication module receives the data from the Lander, it forwards this information by using a "Landing data" message to all the entities that populate the human settlement.

5.4 Conclusion

In the M&S domain many research efforts are focusing on the definition of methods, models and techniques to support the reuse and interoperability of simulation models and their execution on distributed computing environment. Two of the most popular efforts going in these directions are FMI (Functional Mock-up Interface) and HLA (High Level Architecture). However, each of the two mentioned proposals addresses part of the above mentioned issues and great benefits derive from their jointly exploitation.

The chapter presented how to combine HLA and FMI from two different perspectives *HLA for FMI* and *FMI for HLA*. With reference to the *HLA for FMI* perspective, the chapter has delineated some possible extensions to the FMI standard to enrich, during the modeling phase, a FMU with HLA features. In more detail, a set of XML tags that could be added in the model-description XML file to natively integrate the HLA concepts in a FMU have been outlined. With respect to the *FMI for HLA* perspective, the chapter has presented two concrete approaches, *Adapter-based* and *Mediator-based*, for reusing a FMU in a HLA Federation without modifying both the structure and the behavior of the FMU. To demonstrate their effectiveness, a case of study, based on the exploitation of the *Adapter-based* approach has been presented.

6

Further contribution on interoperability in distributed simulation

6.1 Introduction

Distributed and Real-Time Simulation plays a key-role in the Space domain for several purposes. It is exploited for analysis and engineering, from mission level down to individual systems and subsystems, where simulation plays a key tool through the whole lifecycle, from the concept exploration phase to mission design and operation. Another example is training of flight crew and flight controllers, where simulation plays a crucial role as the Space domain is characterized by scarce training opportunities, high cost of real equipment, dangerous scenarios and emergency operations. In particular, great benefits derive from the exploitation of distributed simulation approaches as they allow for combining models from the same or different sources (within the same organization or between different organizations), to run simulation between different locations, and to promote scalability, modularization and usability [37, 41]. Indeed, several distributed simulations have been developed for example for docking vehicles with the ISS and for mission training, in many cases with participants from several nations [2, 81, 85].

This chapter presents further contributions focused on the interoperability of simulation models in the space domain. In particular, the experience gained during the definition and development of the *HLA Development Kit (DKF)* (see Chapter 3), the definition of the *MONADS (MOdel-driveN Architecture for Distributed Simulation)* method (see Chapter 4), and the investigation on how to combine the international standards *IEEE 1516 - High Level Architecture (HLA)* and *Functional Mock-up Interface (FMI)* (see Chapter 5), along with the research activities performed at NASA Lyndon B. Johnson Space Center (JSC) and also within the *SISO Space Reference FOM (SRFOM) Product Development Group (PDG)* [69], allowed to focus on the interoperability of space systems in a distributed simulation.

The chapter is structured as follows. Section 6.2 presents a first set of results achieved by the *SISO Space Reference FOM (SRFOM) Product Development Group (PDG)* that aims at providing a Space Reference FOM for

international collaboration on Space systems simulations [69]. The *Java Space Dynamics Library (JSDL)* project is presented in Section 6.3. It stems from the SISO Space Reference FOM standardization initiative and aims at supporting the development and simulation of complex space systems by providing high fidelity models and algorithms to manage them. Differently from proprietary and commercial solutions that require tool-specific knowledge and training, JSDL 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 specific domain aspects. The chapter concludes, in Section 6.4, with some considerations.

6.2 SISO Space Reference FOM

Although HLA is increasingly used in the Space domain to meet the requirements for simulation interoperability in the US, Europe and Asia, so far different organizations and projects have developed incompatible FOMs (see Chapter 2) to meet their specific needs but increasing the long-term cost for interoperability. In this context, the availability of a reference FOM for the Space domain will enable the development of interoperable HLA-based simulators and related joint projects and collaborations among worldwide organizations involved in the Space domain (e.g., NASA, ESA, ASI, JAXA and Roscosmos) [69].

However, there is currently no reference FOM that addresses Space exploration, since, for example, the RPR FOM is restricted to defense operations in a geocentric environment running in real time [68].

To fill this void, a Product Development Group (PDG) has been recently activated in SISO with the aim to provide a Space Reference FOM for international collaboration on Space systems simulations [91]. Members of the PDG come from several countries and contribute experiences from projects within NASA, ESA and other organizations. Participants represent government, academia and industry. Moreover, the PDG benefit from the wide experience gained in the “Simulation Exploration Experience” (SEE) (formally Smackdown) SISO’s university outreach program [31, 90] (see Section 3.3). Indeed, competencies from NASA and other organizations have been reused in the SEE project to create a core Space Reference FOM. Approximately fifteen different university teams have successfully used this FOM for six consecutive integration projects during the last six years, thus providing a solid base for the SISO standardization initiative.

The Space Reference FOM shall support interoperability for space simulations [69]. This includes federations executing in real-time as well as federations executing in logical-time (including as-fast-as-possible). The primary focus is on training, analysis, mission support and engineering although other types of usage, like test and concept exploration may also be supported to some degree. The standard consists of two parts: (i) the *SISO Standard for*

the Space Reference FOM Federation Agreement. This is a natural language, human readable overview, description and specification of the FOM; (ii) The *Space Reference FOM*. This is a set of computer-interpretable HLA IEEE 1516-2010 FOM modules (XML files), intended for consumption by HLA run-time infrastructure and other software tools. These outcomes are expected to make collaboration politically, contractually and technically easier. It is also expected to make collaboration easier to manage and extend. The Space Reference FOM provides for baseline interoperability. Project specific modules that extend it can be added as needed and commonly used extensions can be added to the standard as they mature. The first version of the standard under release focuses on handling of time and space; in particular, the Space Reference FOM provides the following: (i) a flexible positioning system using Coordinate Reference Frames for arbitrary bodies in space, (ii) a naming conventions for well-known Reference Frames, (iii) definitions of common time scales, (iv) federation agreements for common types of time management with focus on time stepped simulation, and (v) support for physical entities, such as space vehicles and astronauts [69].

6.2.1 The Space Reference FOM

The Space Reference FOM defines a hierarchy of *object* and *interaction classes* for HLA that provides interoperability between simulations in the Space systems domain. It is designed to link simulations of discrete physical entities into distributed collaborative simulations of complex Space related systems. Its capabilities include representations of:

- Physical entities such as mobile surface systems, atmospheric flight systems, space flight systems, lifeforms, infrastructure elements, and interactions between them.
- Collections of individual entities collected as a single aggregate entity.
- Environmental objects and processes.
- Communications between entities.
- Emissions generated by entities.
- Logistics, including repair and resupply.

The HLA Object and Interaction classes are grouped in separate FOM modules (XML files) so as to allow for a more flexible and effective management of the standard proposal as well as of its extension.

Figure 6.1 shows the architecture of the SISO Space Reference FOM along with its modules.

In the following Subsections, the five modules of the SISO Space Reference FOM with their UML Class diagrams are described in detail.

The SISO_Space_FOM_switches module

The *SISO_SpaceFOM_switches* module provides configurations settings for the Federation execution by way of global Federation execution wide switches for

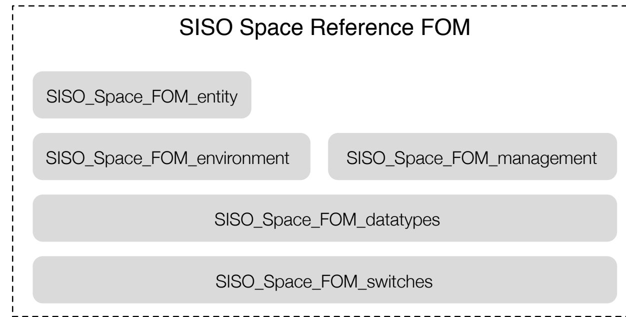


Fig. 6.1: Architecture of the SISO Space Reference FOM.

LRC (Local Run-Time Component) and RTI behavior [53]. Indeed, the 1516-2010 HLA standard defined a set of switches that shall be set in the FOM. These switches regulate the behavior of some of the optional actions the RTI can perform on behalf of the Federate, such as automatically requesting updates of an instance *attribute* when an *object* instance is discovered or advising the Federates when certain events occur. To facilitate easy replacement of these settings, for the modular version of the HLA 1516-2010 Space FOM the switches have been confined to the *SISO_SpaceFOM_switches* FOM module. It is expected that federations might choose to update this module based on their federation agreement.

Figure 6.2 shows the architecture of the *SISO_SpaceFOM_switches* module through the use of an UML Class diagram.

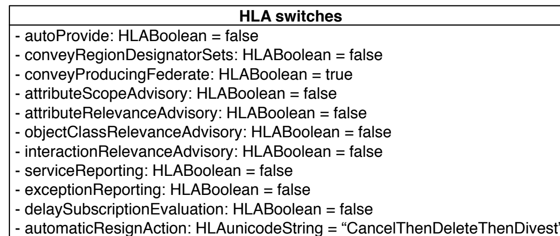


Fig. 6.2: UML Class diagram of the *SISO_SpaceFOM_swithces* module.

The SISO_Space_FOM_datatypes module

Figure 6.3 shows the architecture of the *SISO_SpaceFOM_datatypes* module through the use of an UML Class diagram.

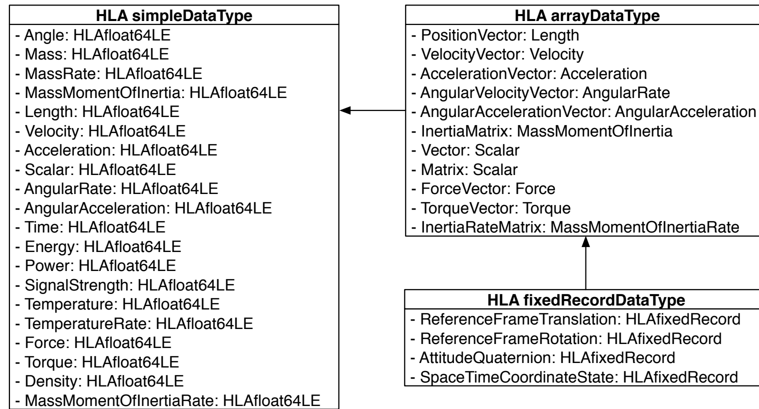


Fig. 6.3: UML Class diagram of the *SISO_SpaceFOM_datatypes* module.

The *SISO_SpaceFOM_datatypes* module provides the definitions of: (i) *HLA simpleDataTypes*, for handling the main scalars physical quantities (Angle, Mass, MassRate, MassMomentOfInertia, Length, Velocity, Acceleration, Scalar, AngularRate, AngularAcceleration, Time, Energy, Power, SignalStrength, Temperature, TemperatureRate, Force, Torque, Density, MassMomentOfInertiaRate); (ii) *HLA arrayDataTypes*, for handling vectors physical quantities (PositionVector, VelocityVector, AccelerationVector, AngularVelocityVector, AngularAccelerationVector, InertiaMatrix, Vector, Matrix, ForceVector, TorqueVector, InertiaRateMatrix); (iii) *HLA fixedrecordDataTypes*, for handling the spacetime coordinates and states of reference frames (see the *SISO_Space_FOM_environment* module subsection) . Moreover, the definition of the HLA logical timestamp and lookahead time are also provided (both represented as 64 bits integers: *HLAinteger64Time*) [53]. The above introduced *data types* are used for *object attributes* as well as *interaction parameters* and adopt the International System of Units (SI) wherever possible.

The SISO_Space_FOM_environment module

The *SISO_SpaceFOM_environment* module provides the fundamental data types used to represent the basic physical environmental properties associated with space-based simulations. In particular, it defines the *ReferenceFrame* object class that represents a fundamental concept for representing when and where any physical entity exists in time and space [69].

