An Execution Environment for Distributed Simulations

MARTIN EKLÖF, RICHARD-MOE GUSTAVSEN*
ASMUND HJULSTAD*, OLE-MARTIN MEVASSVIK* (*FFI)

FOI
Defence Research Agency
Systems Technology
SE-164 90 Stockholm

Phone: +46 8 555 030 00
Fax: +46 8 555 031 00

www.foi.se

FOI-R--2114--SE
ISSN 1650-1942

Methodology report
November 2006

Systems Technology

Martin Eklöf, Richard-Moe Gustavsen*, Asmund Hjulstad*, Ole-Martin Mevassvik*

An Execution Environment for Distributed Simulations

FOI-R--2114--SE Methodology report Systems Technology
ISSN 1650-1942 November 2006

Issuing organization	Report number, ISRN	Report type
FOI – Swedish Defence Research Agency Systems Technology SE-164 90 Stockholm	FOI-R--2114--SE	Methodology report
	Research area code 2. Operational Research, Modelling and Simulation	
	Month year November 2006	**Project no.** E6023
	Sub area code 21 Modelling and Simulation	
	Sub area code 2	

Author/s (editor/s)	Project manager
Martin Eklöf Richard-Moe Gustavsen Asmund Hjulstad Ole-Martin Mevassvik	Farshad Moradi
	Approved by Nils Olsson
	Sponsoring agency FM
	Scientifically and technically responsible Nicklas Wallin

Report title

An Execution Environment for Distributed Simulations

Abstract

This report documents a study performed by the Swedish Defence Research Agency (FOI) and the Norwegian Defence Research Establishment (FFI). The work was as a collaborative project based on the Memorandum of Understanding (MoU) concerning co-operation in defence research between Danish Defence Research Establishment, Finnish Ministry Of Defence, FFI and FOI. The topic of the study is execution environments for distributed simulations. Distributed simulation technology is important in several military applications. Currently its use is dominated by military training. However, applications such as (simulation based) acquisition, decision support and concept development and experimentation are becoming more and more important. The scope and objectives of this study is to outline a common, interoperable framework to support the execution of large-scale heterogeneous distributed simulations. Further more, this study investigates how the advances in Grid technologies, Web Services and Semantic Web technologies, can contribute to improved availability and reliability of complex distributed simulations. Finally, a preliminary design of a core set of services that should form the basis for the envisioned framework is presented.

Keywords

Distributed Simulations, HLA, Services

Further bibliographic information	Language English
ISSN 1650-1942	**Pages** 70 p.
	Price acc. to pricelist

Utgivare FOI - Totalförsvarets forskningsinstitut Systemteknik 164 90 Stockholm	**Rapportnummer, ISRN** FOI-R--2114--SE	**Klassificering** Metodrapport
	Forskningsområde 2. Operationsanalys, modellering och simulering	
	Månad, år November 2006	**Projektnummer** E6023
	Delområde 21 Modellering och simulering	
	Delområde 2	

Författare/redaktör Martin Eklöf Richard-Moe Gustavsen Asmund Hjulstad Ole-Martin Mevassvik	**Projektledare** Farshad Moradi
	Godkänd av Nils Olsson
	Uppdragsgivare/kundbeteckning FM
	Tekniskt och/eller vetenskapligt ansvarig Nicklas Wallin

Rapportens titel

Exekveringsmiljö för distribuerade simuleringar

Sammanfattning

Denna rapport dokumenterar en studie som genomförts av FOI och FFI (Norwegian Defence Research Establishment). Det genomförda arbetet utgår från det MoU (Memorandum of Understanding) gällande försvarsforskning som undertecknats av Sverige, Norge, Finland och Danmark. Studiens tema är exekveringsmiljöer för distribuerade simuleringar. Distribuerad simulering är en viktig del av olika typer av försvarsrelaterade applikation. Idag används det i stor utsträckning i träningssammanhang (exempelvis sammankoppling av flygsimulatorer) men inom områden som simuleringsbaserad anskaffning (SBA) och beslutsstödssystem blir dess användning alltmer betydelsefull. Syftet med denna rapport är att beskriva ett ramverk för exekvering av distribuerade simuleringar som tagits fram inom studien. Vidare beskrivs hur Grid/Webb-tjänster och teknologi från den semantiska webben kan appliceras för att skapa en grund till ramverket, exempelvis bidra till förbättrad tillgänglighet till modeller och tillförlitlighet vid exekvering. Slutligen beskrivs en övergripande design, en mängd centrala tjänstetyper, av det tänkte ramverket.

Nyckelord

Distribuerad simulering, HLA, Tjänster

Övriga bibliografiska uppgifter	**Språk** Engelska
ISSN 1650-1942	**Antal sidor:** 70 s.
Distribution enligt missiv	**Pris: Enligt prislista**

Table of Contents

1. Introduction

This report documents a study performed jointly by the Swedish Defence Research Agency (FOI) and the Norwegian Defence Research Establishment (FFI) during a period of 9 months from March to November 2005. The work was as a Collaborative Project based on the Memorandum of Understanding concerning co-operation in defence research between Danish Defence Research Establishment, Finnish Ministry of Defence, FFI and FOI.

The goal of the project was to establish a preliminary design of a system for execution of distributed simulations, referred to as Execution Environment (EE) in this report. Distributed simulation technology is important in several military applications. Currently, its use is dominated by military training. Applications such as (simulation-based) acquisition, decision support and concept development and experimentation are becoming more important. However, the use of simulations in a distributed and heterogeneous environment may be complex and involves a lot of manual work. The purpose of the EE is to ease simulation set-up and execution management.

The major international standard for military distributed simulation, the High Level Architecture (HLA), defines services allowing efficient data exchange between loosely coupled simulation components while ensuring a consistent simulation. The EE supports tasks normally being performed by the user in order to set up and use an HLA-based simulation. This means that the functionality and services of the EE are orthogonal to those defined by the HLA.

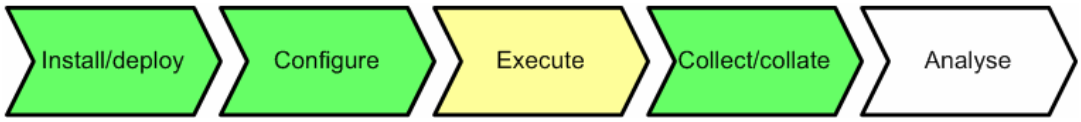


Figure 1. Phases in simulation utilisation. The Execution Environment manages green phases, while supervising the yellow Execute phase.

Figure 1 shows a generic workflow associated with the use of simulations in order to get an answer to a particular question or study a problem. We assume that the simulation system is planned for in advance, including which simulation components to use, and that scenario and configuration files have been developed.

- The install/deployment phase deals with the installation of simulation components and other necessary software on available computers. During this phase a detailed deployment design will be made, and software will be physically moved to and installed on computers according to this design. We assume that the network is set up and properly configured.
- During the configuration phase, configuration files and databases (e.g. terrain databases) are tailored in accordance with the scenario and the selected software and hardware components. The configuration files may e.g. specify how the models will behave in the scenario and the responsibility of each simulation component with respect to which entities to simulate.
- During the execution phase the user will start and stop the simulation system, and monitor and supervise it. Special support is needed with respect to error-handling.

- When the simulation execution has finished, data logs, associated with the scenario, must be collected from the network. This is done in the collect/collate phase.

Figure 2 gives an overview of the EE. The EE views a simulation system as a set of computer programs with associated configuration files and a set of data logs that are produced during simulation. This is in contrast to HLA that focuses on the role of individual simulation components of the simulation and the data and services to be exchanged within the simulation system. An execution environment should be independent of the simulation middleware being used, that is, several types of middleware (e.g. Distributed Interactive Simulation (DIS) and HLA) should be supported.

In order to utilise simulation resources in an effective manner a directory service provides information concerning simulation components, configuration files and available computers. The storage service provides long-term persistent storage for software and data. The computing services provide computing power and should be able to host a variety of components, including simulators, analysis tools and loggers.

1.1 Motivation

Modelling and Simulation (M&S), and distributed simulation in particular, have several military applications, such as training, acquisition, decision support and military experimentation.

The main use of simulations in the defence has traditionally been training applications. The current trends are towards the use of commercial game technology and the interconnection of existing training systems in order to create large simulated exercises across military services. The latter imposes geographically distributed simulation potentially involving a large number of different systems.

The concept of Simulation-Based Acquisition (SBA) relies on using simulation models and simulation throughout the entire acquisition process of military equipment, from concept definition to the phase where the system is taken into service. The goal of SBA is to improve manageability of complex acquisitions, lower procurement and development costs and mitigate risks.

M&S is a key method in Concept Development and Experimentation (CD&E). Experiments may take several forms, ranging from model-based experiments, solely relying on the use of computer based simulation, to experiments with human-in-the-loop simulations, and mixed environments including real platforms and systems. M&S-based experiments will in particular be an important tool in the development of a Network Based Defence (NBD).

Simulation has a potential of becoming an important decision support tool in an NBD environment with distributed applications and decision-makers connected through a heterogeneous network. Thus, it important to look at requirements that an NBD infrastructure will enforce on simulation support systems. For instance, in an NBD framework, decision makers will require transparent access to decision support tools to aid their analyses. This calls for a high availability of simulation models and a supporting infrastructure that will promote reliable simulation execution and general management tasks.

Emerging technologies such as Web or Grid services could form the foundation for an infrastructure that could leverage simulation capability for use in an NBD framework. These

services will help increase the availability of simulation models, ease the often complex task of setting up a distributed simulation, monitor a simulation execution, take precaution in case of failure of critical components and also provide control over the simulation execution on behalf of the simulation consumers.

However, to benefit from the Web and Grid service concepts, the integration of M&S in this new context needs to be explored further. Challenges include development of mechanisms for deployment of simulations in Grid/Web service-based environments to facilitate availability and reuse of simulation models, and also to simplify, or even automate, aspects such as initialization, monitoring and recovery of simulations etc.

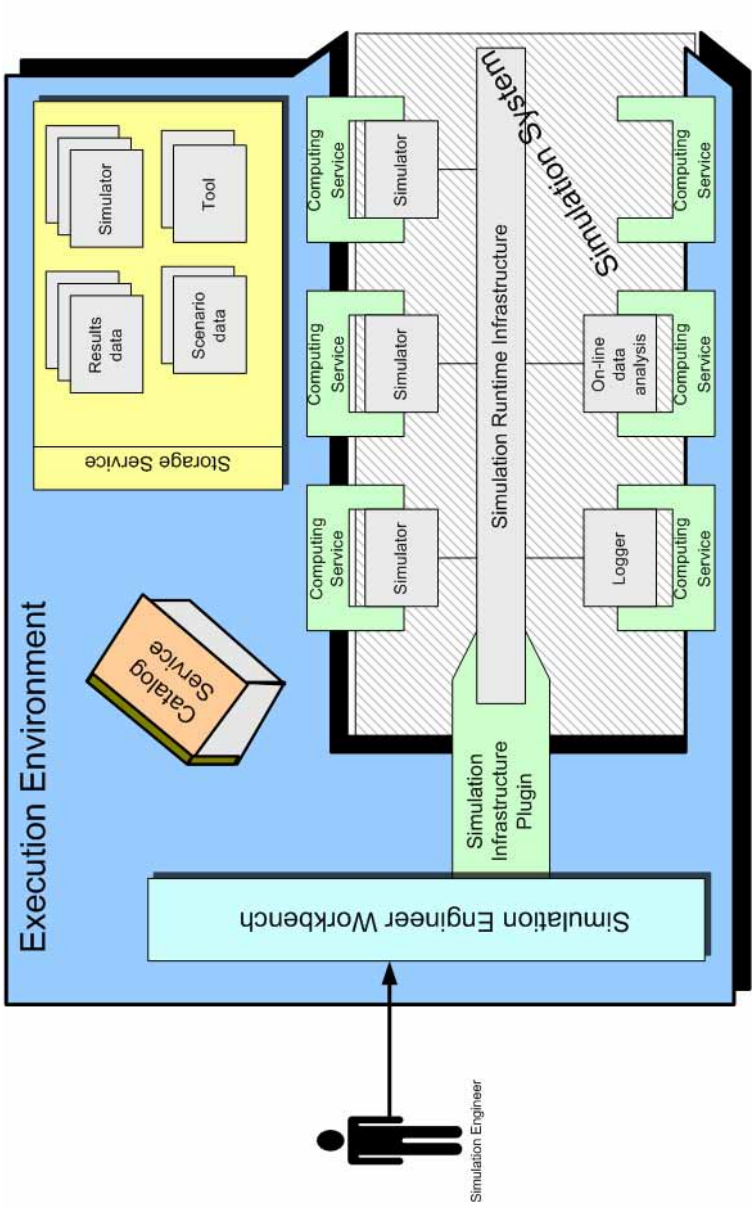


Figure 2. Overview of the Execution Environment.

1.2 Scope of work

The scope and objectives of this study are to:

- Outline a common, interoperable framework to support the execution of large-scale heterogeneous distributed simulations
- Investigate how advances in Grid and Web Services technologies and the Semantic Web can contribute to improved availability and reliability of complex distributed simulations, and facilitate the use of such simulations
- Outline a preliminary design of a core set of services that should form the basis for such framework

Equally important, the report identifies and analyses the data requirements of services in the execution environment, both the necessary input data describing available assets and computing resources as well as deployment and execution plans; data created in the context of the framework. The information is described both in general, technology independent terms, as well as in the perspective of Semantic Web technology.

The preliminary design covers the deployment and execution phase in distributed simulation and does not consider composition of simulations. Also, simulations considered for execution in the environment will be based on the High Level Architecture (HLA).

1.3 Content outline

The report is organized as follows:

Chapter 1 starts by presenting and briefly discussing previous work relating to tools, methods and approaches for managing distributed simulations, and continues with an introduction to the two technology sets applied in this work.

Requirements for the execution environment are elicited in chapter 3 from four use cases designed to describe a typical application of the EE. Non-functional requirements are also discussed here.

The information view of the EE is treated in chapter 4, where technology independent modelling techniques are used to describe the content and structure of the necessary input data in the form of asset descriptions and its like, in addition to data representing a specific deployment configuration.

Building on this foundation, chapter 5 describes the conceptual design of the EE. Chapter 6 discusses how Grid and Semantic Web technologies can be applied in the proposed system, with chapter 7 going into the details of the design.

Finally, conclusions are presented in chapter 8.

2. Background

Initially, this chapter briefly discusses previous work in this field. The main part of this chapter is the introduction to Service Oriented Architectures and Semantic Web technologies. The intention is to provide background and information for readers unfamiliar to the technology sets as well as references for further reading.

2.1 Previous work

Some approaches for managing computing resources related to distributed simulation have previously been reported. In (Lüthi and Großmann 2001) a Resource Sharing System (RSS) is presented that utilizes idle processing capacity in a network of workstations to execute HLA federations. In their approach, the owners of the workstations within a LAN can control the availability of their computers, through a client user interface, for execution of individual federates of a federation. Computers that are willing to share their resources are registered with the RSS manager that performs elementary load balancing. The RSS is built around a centralized manager that relies on an ftp-server for storage and transfer of federates. The article describes no extensive fault tolerance mechanisms included in the RSS implementation, but as this is an important feature of distributed simulations, and not well supported in the HLA, the RSS is planed to include functionality for checkpointing and management of replicated federates and fault detection.

In (Cai, Turner and Zhao 2002) an alternative approach to dynamic utilization of resources for execution of HLA federations is presented. The design is based on Grid technologies, more specifically services of the Globus Grid toolkit. Each federate is embedded in a job object that interacts with the Run-Time Infrastructure (RTI) and a Load Management System (LMS). The LMS performs two major tasks through use of a job management system and a resource management system. These systems carry out load balancing whenever necessary and discovery of available resources on the Grid.

(Bononi, D'Angelo and Donatiello 2003) proposes an adaptive approach, Generic Adaptive Interaction Architecture (GAIA), where model entities can be allocated dynamically to federates in an HLA-based simulation. The potential benefit is the reduction of messages being communicated among separate execution units. This is achieved by a heuristic migration policy that assigns model entities to executing federates as a trade-off between external communication and effective load-balancing. Load-balancing is required to avoid concentration of model entities over a small number of execution units, which would degrade simulation performance. The proposed mechanisms proved beneficial in simulating a prototype mobile wireless system by reducing the percentage of external communication and by enhancing the performance of a worst-case scenario.

In (Eklöf, Sparf and Moradi 2004) an execution environment for distributed simulations, based on peer-to-peer technology is presented. The environment is implemented using the JXTA technology and enables management of simulation components and execution nodes. Further, the environment supports run-time migration of federates, triggered on requests from workstation owners. In (Eklöf, Ayani and Moradi 2005) a refined version of the execution environment, based on web services, is presented. This version supports fault-detection and recovery in time-warp-based federations.

2.2 Service Oriented Architecture (SOA)

The concept of utilizing services and distributed architectures for implementing web applications originates from experiences of developing distributed systems over considerable number of years. A distributed system is defined as (Tanenbaum and van Steen 2002):

> "a collection of independent computers that
> appears to its users as a single coherent system"

This definition pinpoints two important aspects; the computers within the system are autonomous and user experience is equal to that of ordinary, locally installed software. The advantages of a distributed system are numerous, for example (Nagappan, Skoczylas and Sriganesh 2003):

- Parallel and distributed computing can provide more computing power to the individual user. This means that computational intensive tasks, requiring more capacity than available from the user's machine, can still be executed.
- Through distribution of a system, a higher level of robustness and availability is gained. One goal of a distributed system is to avoid a single point of failure and always maintain a certain quality of service.
- A software component, exposing general functionality, can be used by several applications and be reused in an efficient way.
- Distribution of a system may also be beneficial from an economic perspective since resources are shared among several users, instead of duplicated.

2.2.1 Web Services

Today, numerous technologies for implementing distributed systems exist, for instance Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI) and Microsoft Distributed Component Object Model (DCOM). The rapid development of Internet has created a demand for service oriented architectures supporting loosely coupled business-to-business and application-to-application communication, within and between disparate domains. Today, standardized web technologies are used to implement this communication through deployment of applications in the form of services, i.e. Web Services. These services communicate through standardized message formats in the form of XML messages (eXtensible Mark-up Language). Web Services unifies the communication mechanism used by disparate applications, which is fundamental for creating interoperability between applications implemented in different languages and/or for different hardware or software architectures.

The basic concept of Web Services is illustrated in figure 3. Three fundamental component types are identified, namely Service Requestor, Service Broker and Service Provider. A Service Provider is responsible for the development and deployment of a service, but also registration of the service at a Service Broker. The Service Broker is responsible for registration and localization of deployed services, which includes management of an inventory of services and their descriptions. Finally, the user of a service is called Service Requestor. The Service Requestor localizes available services at the Service Broker and then executes desirable services at a Service Provider (Nagappan et al 2003).

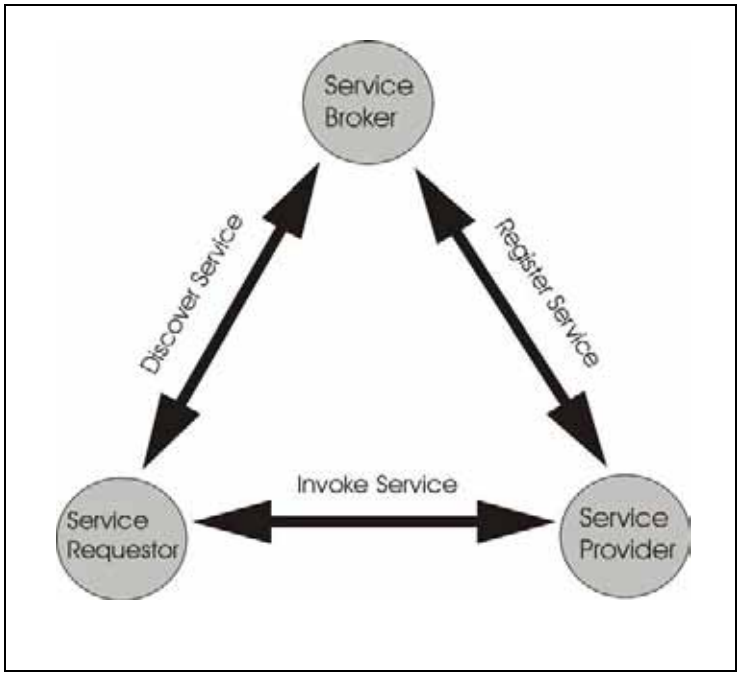


Figure 3. Main components of a Web Services-based architecture.

