UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

João Gabriel da Cunha Schittler

UMA METODOLOGIA PARA DESENVOLVER SIMULAÇÕES DISTRIBUÍDAS HLA USANDO DSEEP GUIADO A MODELOS COM OPM E UML

Santa Maria, RS
2024

João Gabriel da Cunha Schittler

UMA METODOLOGIA PARA DESENVOLVER SIMULAÇÕES DISTRIBUÍDAS HLA USANDO DSEEP GUIADO A MODELOS COM OPM E UML

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

Orientador: Prof. Raul Ceretta Nunes

Santa Maria, RS
2024

João Gabriel da Cunha Schittler

UMA METODOLOGIA PARA DESENVOLVER SIMULAÇÕES DISTRIBUÍDAS HLA USANDO DSEEP GUIADO A MODELOS COM OPM E UML

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

Aprovado em 19 de julho de 2024:

Raul Ceretta Nunes, Dr. (UFSM)
(Presidente/Orientador)

Lisandra Manzoni Fontoura, Dr. (UFSM)

Jürgen Znotka, Dr. (WHG) (videoconferência)

Santa Maria, RS
2024

RESUMO

UMA METODOLOGIA PARA DESENVOLVER SIMULAÇÕES DISTRIBUÍDAS HLA USANDO DSEEP GUIADO A MODELOS COM OPM E UML

AUTOR: João Gabriel da Cunha Schittler
Orientador: Raul Ceretta Nunes

Simulações distribuídas visam recriar algum comportamento real através de redes de simuladores independentes, muitas vezes geograficamente distribuídos. Uma abordagem amplamente usada para desenvolver simulações distribuídas é adotar algum padrão pré estabelecido de comunicação entre simuladores, como o *High-Level Architecture* (HLA) (IEEE, 2010). Porém, o desenvolvimento de código fonte pode ser complexo e sujeito a erros, motivando o uso de geradores de código fonte. A arquitetura orientada a modelos (MDA) (OMG, 2023) tem sido explorada na geração de modelos bem definidos a partir de modelos de alto nível de abstração, potencializando a geração eficiente de código fonte. Porém, a etapa de modelagem conceitual inicial permanece sendo um desafio que requer especial atenção aos objetivos da simulação e à construção de um modelo conceitual dos elementos chave da simulação. Este trabalho, centrado em desenvolvimento alinhado ao padrão HLA, apresenta uma proposta de exploração da *Object-Process Methodology* (OPM) (DORI, 2002) em uma metodologia de desenvolvimento de simulações distribuídas com uma etapa de modelagem conceitual natural, compreensível à *stakeholders* e que garanta uma transição automatizada de especificações de alto nível (*human-friendly*) para código fonte HLA da simulação especificada. A metodologia proposta está alinhada ao Distributed Simulation Engineering and Execution Process (DSEEP) e mantém opção de emprego de UML como linguagem de modelagem. Este trabalho contém também uma série de experimentos que visam validar as contribuições, mostrando que a metodologia desenvolvida consegue atingir seus objetivos e que seu uso é apropriado para o desenvolvimento de simulações distribuídas.

Palavras-chave: Simulações Distribuídas. Arquitetura Orientada a Modelos. Transformações entre Modelos. Metodologia de Objetos e Processos.

ABSTRACT

A METHODOLOGY TO DEVELOP HLA DISTRIBUTED SIMULATIONS USING MODEL-DRIVEN DSEEP WITH OPM AND UML

AUTHOR: João Gabriel da Cunha Schittler
ADVISOR: Raul Ceretta Nunes

Distributed simulations aim to recreate some real behavior using computer networks; they are often geographically distributed as well. One widely used approach to develop distributed simulations is to use a pre-established standard for network communication between simulators; one such standard is High-Level Architecture (HLA) (IEEE, 2010). However, the development of source code for such standards is often complex and error-prone, making code generators a good option for its development. Model Driven Architectures (MDA) (OMG, 2023) have been explored to generate efficient source code based on high-level conceptual models. Nevertheless, using MDA for code generation creates another challenge: the specification and development of accurate high-level models of the simulation, which can be quite difficult if the project stakeholders cannot properly understand the models. This work, centered in HLA project development, explores the use of Object-Process Methodology (DORI, 2002) in a methodology for developing distributed simulations with a conceptual modeling step that is natural, understandable to stakeholders and that guarantees automatic transformation from high-level (human-friendly) specifications into HLA source code for the specified simulation. The proposed methodology is aligned to the Distributed Simulation Engineering and Execution Process (DSEEP) and maintains the possibility of using UML as a modeling language. This work also contains a series of experiments that aim to validate the contributions, showcasing that the developed methodology can achieve its objectives and that its use is appropriate for developing distributed simulations.