Figure 6.4 shows the architecture of the *SISO_Space_FOM_environment* module through the use of an UML Class diagram.

The *ReferenceFrame* object class is defined as an observational reference frame along with a companion right-handed orthogonal set of coordinate axes

that are fixed in the frame. It is characterized by the three attributes: (i) *name*, a unique name for a reference frame instance; (ii) *parent_name*, a string that must correspond to the name attribute of some other *ReferenceFrame* object instance in the simulation or empty for a 'root' reference frame; and (iii) *state*, a four dimensional representation of the *space-time coordinate state* of a reference frame with respect to its parent reference frame and expressed by using a *SpaceTimeCoordinateState* fixed record data type (see *the SISO_Space_FOM_datatypes* module subsection). If the *parent_name* is an empty string, then only the time dimension has meaning.

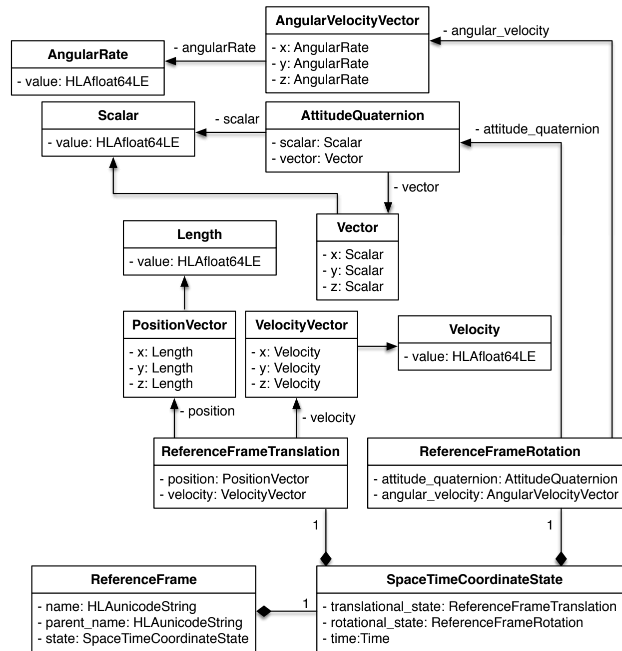


Fig. 6.4: UML Class diagram of the *SISO_SpaceFOM_environment* module.

The *time* field in the *SpaceTimeCoordinateState* specifies the simulated physical time, which represents the time dimension associated with a reference frame state. The other fields in a *SpaceTimeCoordinateState* are the *translational_state* and *rotational_state*. Indeed, many applications require knowledge of the relative attitude of one frame with respect to another. This results in three dimensions of position (*translational_state*), three dimensions of attitude (*rotational_state*) and one dimension of time. The *translational_state* field represents the reference frame's translational state with respect to its parent frame (if the frame has no parent, this attribute is meaningless) in terms of:

(i) the *position* (a *PositionVector*) of the subject frame origin with respect to the referent origin with components expressed in the referent coordinate axes; (ii) the *velocity* (a *VelocityVector*) of the subject frame origin with respect to its referent origin with components expressed in the referent coordinate axes. The *rotational_state* field represents the rotational state of a reference frame with respect to a 'referent' frame in terms of: (i) an *attitude_quaternion* (an *AttitudeQuaternion*) that specifies the orientation of the subject frame with respect to the referent; (ii) the *angular_velocity* (an *AngularVelocityVector*) of the subject frame with respect to the referent with components resolved onto the subject coordinate axes.

In a given simulation scenario (e.g., a mission to Mars), each reference frame has a parent reference frame in which its position and attitude are expressed (except for a *root* reference frame). Thus, it is possible to organize the set of reference frames, useful to represent the coordinates of the involved space entities, in a *rooted tree* structure (a *rooted directed acyclic graph*), provided that there is only a root reference frame and the others have at least that root as *highest common ancestor* (an example of reference frame tree is shown in Figure 6.5).

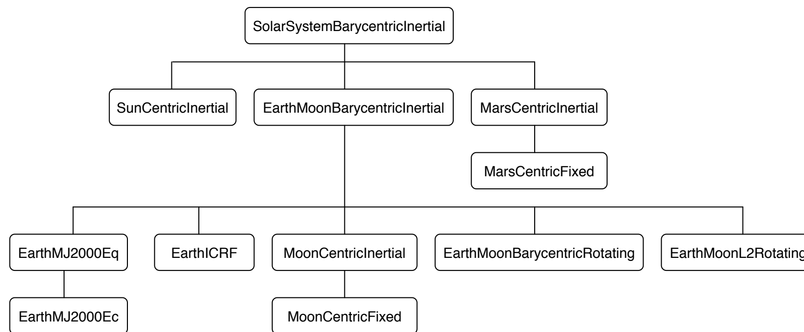


Fig. 6.5: A Reference Frame Tree.

Given a *reference frame rooted tree* structure, it is possible to transform a *space-time coordinate* expressed in a starting reference frame to those expressed in a target reference frame. The transformation is performed by following in the tree the path that goes from the source to the target reference frame through the *lowest common ancestor* and by using the information provided by the *state* of the *ReferenceFrames* along the path. Those transformations are based on well-known formulas from quaternion algebra [60]. This capability is very important when the mission under consideration involves entities operating in different and distant regions of space (e.g., on or close to different celestial bodies in the solar system) and that can also travel among them. Indeed, for expressing the *space-time state* of each entity with

the adequate precision it is required to refer to the reference frame centered in the closest point to that entity. As an example, if a spacecraft is orbiting the Moon a good choice could be to represent its coordinate in the *MoonCentricInertial* reference frame (a reference frame centered in the Moon and with fixed axes directions independent of the Moon's rotation), then when the spacecraft leaves the moon to travel to Mars the reference frame can change to *SolarSystemBarycentricInertial* (a reference frame centered in the center of mass of the solar system and with fixed axes directions); finally, when the spacecraft reaches Mars, the right choice might be *MarsCentricInertial* (a reference frame centered in Mars and with fixed axes directions). A similar situation happens when considering a simulation involving entities operating on all the above mentioned celestial bodies.

In order to be compliant with the Space Reference FOM the following rules shall be respected: (i) all reference frames used in a Federation execution shall be documented in the associated Federation Agreement; (ii) only one root reference frame shall exist within a Space FOM compliant federation execution; (iii) all reference frame parent frames shall exist as owned published object instances when the federation execution is running (e.g., advancing time). Moreover, along with the Space FOM, a recommended set of standard reference frames is provided as well as naming conventions, defined using EBNF (Extended Backus-Naur Form) notation, to correctly construct the name of any non-standard reference frame according to the Space FOM recommendations. These guidelines should enable a-priori interoperability without limiting the flexibility in the definition of Space FOM compliant Federations [69].

The SISO_Space_FOM_management module

The *SISO_Space_FOM_management* module provides the specifications for execution control and management objects, interactions and synchronization points. These are used to convey federation mode transition information and to coordinate federation mode transition behavior [69].

The UML Class diagram in Figure 6.6 shows the architecture of the *SISO_Space_FOM_management* module.

The *ExecutionConfiguration* is a standard HLA ObjectClass which defines the base set of parameters necessary to coordinate federation and federate execution time lines and execution mode transitions in a Space Reference FOM compliant federation execution. It is composed of six attributes: (i) *root_reference_frame*, which specifies the name of the root reference frame in the federation execution's reference frame tree. This frame shall remain fixed throughout the federation execution; (ii) *scenario_time_epoch*, which is the beginning epoch of the federation execution expressed in Terrestrial Time (TT), using the Truncated Julian Date (TJD) origin (1968 May 24, 00:00:00 UTC) as the TT epoch; (iii) *current_execution_mode*, which represents the current running state of the federation execution in terms of a finite set of states expressed as a *ExecutionModeEnum* enumeration value; (iv) *next_execution_mode*, which

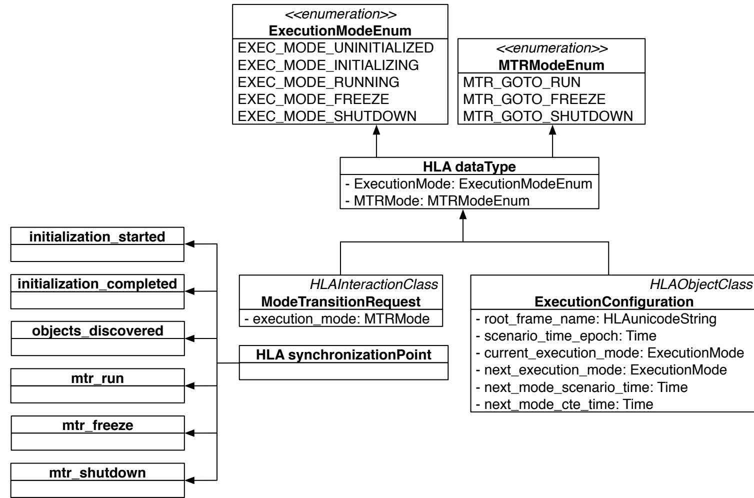


Fig. 6.6: UML Class diagram of the *SISO_SpaceFOM_management* module.

is the next running state of the federation execution in terms of a finite set of states expressed in the *ExecutionModeEnum* enumeration value. This is used in conjunction with synchronization point mechanisms to coordinate federation execution mode transitions; (v) *next_mode_scenario_time*, which defines the time for the next federation execution mode expressed as a simulation scenario time [69]; and, (vi) *next_mode_cte_time*, which is the time for the next federation execution mode change expressed as a Central Timing Equipment (CTE) time reference [69].

The *SISO Space Reference FOM* defines the concept of *Master federate* that represents the principal control federate in the federation execution (see Subsection *Execution control of a compliant federation*)[69]. It is responsible for coordinating and controlling the execution state of the federation through the use of the *ExecutionConfiguration* ObjectClass and a collection of synchronization points (*initialization_started*, *initialization_completed*, *objects_discovered*, *mtr_run*, *mtr_freeze* and *mtr_shutdown*) [69]. The *ModeTransitionRequest (MTR)* InteractionClass is used by participating federates to request a federation execution mode transition to the Master federate. The MTR has one parameter: *execution˙mode*, which represents an execution mode requested defined in terms of a finite set of states expressed in the *MTRModeEnum* enumeration value.

The SISO_Space_FOM_entity module

The *SISO_SpaceFOM_entity* module provides the definitions of vehicle related object classes. In particular, it defines the *PhysicalEntity* object class that

represents a man-made vehicle or a major sub-element of a man-made vehicle [69]. Figure 6.7 shows the architecture of the *SISO_SpaceFOM_entity* module through the use of a UML Class diagram.

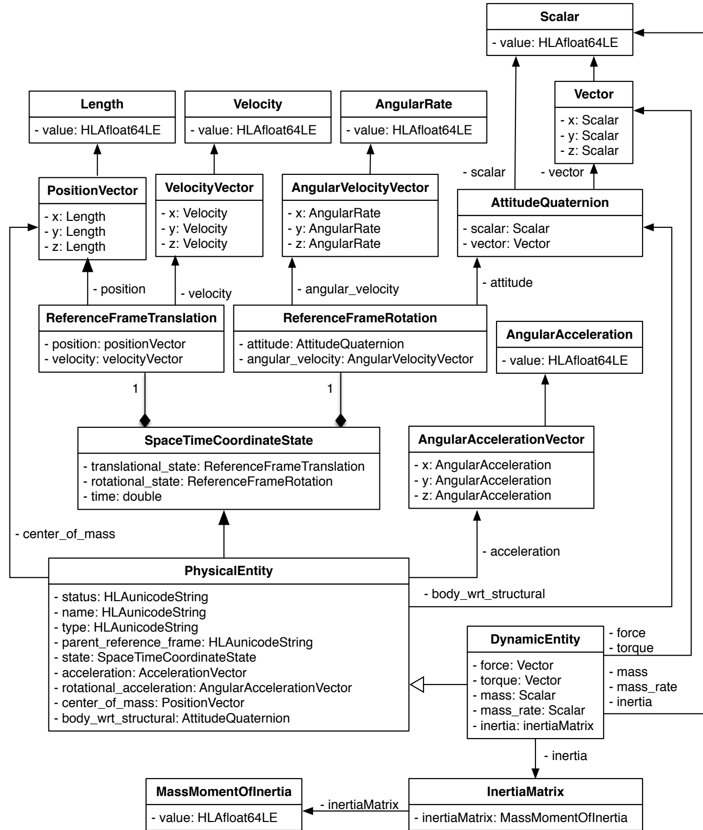


Fig. 6.7: UML Class diagram of the *SISO_SpaceFOM_entity* module.