2.2.2 Grid Services

Decentralization and distribution of resources require new abstractions and concepts that enable accessing and sharing of services and information across large, physically distributed networks so that different capabilities can be delivered in standard ways without regard to physical location or implementation. However, security, resource management and other quality of service properties must still be provided. Grid technologies have been adopted in the scientific community as a way of achieving the above mentioned capabilities (Foster et al 2002).

A Grid is a collection of distributed resources available through a network, but appears to a user as a single system. Grid technologies support coordinated use of resources in distributed virtual organizations (VO), in which distributed components from different organizations with different security policies can be used in unison. Thus, Grids cross boundaries between organizations, hardware and software (Zhang, Chung and Zhou 2005).

The Global Grid Forum (GGF) is a forum for information exchange and specification of Grid technology standards. It is defining the Open Grid Services Architecture (OGSA), which is a distributed architecture ensuring the interoperability of heterogeneous systems. OGSA is based on Web Services, and composed of a set of services that facilitate distributed resource sharing and access. These services can be organized into four layers of abstraction. Services in the lowest layer are those that interact directly with diverse resources such as computers, networks, sensors etc. The next layer provides services that enable uniform and secure access to individual resources and services. The third layer provides services for managing resources and services collectively, for example service brokering and monitoring. The top layer contains user applications and tools (Foster and Kesselman 2003). Services on each layer may interact with all services in the same and lower layer, i.e. not only services in the same layer as themselves.

OGSA (Foster et al 2004) is based on the Open Grid Services Infrastructure (OGSI). OGSI is mainly composed of a set of standard interfaces. An OGSA service typically implements a subset of these interfaces. The interfaces are well-defined with standard mechanisms for discovery, dynamic service creation, lifetime management and notification etc. These interfaces are defined using a specific extension of the Web Service Description Language (WDSL) version 1.1, called GWSDL (Grid WDSL). The extension adds the ability to describe interface inheritance and additional information elements within a *portType*-element (the top interface element). This ability will be included in the next version of WSDL (Tuecke et al 2003).

OGSI (Tuecke et al 2003) specifies both required and optional interfaces. A Web service that implements the required interfaces is a *Grid service*. Grid services are characterized by the capabilities they offer, i.e. the interfaces they implement. Unlike Web services, Grid services can be instantiated and can maintain its internal state. The interfaces that define a Grid service are particularly concerned with the management of transient service instances. Transient service instances can be very lightweight, created to manage short-lived activities, for example a query against a database or a transfer of data. From now on, all occurrences of just "service" will refer to "Grid service".

The Globus Toolkit has emerged as a de facto standard for constructing Grid systems. The Globus Toolkit, version 4 (GT4), is an open-source implementation of OGSI and parts of OGSA. It can be seen as a set of building blocks for developing Grid applications. Implementations of OGSI interfaces and protocols constitute the core of GT4, and basic services for security, execution and data management etc are provided to better facilitate application development.

Although it contains a lot, GT4 is not yet a complete OGSA implementation. Still, the most recent version of the toolkit provides support for job management, file transfer, delegation of credentials and more.

2.3 The Semantic Web

The Semantic Web constitutes two aspects. Firstly, it is a vision of the next generation web, opening new possibilities in areas of search and retrieval facilities, customizing and adapting content to the individual user, and allowing for more automated processing of the information on the web. Secondly, the Semantic Web is also a set of technologies, intended to support or eventually realize a part of that vision.

The following sections will describe and briefly discuss these technologies using comparisons to other related, perhaps "competing" technologies, in order to better explain their possible role in an execution environment for distributed simulation.

Initially we will lend some time to the vision of the Semantic Web; a treatment of the Semantic Web should include a brief presentation of its background (Gagnes 2004).

The Semantic Web is the vision of T. Berners-Lee, the founder of the World Wide Web, a vision he described as:

> "[...] an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation". (Berners-Lee, Hendler and Lassila 2001)

The essence is in the term “well-defined meaning”. A part of the Semantic Web is syntax for referencing a particular, unique concept. This enables users of the web to decide whether two web pages about the city’s “loose birds” actually deal with ornithological concerns.

Any attempt to standardize and forever fix a single meaning for a specific word is futile, as should be learned from previous attempts. It is also highly unlikely that users of the Semantic Web will be able to agree upon a common viewpoint of the world. It may not even be desirable.

Therefore, the remaining goal is to establish a flexible way for users to collaborate and exchange information about concepts where an exact meaning can be agreed upon. As people’s views change over time, and even sometimes differ, the language and tools used must support this dynamic, and perhaps inconsistent, reality.

Even though it is impossible to permanently fix the meaning of a specific English word, it is possible to define a specific meaning and attach an identifier to that definition. If Alice defines the type CAR to be “any four-wheeled gasoline-engine driven transportation vehicle that can have human driver”, and Alice uses this consistently, it is unlikely that too many misunderstandings can occur when only Alice is talking. Someone else, say Bob, might be pickier, insisting that neither a truck nor a van is a car. In order to allow both of these definitions to coexist, a way is needed to distinguish them from one another. An established syntax for disambiguating these–possibly identically named terms is found in the XML world in the notion of namespaces.

Having a way of uniquely identifying concepts, how then describe relevant relations between and constraints on these concepts? How does one put this domain model on paper? A multitude of approaches exist to this problem, including languages such as the SQL Data Definition Language (SQL DDL), XML Schema, UML class diagrams etc. A common characteristic of all these is that they use a rather large set of constructs or primitives.

In Semantic Web technology, data models are built of a very specific form of statements called *triplets*. Each triplet consists of a *subject (s)*, an *object (o)* and *predicate (p)*. A triplet is a single, simple, statement, saying things like “The car (*s*) has colour (*p*) green (*o*)”, or “A lion (*s*) is (*p*) a vertebrate (*o*).” The first statement is about specific *things* in the world, the second about the *types* one uses in order to make sense of the world. Arbitrarily complex statements can be built using only triplets, typically by linking triplets to each other. The meaning of a triplet can only be understood when knowing the meaning of all three parts of a triplet. The second example above uses the very common concept of “is”. It appears reasonable to standardize on certain core concepts. This is what Resource Description Framework (RDF) does. In addition, it specifies a concrete syntax for representing triplets in machine and human readable form.

An ontology is a consistent set of defined concepts or types, and the relations between them. Ontologies are nothing new, their use and related tools have been researched long before the advent of XML. One notable example is DAML+OIL, a language for communication between agents. This language has a general syntax for describing constraints on types and relations. Based on the experiences from DAML+OIL, and using RDF as a starting point, the Web Ontology Language (OWL) has been developed. Previously defined ontology languages, such as DAML+OIL, had a strong link with logic and automated reasoning engines. This inheritance is carried along, and applied to ontologies described in OWL. Reasoning engines, query engines or database engines, are all tools for extracting information

from a database, with differing capabilities, often using different approaches or with a different focus.

OWL also benefits from character sets and encodings found in XML. XML standardization includes ways of identifying and declaring character encodings in documents and document fragments. Uniform Resource Identificators (URIs) are used throughout as syntax for identifying concepts, namespaces, relations etc. Standard parsers are available implementing this functionality, relieving the developer and/or systems integrator of an otherwise tedious and error-prone task.

The ubiquity of XML syntax, namespaces and URIs make tools easily available, providing a solid foundation for the developer on the syntax side.

2.3.1 Resource Description Framework (RDF)

RDF (Manola and Miller 2004) consists of several parts. First it standardizes a model and syntax for referencing concepts in a global namespace, in a machine-readable way. In RDF parlance, a concept is termed a "resource" (hence the name RDF). The syntax draws elements from the URI standard. An RDF identifier is a valid URI, typically looking something like this:

```
http://www.w3.org/1999/02/22-rdf#Description
```

Using XML namespaces, a document may then reference this element as

```
ns:Description
```

Nevertheless, documents with RDF/XML syntax can be very verbose, and not very suitable for manual editing.

The second part of RDF is the concept of *triplets*, mentioned earlier, used to describe resources. The predicates used when describing a resource are referred to as *properties*. Properties are identified in the same way as resources. RDF triplets consists of a two resources (a subject and an object) linked with a property. A set of such triples form a directed graph, with resources as labelled nodes and properties as labelled edges.

RDF, together with RDF-Schema (Brickley and Guha 2004), make it possible to describe subtype/super type-relations between resources and properties, creating two type lattices. (A lattice is a special kind of directed graph.) An example type lattice from the military modelling and simulation (M&S) domain is given in figure 4. Each box represents a unique type that can be identified by a RDF resource. The generalization relationships can be represented using standard RDF properties. The result is a directed graph with types and sub/super types. I addition, the standards provide methods for specifying additional textual descriptions as well as some list and set structures.

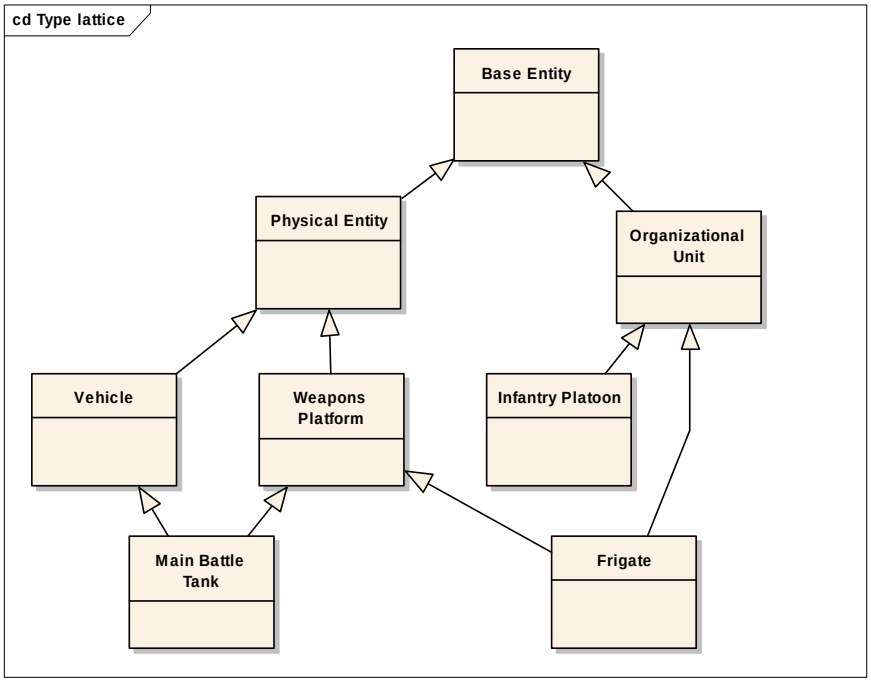


Figure 4. Example type lattice in the military M&S domain.

2.3.2 Web Ontology Language (OWL)

Formal languages for describing ontologies predate the web. OWL aims to realize the abstract syntax from these languages in one language, building on RDF.

When using OWL, one can be faced with a trade-off between using the full expressive power of OWL or being able to use an automatic reasoner. Supporting this trade-off, OWL comes in three dialects or variants: Lite, DL and Full, ordered in increasing expressive power. Of perhaps the most interest is the DL variant in this context. The name is due to its relation with *description logics*. The dialect contains all OWL language constructs, but places some restrictions on their use. As a result it is possible to guarantee some useful properties; all conclusions are guaranteed to be computable, and all computations will finish in finite time. Several tools target the OWL DL variant.

OWL also provides a standard top-level meta-ontology; definitions of various generalized relations between concepts, further enriching RDF and RDF-Schema. Examples of expressivity added through OWL is “an animal is *either* a fish, bird or mammal”, stating that the types are disjunctive, which is not possible using only RDF-Schema. There is also a constraint language embedded in OWL, inherited from previous logic-like languages.

An OWL document is a structured text document in OWL syntax and consists of three main parts. First, the header identifies the document type and all reference documents in a standard XML way. Namespaces are declared here, used to declare references to established ontologies used in the document.

Following the header is the schema of the data in the document, possibly by referencing an external source. This section describes the meaning of the XML tags used in the remaining document.

Lastly comes the instance data itself. The form of this data is not unlike any other data-containing XML document. What distinguishes it from other XML data is that the precise meaning of the various tags is explicitly declared previously in the document.

The format of an OWL document thus makes it possible to store schema and data in the same, or in separate documents.

2.3.3 Query, retrieval and reasoning

Previous sections described standard methods for specifying domain models and instance data, and encoding this in a concrete syntax. How to query and retrieve data from a database has not been dealt with so far.

If the only requirement is simple query and retrieval of previously inserted statements in a simple schema or domain model, Semantic Web technology provides few, if any, significant advantages over SQL databases. The advantages are first seen when more complex queries or questions are asked. For example, try formulating queries such as “find all first and second cousins of the kids in this family” against a genealogy database using a single query in the SQL query language. As the problem complexity increases–the kids may have different parents, be adopted, etc.–a SQL query will quickly become unmanageable.

Here is where reasoning engines enter. A query against a reasoning engine might be as simple as: “A parent is someone who is either the biological or adoptive parent. A grandparent is the parent of a parent. A grand-grandparent is the parent of a grandparent. A first cousin is someone with a common grandparent. A second cousin is someone with a common grand-grandparent“. Give me the list of all first and second cousins of X.”

That query consisted of six sentences, and should map to six (equally simple) statements sent to the database. It is only the last sentence of the query (“Give me the list of...”) that actually should return any data. Typically, the first five are already known to the database, being parts of the domain model. A database engine capable of responding to such a query is termed a *reasoning engine*.

Reasoning engines come in different varieties. One major distinction is between forward-chaining and backward-chaining reasoners. Forward-chaining reasoners use a rule set to add new statements to the knowledge base, possibly triggering new rules. This process continues until no further rules can be triggered. The result is a knowledge base containing all possible true statements, given the original knowledge base and rule set. Such an engine would, using the example above, continuously maintain a set of lists with everybody’s first and second cousins. This approach sacrifices storage space and insertion speed for simple implementation and fast retrieval.

Backwards-chaining systems operate differently. Starting with a rule set, a knowledge base and a query, it attempts different combinations of the rules to locate true statements matching the query. The solver in a Prolog system is an example of a backwards-chaining system, but this in only one example. More specialized reasoning engines are available for OWL data.

An SQL database can also in some ways be considered backwards-chaining, with the notable distinction that it does not (automatically) maintain intermediate results, it only combines existing tables according to specific instructions by the user (or programmer). An SQL database with triggers could be considered a hybrid of the two approaches, but still requiring significant manual work by the database administrator.

Semantic Web technology is thus more powerful than SQL in the case of complex schemas and reasoning.

2.3.4 Semantic Web querying vs. relational databases

The main benefits of using RDF, OWL and XML to describe meta-data are machine-readable, tool-independent encodings. Any OWL data could easily be stored in a relational database, an approach likely for any successful large-scale storage solution. The portable encoding is what adds something new and truly useful.

In addition, schemas for relational databases come in many varieties, with partly overlapping functionality. A common standard, such as OWL DL, provides a common grammar with well-known and well-specified expressive power. Furthermore, it is doubted that any relational database schema has the expressive power of OWL DL, and not the least that of the Full variant.

This does not imply that the use of relational database is in conflict with the choice of OWL. It seems likely that any OWL DL schema can be mapped into a relational schema while preserving scope of the model. The main difference is with regards to the maintainability and expressive power of an OWL DL schema compared to the flat relational database. Simple extensions to the OWL DL schema may require significant alterations to the relational database schema, with changes propagating to queries and updates in marginally related parts of the database.

It even seems likely that an OWL document database may use a relational database as storage back-end, making it possible to leverage existing and robust software.

2.4 Relevance to an execution environment

Semantic Web technologies appears to provide standards, languages and tools that make it possible to more easily create a flexible, extensible and powerful EE. Large scale distributed simulation systems may be highly complex, and Semantic Web technologies may be key to managing this complexity.

Later chapters in this report will explore this further, in particular by showing how OWL can be used for representing information about assets and deployment plans in the context of the EE.

3. Use cases and requirements

3.1 Use cases

The use cases described in the next sections represent the functionality we expect from the EE. From a high level perspective, the environment should provide services for deployment, execution and post-execution of a distributed simulation. In addition, execution services include a special use case to highlight issues concerning asset migration.

In this report, we assume that the user knows which federates to use in a planned federation. This is a precondition for the use cases, and consequently, a precondition for the EE. What the user may not know is what computers those federates should execute on, required dependencies that may exist between federates, which other assets they may depend on, and how to start, stop, pause or even migrate those federates. These operations are within the scope of the EE. The following use cases will elaborate and explain those services in more detail.

For convenience and ease of explanation, it is assumed that simulations in this report are based on HLA. Thus, simulations are often referred to as federations, and simulation components are often referred to as federates. Regardless, the EE is designed to be simulation technology independent. It is also assumed that simulations may include resources from multiple organizations, and that simulations may execute in a wide area network. However, information about the underlying network topology is assumed as known (or collected elsewhere), and not further discussed in the report. In effect this means that the EE will not help users to reason and select between available computing resources based on network topology information.

To describe the use cases, we will use the Uniform Modelling Language (UML). The use cases will only portray what functionalities to expect from the environment, and not how these are made available. The intention is to provide a basis for later eliciting necessary requirements for the proposed system.

We have identified four main use cases: Deployment, Execution, Post-execution and Migration. The first three are executed sequentially by default, while the latter is an extension to Execution. The four operations are transformed into use cases and described below (see figure 5).

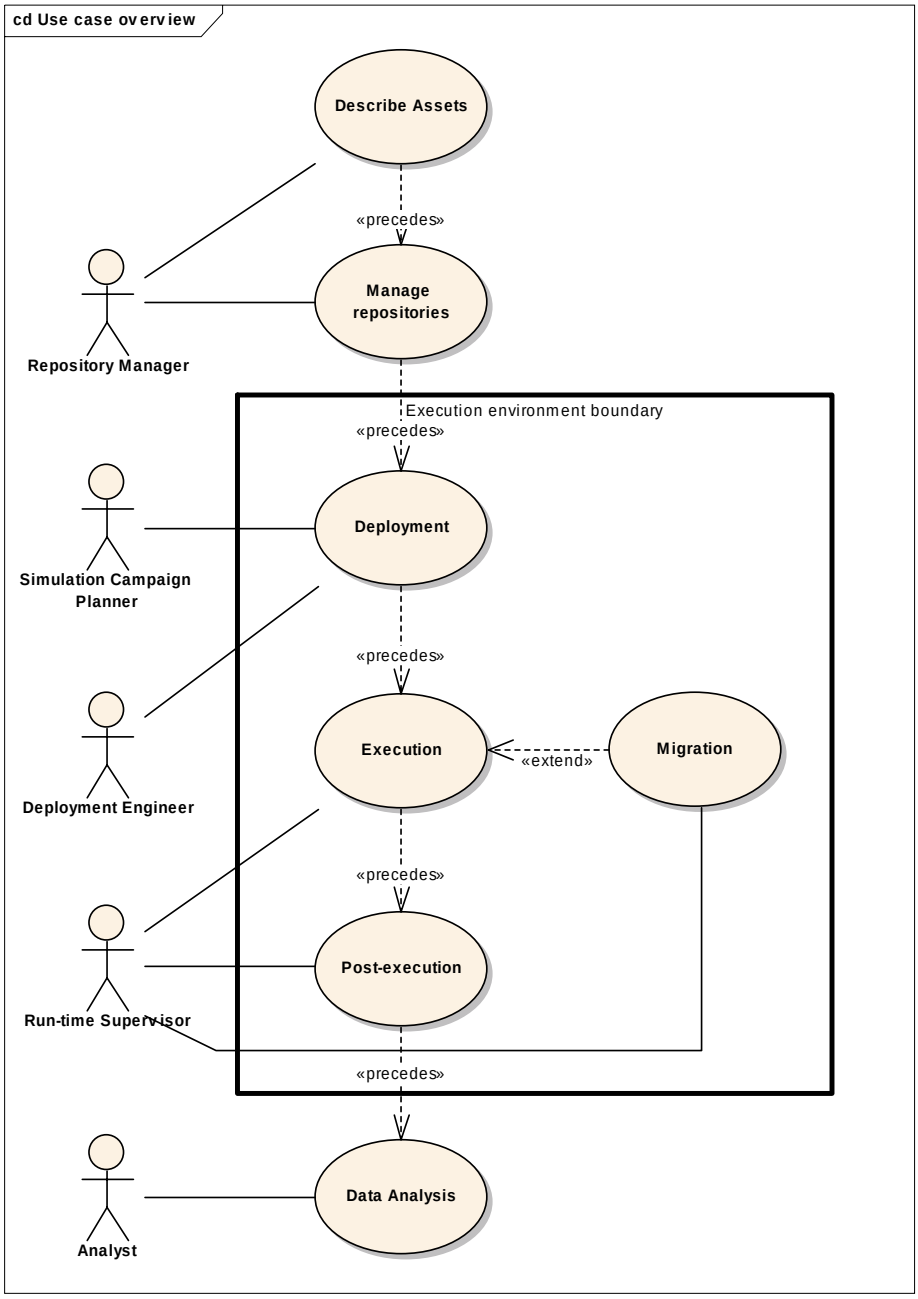


Figure 5. Use case operations supported by the EE: Deploy, Execution, Post-execution and Migration.