Keywords: Distributed Simulation. Model Driven Architecture. Model to Model Transformation. Object-Process Methodology.

LIST OF FIGURES

Figure 1 – Example Object Process Diagram. 19
Figure 2 – Example Object Process Language Segment. 19
Figure 3 – OPCat UML generation from OPM model. 20
Figure 4 – Proposed Development Methodology 26
Figure 5 – OPD that implements RQ1 30
Figure 6 – Alpha Simulator OPD (RQ2 to RQ6). 31
Figure 7 – Beta Simulator OPD (RQ7 to RQ10). 32
Figure 8 – Delta Simulator OPD (RQ11, RQ12 and RQ13). 33
Figure 9 – Object Classes OPD 35
Figure 10 – Interaction Classes OPD 36
Figure 11 – Implementation of the Proposed Methodology 39
Figure 12 – Part of the OPCatUML Meta-Model 42
Figure 13 – HLA Extension of the OPCatUML meta-model 44
Figure 14 – Main OPD and OPL of the Subject Model. 55
Figure 15 – STat Centered OPD and OPL of the Subject Model. 56
Figure 16 – STat Publishing centered OPD and OPL of the Subject Model. 57
Figure 17 – SCon Centered OPD and OPL of the Subject Model. 58
Figure 18 – SCon Publishing centered OPD and OPL of the Subject Model. 59
Figure 19 – Distributed Simulation OPM Diagram. 62
Figure 20 – Alpha Simulator OPD. 62
Figure 21 – UML Class Diagram for the Simulation. 63
Figure 22 – UML Activity Diagram for publishing AirVehicle. 64
Figure 23 – UML Activity Diagram for publishing RefuelRequest/RefuelResponse. 65
Figure 24 – Beta Simulator OPD after the addition of the *HybridVehicle* Class. 67
Figure 25 – Part of the Object Classes OPD after the addition of the *HybridVehicle* Class. 68
Figure 26 – First Half of the Activity Diagram for *HybridVehicle*. 68
Figure 27 – Gamma Simulator OPD. 69
Figure 28 – Second Half of the GroundVehicle Activity Diagram after adding Gamma. 70
Figure 29 – Main screen of developed GUI Program. 71
Figure 30 – UI Button to Proposed Process equivalence. 72
Figure 31 – OPCat Environment and the integrated OPM-UML transformer. 73
Figure 32 – UML model after the OPCatUML Adjuster program. 73
Figure 33 – UML Model after the Annotation Process. 73
Figure 34 – Annotation Helper Screen of GUI Program. 74
Figure 35 – Code Generator UI after FOM insertion. 75

Figure 36 – Generated Code Connecting to Pitch RTI. 76
Figure 37 – Generated Code Connecting to MAK RTI. 77
Figure 38 – Publish/Subscribe ObjectClass test for generated code. 79
Figure 39 – Histogram of the total values of each derived metric. 82

LIST OF TABLES

TABLE 1 – OPD and OPL Representation of Structural Links. 18
TABLE 2 – OPD and OPL Representation of Procedural Links. 18
TABLE 3 – Comparative table between related works. 22
TABLE 4 – Transformation Information Source for Generating a FOM. 47
TABLE 5 – Prerequisites of the distributed simulation with *SCon* and *STat*. 52
TABLE 6 – Summarized requirements of the distributed simulation with *SCon* and *STat*. 53
TABLE 7 – Comparative frame between the scalability of the OPM and UML models. . 70
TABLE 8 – Halstead Derived Measures. 80
TABLE 9 – Halstead metrics for our code generator. 81
TABLE 10 – Halstead metrics for Pitch Developer Studio code generator. 81

LISTA DE SIGLAS

DS	Distributed Simulation
HLA	High-Level Architecture
FOM	Federate Object Model
DSEEP	Distributed Simulation Engineering and Execution Process
MDA	Model-Driven Architecture
PIM	Platform Independent Model
PSM	Platform Specific Model
OPM	Object-Process Methodology
OPL	Object-Process Language
OPD	Object-Process Diagram
UML	Unified Modeling Language