Space vehicles have two reference frames intrinsically attached to them: a 'body frame' and a 'structural frame'. The body frame origin is the vehicle center of mass, whereas the structural frame is located at some well-defined point on the vehicle, but this point is not specified in the FOM. The offset of the body frame origin from the structural frame origin is captured as the vehicle's center of mass location attribute. The relative orientation of the structural frame with respect to the body frame is assumed fixed (not time varying), but it is not specified in the FOM. All dynamics of the vehicle are calculated by propagating the body frame with respect to the vehicle's 'parent

reference frame' which is an object instance in the *ReferenceFrame* hierarchy and is named by the vehicle's *parent_reference_frame* attribute.

The *PhysicalEntity* object class is designed to provide a basis for the individual entities that are the principal participants in Space FOM federations. The current definition of the *PhysicalEntity* object class is based on the prototype that has been used in the SISO SEE project [90] and that is going to be improved and extended during the standardization activity. The core attributes shared by all entities include the entity's *state* with respect to a defined *parent reference frame*.

The *DynamicalEntity* object class extends the *PhysicalEntity* object class to provide additional attributes associated with a maneuvering spacecraft. Specifically, it provides additional force and torque attributes used to provide additional information associated with vehicle effectors and environmental effects. These can be used for both visualization and to improve state propagation between updates. Other extensions of the *PhysicalEntity* object class, as well as of the *DynamicalEntity* object class, can be defined on the basis of the specific simulation needs.

6.2.2 Execution control of a compliant federation

The SISO Space Reference FOM does not define only the above described modules but also sets guidelines on how to define, create and execute a compliant federation.

Every HLA simulation has an executive that controls the execution of the simulation as it starts up, goes through a defined initialization sequence, transitions into various running states, and ultimately goes through a defined shutdown sequence. Interoperability between federates in a federation execution requires not only the specification of the information exchange between federates but also the specification of executive behavior. In this context, the Space Reference FOM defines some specifics of the execution control required for a Space FOM compliant federate. More in detail, the Space Reference FOM designates the role of:

- *Master Federate*, which represents the principal federate for controlling and coordinating the federation execution. It makes use of three principal HLA mechanisms to manage execution control: Execution Control Objects, Mode Transition Request (MTR) Interactions, and coordination synchronization points (see the *SISO_Space_FOM_management* module Subsection).
- *Pacing Federate*, which is a federate in the federation execution that manages the advancement of HLA logical time during the simulation execution. At the moment, the SISO Space Reference FOM provides functionalities to manage only time-stepped federations [69].
- *Root Reference Frame Publisher Federate*, which is responsible for the registration and management of the reference frame tree (see the *SISO_Space_FOM_environment* module Subsection).

- *Early Joiner Federate*, which is a federate that joins the federation execution during the initialization of the simulation execution (e.g. required federates).
- *Late Joiner Federate*, which is a federate that joins into the federation after initialization has been completed.

For each Federate role, the SISO Space Reference FOM standard defines the schemes of initialization, execution with related life cycle and termination. Thus, whatever the role played by a federated, it must respect the constraints defined in the schemes to be compliant with the standard [69].

6.2.3 Supporting Software

The Space Reference FOM is a SISO effort that will result in a published standard, not in software implementations; however, during the SEE Project [90] and the standardization effort, several pieces of software have been developed to support and test the principles of the standard; in particular: NASA has provided federates for the space environment (reference frames) as well as for visualization; COTS providers, like Pitch, ForwardSim and VT MÄK, have provided supporting software tools, both general (RTI, FOM development, federate development, data logging) and Space FOM specific. A tailor version of the HLA Development kit framework (see Chapter 3) that aims at easing the development of Federates and Federations compliant with the SISO Space Reference FOM is under developing.

By using these tools, the developers could focus on the specific aspects and behaviors of their federates by delegating to the services provided by the underlying software layers the management of the common aspects and functionalities related to the standard.

6.2.4 Conclusion

The SISO Space Reference FOM standardization initiative presented in this chapter aims at supporting the development of interoperable simulations of complex space systems and missions, and enhance a priori interoperability among Space FOM users. The principal intended areas of use are training, analysis, mission support and engineering. However, other areas of use, like test and concept exploration, are also supported. The benefits of the proposed Reference Space FOM include: (i) *Interoperability*, the ability for several simulations, each focusing on particular tasks, to interoperate and jointly create a collaborative simulation with wider and richer contexts; (ii) *Composability*, the ability to build collaborative simulations from components that can be combined in different ways, with new or existing simulations, to reach a particular goal; and, (iii) *Reusability*, the ability to use existing simulations in new contexts. It will be possible to build generic and reusable simulations and tools for the Space domain based on the Space FOM.

The SISO Space Reference FOM initiative builds upon many years of simulation experience by professionals in government organizations, industry and academia. Early prototypes of the Space FOM have been tested in the SISO/SCS programs called "Smackdown" and "Simulation Exploration Experience". The SISO working group is going to promote and fully experiment the first release of the standard in ongoing projects involving worldwide organizations active in the Space domain (e.g., NASA, ESA, Roscosmos, and JAXA).

6.3 Java Space Dynamics Library (JSDL)

6.3.1 Introduction

The space flight dynamics domain is one of the many technology fields involving many actors from several organizations belonging to different scientific domains such as mathematical, physical, aerospace and software engineering.

Due to the increasing complexity of space systems and thus of the related engineering problems; new methods, tools and software libraries have been developed in each of these organizations primarily for specific needs and later generalized so as to make them modular, flexible and reusable [51, 87]. These available, commercial and noncommercial, solutions support one or more of the phases in the development of space systems such as flight mechanics, propulsion, orbit controls and data analysis, however none of them seems capable of providing complete coverage of the whole development process in a flexible way [84].

In this context, there is an increasing need for efficient and flexible solutions capable of covering all the steps in the design and develop of space systems, especially for supporting modeling and simulation systems where modularity, flexibility and reusability are key features to provide [84].

In this Section the *Java Space Dynamics Library (JSDL)* project is described, emphasizing its flexibility and showing the set of services provided to define and build space systems such as satellites and vehicles. The rest of the Section is structured as follows. Related works are discussed in Subsection 6.3.2. Subsection 6.3.3 presents the *Java Space Dynamics Library (JSDL)* project with particular focus on the architecture and main services provided. Finally, in Subsection 6.3.4 conclusions are drawn and future research directions are delineated.

6.3.2 Related works

There are several research efforts on the development of methods, tools and libraries in the astrodynamics field, mainly aiming at providing a robust and flexible way for defining, building and simulating complex elements in space. The most applicable solutions have been developed after the mid-1960's when

space missions were the attention of media and computers become prevalent in academia and industry.

The *Java Astrodynamics Toolkit (JAT)* is an open source library of reusable components, distributed under the GNU General Public License (GLP). It is implemented in the Java language and helps developers to create their own application programs and solve problems in astrodynamics, mission design, spacecraft navigation, guidance and control. It provides functionalities that allow the rapid development of spacecraft simulations including 2D and 3D visualization capabilities. Possible applications of JAT include: (i) Design and analysis of space missions, including trajectory optimization; (ii) Simulation of spacecraft navigation, guidance and control as well as its visualization in a 3D environment; and, (iii) Simulation of the motion for basic rigid and flexible spacecraft dynamics [44].

Another software library that enables developers to effectively define and manage elements in space is *Orbits Extrapolation Kit (Orekit)* [19]. Orekit is implemented in the Java language and aims at providing accurate and efficient low level standard astrodynamical models (e.g., time, frames, orbital parameters, orbit propagation, attitude and celestial bodies) and algorithms (e.g., time conversions, propagations and pointing) for the development of flight dynamics applications. It is designed to be easily used in very different contexts, from quick studies up to critical operations. It was developed in 2002 at CS Systémes d'Information and was officially released as an open source software, under the Apache License Version 2.0, in 2008 [83].

European Space Agency (ESA) engineers have been developing several spacecraft simulation tools that form the *Mission - Customer Furnished Item (CFI) Software (Mission CFI)*. It includes the following products [27]:

- The *Earth Observation CFI (EOCFI)* software, which is a collection of multiplatform precompiled C libraries for timing, coordinate conversions, orbit propagation, satellite pointing calculations, and target visibility calculations, specifically parametrized and configured for EO satellites.
- The *EO Orbit and Attitude Adapter (EO Adapter)*, which is part of the Earth Observation Mission Software Suite. It is a tool/library to generate Orbit and Attitude files compliant with EOCFI format using data extracted from one or more binary files, for example files containing Telemetry packets including Orbit and Attitude information.
- The *Envisat CFI* software, which is a collection of multiplatform precompiled C libraries for timing, coordinate conversions, orbit propagation, satellite pointing calculations, and target visibility calculations, specifically parametrized and configured for the Envisat satellite.

The JSDL project presented in this Section stems from the SISO Space Reference FOM standardization initiative carried out by the SISO Space Reference FOM (SRFOM) Product Development Group (PDG) (see Section 6.2). JSDL aims at supporting the development of complex space systems by providing high fidelity models and algorithms to manage them. Differently from

proprietary and commercial solutions that require tool-specific knowledge and training, JSDL 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 specific domain aspects. This license allows anybody to build both commercial and noncommercial applications without restrictions or limitations from the use of JSDL.

In the following Subsections, the JSDL project is described in details by highlighting its architecture and functionalities.

6.3.3 The JSDL project

Java Space Dynamics Library (JSDL) is a low level space dynamics library that facilitates the design and development of space systems, such as space vehicles and satellites. The open source nature of the library allows developers to investigate and customize the architecture and functionalities defined in the source code to fit their own needs.

The JSDL 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 SISO Space Reference FOM (SRFOM) Product Development Group (PDG) (see Section 6.2). The primary goal of JSDL is to provide 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. It is fully implemented in the Java programming language and provides a consistent set of functionalities for developing and running complex elements in space such as, time scales, reference frames, orbital parameters, orbit propagation, and attitude.

The JSDL provides to developers the following resources: (i) a *technical documentation* that describes the library with its philosophy and mission; (ii) a *user guide* to support developers in the use of the library; and (iii) a set of *reference examples* that show how to create space objects.

In the following, the attention is focused on the architecture and services provided by the library.

Architecture of the JSDL

The JSDL library depends only on the Java Standard Edition version 7 (or above), *Apache Commons Math* [97] version 3.6 and *JDateTime* [101] version 3.8 libraries at runtime. The JSDL provides a set of services, each of which defines some Java classes and interfaces that enable specific functionalities. The JSDL architecture is shown in Figure 6.8.

Space Applications. Contains the space applications that are built using the functionalities provided by the JSDL. An application can interact with the *Apache Commons Math* and *JDateTime* directly or through the JSDL library.