3.1.1 Use case 1 (Deployment)

A central part of the EE is to offer services for deploying a distributed simulation. With deploy we mean to identify available computers, federates, assets and dependencies. It also includes the process of actually copying and installing the identified federates and assets onto the selected computing resources. Such operations, like most of the services offered by the EE, may be initiated, monitored and controlled from a central location. The latter is important, as this is an underlying concept of the EE. Deploying a larger amount of software

to multiple locations can otherwise be a laborious process. It may involve several repeated, manual steps, and may include time consuming travelling if working with a geographically distributed simulation. Automating this process can significantly reduce time, allowing for shorter cycles during simulation development and test.

A deployment engineer is the principal actor in this use case. A deployment engineer is a person who is expected to possess good knowledge of the simulation system, as well as knowledge on how to configure and use computers and networks in general.

Before a deployment engineer can decide where to deploy and install assets, he must use the EE to fetch information about all assets and computers he may be in need of. This information must be formalized and made available so that that the EE, as well as the engineer, can reason about it. With reason we mean checking that the requirements posed by each asset will be satisfied. A federate might specify that it must be run on a Linux-based computer with at least 500MB memory. Knowing this, the EE may not only warn the engineer if this requirement gets violated, but also suggest available computers for use. Information about assets and computing resources is regarded as input to the EE.

A typical use of Deployment services begins with the engineer suggesting an initial configuration of assets and computing resources. He then lets the EE validate the set-up, which might reveal that some components are in need of additional configuration files, databases, or similar in order to work. He will continue by adjusting his initial design, revalidate it, and so on, until he is satisfied. When finished, he triggers the actual physical distribution, which will copy and install assets on the selected computers. This includes federates, configuration files, execution scripts, data, etc.

The deployment engineer is assumed to know the underlying network topology of the target simulation system. The EE will not provide any services regarding this issue, and will consequently not need this information.

All participating federates in the federation should be identified in advance. However, this is not a requirement. The EE will not perform any semantic compliancy checking between federates, and it will not interfere with their responsibilities regarding the scenario.

Preconditions

- Technical information about assets that may be used in the simulation is stored according to a known standard and made available for the users as well as the EE. The EE will use this information to help creating a valid deployment design. A data model for describing such information is presented later in the report.
- The underlying network of the target simulation system must be properly configured and up and running. This is a precondition because the EE may need to query the computing resources on the network for technical information. It will also need to know what assets they might contain already. The EE will also eventually use the network for installing selected assets on selected computing resources.

Postconditions

- The simulation system has been deployed, and is ready for execution. This means that all necessary software, including federates, has been copied, installed and properly configured on selected computers. If needed, the environment may contain several different selectable start-up and run-time configurations.

- Information about the deployed simulation system is documented in a known standard, and made available for subsequent EE services, as well as the users of the simulation system. A data model for describing such information is presented later in the report.

3.1.2 Use case 2 (Simulation Execution)

After deploying a simulation system (Use case 1), the next step will be to operate it. This is the job of a run-time supervisor. A run-time supervisor is a person who is expected to possess knowledge of computers and networks, enough to be able to spot faulty behaviour from monitoring services. Knowledge of the simulation system will be advantageous, but not required.

Operating the simulation system includes executing, managing and, eventually, ending the processes constituting it. Without an EE, the process may typically involve one or more operators at each location, making it necessary to divide the responsibilities between them. In addition, if the execution is to be repeated several times (which is normal during development), starting, managing and stopping the simulation system can be tiresome and unpractical. In this context, “start, manage and stop” refers to computer processes, not to be confused with starting and stopping a simulation with regard to simulation time (inside a federation execution). Such operations are part of simulation management, usually conducted from a designated federate.

The deployment engineer may have provided one or more start-up configurations for the supervisor to use. When a proper start-up configuration has been selected, and all computer processes of the simulation system are up and running, the environment will be ready for simulation execution. During this phase, the run-time supervisor will monitor assets with respect to CPU load, network load, status, etc. If any of these properties are found unsatisfactory, the supervisor can choose to migrate assets, moving them to other computers. Migration is handled as a use case of its own, and described below.

When the simulation execution has completed, the simulation engineer will trigger a “stop” command. The intended result is that each federate (and asset) will shut down in an appropriate way. Depending on the deployment configuration for each federate, they will continue executing without being joined to the federation, or stop executing. Log-files at each location should be saved and made available. If any assets fail to stop, the supervisor may choose to remotely end them by killing necessary processes explicitly.

A typical use of the services described by Simulation Execution begins with the run-time supervisor selecting a proper run-time configuration. This will lists the participating assets and their start-up order. He may choose to start execution of processes one asset at a time, or as a batch operation. When all assets are up and running, and monitoring software reports no difficulties, he may finally choose to start the simulation execution.

When the simulation ends, or the supervisor chooses to end it, he may issue a shutdown command. This will reverse the start-up process by stopping the assets, and preparing the environment for later executions. Log-file producing assets will store data on predetermined locations, available for subsequent EE services. The format of such logs will be native for each component, and the logs will not be analysed or used by the EE.

Precondition

- The simulation has been deployed. This will include the fulfilment of all pre- and postconditions of the use case Deploy.

Postcondition

- One simulation run has been completed. All federates have resigned from the federation execution, and the execution is destroyed. Produced data-logs are made available for subsequent EE services, as well as users of the simulation system. Shutdown of asset processes belongs to the next use case (Post-execution), so assets may still execute.

3.1.3 Use case 3 (Migration)

Failure of a critical federate in a simulation execution is often unacceptable, for example in a military decision support system. Thus, in a simulation system, it is essential to provide support for detection and recovery of failed assets in a way that will cause minimal interference to the simulation. Providing a robust environment is important when considering the trustworthiness of a system. The type of failure considered in this report comprises a “lost component”. This could be caused by a lost network connection, a faulty host-environment (hardware or software failure), loss of power, or that the federate itself stops executing for some reason. It should be noted that software errors, in terms of a simulation model producing erroneous result, are not considered. Automatic discovery of this type of failures is complex and will require additional measures.

The EE will provide services for monitoring a simulation system during simulation execution The intention is to take appropriate actions in case of failure. Upon such detections, the EE notifies the operator of these services, which is the run-time supervisor. Given the nature of the failed asset, automatic or manual repair is carried out to resume normal execution.

A special recovery operation supported by the EE is migration; the possibility of moving the execution of federates between computers. If a federate becomes unstable during simulation execution, the EE can try to relocate this asset. This typically means to install the concerned asset on some other computer, and let the new federate continue where the other one left off. Exactly how this can be achieved will probably differ greatly from one federate to another, and the EE must be designed to allow for different strategies. Note that migration of federates can also be triggered by the operator to gain better performance.

Preconditions

- The simulation has been deployed and is in the execution phase. Consequently, all preconditions of Execution have been fulfilled.
- Assets designed to be fault-tolerant must be able to respond to, and follow, migration requests. A design template for such services is discussed later in the report.

Postcondition

- The simulation resumes normal execution after restoration of a failed asset.

3.1.4 Use case 4 (Post-execution)

After a simulation run has completed, participating assets may have generated data logs and placed them on different locations. These will need to be collected, labelled and stored in a common repository. Because simulation systems may be geographically distributed, it must be possible to trigger, control and supervise this process from a central location. In some cases, simulation runs may also be executed several times in sequence (e.g. for Monte Carlo simulations). This repetitive process should be automated. A main objective of Post-execution is to provide services for gathering such log-files, and store them in a repository for later analysis. These services are utilized by a run-time supervisor.

If no further runs are planned, the supervisor may choose to shut down and remove all previously installed software from selected computing resources. By doings so, he resets the whole environment back to the initial state as it were before conducting Deploy. Otherwise, if further runs are planned, he may continue by using the services in use case Execution.

Precondition

- A simulation run has been completed. All preconditions of Execution have been fulfilled.

Postconditions

- Produced data logs and results from the assets are collected, labelled and stored in a common repository available for subsequent EE services, as well as users of the simulation system.
- If the supervisor is done with the environment: All processes belonging to the simulation system have finished, utilized resources have been freed and made available for later executions. If no more runs will be conducted, some, or all, of the used assets may be uninstalled and removed from the computing resources. The latter will remove the simulation system from the network, making it unavailable to the EE.

3.1.5 An example of deployment and execution

As deployment engineers and run-time supervisors perform their roles, the EE will move between different states (see figure 6). We see from the diagram that an EE first resides in an initialised state. In this state all necessary components for *controlling* the EE has been installed on all available computers, and all necessary information about assets has been placed in searchable repositories. The simulation system has not yet been deployed and the deployment engineer has not used any of the available services provided by the EE. The initialised state is considered as an outer boundary for the scope of the EE.

The deployment engineer in now ready to perform deployment. He will start by fetching technical information about all federates needed in the simulation. Which federates this constitutes is known before entering the Deploy phase.

Upon receiving technical information, a list of dependencies towards other federates and assets will be included. Each federate would e.g. require that an RTI is present in order to work. A logger federate might also request an SQL database of some sort. The next step for the engineer is therefore to search and select assets conforming to those requirements. Assets found from this process might spawn new dependencies on their own, which will further need to be satisfied. This incremental process might continue for several iterations. To speed

up the process, the EE will continuously try to satisfy all such dependencies on its own, and present design suggestion to the engineer along the way.

In addition to assets, the engineer must also search and select between available computers. Every asset needed by the simulation must be installed on a computer; this is part of the design conducted by the engineer. The process of finding computers is almost identical to the process of finding assets. In fact, when satisfying assets requirements, assets will often detail what kind of computer they must execute on (e.g. Intel or Sparc, Windows or Linux). Perhaps some of them also need to execute on pre-identified computers because of specialized hardware. The process of identifying assets and identifying computers will, as such, most likely be done in parallel.

When all assets and computers are identified, a deployment plan must be completed. This design specifies on what computers each asset should execute, and how the different requirements of each asset have been fulfilled. The latter might e.g. be that “the RTI needed by the logger on computer 192.168.1.100 will be found on computer 192.168.1.104”. This design will be used for the automatic transfer and installation of all assets, linking them together as needed. This latter operation will, from the engineers point of view, be as simple as pushing a deploy-button on his graphical user interface.

After deployment, the EE moves to the “Ready for start-up of asset processes” state (see figure 6). The simulation is now almost ready for execution. Several different run-time configurations may exist (depending on the scenario), and the supervisor must select which one to use for the current run. Creating a separate run-time configuration may include installation of additional assets and has to be prepared and installed during “Deployment set-up” and “Deploying”. However, the supervisor is offered a large degree of freedom when it comes to override configuration settings. If he e.g. wants to skip using a federate for a certain run, he can easily do that.

When a run-time configuration is selected, the engineer issues a start command and the EE moves to “All processes running”. This does not mean that the federation will start to move forward in simulation time. Simulation execution depends on the Simulation System and the configuration chosen for each federate. Simulation management is also included as a part of the EE, which means that the supervisor must explicitly push the “start simulation” button on his GUI if the simulation is to execute. Other federates might still retain execution.

While the EE is in “All processes running”, the supervisor can monitor the simulation system. If any assets or computers are malfunctioning, he can choose to “move” affected assets to other locations in the network (see use case ‘Migration’). When the supervisor decides that he is done with the current simulation run, he will issue a command for stopping the simulation system, which will bring the EE to “Stopping asset processes”. Note that stopping processes usually implies that the simulation execution also has been stopped, although this is not required.

The supervisor will now continue with post-execution operations (as described in use case Post-execution). This mainly includes copying data logs created from the simulation system and store them in predetermined repositories. This will be done as an automatic process provided by the EE.

Finally the supervisor must decide if he is done with the simulation system or if further runs are required. If he is done, the EE will move to “Removing deployed assets”, which in effect will uninstall the simulation system from the network. In that case, the EE will move back to

"EE Initialised". Otherwise, if further runs are planned, the EE will move to "Ready for start-up of asset processes".

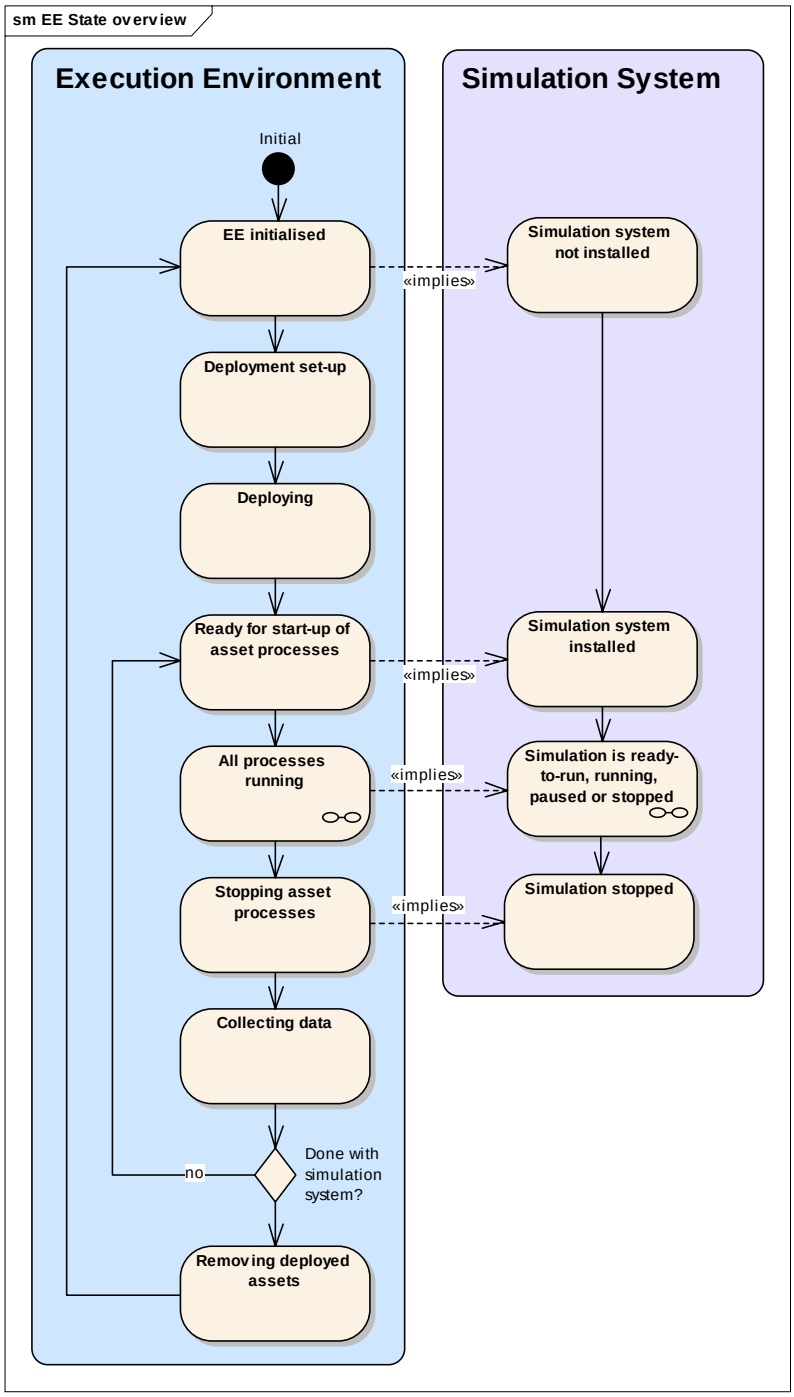


Figure 6. The main states of the EE are shown to the left. The corresponding states of the Simulation System are shown to the right. The EE is aware and dependent of the state of the Simulation System, but the Simulation System may not be aware of the EE.

3.2 Functional requirements

The next sections will describe different requirements that can be extracted from the use cases. The following requirements point out in more detail the necessary functionality of the EE, while maintaining a link to the use cases.

Requirements are numbered sequentially in the text using numbers in square brackets.

3.2.1 Deployment

In order for a deployment engineer to deploy a simulation system, the EE has to satisfy various requirements. Deploying a simulation can be a centralized process [1], or may be performed from multiple, arbitrary workstations [2]. Users may be located at one or more locations in reach of all tools and information needed for accomplishing the task.

The main activity for deploying a simulation is to be able to search for [3], select [4] and install [5] assets and computers that should participate in the simulation. For this to be possible, assets and computers must be documented in a uniform way. There will be a need for a data model combined with a formal language [6] that can include information about assets and computers. This includes network resources, RTIs, federates (models, supporting tools), FOMs, license servers, databases, and other hardware such as projectors [7]. The latter is important to include if planning a simulation for demonstration purposes.

Assets and computers are, before deployment, made available and described according to a given data model [8]. Information about computers can be acquired in the same way as asset descriptions are obtained [9]. Tool support for describing assets and computers is outside the scope for the EE.

Some assets and computers may be transferable, while others will remain stationary. Some will contain strict security policy regarding who, how and when they may be used. Location information [10] and security issues [11] must consequently be included for all assets and computers. The same is true for licensing information [12]. If interrelations or dependencies exist between the assets, they must be documented together with the assets [13]. The EE will provide services to help realize such dependencies, and provide suggestions for deployment designs [14].

Allocation of assets to computing resources is to be supported by tools [15]. These tools may provide some automatic allocation mechanisms [16]. It must be possible to override any automatic allocation [17]. If the engineer has performed manual allocation, he can still query if his allocation is valid [18]. This includes checking if chosen assets can run on selected resources (with respect to technical compatibility), and that all the dependencies of the assets are satisfied.

3.2.2 Execution

Starting a simulation system can be done from a central location [19]. The intended way for a run-time supervisor to start a simulation system is to select a predefined configuration [20]. The EE must be highly flexible regarding how it is operated. While this may be done automatically, manual use must be supported [21]. The supervisor may choose to execute a subset of the assets in a configuration [22]. Either way, the supervisor can choose to start the simulation system as a batch operation [23], or asset by asset [24]. It should be possible to

shut down the simulation system at any time from the same interface as was used to start it [25].

The run-time supervisor must be able to monitor the simulation system during execution [26]. This requires that technical information about computing resources in the environment is continually available [27]. Relevant information in this respect includes (as a minimum) hardware and network status [28].

3.2.3 Post-execution

After execution, deployed assets can be uninstalled [29]. Alternatively, they can be reset to their original, pre-simulation state [30]. The supervisor can choose to uninstall the simulation system in a batch operation [31], or asset by asset [32].