CONTENTS

1 INTRODUCTION 11
2 RESEARCH PROBLEM 14
3 BACKGROUND 15
3.1 HLA 15
3.2 DSEEP 16
3.3 MODEL DRIVEN ARCHITECTURE 16
3.4 OPM 17
3.4.1 OPCat 19
4 RELATED WORKS 21
5 OPM-UML BASED MODEL-DRIVEN DSEEP 24
5.1 PLANNING THE METHODOLOGY'S ACTIVITIES 24
5.2 PROPOSED METHODOLOGY 25
5.3 OPM MODELING 28
5.3.1 Why use OPM? 28
5.3.2 Modeling OPM from Requirements 29
5.3.3 OPM Modeling Review 37
6 AN IMPLEMENTATION FOR THE PROPOSED METHODOLOGY 39
6.1 DEVELOPING A PLATFORM INDEPENDENT MODEL USING OPM 40
6.2 TRANSFORMING THE OPM PIM INTO A UML PSM 40
6.2.1 Transforming the OPM PIM into a UML PIM 41
6.2.2 Transforming the UML PIM into a UML PSM 41
6.3 TRANSFORMING THE UML PSM INTO THE FOM 46
6.4 TRANSFORMING THE FOM INTO HLA SOURCE CODE. 49
7 VALIDATION 50
7.1 EXPERIMENT 1: MILITARY CASE STUDY 50
7.1.1 Objective Defining Meeting 50
7.1.2 Simulation Specification Meeting 52
7.1.3 OPM Model Development 54
7.1.4 OPM Model Review 59
7.2 EXPERIMENT 2: MODELING COMPARISONS BETWEEN OPM AND UML 60
7.3 EXPERIMENT 3: PROPOSED MDA METHODOLOGY APPLIED TO AN EXAMPLE 71
7.4 EXPERIMENT 4: ANALYSING AND TESTING GENERATED CODE 74
7.4.1 HLA Code Tests 75
7.4.2 HLA Code Metrics Analysis 79
8 CONCLUSIONS 83

BIBLIOGRAPHY 85

1 INTRODUCTION

In the realm of distributed simulations (DS), seamless interoperability among simulators is an inherent requirement (TOPÇU et al., 2016). In order to facilitate efficient and objective-focused simulator interoperability, the establishment of specific communication standards is imperative. One of the most prevalent standards for achieving this interoperability is the High-Level Architecture (HLA) (IEEE, 2010), which leverages the publish/subscribe architecture as its foundational framework.

Within the HLA framework, a distributed simulation is referred to as a "Federation," and each interconnected simulator assumes the role of a "Federate" within that Federation. HLA further mandates the utilization of a Federate Object Model (FOM) file to delineate the objects, interactions, and data types that will be exchanged over the network. The widespread adoption of HLA transcends various sectors, encompassing domains such as healthcare (PETTY; WINDYGA, 1999), military (LEE; YOO; JEONG, 2005), and space (CRUES et al., 2022).

There are two widely known challenges when developing HLA distributed simulations: i) Crafting an adequate and concise FOM file that aligns with the simulation's objectives and ii) Accurately implementing the simulator source code to conform to the established FOM, thereby ensuring effective utilization within the simulation environment (MöLLER; KARLSSON; LöFSTRAND, 2006; GRAHAM, 2007).

The Distributed Simulation Engineering and Execution Process (DSEEP) (IEEE, 2022) is a widely recognized development process for distributed simulations that helps HLA-based developing process. DSEEP defines clear steps to be followed to create a distributed simulation, from the definition of the simulation objectives and the conceptual modeling to the development and testing of the simulation.

In recent works, Bocciarelli (BOCCIARELLI, P. et al, 2019) and D'Ambrogio (DAMBROGIO et al., 2019) present development approaches for HLA simulations that unite the concepts of DSEEP and Model Driven Architecture (MDA) (OMG, 2023). MDA is a well-known approach for software development that transforms conceptual models into final code. These works use the DSEEP steps as a base and add automatic model transformations starting from the conceptual model of the simulation, transforming it into the simulation FOM and its source code. These approaches require that the initial conceptual model represent what is expected of the simulation. Ideally, this model should be easily obtainable and understood by the organizers of the project and its stakeholders, even if they don't have experience with modeling languages so that the requirements of the project are well interpreted by both parties, avoiding future problems relating to them (MALL, 2018).