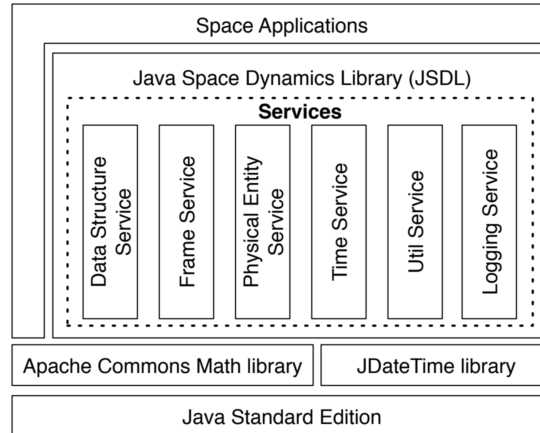


Fig. 6.8: The Architecture of the JSDL library.

Java Space Dynamics Library (JSDL). It is the core library for creating Space applications. It provides a set of features useful for modeling objects in space. The complexity of the features provided is hidden behind an intuitive set of APIs.

Apache Commons Math library. It is a standard library of lightweight, self-contained mathematics and statistics components addressing the most common practical problems not immediately available in the Java programming language [97].

JDateTime library. It is a library that offers a very precise way to track dates and time. It uses well-defined and proven astronomical algorithms for time manipulation [101].

In the following Subsections, the six JSDL services with their UML Class diagrams are described in detail.

Data Structure Service

The *Data Structure Service* defines functionalities that ease working with complex data structures. It provides a very useful set of data structures (tree and queue) to build and manage *Reference Frames* and *Physical Entities* with their transformations.

The structure of the *Data Structure Service* is shown in Figure 6.9 by using a UML Class Diagram.

The *LinkedNTree* is a generic class that stores elements hierarchically where each element has a parent element and zero or more children elements. It implements the *Tree* interface that defines some functionalities to handle a tree such as *height()*, *depth()*, *root()* and *size()*. Moreover, all the common

traversal schemes for trees are provided: *LevelOrderIterator*, *PreOrderIterator*, *InOrderIterator* and *PostOrderIterator*.

The *Queue* class provides a *queue* data structure that follows the First-in First-out (FIFO) strategy. Elements can only be added to the end (enqueue) and only be removed from the front (dequeue). The *queue* has been implemented by using a Java standard *LinkedList* and provides two methods *enqueue()* and *dequeue()* to perform each task respectively.

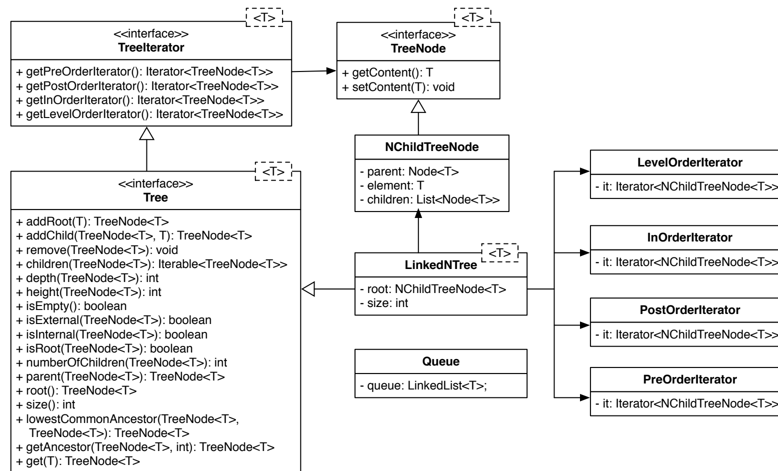


Fig. 6.9: The architecture of the Data Structure Service.

Frame Service

Reference frame is a fundamental concept for representing when and where a physical entity exists in time and space [69]. This representation is referred to as the state of the entity. In order to represent the state of something, it is necessary to express that state with respect to some time scale and some referent coordinate system. This combination of time and coordinate system is referred as a *Space-Time Coordinate* or *Reference Frame*.

The structure of the *Frame Service* is shown in Figure 6.10 by using a UML Class Diagram.

The *Frame Service* provides functionalities to handle *Reference Frames*. It includes the fundamental *ReferenceFrame* class that represents a single frame. Each *Reference Frame*, as defined in the *SISO Space Reference FOM* (see Section 6.2), is composed of three attributes:

- *name*, which represents the unique name of the reference frame.

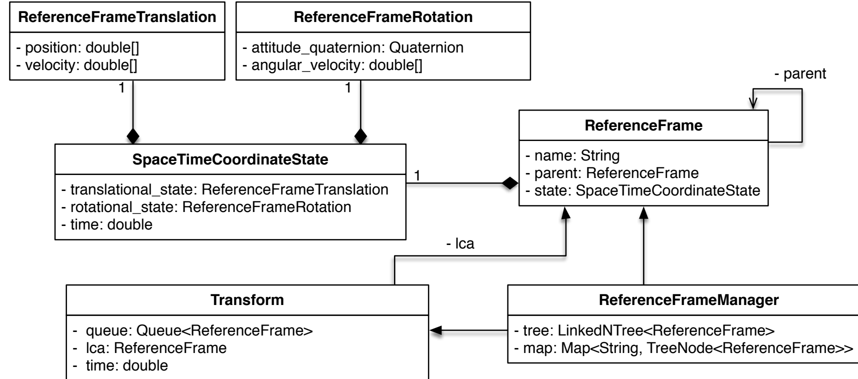


Fig. 6.10: The architecture of the Frame Service.

- *parent*, which is the parent *Reference Frame*. If the *parent* attribute is 'null', the *Reference Frame* is the *root* frame.
- *space-time coordinate state*, which defines through the *SpaceTimeCoordinateState* class a four-dimensional representation of the space-time coordinate state with respect to its parent reference frame [69]. It consists of:
  (a) *Translational state information*, which provides through the *ReferenceFrameTranslation* class a position vector $\vec{r}$ from the origin of the parent reference frame to the origin of the reference frame. It also provides a velocity vector $\vec{v}$ for the motion the reference frame with respect to the parent frame. Both of these vectors are expressed with respect to the parent reference frame. These vectors can be used to describe the translational position and motion of a frame with respect to its parent.
  (b) *Rotational state information*, which provides through the *ReferenceFrameRotation* class an attitude quaternion $\tilde{q}$ that describes the attitude of the reference frame with respect to its parent frame. It also provides an angular velocity vector $\vec{\omega}$ that describes the rotational motion of the reference frame with respect to the parent frame expressed in the subject frame's coordinates. $\tilde{q}$ and $\vec{\omega}$ can be used to describe the attitude and rotational motion of a frame with respect to its parent.
  (c) *Time*, which contains information about the time t to which the *space-time coordinate state* corresponds.

As shown in Figure 6.11, all *Reference Frames* are organized as a tree that is formed from a single base root node with directed paths from an arbitrary number of child nodes. These child nodes can then have directed paths from other arbitrary sets of child nodes.

The translational and rotational information can be used to transform a generic vector expressed in a given reference frame $\vec{r}_{child}$ into a vector ex-

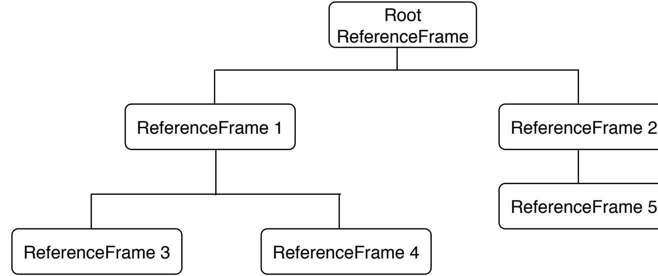


Fig. 6.11: Tree of ReferenceFrames.

pressed in its parent frame $\vec{r}_{parent}$. In turn, the vector $\vec{r}_{parent}$ now expressed in the parent frame can be expressed in the parent's parent frame or in another child frame of the parent frame. Chaining together sequences of transformations using the relationships established in the reference frame tree allows for transformation between any pair of frames in the reference frame tree.

Transformations are defined and managed by the *Transform* and *ReferenceFrameManager* classes. In particular, a transformation is computed by merging individual transforms while walking the shortest path between them. The walking/merging operations are handled transparently by the library. Developers only need to select the frames, provide the date and ask for the transformation, without knowing how the frames are related to each other. Transformations are defined as operators that when applied to the coordinates of a vector expressed in the *initial Reference Frame*, provide the coordinates of the same vector expressed in the *final Reference Frame*.

Equation 6.1 gives the transformation of a position vector expressed in a child reference frame into a position vector expressed in the parent reference frame [60].

$$\vec{r}_{parent} = \vec{r}_{0_parent} + \tilde{Q}(\vec{r}_{child}) \tag{6.1}$$

where $\vec{r}_{child}$ is the position vector expressed in the child reference frame, $\tilde{Q}(\vec{r}_{child})$ is the quaternion rotation operator associated with the attitude quaternion $\tilde{q}$ that defines the attitude of the child reference frame with respect to the parent reference frame, $\vec{r}_{0_parent}$ is the vector giving the position of child reference frame origin with respect to the parent reference frame origin expressed in parent reference frame coordinates, and $\vec{r}_{parent}$ is the position vector of the entity expressed in parent reference frame coordinates.

With reference to the $\tilde{Q}(\vec{r}_{child})$ operation, it is the canonical way of multiplying a quaternion $\tilde{q}$ by a vector $\vec{x}$ as given by expression 6.2 [60],

$$\tilde{Q}(\vec{x}) = \tilde{q} \cdot \vec{x} \cdot \tilde{q^*} \tag{6.2}$$

where $\tilde{q^*}$ is the conjugate of $\tilde{q}$.

The relative motion between a child reference frame and a parent reference frame is provided by the velocity $\vec{v}$ and angular velocity $\vec{\omega}$ vectors. Equation 6.3 gives the velocity of an entity expressed in the parent reference frame given the velocity of the entity expressed in the child reference frame [60].

$$\vec{v}_{parent} = \vec{v}_{0_parent} + \tilde{Q}(\vec{v}_{child} + (\vec{\omega}_{child} \times \vec{r}_{child})) \tag{6.3}$$

where $\vec{v}_{child}$ is the velocity vector of an entity expressed in the child reference frame, $\vec{\omega}_{child}$ is the angular velocity vector of the child frame with respect to the parent frame and expressed in child frame coordinates, $\vec{v}_{0_parent}$ is the velocity of the child frame with respect to the parent frame expressed in parent frame coordinates, and $\vec{v}_{parent}$ is the velocity of an entity expressed the parent reference frame.

In most cases, the position and velocity relationships above are sufficient. However, acceleration is sometimes needed and is included here for completeness. Equation 6.4 gives the acceleration of an entity expressed in the parent reference frame given the acceleration of the entity expressed in the child reference frame [60].

$$\begin{aligned}\vec{a}_{parent} = \vec{a}_{0_parent} + \tilde{Q}(\vec{a}_{child} + (\vec{\omega}_{child} \times (\vec{\omega}_{child} \times \vec{r}_{child})) \\ + (2\vec{\omega}_{child} \times \vec{v}_{child}) + (\vec{\alpha}_{child} \times \vec{r}_{child}))\end{aligned} \tag{6.4}$$

where $\vec{a}_{child}$ is the acceleration of an entity expressed in the child reference frame, $\vec{\alpha}_{child}$ is the angular acceleration of the child frame with respect to the parent frame and expressed in child frame coordinates, $\vec{a}_{0_parent}$ is the acceleration of the child frame with respect to the parent frame expressed in parent frame coordinates, and $\vec{a}_{parent}$ is the acceleration of an entity expressed in the parent reference frame.