Some facility is needed for running an inventory list over asset-generated data [33]. Such data must be labelled and related to a specific simulation run [34]. Retrieval according to label should be supported [35]. It is necessary to be able to copy and store the data in a common repository [36]. In some situations, either caused by low available bandwidth or for confidentiality reasons, this may not be possible or desirable. The system should therefore support multiple data collection nodes [37].

3.2.4 Migration

The EE should provide services for conducting robust and fault-tolerant executions of composed simulations [38]. An essential service in this respect is mechanisms for fault-detection [39]. First, a failed component should be detected [40]. Second, a simulation component itself should detect if it has been detached from the environment, in order to trigger rejoin [41]. The EE should signal a detected failure to the run-time supervisor [42].

To enable robust executions of simulation systems, mechanisms for fault detection and recovery must be supported by the EE and the concerned assets [43]. This implies that developers of assets designated for execution within the environment, must be supplied with relevant guidelines and APIs to implement the chosen approach [44]. Thus, individual simulation components that will participate in a fault-tolerant execution must conform to certain requirements [45]. In case of failure of a simulation component, the environment must provide mechanisms for its recovery [46]. The EE should provide mechanisms for manual or automatic recovery of a simulation component in its current host environment [47], as well as recovery in a new host environment [48]. Regardless of manual or automated recovery of a simulation component, the EE should provide means of transferring a simulation component's state between host environments (migration) [51].[1]

The environment will not manage software errors in terms of federates producing erroneous results. It will target failures caused by lost network connections to remote sites, failed hardware of a host environment, failed OS of a host environment or malfunctions in software of an asset [52]. The supervisor should be able to act if an individual asset behave in an unexpected or undesirable way [53], for instance by triggering restoration of the concerned asset.

[1] Requirements numbered 49 and 50 have been edited out.

3.2.5 Non-functional requirements

The non-functional requirements of the EE describe services that should be present without reference to any specific process or task. This includes issues such as network and security. Security is one of the key questions regarding the EE, since the correct functioning and mutual trust of many involved systems will be essential to mission success. Security is usually viewed from five different aspects: authentication, authorization, confidentiality, integrity and availability.

Authentication is any process through which the identity of a participating actor is verified. For instance, each user must be able to verify that each received message originated from a known and trusted source. The EE must be able to provide such authentication for messages, events, updates etc. [54]. Authentication typically involves a username and a password, but other methods of demonstrating identity exist, such as smart cards, digital signatures and retina scans. The authentication protocol should not be sensitive to eaves-dropping [55].

Authorisation is any process of granting or denying participating actors permission. For instance, not all users may have permission to view classified information. The EE must thus provide authorisation for different users [56]. Authorisation consists of setting up permissions and access control. Each time a user tries to use a resource, or access information, a control of permission should be performed in order to prevent unauthorised access. Several methods to provide authorisation exist, such as Access Control Lists (ACL) and capability lists.

Ensuring confidentiality is any process through which unauthorised disclosure of information is prevented. The EE must ensure that information can be passed back and forth without being disclosed to an unauthorised third party [57]. The most common, and in a distributed environment most practical, method is to use encryption.

Ensuring integrity is any process through which unauthorised alteration or destruction of information is prevented. For instance, a receiver of a message must be able to ensure that the message received is exactly what was sent, no more and no less. The EE must therefore provide means for participants to verify that received information has not been altered in any way [58]. Methods to ensure integrity include message digests and encryption.

The availability of a component, such as a server, is often defined as the percentage of time it is able to accommodate requests for information or other functionality. Since the infrastructure may well be used during the execution of mission critical tasks, the EE must provide high availability [59]. Methods to provide high availability include replication and clustering. Availability will also include services regarding allocation of assets and computers used by a simulation system [60]. E.g. a computer might be physically available in the network, but another project might have reserved it (and as such, made it unavailable) for a specific period of time. The EE supports distributed simulations, so the ability to function in local area networks is a minimum requirement [61]. In order to be of inter-organizational use, it must also function in wide area networks [62]. Consequently, it must be able to operate across larger networks, without being hindered by firewalls, NAT-servers, etc. [63]. The EE must be able to execute in a heterogeneous environment since it will support distributed simulations. The most common operating systems should be supported, such as Windows XP/NT/2000, Linux and Solaris [64]. Finally, the EE must be independent of the infrastructures it is meant to support [65]. While we in this report discuss the EE with respect to HLA, this is primarily for explanatory reasons. If the EE is to survive over time, it must be able to accommodate for today’s simulation technologies, as well as succeeding technologies not yet invented.

4. Specification of assets

The purpose of this chapter is to identify what needs to be known about assets and deployment plans. This knowledge must be rich enough so that every possible asset and the various dependencies that may exist between them can be described. Such a dependency can be a federate requiring a particular FOM, or two federates in the same federation needing the same FOM. A part of this information is regarded as input to the EE, and will be called the input model. When new assets become available to the EE, the input model will point out what we need to know about them.

The input model describes the type of input we can expect when entering the first phase of the EE (Deploy). Subsequent phases will also make use of this knowledge, but they will in addition need the results produced from Deploy. This information states where assets have been deployed and is therefore vital for the execution phase. As such, we need to develop a data model that can capture what we need to know about a deployed simulation environment as well. This model will specify information that must be available, in addition to static asset descriptions, before entering the execution phase. We have chosen to call this the environment model.

4.1 Input model

The input model details what we need to know about assets before entering the Deploy phase. Consequently, it must be able to describe all kinds of assets that may participate in a simulation environment. Specifically, it must capture the different types of dependencies that may exist between them, possible deployment restrictions, run-time information, etc. To explain the various parts of the model, we will discuss an example of a simple federation.

In our example we will make use of two components interconnected through the use of HLA; a CGF and a data logger. The CGF consists of two main components: a graphical user interface and a simulation engine. In order to simulate a particular scenario, the CGF utilises an object parameter database describing the default properties of the entity types involved. Further, the CGF makes use of an order-of-battle file (or files) describing the forces participating in the simulation and their plans or orders, a terrain database and a scenario file tying all the different files together. The data logger stores data into an SQL database. In addition, an RTI implementation must be available to both the CGF and the logger. The RTI is configured through the use of a federation object model file and an RTI configuration file.

Looking at the example, there are evident asset candidates. Each federate should clearly be an asset. This is true because a federate is something that will be a part of our simulation execution, and must be deployed to a computational resource. In the example, we need to distinguish between the information describing a deployed asset, and the asset itself. Information about the latter will be static properties, like asset name, vendor, supported FOM, static dependencies to other assets, a link for downloading the executables, licence files, etc. Information about the former belongs to the environment model, and is discussed later.

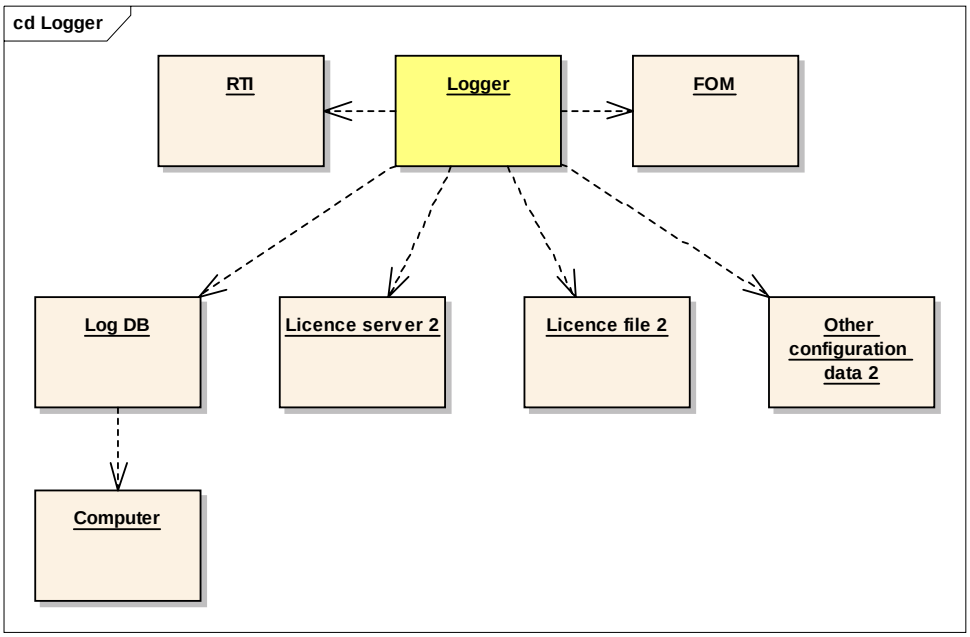
In the example, all the various databases are considered as assets, and the same is true for the RTI. Not all assets are as evident as these. For example, configuration files used by the

different components must be deployed in the same way as federates. Because such data may be shared among several different components without belonging specifically to any of them, we consider them as independent assets. However, some configuration files will always be tightly coupled to a specific component, and in such cases, the EE should see them both as single assets.

If we were to describe the input so far, we would need technical information about which federates to use, available computers, configuration files, licence files, databases, and RTIs, and we would need to specify how they depend on each other. An example of such a description is depicted in figure 7. Here the simulation components are coloured yellow. The arrows pointing away from them indicate their dependencies towards other assets. This should be interpreted as if an engineer where to use e.g. the Logger component in a simulation, he would also need to deploy a specific RTI, a specific FOM, a specific licence server, etc.

An information model embracing the example input must obviously reference many different assets. And, as shown in the figure, it must describe the various types of dependencies between them. One kind of dependency could be “must use”, while another could be “must be deployed on”. In addition we need dependencies that do not include a specific target asset. A federate might e.g. be described as having a dependency “must use” to an RTI of a certain version. However, exactly which RTI (vendor) to use might not be important. In such cases, it might be left to the deployment engineer to select one he finds suitable upon deployment.

When deploying a set of assets, there should be enough information about them to ensure that all necessary dependencies are satisfied. If we fail to see them as a whole, one might end up selecting different RTIs for different federates, although they are meant to participate in the same federation. As this is not a restriction belonging to a single federate, but rather to the set of federates in the same federation, the input model needs to include a set of rules specifying interoperability issues with respect to deployment and execution. A diagram depicting such dependencies is shown in figure 8.



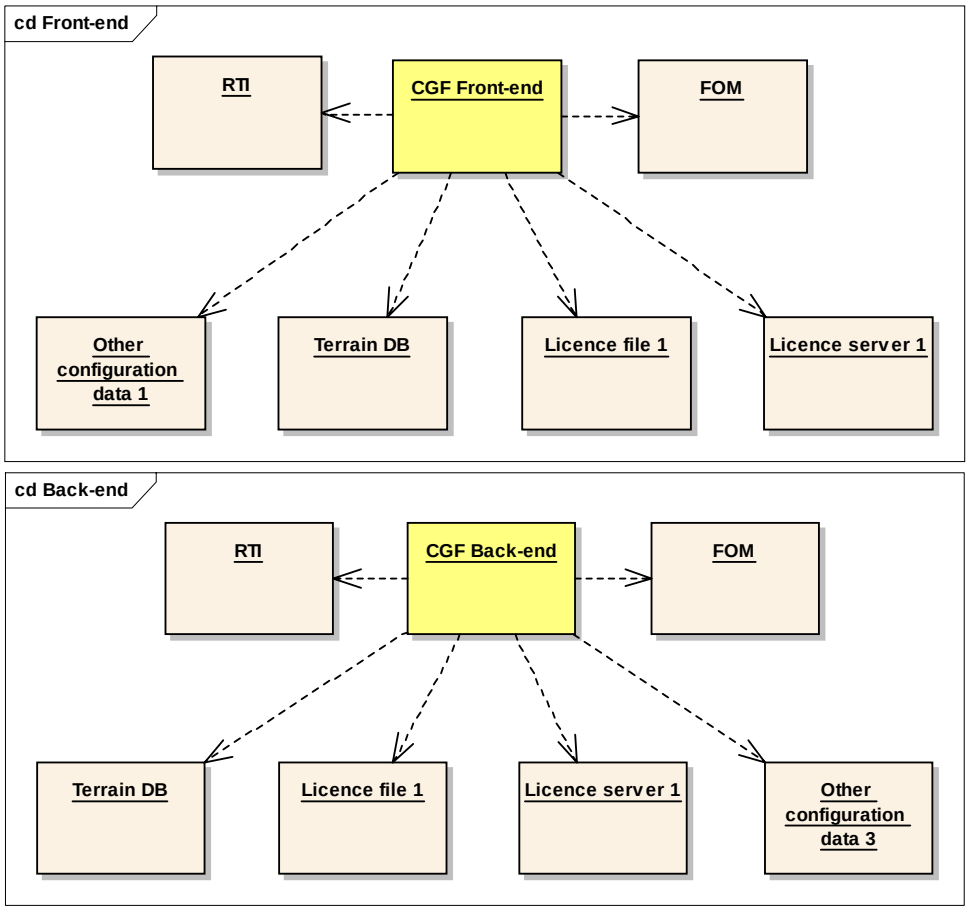


Figure 7. The yellow objects are the different simulation components identified as assets in the example. The figure exemplifies how these depend on other assets if they should operate in a simulation environment.

Looking closer at the identified assets, we realize that some of them share common properties and restrictions. Some assets might e.g. not be able to move between computers. This is important information during deployment. A licence server, or perhaps a simulator running on proprietary hardware, might contain such a restriction. Some of the assets might not be executable, like a database realized as a binary file. Assets can also have combinations of the described properties, being e.g. both executable and moveable. In our example the CGF would be both executable and moveable, and the corresponding terrain database would be movable, but not executable. Whether an asset is movable and/or executable is a property of the asset, and must be referenced in the input model.

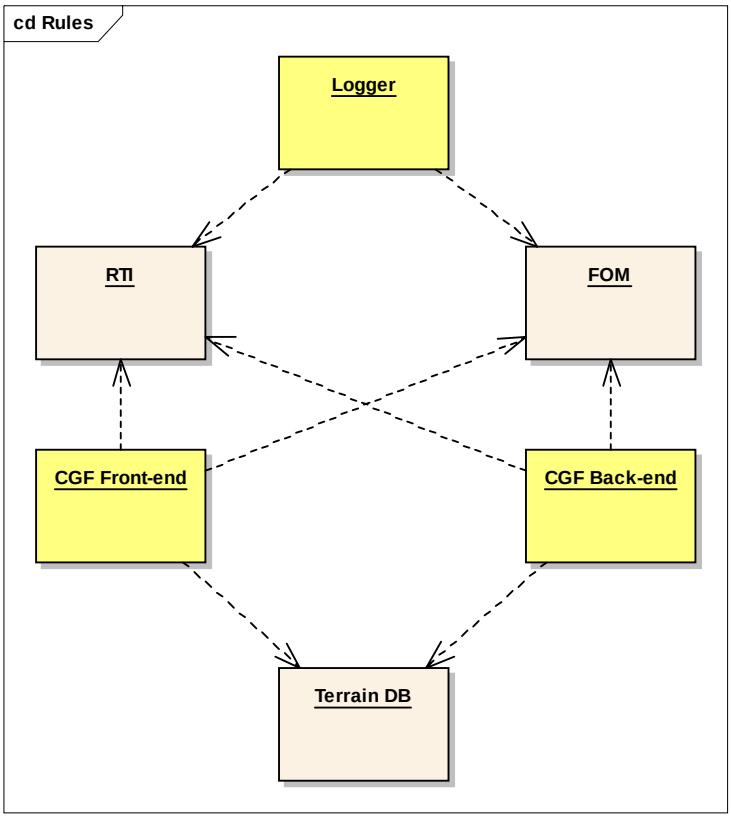


Figure 8. Independent assets may be constrained by additional rules when deployed. The figure specifies (among others) that each federate must use the same RTI and FOM in the same federation.

4.1.1 Description of the input model

Figure 9 gives an overview of the input model. It avoids the notation of existing asset types, such as a "Federate" or "RTI", because using them would imply the use of HLA. Instead, it concentrates on common properties found for every asset (with respect to the EE), and only the collecting term "asset" is used. By modelling the information this way, a generic infrastructure without a bias towards specific simulation technologies can later be achieved. The final result will hopefully be an infrastructure able to utilise future simulation technologies, as well as those in use today.

Because the model avoids referring to a specific technology, it might look a bit unexpected at first glance. Essential in the model is the notation of an `Abstract Asset`. An `Abstract Asset` is here considered as an asset being abstract, which means that it will never represent something that is concrete. E.g. the general notion of an RTI can be considered as abstract, while the DMSO RTI1.3NG-V6 is concrete. Concrete assets are realized as `Product Versions`, which is a subtype of `Abstract Asset`. By separating the notion of abstract assets from concrete assets this way, we are able to specify rules and constraints concerning all assets of a general type, as well as specific assets from specific vendors.

A `Product Version` is usually a version of a software product, and hence, has an association to a `Software Product`. A `Software Product` describes common information valid for all versions of the same product. Moreover, only concrete assets may be

deployed somewhere, so only `Product Version` has `Deployment Info` associated to it. `Deployment Info` contains information about where the product can be found (or downloaded), and whether or not it can be moved to other locations (deployed) and later executed.

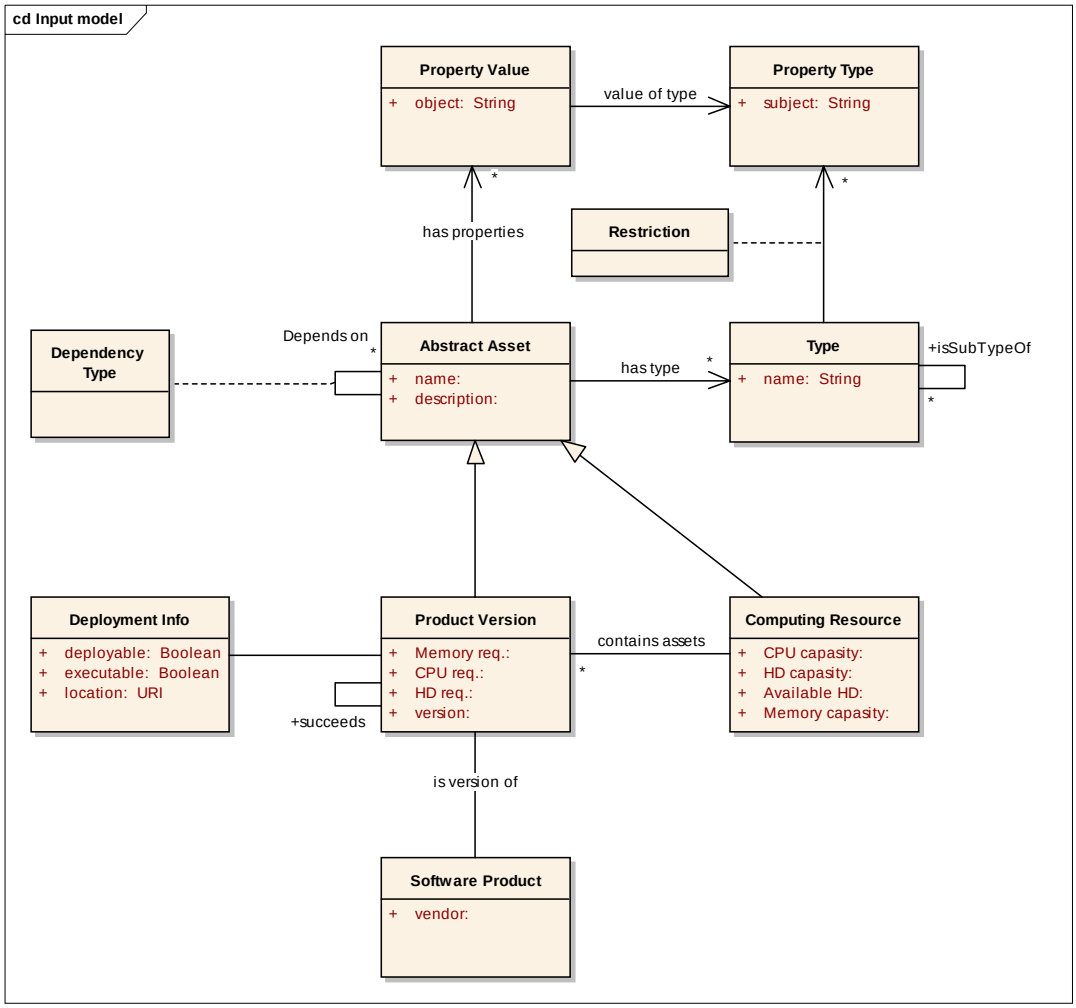


Figure 9. Input model. It shows the information expected as input to the EE. UML 2.0 Class diagram notation is used (OMG 2005).