Bocciarelli and D'Ambrogio used SysML (SysML Org, 2022) as their sole modeling language. Transforming a conceptual SysML model into the simulation FOM and the

source code. SysML was created using UML as a base, focusing more on system modeling. However, SysML may not be the most appropriate language for the first conceptual model due to needing multiple diagram types to properly represent the system's structure and behavior. There are many known challenges in User Comprehension and Complexity associated with using multiple diagram types (ONG; JABBARI, 2019). Furthermore, the high information density required to fill the many diagrams that compose a SysML model may not be well defined at the initial modeling (ŠENKỲR; KROHA, 2021). Therefore, using a conceptual modeling language that requires less information to describe the simulation, is more readable, and is easily obtainable can be highly beneficial in the first steps of software development approaches (BASNET et al., 2020).

Even with an established FOM, implementing the simulation remains a challenge due to the high complexity of the source code that manages low-level communication between federates. This complexity makes manual programming of the code highly error-prone. (MöLLER; KARLSSON; LöFSTRAND, 2006; GRAHAM, 2007). Recently, this problem has been circumvented with the use of model-oriented solutions that use abstract models to automatically generate the more complex parts of the simulation's code (BOCCIARELLI, P. et al, 2019; DAMBROGIO et al., 2019), which indicates a tendency for the automatic generation of FOM files as well. It is important to note that the FOM can be interpreted as an intermediate model between a high-level conceptual model and the source code. Some examples of HLA code generators are: *Pitch Development Studio* (Pitch Technologies, 2022), *MAK VR-Link* (MAK Technologies, 2023), Bocciareli's (BOCCIARELLI, P. et al, 2019).

This work seeks for a distributed simulation development methodology based on DSEEP, incorporating a modified conceptual modeling step designed to reduce modeling errors for distributed simulations without losing the detailed precision needed for code generation.

This methodology uses Object-Process Methodology (OPM) (DORI, 2002), a modeling language that facilitates the creation of high-level system models, in conjunction with UML to make a conceptual modeling process that is more readable for non-technical people while still containing all of the information required to describe the system.

With this methodology, after the acquisition of the simulationt's requirements, an OPM model is developed, containing complete diagrams for the abstract views of the system and partially complete diagrams for less abstract parts of the system (depending on the amount of information described in the requirements). This OPM model is reviewed by the stakeholders and the missing information is filled. The finished model is then transformed into a UML class diagram that goes through a series of automatic transformations until it becomes the simulation's FOM and HLA source code.

In summary, the contributions of this work are as such:

- A proposition of a new HLA distributed simulation development methodology that con-

forms to DSEEP and uses OPM and UML as the modeling languages. The methodology incorporates a new conceptual modeling step that is more readable for people with less knowledge of modeling languages, which can be the case for project stakeholders. With greater model readability, greater is the understanding of the readers in relation to the simulation's conceptual vision, increasing the alignment between what the stakeholders want the project to be and what the developers interpreted, reducing the risks of changes in requirements as the project is developed.

- A complete implementation of such HLA simulation development methodology from high-level modeling step to source-code generation. The implementation explores QVT-Operacional (OMG, 2024) as the language used for model-to-model transformations, and the StringTemplates template engine for code generation.

The rest of this work is organized as follows: Section 3 presents the concepts required to understand this work's proposition; Chapter 4 discusses and compares related works; Chapter 5 provides an in-depth explanation of this work's proposed methodology; Chapter 6 presents one implementation of the discussed methodology; Chapter 7 features the conducted experiments and their results; Finally, Chapter 8 provides this work's conclusions.

2 RESEARCH PROBLEM

Developing HLA simulations comes with its own set of challenges. For example, due to the special rule for encoding and decoding each data type and the need to know the correct RTI API calls, even senior programmers can make mistakes. Thus, the implementation of the simulation source code to align with the HLA standard can be quite error-prone and complex. To mitigate this issue, many efforts have already been made, from the development of auxiliary libraries that obfuscate the encoding/decoding process (MöLLER; KARLSSON; LöFSTRAND, 2006) to automatic HLA code generators that use abstract models to generate the low-level encoding/decoding and RTI API calls from abstract models (Pitch Technologies, 2022).

Using automatic code generators to create HLA code requires at least one form of abstract model to be used as a base from which the code will be generated. That model can be, for example, the federation's FOM (SANTOS; NUNES, 2022) or a set of SysML diagrams (BOCCIARELLI et al., 2019). However, this approach does not completely solve the issue of developing HLA simulations because the abstract model definition task can often be a complex task. Whenever the abstract model does not produce the expected requirements of the simulation, it will delay the conclusion of the software project (MALL, 2018).