Concerning reverse transformations; using the child to parent vector transformation equations above defined along with some vector and quaternion algebra, the resulting equation 6.5 gives the transformation of a position vector expressed in a parent reference frame into a position vector expressed in the child reference frame [60].

$$\vec{r}_{child} = \tilde{Q}^*(\vec{r}_{parent} - \vec{r}_{0_parent}) = -\vec{r}_{0_child} + \tilde{Q}^*(\vec{r}_{parent}) \tag{6.5}$$

where $\tilde{Q}^*(\vec{r}_{parent})$ is the conjugate quaternion rotation operator associated with the attitude quaternion $\tilde{q}$ that defines the attitude of the child reference frame with respect to the parent reference frame, and $\vec{r}_{0_child}$ is the vector giving the position of child reference frame origin with respect to the parent reference frame origin expressed in child reference frame coordinates [60].

Similar relationships can be derived for velocity (equation 6.6) and acceleration (equation 6.7) [60]:

$$\begin{aligned}\vec{v}_{child} &= \tilde{Q}^*(\vec{v}_{parent} - \vec{v}_{0_parent}) - (\vec{\omega}_{child} \times \vec{r}_{child}) \\ &= -\vec{v}_{0_child} - (\vec{\omega}_{child} \times \vec{r}_{child}) + \tilde{Q}^*(\vec{v}_{parent})\end{aligned} \tag{6.6}$$

$$\begin{aligned}\vec{a}_{child} &= \tilde{Q}^*(\vec{a}_{parent} - \vec{a}_{0_parent}) - (\vec{\omega}_{child} \times (\vec{\omega}_{child} \times \vec{r}_{child})) \\ &\quad - (2\vec{\omega}_{child} \times \vec{v}_{child}) - (\vec{\alpha}_{child} \times \vec{r}_{child}) \\ &= -\vec{a}_{0_child} - (\vec{\omega}_{child} \times (\vec{\omega}_{child} \times \vec{r}_{child})) \\ &\quad - (2\vec{\omega}_{child} \times \vec{v}_{child}) - (\vec{\alpha}_{child} \times \vec{r}_{child}) + \tilde{Q}^*(\vec{a}_{parent})\end{aligned} \tag{6.7}$$

Physical Entity Service

The structure of the *Physical Entity Service* is shown in Figure 6.12 by using a UML Class Diagram.

PhysicalEntity is the highest-level object class in the JSDL entity hierarchy. This class provides attributes to describe an entity's location in time and space. It also contains attributes to uniquely identify it individually from all other physical entities.

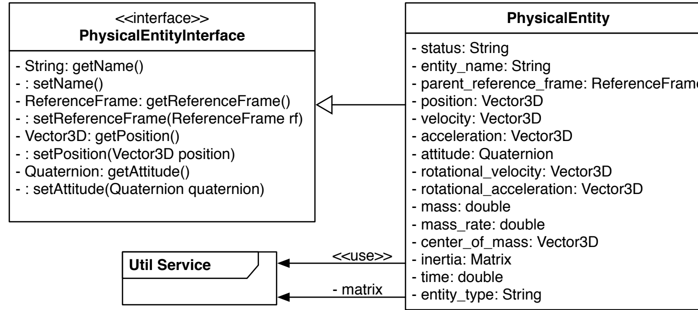


Fig. 6.12: The architecture of the Physical Entity Service.

Physical entities have two intrinsically associated reference frames: (i) a *structural frame*; and (ii) a *body frame*. These are not registered in the reference frame tree but are used to place and orient the entity in space with respect to a reference frame in the tree. The origin of the *structural frame* is located at some arbitrary but known point on the entity [69]. The *body frame* origin is at the entity's *center of mass* and is located with respect to the entity's structural reference frame by a vector from the origin of the structural reference frame to the center of mass of the entity. This vector is expressed in the entity's structural reference frame. The orientation of the entity's body frame with respect to the entity's structural reference frame is defined by an attitude quaternion [88].

The *Physical Entity Service* is designed to provide functionalities for space objects such as satellites, asteroids and vehicles. The core attributes defined in the *PhysicalEntity* class includes the position and orientation with respect

to a defined parent reference frame, which must be a reference frame instance in the reference frame tree, and a time tag in a defined time scale. This information is sufficient to unambiguously represent an entity in time and space.

Time Service

The *Time Service* allows to manage epochs, time scales, time units and to compare time instants. The structure of the *Time Service* is shown in Figure 6.13 by using a UML Class Diagram.

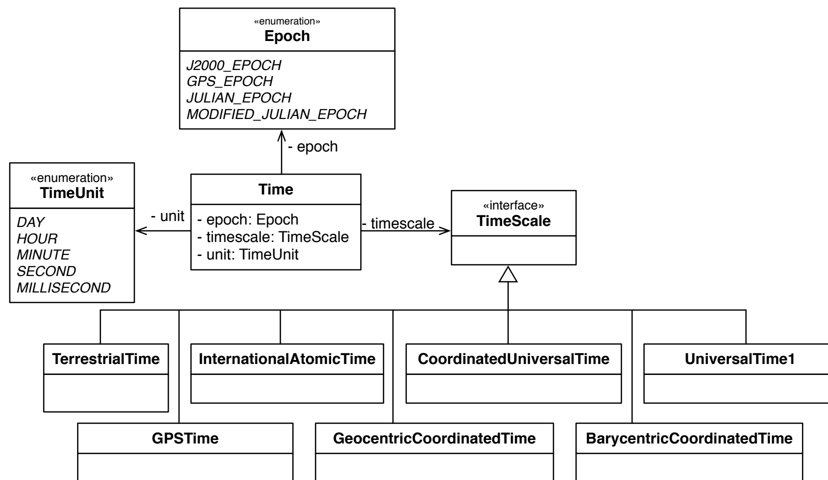


Fig. 6.13: The architecture of the Time Service.

The principal class is *Time* that represents a unique instant in time defined by specifying a point in a specific epoch (e.g., J2000, GPS and Julian epoch), time scale and time unit [69]. The *TimeScale* interface defines a set of predefined time scales:

- *Universal Time (UT)*. It is a time standard based on Earth's rotation, defined as the Mean Solar Time at the Royal Observatory in Greenwich, England. There are three variations of Universal Time. UT0 is the observed mean solar time. UT1 is UT0 corrected for polar motion, the motion of the Earth's rotational axis over the surface of the Earth, and UT2 that is corrected for seasonal variations but today it is considered obsolete.
- *International Atomic Time (TAI)*. It was introduced in 1972 and represents a high-precision atomic coordinate time standard based on the notional passage of proper time on Earth's geoid [47]. This time scale is

accurate enough to observe relativistic effects for clocks in motion or accelerated by a local gravity field. One advantage of using TAI is that it is a continuous uniform time scale. Specifically, the rate of time passage for TAI is constant unlike the Earth rotation based scales. This means that the Earth rotation based time scales diverge from TAI over time due to the variations in the Earth's rotation. TAI is exactly 36 seconds ahead of UTC. The 36 seconds results from the initial difference of 10 seconds at the start of 1972, plus 26 leap seconds in UTC since 1972.

- *Coordinated Universal Time (UTC).* It is a 24-hour time standard that is used to synchronize world clocks. UTC is defined by the International Telecommunications Union Recommendation (ITU-R TF.460-6), Standard-frequency and time-signal emissions [86] and is based on International Atomic Time (TAI) with leap seconds added at irregular intervals to compensate for the slowing of Earth's rotation. Leap seconds are inserted as necessary to keep UTC within 0.9 seconds of universal time, UT1 [21].
- *Global Positioning System Time (GPS Time).* GPS Time is the uniform time scale with a starting epoch at midnight between Saturday January 5th and Sunday January 6th, 1980 (1980 January 6, 00:00:00 UTC). GPS Time counts in weeks and seconds of a week from this instant. The GPS week begins at the transition between Saturday and Sunday. The days of the week are numbered sequentially, with Sunday being 0, Monday 1, Tuesday 2, etc. The GPS time scale begins at the GPS starting epoch with GPS week 0. Within each week, the time is usually denoted as the second of the week (SOW). This is a number between 0 and 604,800 (60 x 60 x 24 x 7). Sometimes SOW is split into a day of week (DOW) between 0 and 6 and a second of day (SOD) between 0 and 86400. While GPST is a uniform time scale, it does have rollover. To limit the size of the numbers used in the data and calculations, the GPS Week Number is a ten-bit count in the range 0-1023, repeating every 1024 weeks. As a result, the week number 'rolled over' from 1023 to 0 at 23:59:47 UTC on Saturday, 21st August 1999. This was before midnight UTC because every GPS week contains exactly 604,800 seconds, to keep the calculations consistent. The 13 intervening leap seconds had put UTC behind GPS system time. The next GPS week rollover occurs on April 6th, 2019.
- *Terrestrial Time (TT).* It is an astronomical time standard defined by the International Astronomical Union (IAU) used widely for geocentric and topocentric ephemerides. TT is defined to run at the same rate as TAI seconds but with an offset of 32.184 seconds. This offset is based on preserving continuity with other historical dynamic time scales.
- *Geocentric Coordinated Time (TCG).* It is a coordinate time standard defined in 1991 by the International Astronomical Union (IAU). It is primarily used for theoretical developments based on the Geocentric Celestial Reference System (GCRS). TCG is a relativistic time scale and since the reference frame for TCG is not rotating with the surface of the Earth and not in the gravitational potential of the Earth, TCG ticks faster than

clocks on the surface of the Earth by a factor of $6.97 \cdot 10^{-10}$ seconds. TCG, Barycentric Coordinated Time (TCB) and Terrestrial Time (TT) are defined in a way that they have the same value on January 1st 1977, 00:00:00 TAI (JD 2443144.5 TAI).

- *Barycentric Coordinated Time (TCB)*. It is a time scale, defined in 1991 by the International Astronomical Union (IAU), primarily used for theoretical developments based on the Barycentric Celestial Reference System (BCRS). TCB is a relativistic time scale and since the reference frame for TCB is not influenced by the gravitational potential caused by the Solar system, TCB ticks faster than clocks on the surface of the Earth by $1.55 \cdot 10^{-8}$ seconds. TCB, Geocentric Coordinated Time (TCG) and Terrestrial Time (TT) are defined in a way that they have the same value on January 1st 1977, 00:00:00 TAI (JD 2443144.5 TAI).

Util Service

The *Util Service* defines a number of useful functionalities, primarily transformations ones that are useful for working with *Physical Entities* in space. This service should not be considered merely a utility one that is separate from the rest of JSDL; in fact, JSDL depends directly on several of the classes defined in it. Indeed, it provides services needed to define both *Reference Frame* and *Time* objects with their standard conversions.

The structure of the *Util Service* is shown in Figure 6.14 by using a UML Class Diagram.

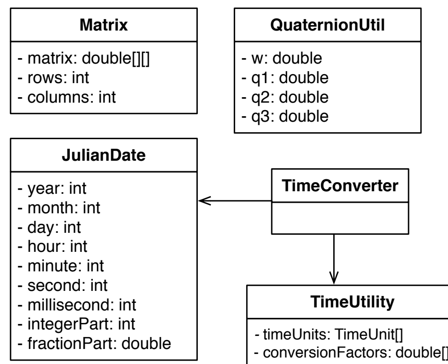


Fig. 6.14: The architecture of the Util Service.

The *Matrix* class represents a mathematical matrix. It provides methods for creating matrices, operating on them arithmetically and algebraically, and determining their mathematical properties such as trace, rank, inverse and

determinant. The *QuaternionUtil* class provides classical methods to manage quaternions such as conjugate, inverse and norm. The *JulianDate* class represents a Julian Date, which is a universal time used by all astronomers to ensure that observations are based on a universal astronomical time. It corresponds to the day, hour and minute of the observation and is the interval of time in days since noon at Greenwich on 1 January 4713 BC. Finally, the *TimeConverter* and *TimeUtility* allow to perform time conversions. Moreover, it is possible to easily convert a *JulianDate* to a standard Java *Calendar* object to have a date/time representation of it through the use of the *toCalendar(JulianDate jd)* method defined in the *TimeConverter* class. For example, the Truncate Julian Date (TJD) 17131.83333333334 can be converted in a Calendar object with value 2015 April 19, 20:00:00 UTC.