From the figure we see that an `Abstract Asset` may depend on several other assets using the `Depends on` association. Because `Product Version` inherits from `Abstract Asset`, this means that an asset can depend on both abstracts assets as well as concrete assets. Because a `Computing Resource` also inherits from `Abstract Asset`, an asset can also depend on a computing resource. This might be handy for e.g. specifying that a concrete visualisation federate must be installed and executed on a concrete computer with a specific graphics card installed. Also note that it might be reasonable to create abstract assets representing both operating systems and hardware, and then later use “must use” dependencies between concrete assets and such abstract assets when hardware and software restrictions apply.

An asset, abstract or not, will always have one or more `Types` associated. These types tell what kind of asset we are describing. If we where to describe a new federate, we would create a type with name “federate” (if we had not described such a type already), and associate it with our new federate asset. When we later receive asset descriptions as input to our EE, we can easily determine their types by reading this association.

A more complex part of the `Abstract Asset/Type` relation is the property system. What the model captures is the fact that an asset of a certain type can contain a set of properties. If we e.g. created a type “Federate”, we might also like to define that a federate should include a property named “HLA version”. By adding such a property to this type, we state that whenever a new asset is of type “Federate”, the asset need to specify what kind of HLA version it supports. The restriction class of the association is used to specify whether or not a value for the property is imperative.

4.1.2 Describing the example by using the model

Let us now revisit the example described earlier, and focus on the logger component. If we pretend that we need to describe this asset using the proposed input model. Firstly, our logger is a concrete asset bought from a specific vendor, and is therefore clearly a `Product Version` (as opposed to an `Abstract Asset`). This means that it is a part of a Software Product line, so general information about the product can be created and inserted into a `Software Product`. We know the repository where the component will be made available, and we know that the logger is both deployable and executable. This information is stored into `Deployment Info`.

The next step is to decide the type of asset we consider the logger to be. First of all, it is a logger, so we create a `Type` with name “Logger”, and associate it with our asset. We also know that our logger is a federate, so we create a `Type` “Federate” as well, and make a similar association. Because we decide that we later might want to query the HLA version of every federate that may exist, we add a property “HLA version” to the “Federate”-type, and make it imperative. Because of this, we also need to add a new `Property Value` to our federate asset with the HLA version of our logger. See figure 10 for an overview.

From figure 10 we see that a logger depends on several other assets in order to work. An RTI and a FOM is e.g. needed. Let us say that the FOM is already described as a `Product Version` of its own. To specify the required dependency, all we need to do is to create an instance of the `Dependency Type` association from the logger to the FOM. The same could be done for the RTI. But let us say that we do not want to tie the logger to a concrete RTI, only to an abstract notion of an RTI. In this way we state that the logger will need an RTI (perhaps of a certain version) to work, but not exactly which one. This decision is instead left out to the deployment engineer when performing deployment. To create an abstract notion of an RTI, the only thing we need to do is to create a new `Abstract Asset` with type “RTI”, if it did not exist already, and link our logger asset to this one instead.

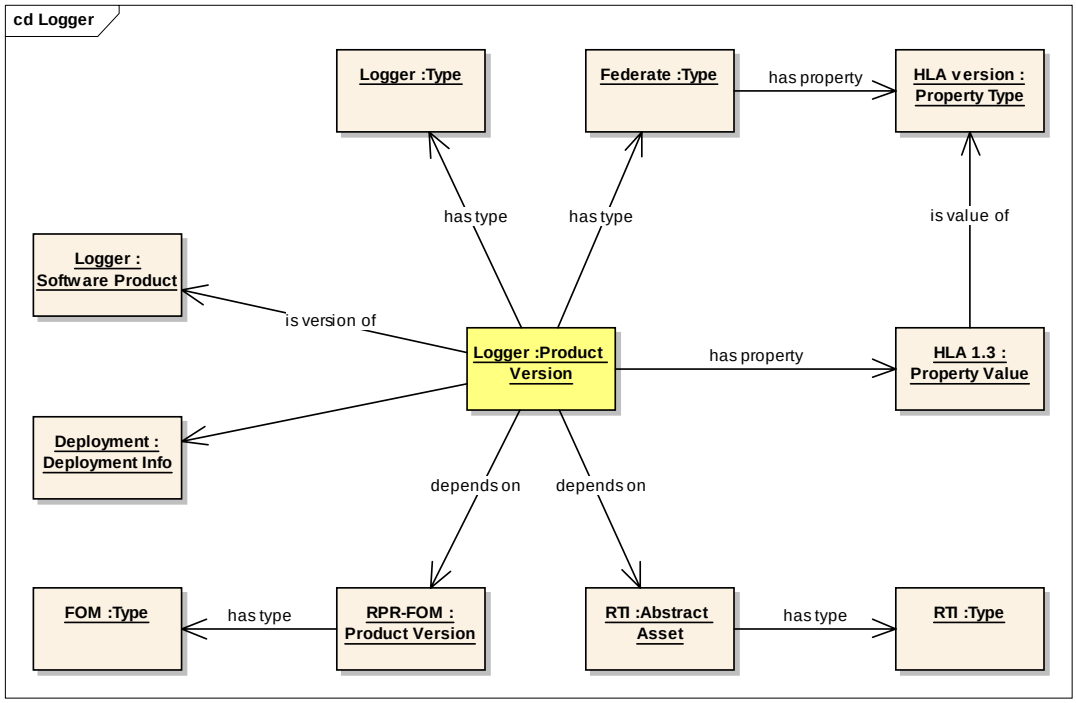


Figure 10. A part of the example federation described with the input model.

4.2. Environment model

So far, this chapter has presented a data model specifying the type of information needed as input to the EE. As already stated, this is only half of what we need. We also require an environment model detailing the kind of information produced by the deployment phase. This information will be used as input to the execution phase, being available to the tools and technologies there. While the input model is the basis for describing assets before entering Deploy, environment information will be produced from the processes described by the identified use cases.

Before reading further, it is important to understand that the environment model only explains the information needed to describe an EE. It is meant as an overview of the elements and dependencies we need to include before we later discuss suitable technologies and implementation. Therefore, the model is not saying anything about how information about an EE will be stored.

To explain the various parts of an EE, we will start with the root element named `Execution Environment` (see figure 11). We have realized that an EE contains two parts. First, we need to explain in detail where different assets will be deployed. Second, we need information about how the same assets may be executed, which configuration files they will use, how to pause or stop them, where their data logs are stored, etc. The first aspect belongs to Deployment, whereas the latter belongs to Execution. To reflect these aspects, an `Execution Environment` consists of a `Deployment Configuration`, and one or more `Runtime Configurations`. The reason for an EE to have more than one run-time configuration is that we may want to choose different versions of the same simulation for each run, perhaps only varying parts of the scenario.

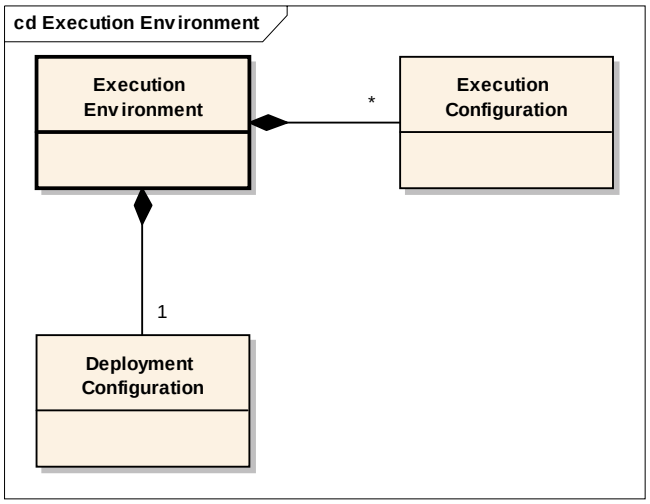


Figure 11. The diagram shows the main information elements for describing a simulation environment.

4.2.1 Deployment configuration

In figure 11 we see that an `Execution Environment` contains a `Deployment Configuration`. A `Deployment Configuration` is a root object for describing where assets will be deployed in this environment. A detailed view uncovering this structure is shown in figure 12. Here we see that a `Deployment Configuration` will contain a list of `Environment Items`. An `Environment Item` will usually be an `Environment Asset`. So what this model roughly says is that a simulation environment will contain a list of environment assets. An important detail in the model is that it separates the notation of an asset from an environment asset. The former describes static information about a specific asset (like a 3D stealth viewer), and is realized as an `Asset` object. The latter is a reference to the same asset, but with a given role attached (as described in the scenario). This means that an asset, like a CGF federate, can be used at several places in the same federation. But each instance might use different configuration files, databases, licence files etc., and be placed on different computers. An `Environment Asset` contains a reference to the `Asset` it instantiates.

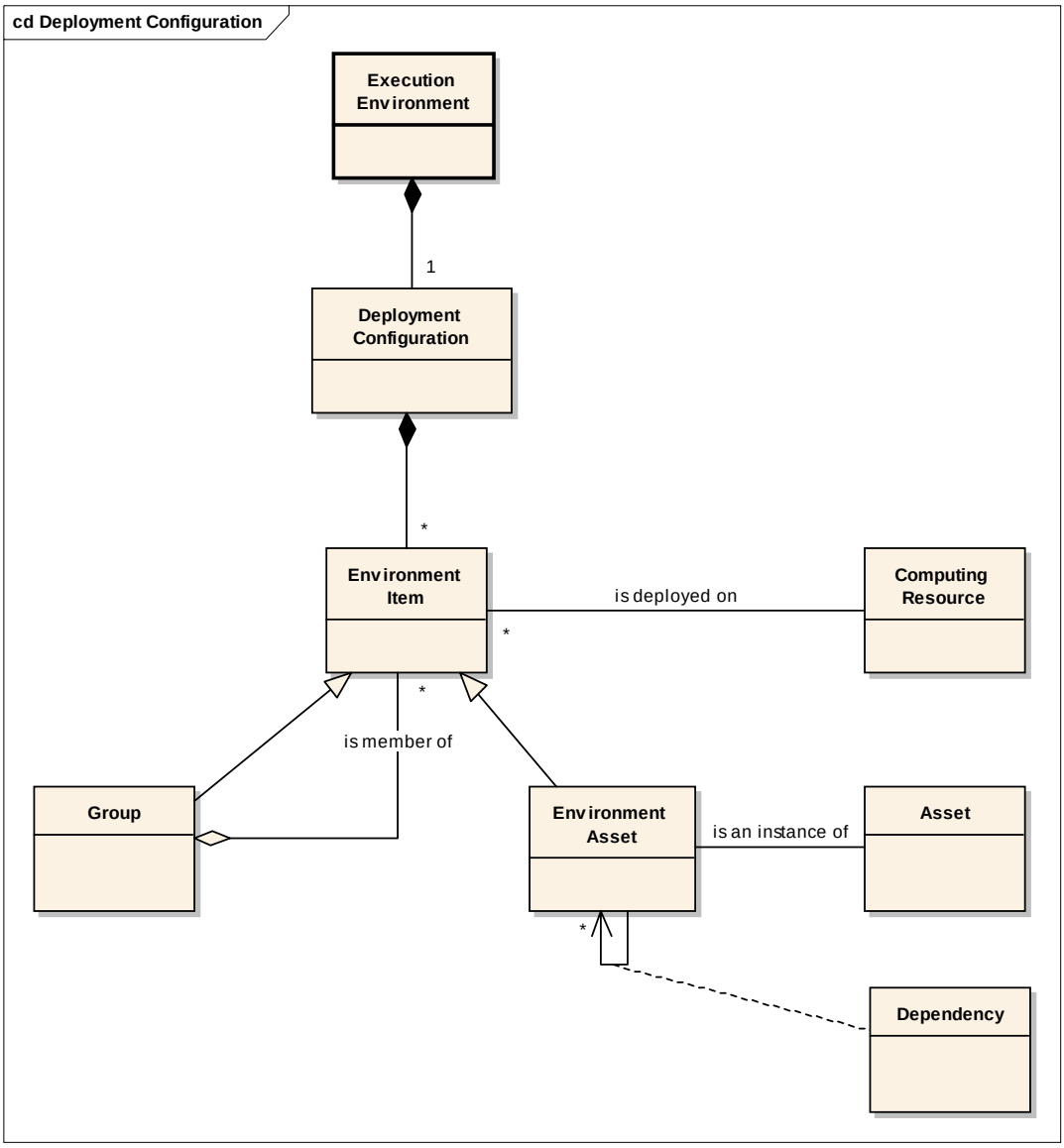


Figure 12. The model shows how a session configuration can be described.

Elaborating further, we realize that not only environment assets can be an environment item, but also groups. Rather than assigning single environment assets to a computer, one at a time, the user should be able to group several items, and assign the group instead, a useful feature when assigning the same set of assets to many computers. When replacing a single, replicated file, such as a network configuration file, this is particularly useful. The file can be replaced in the context of the group, rather than on all utilized computers. The application will automatically, or on demand, substitutes the file on all relevant computers.

Figure 12 shows a `Dependency` relation from an `Environment Asset` to itself. The relation describes the dependencies found between deployed assets. A deployed logger federate may e.g. depend on a deployed database. Fully specifying all the different kinds of dependencies that may exist between assets in a general way is difficult. There may be loosely coupled dependencies, e.g. for semantic interoperability, and there may be more directly used

dependencies, like TCP/IP connections. In some cases, the dependant components must be located at the exact same computer. In other cases, it is sufficient for shared data to be accessible through a network file system. Further analysis and detailed specification of these properties is expected to be highly dependent on design and technology, and is therefore deferred to later chapters.

4.2.2 Execution configuration

Occasionally, there is a need for varying configuration input data between different simulation executions. Perhaps we want to install several terrain databases in our environment, and choose upon execution which one to use. Rather than creating a new EE for each such configuration, thereby duplicating most of the information already available, an environment can contain several different `Execution Configurations`.

Figure 13 shows that we have added a specialisation of the `Environment Asset` called `Executable Environment Asset`. The new type is considered as being executable, which is important, because only executable environment assets can be a part of any run-time configuration. A `Start-up Configuration` indicates that we need to describe a user-ordered list of environment assets that should be executed upon federation execution. Each asset will further need to contain information of some sort telling which files to locally use and execute.

Because different run-time configurations of the same EE probably can be quite similar, we have added a `Depends on` association from `Run-time configuration` to itself. This is for informing about this similarity, and to indicate that only changes should have to be stored from one configuration to the next.

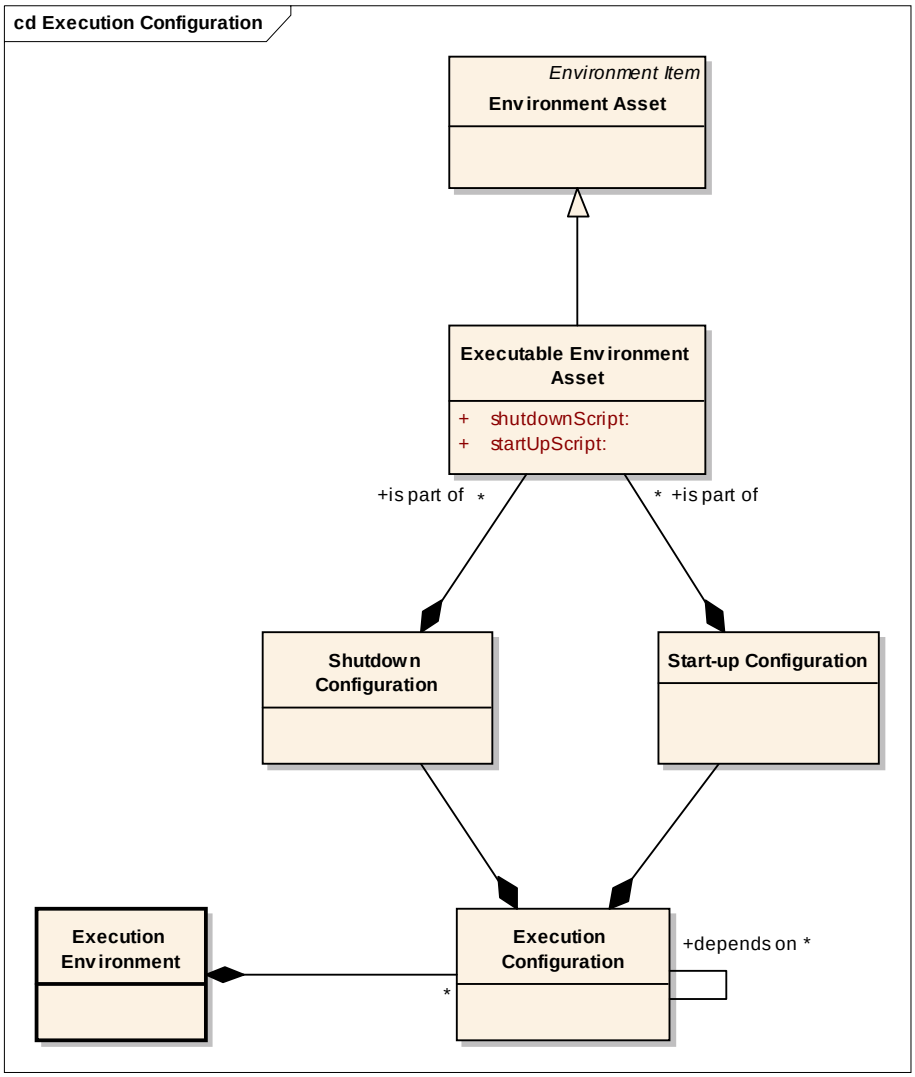


Figure 13. The figure shows how a simulation environment may contain several different run-time configurations.

5. Conceptual design of the EE

The EE should be on a Service Oriented Architecture (SOA). In this architecture, applications make use of services available on a wide and/or local area network. A service provides specific functionality, which is exposed to consumers using a standardized interface. A service can serve an application on its own, but also work in cooperation with other services. In this way, services can be composed to satisfy more complex business demands. SOA is simply an approach to interconnect applications enabling them to take advantage of each other. A special characteristic that differentiate an SOA from other architectures is loose coupling. This means that when a client, either an application or a service, is using a service it does not need to be aware of the implementation details of the service. Regardless of implementation language, or what platform the service is deployed on, the client can utilize it by communicating through a well-defined interface (Ort 2005).

Today, an SOA is usually realized through use of web services. Web services are deployed using a set of standardized protocols and technologies. Recently, extension of the web service concept has emerged through the notion of grid services. This concept is further described in section 2.2.

5.1 Principal service categories

The following section describes principal service categories and their role in establishing a framework for execution of distributed simulations. The EE comprises five main components defined as services, namely:

- Computing service
- Repository service:
  - Storage service
  - Directory service
- Simulation Infrastructure Plug-in

In addition to these services, the EE comprises a Simulation Engineer Workbench component. This component represents a tool-suite, accessible for users of the EE, which is used for utilization of the EE services. Figure presents an overview of the EE components and how they are interrelated. In the following sections, the components are described in more detail.