Logging Service

The *Logging Service* provides functionalities useful to both track down any problems or errors occurred during its use, and understand how the JSDL core services work. This information is stored into the *jsdl_trace.log* file.

The structure of the *Logging Service* is shown in Figure 6.15 by using a UML Class Diagram.

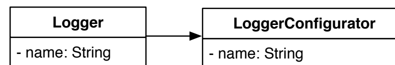


Fig. 6.15: The architecture of the Logging Service.

6.3.4 Discussion and Results

In the space flight dynamics domain many research efforts are focusing on the definition of methods, tools and software libraries, mainly aiming at providing a robust and flexible way for defining, building and simulating complex elements in space.

As discussed in the Section, due to the increasing complexity of space systems and thus of the related engineering problems; new methods, tools and software libraries have been developed in each of these organizations primarily for specific needs and later generalized so as to make them modular, flexible and reusable. The available, commercial and noncommercial, solutions support one or more of the phases in the development of space systems such as flight mechanics, propulsion, orbit controls and data analysis, however none of them seems capable of providing complete coverage of the whole development process of space simulations. To overcome this issue, the Java Space Dynamics Library (JSDL) has been created.

The Java Space Dynamics Library (JSDL) stems from the SISO Space Reference FOM standardization initiative carried out by the SISO Space Reference FOM (SRFOM) Product Development Group (PDG) [69]. JSDL is still evolving and aims at supporting the development of complex space systems by providing high fidelity models and algorithms to manage them.

6.4 Conclusion

This chapter has described further contributions focused on the interoperability of simulation models in the space domain. In particular, a first set of results achieved by the *SISO Space Reference FOM (SRFOM) Product Development Group (PDG)* concerning the Space Reference FOM for international collaboration on space systems simulations has been described in detail [69].

Then, the *Java Space Dynamics Library (JSDL)* project has been presented, emphasizing its flexibility and showing the set of services provided to define, develop and simulate complex space systems.

7

Conclusions

7.1 Main contributions

The research presented in this Thesis aimed at contributing to fill the lack of methods, models and techniques to support, in an integrated way, the *reuse* and *interoperability* of simulation models and their *execution on distributed computing environment*.

The research has been focused on the analysis of two of the most mature and widely used standards going in this direction: *IEEE 1516 - High Level Architecture (HLA)* [53] and *Functional Mock-up Interface (FMI)* [39]. Although these standards start from different objectives and are based on different techniques (see [39, 53, 56, 59]), they have several common features that have been exploited so as to address the three issues above presented and then create a full-fledged solution.

In this context, three main contributions have been provided:

- *HLA Development Kit*, a software framework that aims at facilitating the design and develop of distributed simulators compliant with the IEEE 1516.2010 - High Level Architecture (HLA) standard [53] (see Chapter 3).
- *MONADS*, a Model-Driven method that makes easier for Systems Engineers to design a complex system and simulate it on a distributed simulation environment, without asking them to explicitly deal with the intricacies and difficulties of currently available standards and technologies (see Chapter 4).
- Two methods, *HLA for FMI* and *FMI for HLA*, to address in an integrated way the issues of *reuse*, *distribution* and *interoperability* among heterogeneous simulation components through the integration of the functionalities offered by the HLA [53] and FMI [39] standards (see Chapter 5).

The *HLA Development Kit software Framework (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 [53, 99]. 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 to/from the HLA RTI; the publish, subscribe and updating of *ObjectClass* and *InteractionClass* elements [42, 30, 53]. 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) [71]. It has been successfully experimented in the Simulation Exploration Experience (SEE) project since the 2015 edition [90].

The second contribution has concerned the definition of a Model-Driven method called *MONADS (MOdel-driveN Architecture for Distributed Simulation)*. The MONADS method aims at facilitating the distributed simulation of complex systems, specified by using SysML [74], according to the Model-Driven Systems Engineering (MDSE) paradigm. MONADS leverages the strengths of both the capabilities provided by robust and well-known OMG modeling languages (e.g., UML and SysML), appropriate to define the architectural and behavioral aspects of complex systems, and the functionalities given by the *HLA Development Kit software Framework (DKF)*, suitable for defining distributed simulations according to the IEEE 1516-2010 - High Level Architecture (HLA) standard [53]. Indeed, the HLA simulation code, generated starting from SysML/UML models by a chain of *model-to-model* and *model-to-text* transformations, is based on the DKF. MONADS is the result of an intensive experimentation in several application domains (avionics, aerospace, drones) carried out working in cooperation with the Laboratory of Software Engineering, Department of Enterprise Engineering of the University of Tor Vergata (Rome).

Concerning the third contribution, two methods to address, in an integrated way, the issues of *reuse*, *distribution* and *interoperability* among heterogeneous simulation components through the integration of the functionalities offered by the HLA [53] and FMI [39] standards have been defined. These two methods devoted to fruitfully combine the standards are: (i) *HLA for FMI*, in which a FMU is enriched with HLA features and services. In more detail, a set of XML tags have been defined and added in the model-description XML file to natively integrate the HLA concepts; and, (ii) *FMI for HLA*, in which simulation modules that are available as FMUs are reused in a HLA simulation environment without modifying them. The effectiveness of the *FMI for HLA* method has been evaluated through a case study in the space domain concerning the integration of a FMU module into a HLA simulation. The simulation scenario took place on the lunar surface and concerned a situation in which a lander, defined as a FMU, was landing on the platform located in the Moon base. This research activity has been also experimented in the MODRIO (Model-Driven Physical Systems Operation) ITEA3 project [57].

The experience gained during the definition and development of the *HLA Development Kit (DKF)*, the definition of the *MONADS (MOdel-driveN Architecture for Distributed Simulation)* method, and the investigation on how to combine the international standards *IEEE 1516 - High Level Architecture (HLA)* and *Functional Mock-up Interface (FMI)*, along with the research activities performed at NASA Lyndon B. Johnson Space Center (JSC) and the contribution given to the *SISO Space Reference FOM (SRFOM) Product Development Group (PDG)* [69], allowed to focus on the interoperability of space systems in a distributed simulation. In this context, the *Java Space Dynamics Library (JSDL)* that facilitates the design, development and simulation of space systems, such as space vehicles and satellites has been created (see Chapter 6). It represents a low level space dynamics library and provides 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.

7.2 Ongoing and Future Work

The results presented in this Thesis constitute a starting point for ongoing and future research activities.

One of these concerns how to extend the *HLA Development Kit (DKF)* to provide support for event-driven simulations. The DKF framework has been developed in the context of the "Simulation Exploration Experience (SEE)" project [90] and the current version provides support only for the develop of time-stepped simulations as it is the SEE reference simulation model. The next release of the framework will include this support.

In addition, a tailor version of the DKF that aims at easing the development of Federates and Federations compliant with the SISO Space Reference FOM ongoing standard is under developing.

Another future work includes the improvements of the proposed *MONADS* method, by introducing some approaches and possible patterns to define and simulate complex systems based on the Modelica language [36] and OpenModelica simulation environment [78].

Concerning the integration of the functionalities offered by the HLA [53] and FMI [39] standards, future research efforts will be devoted to further analyze and experiment the joint exploitation of FMI and HLA in different simulation domains so to prove not only the benefits that derive from their combinations but also the effectiveness of the proposed solutions. Moreover, the advantages in the generation of simulators that could derive from the exploitation of Model-Driven approaches that combine HLA and FMI features

will be also investigated.

Regarding the *Java Space Dynamics Library (JSDL)*, the current version has been successfully tested in the 2017 edition of the “Simulation Exploration Experience” project [90]. Future research activities will be devoted to further analysis and experimentations of the effectiveness of the library and to define other models and algorithms so as to follow the evolution of the *SISO Space Reference FOM* that is supported in the context of the related SISO SRFOM PDG [69].

A

Appendix

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

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

```
1 @ScheduledMethod(start=1, interval=1)
2 public void step(){
3     //This updates its map with the next target from the UAV
4     updateMap(ExcavatorMap, UAVx, UAVy);
5     checkCargoLimit(returnOrigin);
6 
7     //This moves the Excavator X,Y
8     moveExcavator(grid.getLocation(this), returnOrigin, ExcavatorMap,
      EXCx, EXCy);
9 }
```

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

```
1 private void publishAndSubscribe() throws RTIexception {
2     // get all the handle information for the attributes of ObjectRoot.
      Excavator
3     this.classHandle = rtiamb.getObjectClassHandle("HLAobjectRoot.
      Excavator");
4     this.exHandle = rtiamb.getAttributeHandle(classHandle, "ex");
5     this.eyHandle = rtiamb.getAttributeHandle(classHandle, "ey");
```

```
6     this.uxHandle = rtiamb.getAttributeHandle(classHandle, "ux");
7     this.uyHandle = rtiamb.getAttributeHandle(classHandle, "uy");
8     AttributeHandleSet attributes = rtiamb.getAttributeHandleSetFactory
      ().create();
9     attributes.add(exHandle);
10    attributes.add(eyHandle);
11    attributes.add(uxHandle);
12    attributes.add(uyHandle);
13    rtiamb.publishObjectClassAttributes(classHandle, attributes);
14    rtiamb.subscribeObjectClassAttributes(classHandle, attributes);
15    ...
16 }
```

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

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

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

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

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

```
for(AttributeHandle attributeHandle : theAttributes.keySet()) {
    // uxHandle and uyHandle hold the UAV Cartesian coordinates and are
     updated in the UAV Federate
    if(attributeHandle.equals(federate.uxHandle)){
        UAVx=decodeInt(theAttributes.get(attributeHandle));
    }

    if(attributeHandle.equals(federate.uyHandle)){
        UAVy=decodeInt(theAttributes.get(attributeHandle));
    }
}
```

The decoder for the above received data is reported below:

```
private int decodeInt(byte[] bytes) {
    HLAinteger32BE value = federate.encoderFactory.createHLAinteger32BE
    ();
    try {
        value.decode(bytes);
        }
    catch(DecoderException de) {
        log("DecoderException:"+de.getMessage());
        }
    return value.getValue();
}
```

A.5 The FOM module of the Excavator Federate

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

```
<objects>
    <objectClass>
        <name>HLAobjectRoot</name>
        <sharing>Neither</sharing>
    <objectClass>
        <name>Excavator</name>
        <sharing>PublishSubscribe</sharing>
        <semantics>NA</semantics>
            <attribute>
                <name>ex</name>
                <dataType>HLAinteger32BE</dataType>
                <updateType>Conditional</updateType>
                <updateCondition>NA</updateCondition>
                <ownership>NoTransfer</ownership>
                <sharing>PublishSubscribe</sharing>
                <dimensions>NA</dimensions>
```

```
17                <transportation>HLAreliable</transportation>
18                <order>TimeStamp</order>
19                <semantics>NA</semantics>
20            </attribute>
21        ...
22    </objectClass>
23 </objects>
```

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

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

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

A.7 Time synchronization

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

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

```
8           rtiamb.evokeMultipleCallbacks(0.1, 0.2);
9       }
10  }
```

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

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

References

1. Adak, M., Topçu, O., Oguztüzün, H.: Model-based code generation for hla federates. Software: Practice and Experience **40**(2), 149–175 (2010)
2. Arguello, L., Dwedari, L., Lauderdale, G., Vankov, A., Chliaev, P.: Esa-nasa distributed simulation experiment: First results and lessons learned, euro-swig. In: 2001 European Simulation Interoperability Workshop, Paper, 01E-SIW, p. 018 (2001)
3. Atkinson, C., Kühne, T.: Model-driven development: A metamodeling foundation. IEEE Software **20**(5), 36–41 (2003). DOI 10.1109/MS.2003.1231149. URL http://dx.doi.org/10.1109/MS.2003.1231149
4. Awais, M.U., Gawlik, W., De-Cillia, G., Palensky, P.: Hybrid simulation using sahisim framework: a hybrid distributed simulation framework using waveform relaxation method implemented over the HLA and the functional mock-up interface. In: Proceedings of the 8th International Conference on Simulation Tools and Techniques, Athens, Greece, August 24-26, 2015, pp. 273–278 (2015). DOI 10.4108/eai.24-8-2015.2260869. URL http://dx.doi.org/10.4108/eai.24-8-2015.2260869
5. Awais, M.U., Mueller, W., Elsheikh, A., Palensky, P., Widl, E.: Using the hla for distributed continuous simulations. In: Modelling and Simulation (EUROSIM), 2013 8th EUROSIM Congress on, pp. 544–549 (2013). DOI 10.1109/EUROSIM.2013.96
6. Awais, M.U., Palensky, P., Elsheikh, A., Widl, E., Stifter, M.: The high level architecture RTI as a master to the functional mock-up interface components. In: International Conference on Computing, Networking and Communications, ICNC 2013, San Diego, CA, USA, January 28-31, 2013, pp. 315–320 (2013). DOI 10.1109/ICCNC.2013.6504102. URL http://dx.doi.org/10.1109/ICCNC.2013.6504102
7. Banks, J.: Handbook of simulation: principles, methodology, advances, applications, and practice. John Wiley & Sons (1998)
8. Banks, J., II, J.S.C., Nelson, B.L., Nicol, D.M.: Discrete-Event System Simulation, 5th New Internatinal Edition. Pearson Education (2010)
9. Bar-Yam, Y.: Dynamics of complex systems. Westview Press (2003)
10. Basili, V.R., Perricone, B.T.: Software errors and complexity: An empirical investigation. Commun. ACM **27**(1), 42–52 (1984). DOI 10.1145/69605.2085. URL http://doi.acm.org/10.1145/69605.2085

11. Bastian, J., Clauß, C., Wolf, S., Schneider, P.: Master for co-simulation using fmi. In: Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany, 63, pp. 115–120. Linköping University Electronic Press (2011)
12. Birta, L.G., Arbez, G.: Modelling and Simulation - Exploring Dynamic System Behaviour, Second Edition. Simulation Foundations, Methods and Applications. Springer (2013). URL http://www.springer.com/computer/image+processing/book/978-1-4471-2782-6
13. Bocciarelli, P., D'Ambrogio, A., Falcone, A., Garro, A., Giglio, A.: A Model-Driven Approach to Enable the Distributed Simulation of Complex Systems, pp. 171–183. Springer International Publishing (2016). DOI 10.1007/978-3-319-26109-6_13. URL http://dx.doi.org/10.1007/978-3-319-26109-6_13
14. Bocciarelli, P., D'Ambrogio, A., Giglio, A., Gianni, D.: A saas-based automated framework to build and execute distributed simulations from sysml models. In: Winter Simulations Conference: Simulation Making Decisions in a Complex World, WSC 2013, Washington, DC, USA, December 8-11, 2013, pp. 1371–1382 (2013). DOI 10.1109/WSC.2013.6721523. URL http://dx.doi.org/10.1109/WSC.2013.6721523
15. Brann, J.J.: Enumeration and bit encoded values for use with protocols for distributed interactive simulation applications. Institute for Simulation and Training (IST) **3280** (2002)
16. Bryant, R.E.: Simulation of packet communication architecture computer systems (1977)
17. CeRTI Project: The simulation toolkit home page (2016). URL http://savannah.nongnu.org/projects/certi. Accessed 30 Nov 2016
18. Chandy, K.M., Misra, J.: Distributed simulation: A case study in design and verification of distributed programs. IEEE Trans. Software Eng. **5**(5), 440–452 (1979). DOI 10.1109/TSE.1979.230182. URL http://dx.doi.org/10.1109/TSE.1979.230182
19. CS Communication & Systémes: The Orbits Extrapolation Kit (Orekit) home page (2016). URL https://www.orekit.org/. Accessed 30 Nov 2016
20. D'Ambrogio, A., Iazeolla, G., Pieroni, A., Gianni, D.: A Model Transformation Approach for the Development of HLA-based Distributed Simulation Systems. In: SIMULTECH 2011 - Proceedings of 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Noordwijkerhout, The Netherlands, 29 - 31 July, 2011, pp. 155–160 (2011)
21. Department, T.S.: United States Naval Observatory. Leap Seconds home page. Retrieved 17 July 2011 (2016). URL http://tycho.usno.navy.mil/. Accessed 30 Nov 2016
22. Diallo, S.Y., Tolk, A., Graff, J., Barraco, A.: Using the levels of conceptual interoperability model and model-based data engineering to develop a modular interoperability framework. In: Winter Simulation Conference 2011, WSC'11, Phoenix, AZ, USA, December 11-14, 2011, pp. 2576–2586 (2011). DOI 10.1109/WSC.2011.6147965. URL https://doi.org/10.1109/WSC.2011.6147965
23. Eclipse Foundation: The Acceleo project home page (2016). URL http://eclipse.org/acceleo/. Accessed 30 Nov 2016
24. Eclipse Foundation: The Eclipse Project home page (2016). URL http://www.eclipse.org/. Accessed 30 Nov 2016

25. Eclipse Foundation: The Papyrus Modeling Environment project home page (2016). URL http://eclipse.org/papyrus/. Accessed 30 Nov 2016
26. Eclipse Foundation: The QVT Operational component home page (2016). URL http://projects.eclipse.org/projects/modeling.mmt.qvt-oml. Accessed 30 Nov 2016
27. (ESA), E.S.A.: The Mission CFI software homepage (2016). URL http://eop-cfi.esa.int/index.php/mission-cfi-software. Accessed 30 Nov 2016
28. Falcone, A., Garro, A.: 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), part of the 2016 Spring Simulation Multiconference, SpringSim '16, MSCIAAS '16, pp. 1:1–1:8. Society for Computer Simulation International, San Diego, CA, USA (2016). URL http://dl.acm.org/citation.cfm?id=2962664.2962665
29. Falcone, A., Garro, A.: Using the HLA standard in the context of an international simulation project: The experience of the "smashteam". In: 15th International Conference on Modeling and Applied Simulation, MAS 2016, Larnaca, Cyprus, September 26-28, 2016, pp. 121–129 (2016)
30. Falcone, A., Garro, A., Anagnostou, A., Chaudhry, N.R., Salah, O., Taylor, S.J.E.: Easing the development of HLA federates: The HLA development kit and its exploitation in the SEE project. In: 19th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2015, Chengdu, China, October 14-16, 2015, pp. 50–57 (2015). DOI 10.1109/DS-RT.2015.18. URL http://dx.doi.org/10.1109/DS-RT.2015.18
31. Falcone, A., Garro, A., Longo, F., Spadafora, F.: Simulation exploration experience: A communication system and a 3d real time visualization for a moon base simulated scenario. In: 18th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2014, Toulouse, France, October 1-3, 2014, pp. 113–120 (2014). DOI 10.1109/DS-RT.2014.22. URL http://dx.doi.org/10.1109/DS-RT.2014.22
32. Falcone, A., Garro, A., Taylor, S.J.E., Anagnostou, A., Chaudhry, N.R., Salah, O.: Experiences in simplifying distributed simulation: The HLA Development Kit framework. Journal of Simulation **10**(37), 1–20 (2016). DOI 10.1057/s41273-016-0039-4. URL http://dx.doi.org/10.1057/s41273-016-0039-4
33. FMI Toolbox for Matlab/Simulink: The FMI toolbox home page (2016). URL http://www.modelon.com/products/fmi-toolbox-for-matlab. Accessed 30 Nov 2016
34. Fortino, G., Garro, A., Russo, W.: From modeling to simulation of multi-agent systems: An integrated approach and a case study. In: Multiagent System Technologies, Second German Conference, MATES 2004, Erfurt, Germany, September 29-30, 2004, Proceedings, pp. 213–227 (2004). DOI 10.1007/978-3-540-30082-3_16. URL http://dx.doi.org/10.1007/978-3-540-30082-3_16
35. Fortino, G., Garro, A., Russo, W., Caico, R., Cossentino, M., Termine, F.: Simulation-driven development of multi-agent systems. In: Workshop on Multi-Agent Systems and Simulation, Palermo, Italia (2006)
36. Fritzson, P.A.: Principles of object-oriented modeling and simulation with Modelica 2.1. Wiley (2004)

37. Fujimoto, R.M.: Parallel and distributed simulation systems, vol. 300. Wiley New York (2000)
38. Fujimoto, R.M., Malik, A.W., Park, A.: Parallel and distributed simulation in the cloud. SCS M&S Magazine **3**, 1–10 (2010)
39. Functional Mock-up Interface: The Functional Mock-up Interface standard home page (2016). URL http://www.fmi-standard.org. Accessed 30 Nov 2016
40. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
41. Garro, A., Falcone, A.: 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, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015, pp. 9–16 (2015). URL http://dl.acm.org/citation.cfm?id=2872967
42. Garro, A., Falcone, A., Chaudhry, N.R., Salah, O., Anagnostou, A., Taylor, S.J.E.: A prototype HLA development kit: Results from the 2015 simulation exploration experience. In: Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation, London, United Kingdom, June 10 - 12, 2015, pp. 45–46 (2015). DOI 10.1145/2769458.2769489. URL http://doi.acm.org/10.1145/2769458.2769489
43. Garro, A., Tundis, A.: On the reliability analysis of systems and sos: The RAMSAS method and related extensions. IEEE Systems Journal **9**(1), 232–241 (2015). DOI 10.1109/JSYST.2014.2321617. URL http://dx.doi.org/10.1109/JSYST.2014.2321617
44. Gaylor, D., Page, R., Bradley, K.: The java astrodynamics toolkit (jat) home page (2016). URL http://jat.sourceforge.net/. Accessed 30 Nov 2016
45. Google Developers: The CodePro AnalytiX project home page (2016). URL https://developers.google.com/java-dev-tools/codepro/. Accessed 14 Oct 2015
46. Grogan, P.T., de Weck, O.L.: Infrastructure System Simulation Interoperability Using the High-Level Architecture. IEEE Systems Journal **PP**(99), 1–12 (2015). DOI 10.1109/JSYST.2015.2457433
47. Guinot, B.: Is the international atomic time tai a coordinate time or a proper time? Celestial mechanics **38**(2), 155–161 (1986)
48. Haouzi, H.E.: Models simulation and interoperability using MDA and HLA. CoRR **abs/cs/0606111** (2006). URL http://arxiv.org/abs/cs/0606111
49. Haskins, C., Forsberg, K., Krueger, M., Walden, D., Hamelin, D.: Systems engineering handbook. In: INCOSE Systems Engineering (2006)
50. Hodson, D.D., Hill, R.R.: The art and science of live, virtual, and constructive simulation for test and analysis. The Journal of Defense Modeling and Simulation **11**(2), 77–89 (2014)
51. Ido, H.: Space flight dynamics as a service (2012)
52. IEEE Std. 1278.3-1996: IEEE Recommended Practice for Distributed Interactive Simulation - Exercise Management and Feedback (1997). DOI 10.1109/IEEESTD.1997.82357
53. 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) (2010)