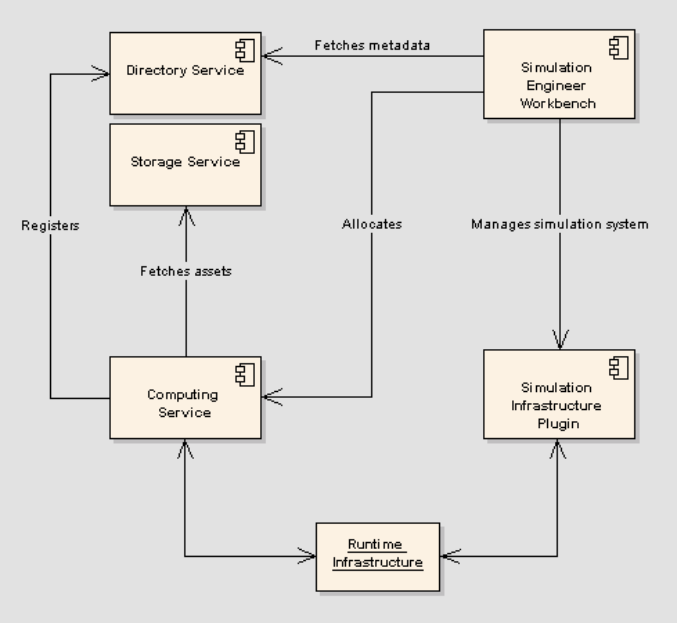


Figure 14. Main components of the execution environment.

5.1.1 Computing Service

A computing service is deployed to utilize the computing capacity of a machine for execution of jobs. A computing service can receive an arbitrary number of jobs for execution, as long as its hardware and software specification meets defined requirements. The computing services are registered in a repository to indicate their availability and features. A job in this case refers to a component of a simulation. If the HLA RTI is used as simulation infrastructure, a job represents a federate. Depending on the capacity of the machine, where a computing service is deployed, or the preferences of the machine's owner, the computing service may process a variable number of jobs. A computing service may also represent an asset bound to a specific piece of hardware i.e. an asset that can not be migrated. This may include flight simulators, RTI components etc.

5.1.2 Repository Service

The execution environment is highly dependent on a repository for storage of assets and descriptions of computers. The purpose of the repository is to enable sharing of resources within and between organizations in a transparent way. The repository will primarily manage assets such as simulation components, data and computers. A fundamental aspect of the repository is an ontology capturing required aspects of assets and computers. This ontology forms the basis for tagging assets with meta-data, and describing computers, which eases their localization and use. The repository of the EE comprises two basic services; the directory service and the storage service. The directory is responsible for registering and querying of metadata, whereas the storage manages sharing of the actual assets.

5.1.3 Simulation Engineer Workbench

The simulation engineer workbench is the main point of entry for users of the EE. Through this component, users get access to various tools that support the process of executing a

simulation system. Thus, these tools can be seen as means of coordinating the services. The tools of the workbench are used to deploy and execute a predefined simulation system. This process involves querying of the directory to obtain metadata describing assets and computers. Based on the features of available computation service instances and the requirements of each job, the management service determines a preliminary allocation and tries to deploy the jobs at suitable computation services. Given that the allocation process succeeds, the jobs are executed upon request from the management service that monitors the execution, for instance detecting failures, continuously.

5.1.4 Simulation Infrastructure Plug-in

The Simulation Infrastructure Plug-in enables control over a simulation execution from the perspective of the Simulation Engineer Workbench. This involves general tasks such as starting, stopping, pausing and resuming a simulation execution (simulation management). Further, it provides information to the Simulation Engineer Workbench concerning the status of a simulation execution. This includes signalling of lost simulation system components in order to enable recovery of a simulation system. The Simulation Infrastructure Plug-in is tightly coupled with the simulation infrastructure used for simulation execution. Thus, if the HLA RTI is employed, the plug-in is a member of the current federation execution, i.e. the plug-in is a federate.

6. Applying existing technology

6.1 GRID Technology

6.1.1 Requirements

In this section the requirements for the planned execution environment will be discussed. The definition of OGSA is based on a set of requirements, which in turn were extracted from a set of use cases. One of these use cases was an HLA-based distributed simulation, or more specifically a “distributed collaborative environment for developing and running simulations across administrative domains” (Foster et al 2004). This suggests that the concepts on which OGSA is based are similar to those of the planned HLA-based execution environment for distributed simulations by FOI and FFI. In the following sections the feasibility of OGSI and OGSA as the basis for this execution environment will be discussed. The main focus will be on how the requirements of the FOI-FFI EE can be met.

Deployment

Deployment of a simulation as a centralized process is not a problem since an individual Grid service can be used as the input valve into the Grid, or as a coordinator. Once deployed the simulation components will be distributed in the grid.

Discovery, monitoring and resource provisioning is fundamental to OGSA, so searches for assets and resources, along with selection and installation are certainly feasible. OGSA also contains Information Services that provide the important OGSA capability of being able to access and manipulate information about resources and services in the Grid. Information Services may associate some meaning with data but no specific format of the data is prescribed. GT4, (Foster 2005), provides standardized mechanisms for associating XML-based information with grid entities to facilitate discovery. These mechanisms are also easily incorporated into user-developed services. Therefore it is possible to implement descriptions of resources and assets using any XML-based syntax, for instance RDF or OWL. GT4 also provides services to collect recent state information from information sources. These services can be configured to collect information from arbitrary kinds of information sources. GT4 provides several means of querying for the collected information as well. All this should allow a data model expressed in a formal language to be used to describe assets and resources. The descriptions can easily contain location, security and licensing information about resources as well as the dependencies between them.

Tools for resource allocation may be added by deploying new Grid services. These tools may provide both manual and automatic allocation mechanisms, in addition to those already present in the Grid environment.

Execution

OGSA contains a type of service called Execution Management Services (EMS). EMS handles instantiation of units of work (jobs) and manages them until completion. Examples of jobs are OGSA applications or non-OGSA applications such as a Java servlet or an HLA-federate. Jobs are handled through:

- Finding candidates for execution locations subject to certain constraints regarding different resources such as memory, CPU, licenses etc.
- Selecting execution location and preparing for execution, considering deployment, installation and configuration.
- Initiating and managing the execution, including monitoring and controlling that it does not fail or fails to meet its goals.

OGSA manages EMS by implementing multiple replaceable components as services. User-defined components may also be used. Subsets of all services can be combined to realize EMS capabilities, for instance making resources easily accessible by automatically matching a grid application's requirements against available resources. Important EMS services are for example a service container where jobs can be run and a Persistent State Handle Service (PSHS) that keeps track of where states of executing entities are located to enable quick access. PSHS thus facilitates both migration and replication. Another important service is a Job Manager (JM), which manages all aspects of executing a job from start to finish. It is responsible for orchestrating necessary services, interacting with containers and configuring monitoring services. A JM can be implemented by for example a queue that accepts, prioritizes and executes incoming jobs. The provided ability to select where to execute different components is also important.

EMS can successfully be used to execute deployed simulation components or subsets thereof. How to stop and control a running simulation through EMS is very much up to the developer of such services.

A typical example of an OGSA information services is a logging service that is used to store all messages sent by an executing job. A job that is to be executed may be published as a service instance in the Grid, and thus be subject to all basic Grid service functionality. If an executing job does not send any messages it may still be queried for information. To provide miscellaneous information continuously to different nodes in a Grid is therefore an easy task.

GT4 provides a set of services, called GRAM, to submit, monitor and cancel jobs on local or remote computing resources. One purpose of GRAM is to be able to run arbitrary programs while performing stateful monitoring and achieving reliable operation.

Post-execution

When a job is finished executing there is a need for cleanup. This may be part of the preparations for another job that will execute, or it may be part of a post-execution process. Such actions are probably best implemented as part of EMS, but may also be implemented as special user-defined services. PSHS described in the previous section enables the state of an executing simulation component to be handled separately, for example to migrate the component or reset it to a stored pre-simulation state.

Monitoring and discovery in a Grid require that information can be collected from multiple distributed information sources. GT4 provides both certain aggregator services to collect recent information from registered information sources and different query methods. Examples of information sources are files, programs and network-enabled services etc. All collected information is maintained as XML, which enables semantic information expressed in formal languages such as RDF and OWL to be stored using XML-based syntaxes. Arbitrary information sources may be used if the information is converted into and stored in an appropriate XML representation. All this makes it possible to store data generated during

a simulation execution, to label and to collect it using distributed components. Since all data is stored as XML it can be retrieved according to labels using standard XML query methods. User-defined services for collecting and querying data may also be provided.

Robustness

The monitoring capabilities in OGSA along with facilities to share information, in both pull and push mode, among resources and services in a Grid provide a solid basis on which to base a robust simulation execution environment. Simulation execution services can be implemented both by using and by extending EMS. Capabilities not included can thus be provided by the developers of the simulation execution environment, such as the ability for a failed simulation component to rejoin, manual and automatic restoration after faults and other specific fault-tolerance mechanisms. EMS services such as PSHS and the fundamental discovery capabilities of OGSA enable restorations of failed components into other hosting environments that meet necessary requirements.

GT4 provides pull modes of information sharing through its query capabilities. Push mode information sharing is accomplished through the OGSI notification interfaces. These interfaces implement a publish-subscribe scheme by allowing some Grid services to act as information sources that send information, and others as sinks that receive it.

Non-functional

Grid applications may span several administrative domains each with their own security policies that must be enforced. All interactions between services within an executing grid application must adhere to both local security policies and to policies established for the virtual organization that the user belongs to. The security components of OGSA aim to support, integrate and unify security mechanisms and protocols so that disparate Grid services can interoperate securely. Audit services offer security logging so that security-relevant events in the Grid may be tracked. Such logs may then be used to verify that security policies are being enforced etc.

GT4 provides authentication and authorization capabilities based on the X.509 (ITU 2000) certificate standard. Proxy certificates may be used to enable temporary delegation, which can help achieve high system availability by automatically providing (authorized) access to alternative resources or services. OGSA also specifies the enforcement of security policies such as redundancy etc to achieve high availability. GT4 offers message-level and transport-level security to ensure confidentiality and integrity.

Since all services in a Grid are interoperable the hosting environments in which published services are implemented may vary substantially.

6.1.2 The Semantic Grid

Both the Grid and the Semantic Web are sometimes said to be the future of the Web. This section focuses on the similarities between these views and the possibility of a merger between them, called The Semantic Grid.

When performing large computations using Grid technologies it is desirable to (automatically) reuse existing data, services and even knowledge. The key to seamless interoperability and automation lies in making knowledge explicit and machine-interpretable, i.e. achieving semantic interoperability. “The Semantic Grid is an extension of the current

Grid in which information and services are given well-defined meaning, better enabling computers and people to work in cooperation" (Roure 2005).

Content in the current Grid is XML-based, but it is possible to provide richer resource descriptions and metadata using ontologies and Semantic Web Services. This will require extensions to the standard OGSI interfaces and new OGSA management services etc. However, it may be possible to use user-defined services in the current Grid to provide semantic metadata represented using an XML-based syntax. This topic needs further investigation.

6.1.3 Summary

The Open Grid Services Architecture (OGSA) provides a Web Service-based framework of management services for developing Grid applications and is an excellent tool for the development of an execution environment for distributed simulations. OGSA is based on the Open Grid Services Infrastructure (OGSI), which provides standard Grid service interfaces with mechanisms for discovery, dynamic service creation, lifetime management and notification etc. Globus Toolkit is an implemention of OGSI and OGSA, and can be seen as a set of programmatic building blocks for implementing Grid applications. Version 4 of Globus Toolkit (GT4) is the most recent and most comprehensive release. The core of GT4 essentially implements OGSI. GT4 also provides a suite of management services that facilitate faster and better development of Grid applications. These services only implement parts of OGSA, but a lot of work could be avoided using Grid technologies to develop the execution environment. An on-going project called The Semantic Grid attempts to merge the current Grid with Semantic Web technologies to enable seamless Grid service interoperability and automation.

6.2 Semantic Web

As stated in section 2.3, "the semantic web" is both a vision and a technology set. This section evaluates the semantic web technology and its applicability to the EE.

To briefly recall, semantic web technology consists of

- XML-based syntax, with parsers, simple query and transformation languages,
- RDF and RDF-Schema, with resources, properties and triplets, as well as simple inheritance,
- OWL, a rich ontology language that comes in several varieties, supported by editing tools
- reasoning or inference engines

In order to evaluate its applicability, the following questions must be asked:

- Is the technology relevant to the task at hand?
- Is the technology powerful enough, in terms of both expressive power and performance?
- Is the technology sufficiently mature, with stable standards, available core functionality and user-friendly tools and documentation?
- Will the technology remain and be strengthened further, or is it likely to be replaced by something else?

This section attempts to answer these questions.

An eventual role of semantic web technology is in the data-oriented perspective of the system. As identified in the requirements, there is “a need for a data model and a formal language that can include information about computing resources, network resources, RTIs, federates (models, supporting tools), federate factories, FOMs, license servers, databases, and other hardware such as projectors”. OWL or RDF/XML may be such a formal language.

The data model can be described in many different formats. Chapter 4 describes a data model using UML syntax. This syntax, although well-known and commonly used for documenting data models, is not easily machine-readable. Standardization efforts for UML-based interchange formats (e.g. XMI) have had a difficult start, and few vendors implement the format in a fully compatible way. Another option is to standardize on an SQL-variant description of the data model, trading some human readability for easy use by computer systems.

OWL data in text form is not at all suitable for direct human editing; tool support is absolutely essential. The proposed data model includes several constructs that strain the expressive power of SQL. These constructs are the use of transitive relations, such as the relation “depends on” in the input model, as well as a possibly complex type graph. Few SQL implementations support recursive queries or inheritance, hardly any support multiple inheritance. If these aspects are to be kept in an implementation model, SQL might not be a good option.

OWL, even in the DL or perhaps Lite variant, should have all the expressive power necessary to describe the data model in a formal language. Tool support is maturing, and is at the moment very close to being, if not already, sufficiently user-friendly and powerful. One should, however, question whether a formal language for the data model and the data is indeed necessary for this application, and if the use of a formal language will make the learning curve steeper and detract from the core functionality. This is especially so since the data manipulation necessary is not particularly complex, and is perhaps easily coded “manually”.

The next issue is regarding “automatic allocation mechanisms” for allocating components to computing resources during initial deployment, or possibly as a result of a computer failure or from load-balancing. In the requirements, the word “may” is used, indicating that this is not an absolute requirement.

The problem of determining a feasible or optimum allocation configuration can be compared to similar problems often successfully solved by constraint solvers. The sound foundations of OWL in logic should make it easier to apply automatic solvers, compared to other representation languages. Using a standard language may also provide wider options for alternative solver engines. Even if the language itself is not used, the prerequisite domain modelling process may prove beneficial when integrating a solver in the application, as it pushes forward the need for a certain rigor in the domain modelling.

Whether a standard or general solver will actually be able to solve the allocation problem is not entirely obvious. Consider for example a large deployment scenario consisting of say 100 computers and perhaps three to six kinds of jobs to be spread across these computers. If any type of job may be assigned to any computer, we have approximately $6 \cdot 10^{77}$ different options. Allowing for arbitrary combination of jobs to computers will greatly increase the solution

space size. It is obviously necessary to heavily prune the solution space in some way. This pruning is done by a solver by utilizing user-specified rules and constraints.

The technology and tool support has matured during the last few years, and even months. It is interesting to note that several tools, e.g. Protégé and others, have long histories of development and use in smaller academic communities, long predating OWL. The advent of OWL, riding the popularity wave together with a family of “hot” buzzwords such as XML, might cause tools like Protégé to reach critical mass in terms of user adoption. Furthermore, W3C recommendations are available, making the technology specification relatively stable, even though no standardisation agency has adopted standards in this area so far.

The last question remaining is whether the technologies, standards and formats are likely to remain, or will be replaced in the near future. At the moment there appears to be no similar, competing technologies on the horizon. Because of this, there are few other options than to evaluate the Semantic Web technology set with regards to the present requirements. This in comparison to other technology battles, such as the rivalry between the .NET Framework and Java Enterprise Edition, where similar, competing, options exist.

As stated above, Semantic Web technologies are still very young, and tool support is lacking and seems immature. Still, we see the benefits gained from using those technologies as substantial and interesting, and mean that that the technologies that exists today will solve the requirements put forward for the EE.

7. Preliminary software design

This section describes a preliminary design of the EE. The design focuses on the primary services as identified in section 5.1, i.e. specification of service interfaces and the main relations among individual services.

7.1 Service communication

Figure 15 illustrates interactions among services of the EE in the deployment phase. These interactions are derived from use case 1 – Deployment. The deployment of a simulation system requires a valid deployment description, stating the configuration to be used for execution. To construct this design the user must resolve all dependencies for assets that are part of the simulation system. Thus, tools of the simulation engineer workbench should allow the user to query the directory service for metadata concerning assets and computers. Constructing the initial design may involve several queries to sort out all dependencies of the assets. During this phase, several assets, such as configuration scripts and parameter files might be altered to suit the current simulation execution. Thus, assets requiring such modifications are downloaded from the storage service. When the deployment design for a simulation system is completed, altered assets are transferred back to the storage service along with metadata that states the context of the assets. Note that altered assets do not replace original assets, but are stored separately along with metadata stating what design they belong to. When this is completed the simulation engineer workbench installs the assets in the selected computing services. Fragments of the design are transferred to individual computing services to enable them to download required executable code, configuration scripts, data files etc., and to ensure proper installation.

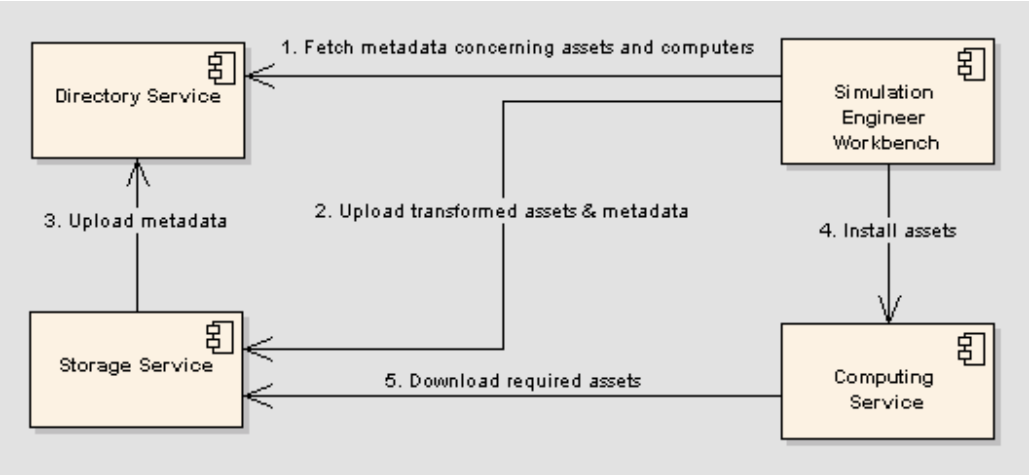


Figure 15. Interactions among services in the deployment phase. The diagram is a UML 2.0 Communication diagram (OMG 2005).

Next, the EE enters the execution phase, as defined in use case 2 – Execution, se figure 16. At this point all assets of the current simulation system has been installed. First, the user uses the tools of the simulation engineer workbench to start-up the required assets. This includes

individual components of the simulation, but also start-up of required runtime infrastructures. These infrastructures are accessed through computing services and are defined as assets, which are not moveable. At computing services responsible for simulation system components, processes are started and the targeted components join the current simulation execution. However, the simulation execution is not started at this point, i.e. the logical time of the simulation is not advanced. The user has control over the simulation system through the simulation infrastructure plug-in, which is accessed using the tools of the simulation engineer workbench. The control involves general management tasks such as starting, stopping and pausing the simulation and changing the wall-clock to logical time ratio. When the user issues a start command the simulation execution starts. During the simulation execution the status of individual computing services can be monitored, e.g. controlling the CPU load, or memory allocation. If the current status is considered unsatisfactory, simulation system components can be migrated. This mechanism is explained in subsequent sections. When the simulation execution is completed, installed assets are stopped. This means that simulation system components resign from the execution and their processes are destroyed.