54. IEEE Std. 1516.3-2003: IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP) pp. 1–32 (2003). DOI 10.1109/IEEESTD.2003.94251
55. IEEE Std. 1730-2010: IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP) (2010)
56. ITEA2 Modelisar Project: The ITEA2 project home page (2014). URL http://www.fmi-standard.org/history. Accessed 15 May 2014
57. ITEA3 MODRIO project: The MODRIO project home page (2016). URL https://itea3.org/project/modrio.html. Accessed 30 Nov 2016
58. javaFMI library: The SIANI javaFMI project home page (2016). URL https://bitbucket.org/siani/javafmi. Accessed 30 Nov 2016
59. Kuhl, F., Weatherly, R., Dahmann, J.: Creating Computer Simulation Systems: An Introduction to the High Level Architecture. Prentice Hall PTR, Upper Saddle River, NJ, USA (1999)
60. Kuipers, J.: Quaternions and rotation sequences: A primer with applications to orbits, aerospace and virtual reality (p. 400) (2002)
61. Law, A.M., Kelton, W.D.: Simulation modeling and analysis, vol. 2. McGraw-Hill New York (1991)
62. Luzeaux, D., Ruault, J.R.: Systems of systems. Wiley Online Library (2010)
63. Maier, M.W.: The role of modeling and simulation in system of systems development. Modeling and simulation support for system of systems engineering applications pp. 11–44 (2015)
64. MÄK VR-Forces: The MÄK home page (2016). URL http://www.mak.com/. Accessed 30 Nov 2016
65. Mathworks software: The Mathworks/Matlab home page (2016). URL http://www.mathworks.com. Accessed 30 Nov 2016
66. Modeling and Simulation Coordination Office (M&S CO): The M&S CO home page (2016). URL http://www.msco.mil/. Accessed 30 Nov 2016
67. Möller, B.: The HLA tutorial v1.0. Pitch Technologies, Sweden (2013). Accessed 30 Nov 2016
68. Möller, B., Dubois, A., Leydour, P., Verhage, R.: Rpr fom 2.0: A federation object model for defense simulations. In: 2014 Fall Simulation Interoperability Workshop,(paper 14F-SIW-039), Orlando, FL (2014)
69. Möller, B., Garro, A., Falcone, A., Crues, E.Z., Dexter, D.E.: Promoting a-priori Interoperability of HLA-Based Simulations in the Space Domain: The SISO Space Reference FOM Initiative. In: 20th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2016, London, United Kingdom, September 21-23, 2016, pp. 100–107 (2016). DOI 10.1109/DS-RT.2016.15. URL http://dx.doi.org/10.1109/DS-RT.2016.15
70. Mutambara, A.G.: Decentralized estimation and control for multisensor systems. CRC press (1998)
71. NASA (The National Aeronautics and Space Administration) Software, Robotics, and Simulation Division (ER): The Simulation and Graphics Branch (ER7) home page (2016). URL http://er.jsc.nasa.gov/ER7/. Accessed 30 Nov 2016
72. North, M.J., Collier, N.T., Ozik, J., Tatara, E.R., Macal, C.M., Bragen, M., Sydelko, P.: Complex adaptive systems modeling with repast simphony. Complex adaptive systems modeling **1**(1), 1 (2013)
73. Object Management Group: MOF Model to Text Transformation Language, version 1.0 (2008)

74. Object Management Group: Systems Modeling Language, version 1.3 (2012)
75. Object Management Group: MDA Guide, version 2.0 (2014)
76. Object Management Group: Meta Object Facility (MOF), version 2.5 (2015)
77. Object Management Group: MOF 2.0 Query/View/Transformation (QVT), version 1.2 (2015)
78. OpenModellica project: The Open Source Modelica Consortium (OSMC) home page (2016). URL https://www.openmodellica.org/. Accessed 30 Nov 2016
79. Paredis, C.J., Johnson, T.: Using omg's sysml to support simulation. In: 2008 Winter Simulation Conference, pp. 2350–2352. IEEE (2008)
80. Peak, R.S., Burkhart, R.M., Friedenthal, S.A., Wilson, M.W., Bajaj, M., Kim, I.: 9.3. 2 simulation-based design using sysml part 1: A parametrics primer. In: INCOSE international symposium, vol. 17, pp. 1516–1535. Wiley Online Library (2007)
81. Phillips, R., Crues, E.: Time management issues and approaches for real time hla based simulations. In: Proceedings of the fall simulation interoperability workshop, Orlando, FL (2005)
82. Pitch Technologies: The simulation toolkit home page (2016). URL http://www.pitch.se. Accessed 30 Nov 2016
83. Pommier-Maurussane, V., Maisonobe, L.: Orekit: an open-source library for operational flight dynamics applications. In: International Conference on Astrodynamic Tools and Techniques (ICATT), ESA/ESAC, Madrid, Spain, pp. 3–6 (2010)
84. Pulecchi, T., Lovera, M.: A modelica library for space flight dynamics. In: In Proceedings of the 5th International Modelica Conference (2006)
85. Rabelo, L., Sala-Diakanda, S., Pastrana, J., Marin, M., Bhide, S., Joledo, O., Bardina, J.: Simulation modeling of space missions using the high level architecture. Modelling and Simulation in Engineering **2013**, 11 (2013)
86. Recommendation, I.: 460-6, standard-frequency and time-signal emissions (question itu-r 102/7). ITU-R Recommendations: Time Signals and Frequency Standards Emission, Geneva, International Telecommunications Union, Radiocommunication Bureau (2002)
87. San-Juan, J.F., Lara, M., López, R., López, L.M., Folcik, Z.J., Weeden, B., Cefola, P.J.: Using the dsst semi-analytical orbit propagator package via the nondywebtools/astrodywebtools open science environment. RdM **6**(1), 2π (2011)
88. Sauceda, F.: Space station reference coordinate systems. International Space Station, Rept. TR-SSP-30219 (2001)
89. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006). DOI 10.1109/MC.2006.58. URL http://dx.doi.org/10.1109/MC.2006.58
90. Simulation Exploration Experience (SEE) project: The Simulation Exploration Experience home page (2016). URL http://www.exploresim.com/. Accessed 30 Nov 2016
91. Simulation Interoperability Standards Organization: SISO Space Reference FOM (SRFOM) Product Development Group (PDG) (2016). URL https://www.sisostds.org/StandardsActivities/DevelopmentGroups/SRFOMPDGSpaceReferenceFederationObjectModel.aspx. Accessed 30 Nov 2016
92. Sokolowski, J.A., Banks, C.M.: Principles of modeling and simulation: a multidisciplinary approach. John Wiley & Sons (2011)

93. Taylor, S.J.E., Fujimoto, R., Page, E.H., Fishwick, P.A., Uhrmacher, A.M., Wainer, G.A.: Panel on grand challenges for modeling and simulation. In: Winter Simulation Conference, WSC '12, Berlin, Germany, December 9-12, 2012, pp. 232:1–232:15 (2012). DOI 10.1109/WSC.2012.6465310. URL http://dx.doi.org/10.1109/WSC.2012.6465310
94. Taylor, S.J.E., Revagar, N., Chambers, J., Yero, M., Anagnostou, A., Nouman, A., Chaudhry, N.R., Elfrey, P.R.: Simulation exploration experience: A distributed hybrid simulation of a lunar mining operation. In: 18th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2014, Toulouse, France, October 1-3, 2014, pp. 107–112 (2014). DOI 10.1109/DS-RT.2014.21. URL http://dx.doi.org/10.1109/DS-RT.2014.21
95. Taylor, S.J.E., Sudra, R., Janahan, T., Tan, G.S.H., Ladbrook, J.: GRIDS-SCF: an infrastructure for distributed supply chain simulation. Simulation **78**(5), 312–320 (2002). DOI 10.1177/0037549702078005553. URL http://dx.doi.org/10.1177/0037549702078005553
96. Taylor, S.J.E., Turner, S.J., Straßburger, S., Mustafee, N.: Bridging the gap: A standards-based approach to OR/MS distributed simulation. ACM Transactions on Modeling and Computer Simulation (TOMACS) **22**(4), 18 (2012). DOI 10.1145/2379810.2379811. URL http://doi.acm.org/10.1145/2379810.2379811
97. The Apache Commons Mathematics Library: The Apache Commons Mathematics home page (2016). URL https://commons.apache.org/proper/commons-math/. Accessed 30 Nov 2016
98. The Forwardsim HLA Toolbox for Matlab: The Forwardsim home page (2016). URL http://www.forwardsim.com/products/hla-toolbox/. Accessed 30 Nov 2016
99. The HLA Development Kit project: The HLA Development Kit project home page (2016). URL https://smash-lab.github.io/HLA-Development-Kit/. Accessed 30 Nov 2016
100. The International Celestial Reference Frames: The International Celestial Reference Frames home page (2016). URL http://www.iers.org. Accessed 30 Nov 2016
101. The Jodd Components: The JDateTime component home page (2016). URL http://www.jodd.org/doc/jdatetime.html. Accessed 30 Nov 2016
102. The PoRTIco project: The PoRTIco project home page (2016). URL http://www.porticoproject.org/. Accessed 30 Nov 2016
103. The SEE HLA Starter Kit project: The SEE HLA Starter Kit project home page (2016). URL https://smash-lab.github.io/SEE-HLA-Starter-Kit/. Accessed 30 Nov 2016
104. Topçu, O., Durak, U., Oguztüzün, H., Yilmaz, L.: Distributed Simulation - A Model Driven Engineering Approach. Simulation Foundations, Methods and Applications. Springer (2016). DOI 10.1007/978-3-319-03050-0. URL http://dx.doi.org/10.1007/978-3-319-03050-0
105. Topçu, O., Oğuztüzün, H.: Layered simulation architecture: A practical approach. Simulation Modelling Practice and Theory **32**, 1–14 (2013)
106. Tyszer, J.: Object-oriented computer simulation of discrete-event systems, vol. 10. Springer Science & Business Media (2012)
107. Van Spengen, J.W.: Fedef: A high level architecture federate development framework. Tech. rep., DTIC Document (2010)

108. Villimann, O.: Cto project, documentation, hla framework. Danish Maritime Institute (1999)
109. Vision, D.: A map to the future of distributed simulation. Prepared by the DIS Steering Committee, Institute of Simulation and Training, University of Central Florida, Orlando, Florida, US (1994)
110. Wainer, G.A., Al-Zoubi, K.: An introduction to distributed simulation. In: Theoretical Underpinnings and Practical Domains, pp. 373–402. Wiley-Blackwell (2010). DOI 10.1002/9780470590621.ch11. URL http://dx.doi.org/10.1002/9780470590621.ch11
111. Weatherly, R., Seidel, D., Weissman, J.: Aggregate level simulation protocol. In: Summer Computer Simulation Conference, Baltimore, Maryland, pp. 953–958 (1991)
112. Wilson, A.L., Weatherly, R.M.: The aggregate level simulation protocol: an evolving system. In: Proceedings of the 26th conference on Winter simulation, WSC 1994, Lake Buena Vista, FL, USA, December 11-14, 1994, pp. 781–787 (1994). DOI 10.1109/WSC.1994.717433. URL http://dx.doi.org/10.1109/WSC.1994.717433
113. Xie, Y., Teo, Y.M., Cai, W., Turner, S.: Towards grid-wide modeling and simulation (2005)
114. XQuery 1.0 and XPath 2.0 Functions and Operators: The XQuery 1.0 and XPath 2.0 standard home page (2016). URL http://www.w3.org/TR/xpath-functions/. Accessed 30 Nov 2016
115. Yilmaz, F., Durak, U., Taylan, K., Oğuztüzün, H.: Adapting Functional Mockup Units for HLA-compliant distributed simulation. In: Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden, 96, pp. 247–257. Linköping University Electronic Press (2014)
116. Yu, S., Zhou, S.: A survey on metric of software complexity. In: Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on, pp. 352 –356 (2010). DOI 10.1109/ICIME.2010.5477581