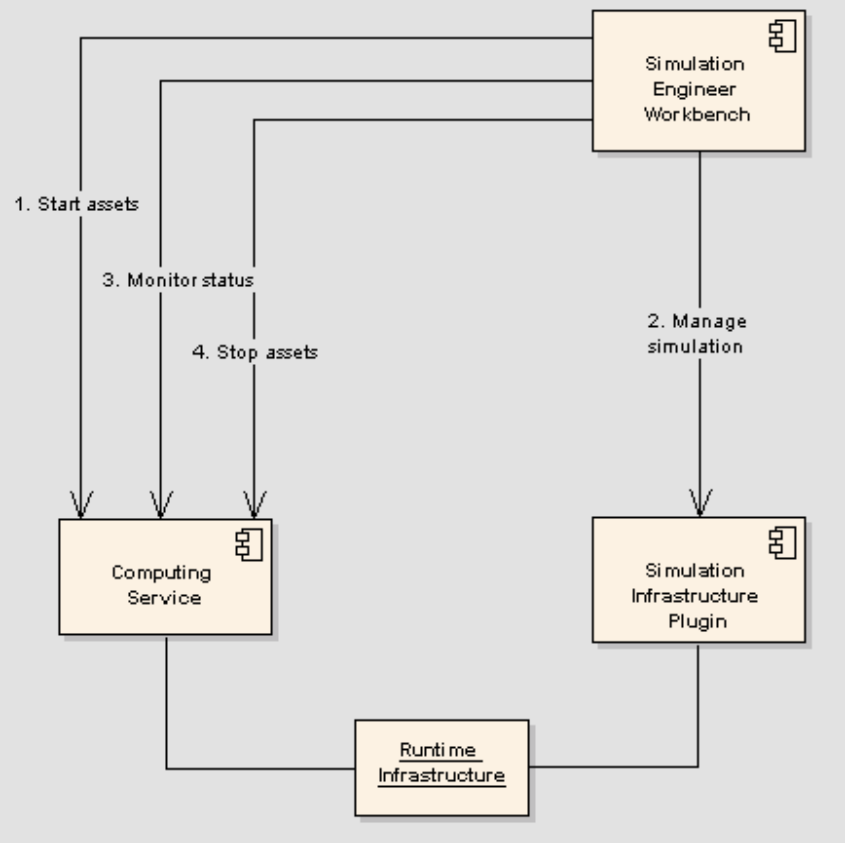


Figure 16. Interactions among services in the execution phase.

To enable migration of failed simulation system components, and to enable restoration of simulations in general, the EE comprises functionality for saving states of individual simulation system components, i.e. check-pointing. Figure illustrates the sequence for saving states in the storage service. In order to produce a global snap-shot of the simulation system, the simulation infrastructure plug-in is used. This plug-in is member of the simulation execution, and thus it has the ability to initiate a save procedure within the simulation system.

When a state has been produced for each simulation component, the simulation engineer workbench issues a request for distribution of checkpoints at each computing service. This means that each computing service uploads a checkpoint, with associated metadata, to the storage service.

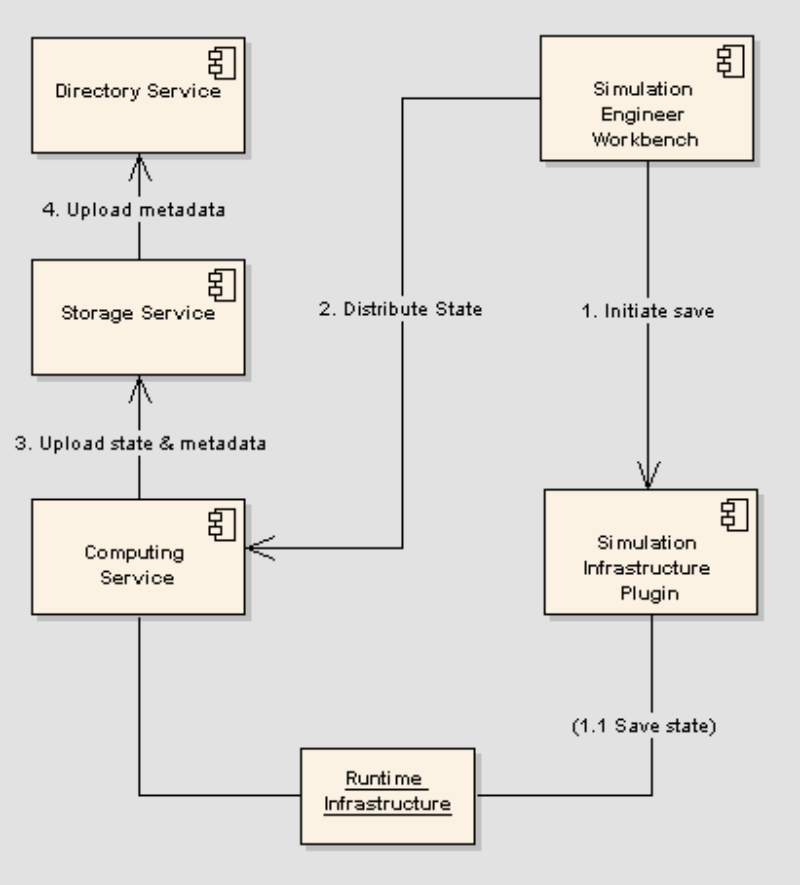


Figure 17. Interactions among services when saving states of simulation components.

The check-pointing as described enables migration (transfer) of simulation components between computing services at runtime. This is crucial in case of failure of a critical simulation component, or for the purpose of making the simulation execution more effective (load-balancing), as described in use case 3 – Migration. Below, two cases of simulation component migration are described; coordinated migration (figure 18) and uncoordinated migration (figure 19).

A coordinated migration is performed in response to user requests, for instance in case of heavy CPU load on a certain computing service that effect the performance of the simulation. In this case the simulation infrastructure plug-in is used to temporarily pause the simulation execution. Furthermore, the concerned simulation system component is instructed to produce its state. Next, the concerned computing service is called to stop the concerned simulation system component and to distribute the recently produced state. The computing service forces the simulation component to resign from the execution and destroys its process. Then the state is uploaded to the storage service along with required metadata. The simulation engineer workbench localizes an alternative computing service that will host the migrated

simulation system component (this can of course be done in advance, or the computing service might already be known). Next, assets are installed in the new computing service, which involves downloading of required executable code, configuration scripts, checkpoints etc. from the storage service. When this is completed the simulation system component is started from the simulation engineer workbench and the simulation infrastructure plug-in is used to restore the concerned simulation system component and to resume the simulation execution.

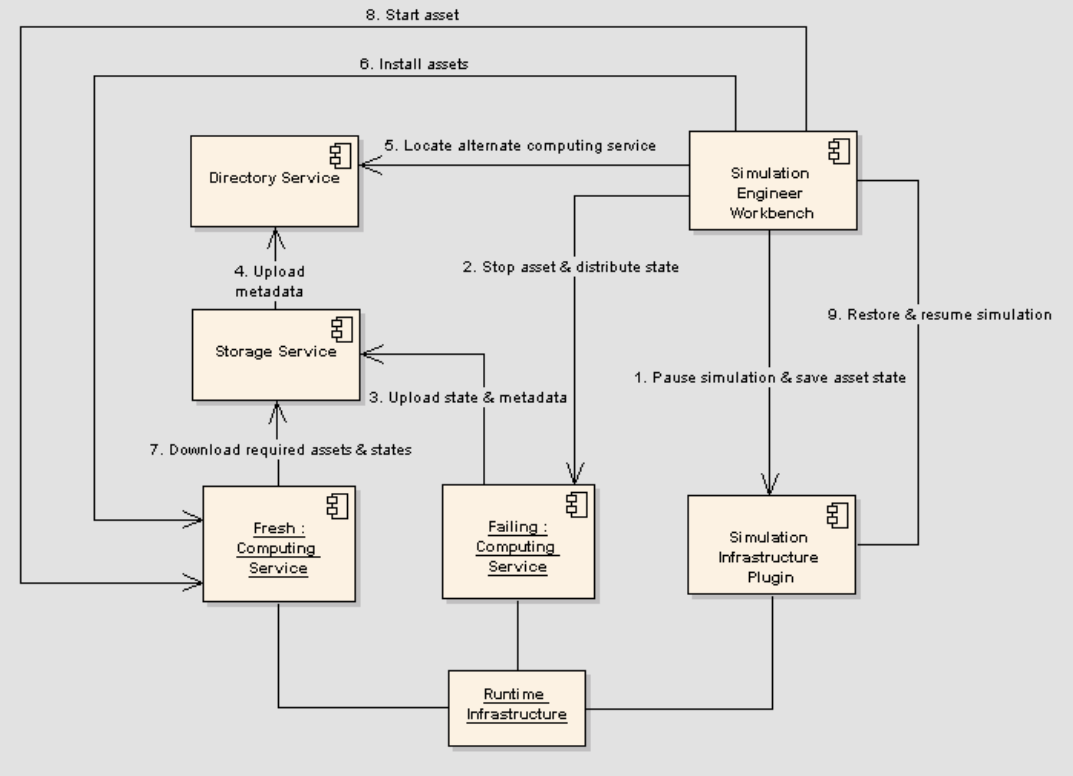


Figure 18. Interactions among service in case of coordinated migration.

An uncoordinated migration is performed in response to asset failure. In this case the simulation infrastructure plug-in signals the failure of a simulation system component to the simulation engineer workbench. In response to this, the simulation infrastructure workbench stops the simulation system components at the concerned computing service. Next, a suitable computing service is identified by querying the directory service. The assets are installed in the new computing service, which downloads required executable code, configuration scripts, previously saved states etc. from the storage service. When this is completed the simulation system component is started at the new computing service. Finally, the simulation infrastructure plug-in is used to restore the federation to a previously saved state, after which the simulation execution is resumed.

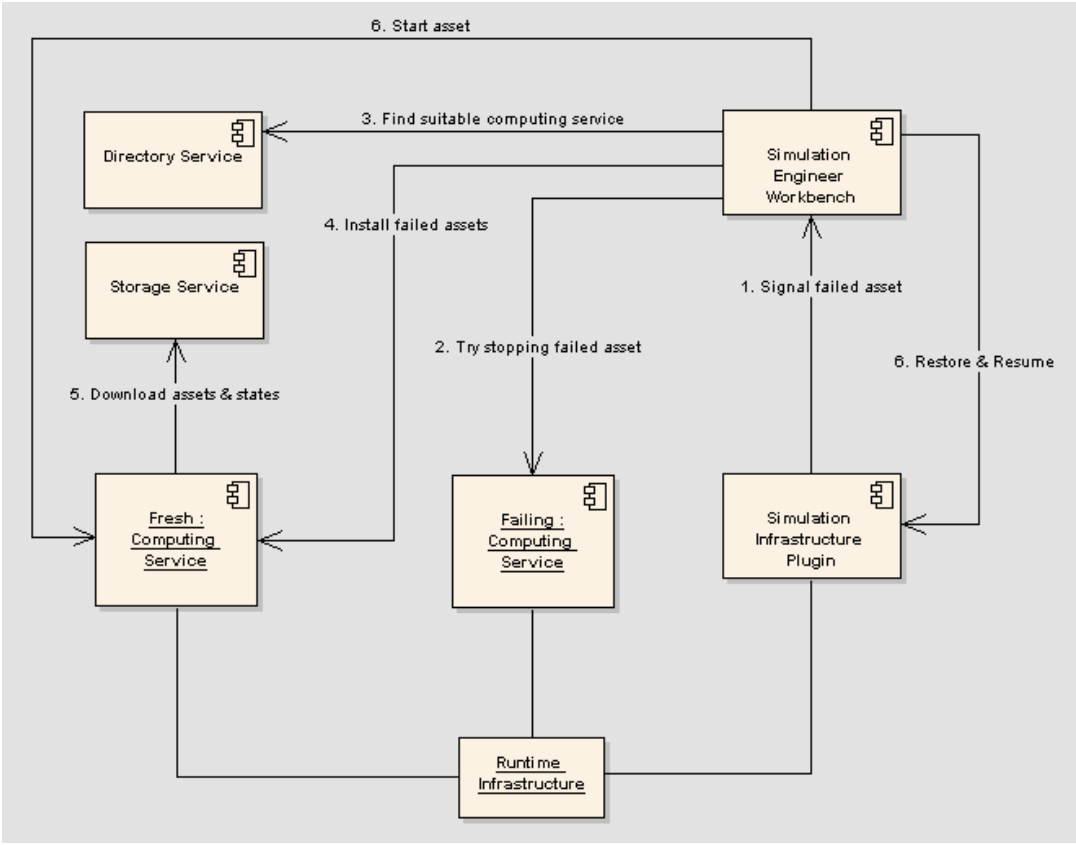


Figure 19. Interactions among services in case of asset failure.

When the simulation execution ends, or when the user chooses to end the simulation execution, the EE enters the post-execution phase as described in use case 4 – Post-execution, see figure 20. The main purpose of post-execution is to make asset logs and produced results available for subsequent analysis. To do this, the simulation engineer workbench issues a distribute logs request at each computing service. This forces a computing service to collect logs and results from all assets that it has been responsible for during simulation execution. When logs and results have been collected it is uploaded to the storage service with associated metadata. Next, the user can choose to uninstall assets from computing services, or leave them installed for subsequent simulation runs.

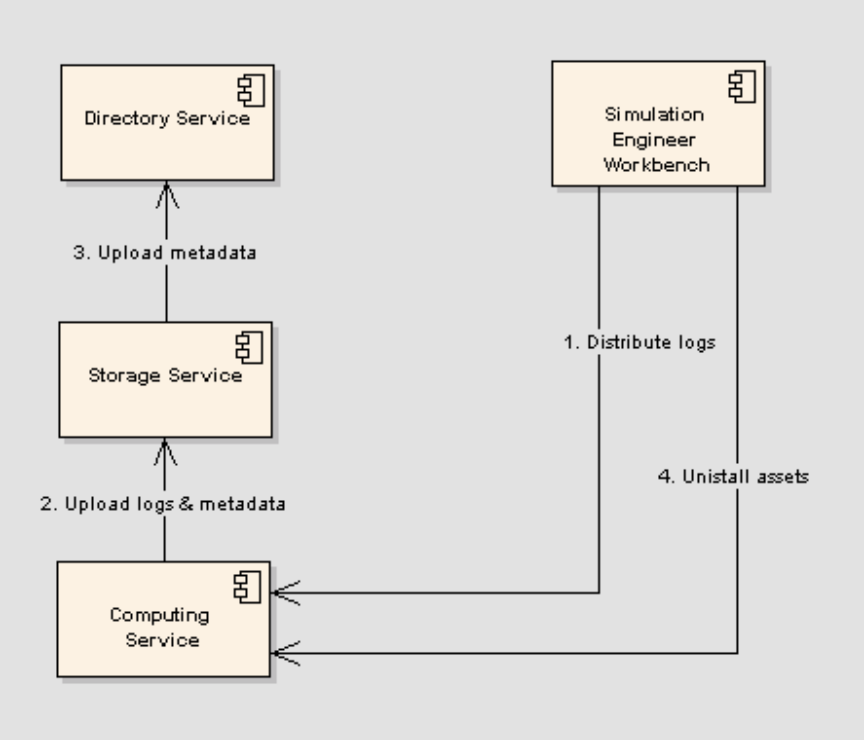


Figure 20. Interactions among services in the post-execution phase.

7.2 Service interfaces

Given the interactions among services described in Figure 15 through Figure 20, service interfaces of the EE can be derived. These interfaces are presented in Figure 21. The directory service interface defines four operations; register, query, delete and update of metadata. The method `registerMetadata` is used by the storage service when new assets are added to the EE, or when a computing service registers its presence on the network. The `queryMetadata` is used through the tool suite of the simulation engineer workbench to fetch information concerning available assets in order to resolve dependencies of a simulation system to be executed. As the names imply, the `deleteMetadata` and `updateMetadata` operations are used to delete and update metadata concerning a particular asset.

The storage service interface defines four operations, these are; download, upload, delete and update asset. The `uploadAsset` operation is used when an asset is shared by a user, but also when other services store intermediate files during the deployment and execution of a simulation system, i.e. upload transformed configuration scripts, states of simulation components etc. In order to fetch a copy of an asset, when it has been localized through the directory service, the `downloadAsset` operation is used. The `deleteAsset` operation is simply used to unshare an asset contained within the EE, whereas the `updateAsset` operation replaces an old version of an asset with a new one.

The computing service interface defines eight operations. First, the interface contains operations for management of assets. This includes operations for installing, starting and stopping an asset. The `installAssets` operation instructs a computing service to download required assets from the storage service and install them according to a pre-defined design. When the `startAsset` operation is called, the process, or processes, of a particular asset are

created, whereas the `stopAsset` operation destroys the processes. The interface also comprises operations for distribution of certain files to the storage service. In order to distribute the current state of a simulation system component the `distributeState` operation is used. When a simulation run has been completed the `distributeLogs` operation is used to disseminate logs and results from individual simulation components. Finally, to enable monitoring of the status of individual computing services the `monitorStatus` is defined.

The simulation infrastructure plug-in contains operations to allow control over a simulation system from the simulation engineer workbench. Thus, this interface includes operations for starting, stopping, pausing and resuming a simulation execution from the perspective of the simulation runtime infrastructure. Furthermore, operations for enabling simulation component restorations are included. This involves operations for instruction of state-saving within a simulation system, through `saveState`, and consequently, operations for restoration and resume of a simulation execution, through `restoreSimulation` and `resumeSimulation` respectively. Finally, the interface defines an operation for setting the speed of a simulation execution, through `setTimeRatio`.

The simulation engineer workbench is not a service in a common sense, but comprises various tools that support utilization of the EE. However, certain operations are needed to enable callbacks from services within the EE. An example of this kind of operation is signalling of a failed asset. In this context the simulation infrastructure plug-in functions as a detector of failed components. In the case of failure, proper precautions must be taken to ensure the integrity of the simulation system, and thus, the simulation engineer workbench is notified, enabling a user, or a tool, to resolve the problem. Other types of operations in the simulation engineer workbench might be needed, but this is not elaborated further in this report.

Figure 21. Service interfaces of the EE.

7.3 Data model implementation design

In chapter 4, specification of assets, a data model for capturing information about assets was developed. The model identified necessary information for the EE, if it were to help the deployment engineer setting up a new simulation system. This information was regarded as input, which means that it should have been made available in some known repository before entering the EE. A data model for describing the environment design was also presented in the same chapter. That model captured the set-up information produced by the deployment engineer during design, information regarded as input to the execution phase of the EE.

During input- and environment model design, we did not consider how the models could be realized, that is, which technologies that could be used for storing and retrieving the information. Several very different solutions may be applied. We could e.g. use a relational database, a specially designed Java application, or just a normal text-file. In this project we decided at an early stage that one of our objectives was to explore the opportunities of the Semantic Web. Consequently, we have decided to use the technologies found there.

For implementation design, we will use OWL DL. One of OWL's strengths is the possibility of distributing the model over a network, and not only the model's data. The model can be extended by various users in the network, and in this way evolve as new types of information are needed. It is unlikely that any single user in the network holds the full picture of the model at any time, as the model is not meant to be centralized in this way. Any component using the model must explore and learn the model at the same time as processing its own queries. This is important to realize as it tells us that we should not try to describe in OWL every type of simulation component needed, in advance. Component types can be added when needed by those needing it. We only need to identify what we explicitly will ask for in our applications, and let an OWL reasoning engine deduce the rest. Because we have chosen to use OWL, we will reference our implementation data models as ontologies below.

7.3.1 Input ontology

As already stated, we will use OWL for creating an ontology based on the input model in chapter 4. The proposed ontology will not diverge much from this model, as OWL contains a very similar set of modelling constructions. Actually, OWL has more rich set of constructions, some of which we will make use of to simplify the ontology. The result is shown in figure 22.

From the figure we can see that an `Asset` has a `depends on` relationship to itself, which means that an asset is dependant on other assets. As discussed earlier in this report, it is difficult, if not impossible, to prescribe all the different types of dependencies that may exist between assets. An HLA federate may depend on an RTI, a specific FOM, a licence file, one or more databases, etc. Different assets have different dependencies. How should those dependencies be implemented upon deployment? The EE may deduce that a certain federate will need a certain FOM-file to execute, but this is not sufficient. The EE must also know how to implement that dependency, which in some cases may be which asset configuration file to alter, and how. In other cases, the path to the FOM-file should be given as a command line argument when executing the asset. Or perhaps the asset checks an environment variable to find the path. The possibilities are numerous.

To address the above, we find it most practical to enable the creation of tailored solutions for each dependency. As a result, we include the possibility for the user to create script files to be

added with the dependency information. Such scripts can be written in e.g. Python, Perl or bash, and include operations for manipulating any files on the host, inside a given frame of security. The script file itself is contained inside a zip-file together with other kinds of data needed, and executed on the host after all assets have been installed in the network. The `DataTypeProperty` realizing this is shown in the ontology (figure 22) inside the `depends on` association.

In addition to a script file, a human readable `description` is also added inside the association class. This text is intended to describe the purpose of the dependency, and instruction on how to realize it if a script file does not suffice. A `same computer` variable is also present, which means that two assets of an association must be installed on the same computer. This information is helpful when aiding a deployment manager through a design process.

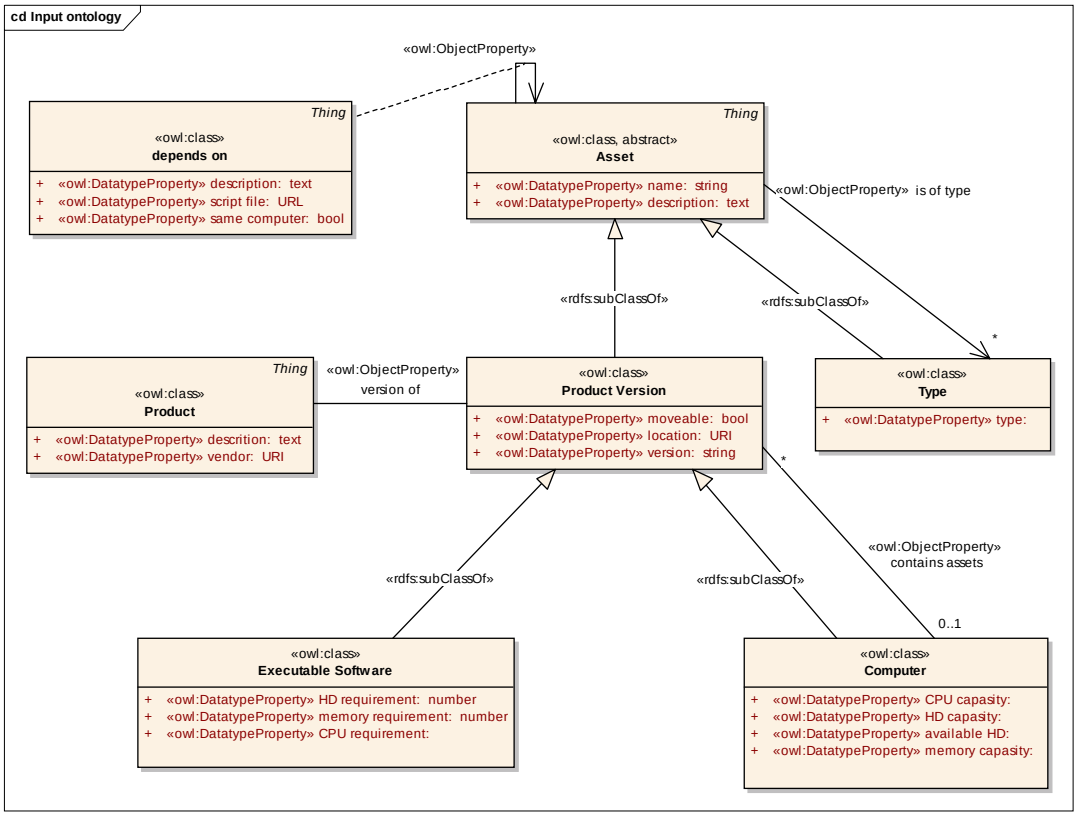


Figure 22. Ontology implementing the input model

The notion of an `Asset` in the ontology is the same concept as the `Abstract Asset` in the input model. We can see from the stereotype that it is still considered as abstract, which means that there should never be descriptions of e.g. simulation components that are only considered as assets. Instead, an asset may either be a `Product Version`, `Executable Software`, a `Computer`, or only a `Type`. The latter has been made a subtype of `Asset` to ease the process of describing dependencies between product versions and asset types, as well as between types themselves. If e.g. the two types “Federate” and “RTI” exist, we could easily add a dependency from “Federate” to “RTI” representing that every federate in the repository will need an RTI in order to work.

Federates, and simulation components in general, will usually be described as `Executable Software`, because they represent something that can be executed. Licence files and other types of data will only be `Product Versions`. From the diagram we see that a `Product Version` can contain an association to a `Product`. A `Product` describes a given artefact more generally, and gives the user an anchor point for finding different version of the same product in the repository.

Looking at both the input model and the ontology, there are some differences. First, we have removed the separate notion of `Deployment Info`, and realized it as a part of `Product Version` and `Executable Software` instead. In this way, computers, simulation components, as well as data, can have deployment properties. Another difference is that only true executable software can possess CPU requirements and alike. Such software will therefore be instances of their own class.

A final major difference between the models is the removal of the meta-types found in the upper left part of the input model. In OWL, additions such as adding new classes and properties can be made dynamically and separately from the reasoning engines. Such engines do not have to know about new extensions to use it. Recall, this is a part of how OWL works. A repository manager that adds a new type of asset to the repository is free to extend the model at any time. He can easily describe this type by sub-classing `Type`. To add a new type "Federate", he would create a sub-class `Federate` from `Type`. If `Federate` needs some specialized properties (e.g. `HLAVersion`), this could be inserted into the class as well. When he later inserts a federate into the repository, the `is of type` relationship would point to a new instance of `Federate`. For this reason, we need not include meta-types in the ontology, since a system for this is already present.

7.3.2 Environment ontology

An environment model was presented in chapter 4. This model expresses the information needed to describe a deployment configuration, and is used both for installing, executing and removing a simulation system. While the input model describes what we need to know in advance, the environment model describes what we will produce during Deploy. A design of the environment model meant for implementation is the topic of this section.

The input ontology and the environment ontology could be realized as two different models. This is similar to how it was presented in chapter 4. By separating the models, we could use a simpler technology than OWL for describing the environment (e.g. just plain XML). However, one effect of this would also be that we were unable to execute queries about both assets and environments at the same time. If we e.g. wanted to know which simulation systems a federate previously had been part of, the input ontology alone would not suffice. And since we value the possibilities for such cross-queries, we have decided to combine the models into one ontology.

A diagram showing the environment part of the resulting ontology is depicted in figure 23. This solution is nearly identical to the model presented in chapter 4, but has been coupled with the input ontology. This can be seen at e.g. `Deployed Product Version`, which has an association towards `Product Version`. The same is also true for `Realized Dependency`, which has an association towards `depends on`. Recall, both `Product Version` and `depends on` was part of the input ontology.

Some classes in the environment model have also been dropped, as they turned out to not carry any valuable information. The class `Deployment Configuration` is one such example.

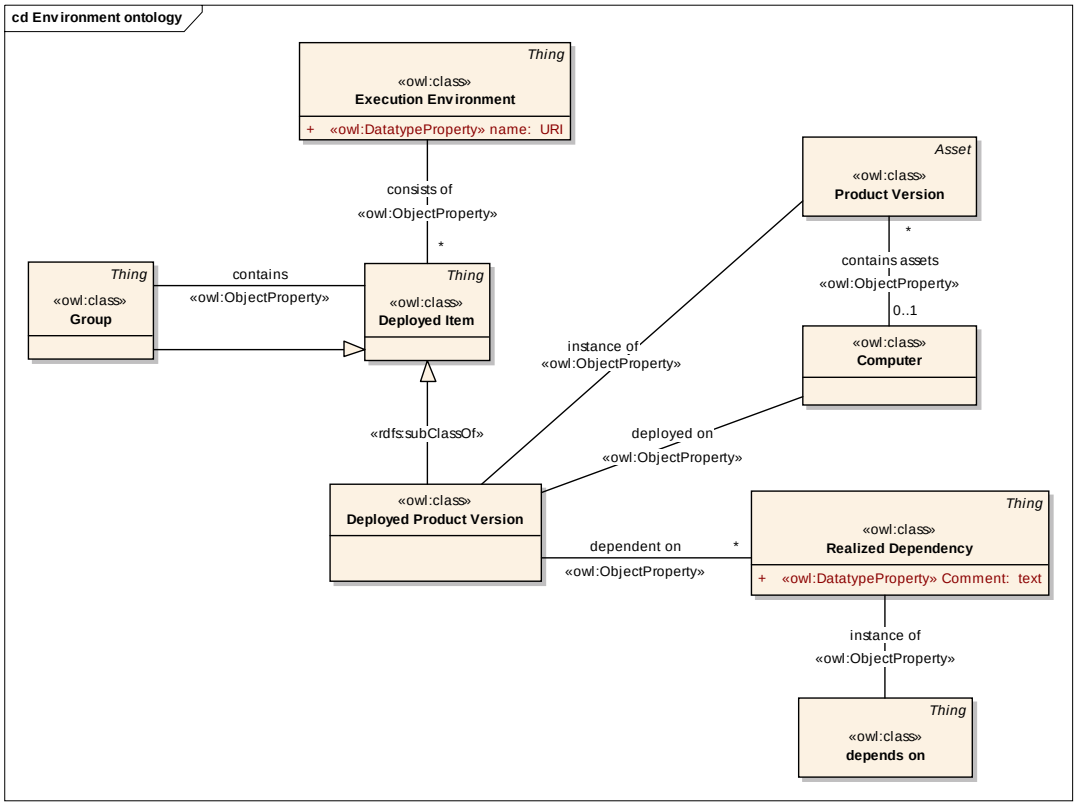


Figure 23. Environment ontology. The ontology is coupled with the input ontology.

Similar to the model in chapter 4, a part of the ontology also contains information concerning the execution phase. This part enables the deployment engineer to specify how an EE should start and stop a simulation system. It also captures the fact that several start-up and shutdown configurations may be desirable.

From figure 24 we see that a `Deployed Product Version` has been specialized into a class `Deployed Executable Software`. This is a consequence of the fact that only executable software can be started and stopped by the EE. Hence, the `instance of` association has also been overridden to ensure this. Finally, because a simulation system might impose certain needs regarding how it may be started, both a `Start-up order-` and `Shutdown order` class has been added.

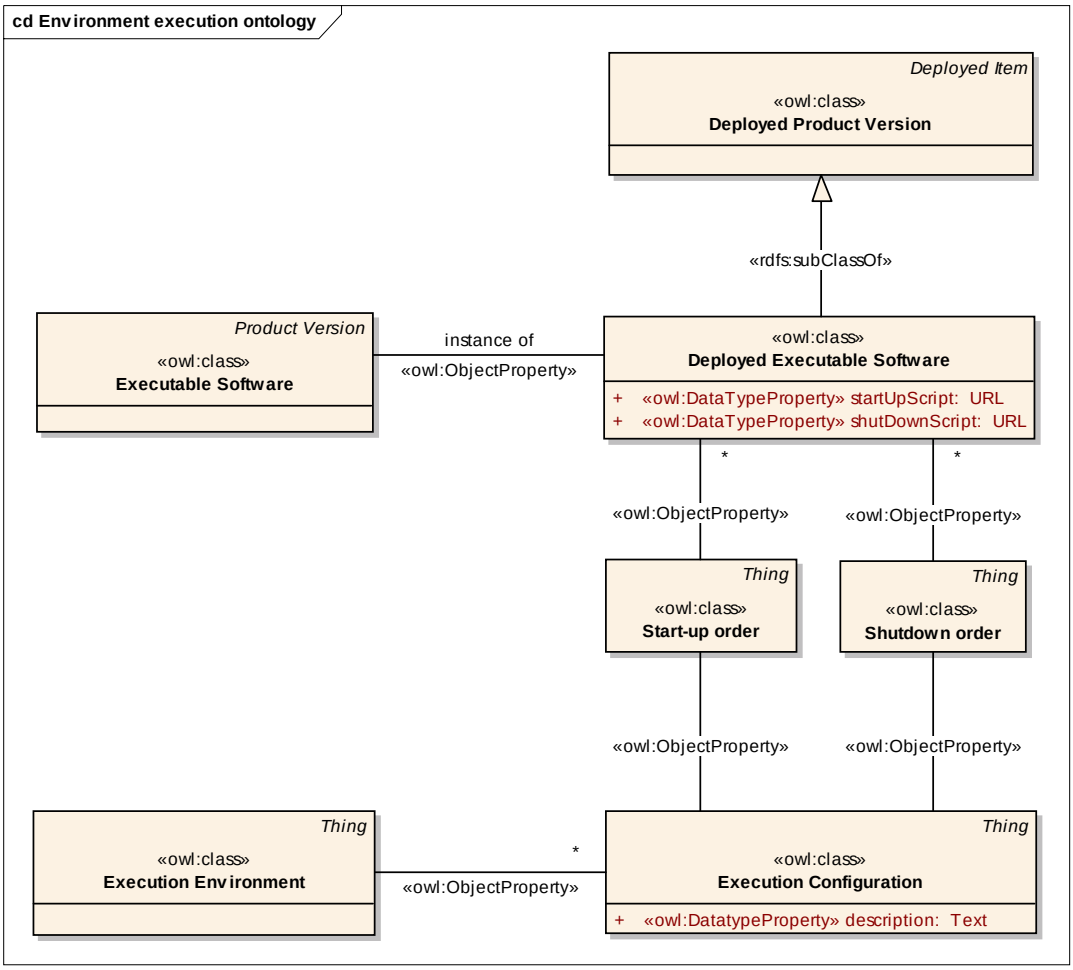


Figure 24. Environment ontology with respect to the execution phase.

8. Conclusion

Exploitation of M&S at a larger scale, for instance in NBD settings, requires flexibility and robustness in the way simulation systems are managed and executed. First, efficient means of sharing resources within and between organisations are required to enable availability and reuse of simulation models and data. Second, methods and techniques for management of distributed simulation systems, which includes configuration, deployment, execution and post-analysis, are needed to facilitate use of M&S by various stakeholders within an organisation. Finally, robustness of simulation system executions must be satisfied to enable response in a timely fashion and to increase the trustworthiness of M&S as a tool in general.

Current developments in Grid and Semantic Web technologies bring new opportunities for utilization of M&S in the way proposed above. At a general level, Grid technologies provide for efficient sharing of resources within and between organisations, which includes sharing of applications, data and computing capacity. Semantic Web technologies present the opportunity to structure and formalize information and knowledge in a common machine interpretable form, to facilitate interoperability at the syntactic and semantic levels. Recent developments target the intersection of these technology areas, creating the foundation for what is called the Semantic Grid. The Semantic Grid will support true reuse of data, applications and knowledge, and provide for seamless interoperability and automation.

Today, technologies and standards emerge from both the Grid and Semantic Web communities, which could facilitate implementation of an execution environment for distributed simulations. The Open Grid Services Architecture (OGSA) provides a Web Service-based framework of management services for developing Grid applications. OGSA is based on the Open Grid Services Infrastructure (OGSI), which provides standard Grid service interfaces with mechanisms for discovery, dynamic service creation, lifetime management and notification etc. There are several implementations of OGSI and OGSA available of which the most well-known is the Globus Toolkit. The use of Semantic Web languages such as RDF and OWL is gaining momentum and work on standardising these technologies is carried out within W3C forum. Through the increased interest in these technologies a wide range of tools, both academic and commercial, are emerging. This includes ontology editors, inference engines, and general APIs for management of RDF and OWL data.

This report presents the design of an execution environment for distributed simulations. This includes design of principal services required to leverage the expected functionality, but also a model (ontology) for description of assets available within the environment. The execution environment comprises four main components; the *computing services* executing parts of a simulations system. To enable lookup of available assets within the environment (simulation components, data etc.) a *repository service* is used. A *simulation infrastructure plug-in* enables management of simulation executions through the tools of a *simulation engineer workbench*, which represents the front-end of the execution environment.

In the implementation of the suggested design, we believe that Grid and Semantic Web technologies are crucial. The analysis indicates that application of these technologies, in the context of the execution environment, is able to cope with, if not all, at least the majority of the requirements enforced. As a next step, it is desirable to implement a “proof-of-concept”

system of the proposed design, but also to initiate research that will facilitate realization of the execution environment within a future NBD framework.

References

(1) Berners-Lee T, Hendler J og Lassila O (2001): The Semantic Web, *Scientific American*, **284**, 34-43.

(2) Bononi L, D'Angelo G og Donatiello L (2003): HLA-based Adaptive Distributed Simulation of Wireless Mobile Systems, in *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, San Diego.

(3) Brickley D og Guha R V (2004): RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

(4) Cai W, Turner S og Zhao H (2002): A Load Management System for Running HLA-based Distributed Simulations over the Grid, in *Proceedings of the 2002 IEEE International Symposium on Distributed Simulation and Real Time Applications.*

(5) Connolly D, van Harmelen F, Horrocks I, McGuinness D L, Patel-Schneider P F og Stein L A (2001): DAML+OIL (March 2001) Reference Description, W3C. http://www.w3.org/TR/2001/NOTE-daml+oil-reference-20011218.

(6) Eklöf M, Ayani R og Moradi F (2005): A Framework for Fault-tolerance in HLA-based Distributed Simulations, in *Proceedings of the 2005 Winter Simulation Conference*.

(7) Eklöf M, Sparf M og Moradi F (2004): Peer-to-peer-based resource management in support of HLA based simulations, *Simulation*, **80**, 181-190.

(8) Foster I (2005): A Globus Primer, Or Everything You Wanted to Know about Globus, but Were Afraid To Ask—Describing Globus Toolkit Version 4 (draft). http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf.

(9) Foster I, Berry D, Djaoui A, Grimshaw A, Horn B, Kishimoto H, Maciel F, Savva A, Siebenlist F, Subramaniam R, Treadwell J og Von Reich J (2004): The Open Grid Services Architecture, Version 1.0. http://www.ggf.org/documents/Drafts/draft-ggf-ogsa-spec.pdf.

(10) Foster I, Kesselman C (2003): The GRID 2, Blueprint for a New Computing Infrastructure, 2nd utgave, Elsevier, Amsterdam.

(11) Foster I, Kesselman C, Nick J og Tuecke S (2002): Grid Services for Distributed System Integration, *Computer*, **35**, 6. http://www.globus.org/alliance/publications/papers/ieee-cs-2.pdf.

(12) Gagnes T (2004): A survey of the current state of the semantic web, FFI/NOTAT-2004/03985, Forsvarets forskningsinstitutt.

(13) ITU (2000): X.509: Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks, X.509 (03/00), International Telecommunication Union.

(14) Lüthi J og Großmann S (2001): The resource sharing system: Dynamic federate mapping for HLA-based distributed simulation, in *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, Lake Arrowhead, California, s. 91-98.

(15) Manola F og Miller E (2004): RDF Primer. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

(16) Nagappan R, Skoczylas R, Sriganesh R P (2003): Developing Java Web services: architecting and developing secure Web services using Java, Wiley, Indianapolis, Ind.

(17) OMG (2005): Unified Modeling Language: Superstructure. Version 2.0, Object Management Group.

(18) Ort E (2005): Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools, *Sun Developer Network Technical Article*. http://java.sun.com/developer/technicalArticles/WebServices/soa2/SOATerms.html#soawhy.

(19) Roure D D (2005): Semantic Grid Vision. http://www.semanticgrid.org/vision.html.

(20) Tanenbaum A S, van Steen M (2002): Distributed Systems: Principles and Paradigms, Vrije University, Amsterdam, The Netherlands.

(21) Tuecke S, Czajkowski K, Foster I, Frey J, Graham S, Kesselman C, Maquire T, Sandholm T, Snelling D og Vanderbild P (2003): Open Grid Services Infrastructure (OGSI) Version 1.0. http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf.

(22) Zhang L-J, Chung J-Y og Zhou Q (2005): Developing Grid computing applications, *IBM Developerworks*. http://www-128.ibm.com/developerworks/grid/library/gr-grid1/.