Some MDA-based methodologies to automatically transform conceptual models into the federation's source code are proposed (BOCCIARELLI, P. et al, 2019; DAMBROGIO et al., 2019). In these methodologies, the initial system model is made with the SysML language, a variant of UML with a greater focus on system modeling. The SysML language requires multiple diagram types to properly convey the system's structure and behavior. However, using modeling languages with multiple diagram types is known to come with its own set of challenges, such as diagram inconsistencies and increased difficulty in user comprehension (ONG; JABBARI, 2019).

The mentioned works have a greater focus on the process of transforming conceptual models onto the federation source code using automatic transformations instead of on the creation of the initial conceptual model required to start the modeling process. As such, the problem of enhancing the collaboration between the stakeholders and project developers of the to-be-developed distributed simulation via creating a conceptual model is partially unsolved and is this work's main focus.

3 BACKGROUND

This chapter will explain some of the underlying knowledge required to fully understand the work that has been developed. Section 3.1 explains the various concepts behind High-Level Architecture (HLA). Section 3.2 provides a general overview of DSEEP. Finally, Section 3.4 explains the Object-Process Methodology modeling language alongside the tool used to develop models in this language (OPCat).

3.1 HLA

High-Level Architecture (HLA) is a standardized solution with the objective of allowing interoperability between different simulators with one common architecture and standardized interfaces (IEEE, 2010). HLA provides developers with a publish-subscribe framework to structure and model simulation systems to guarantee interoperability with other simulators. HLA defines the distributed simulation as a 'Federation', composed of "Federates" (simulators) that communicate with each other via a run-time infrastructure (RTI). The following documents describe the HLA specification: "HLA Framework and Rules Specification" (IEEE, 2010), "HLA Object Model Template" (IEEE, 2010), and "HLA Federate Interface Specification" (IEEE, 2010).

HLA operates with `ObjectClasses`, objects that can have many attributes and persist within the simulation, and `InteractionClasses`, one-time events that also have their attributes but do not persist in the simulation. If a new federate joins the simulation, they will be able to know all the objects that are "existing" there, but they will only be able to know of interactions that happen after they join.

HLA defines what data can be transferred between federates in a federation agreement named Federate Object Model (FOM) file. The FOM file defines all the `ObjectClasses`, `InteractionClasses`, attributes and Datatypes that are able to be transferred in the federation. One federate participating in the federation does not need to implement the entire FOM, only the parts that matter for its simulation purpose. The subset of the FOM implemented by a federate is called Simulation Object Model (SOM).

When developing HLA distributed simulations, the FOM design needs to be aligned with the simulation's goals and the simulation code must be correct and efficient. To facilitate this, the creation of conceptual models and the exploration of model transformations until the automatic code generation has been explored on HLA-based simulation development.

3.2 DSEEP

The Distributed Simulation Engineering and Execution Process (DSEEP) (IEEE, 2022) is a widely known and utilized development process to construct distributed simulations. It consists of the following seven steps:

1 Define Simulation Environment Objectives.

2 Perform Conceptual Analysis.

3 Design Simulation Environment.

4 Develop Simulation Environment.

5 Integrate and Test Simulation Environment.

6 Execute Simulation.

7 Analyse Data and Evaluate Results.

These steps provide a clear guideline of how one should aim to develop a distributed simulation. DSEEP is normally used alongside interoperability technologies such as HLA, Distributed Interactive Simulation (DIS) (IEEE, 2015) or Test and Training Enabling Architecture (TENA) (TENA Software Development Activity, 2023) when developing distributed simulations.

3.3 MODEL DRIVEN ARCHITECTURE

Model Drive Architecture (MDA) is a software development standard proposed by OMG (OMG, 2023) that presents a software development guideline using different abstract representations (models) of the final software. MDA defines three types of abstract models:

i) The Computation-independent Model (CIM) conceptually defines the system without specifying computational aspects, such as the separations of functionalities in different sub-systems;

ii) The Platform-independent model (PIM) also conceptually defines the system, taking into account the computational aspects involved, providing with a view that shows the general structure of the system. However, the PIM model does not show the specific technologies used in different parts of the system. For example, it may show that a distributed simulation will use a publish-subscribe architecture but will not specify that it will be HLA.

iii) The Platform-specific model (PSM) defines the system, taking into account the computational aspects, and specifies the different technologies used within the system.

MDA indicates that the software should start as an abstract view of itself and be subjected to transformations through the defined abstract models until it is transformed into the final software.

3.4 OPM

Object-Process Methodology (OPM) (DORI, 2002) is a systems modeling language specified as ISO/PAS 19450. The modeling methodology is supported by two modeling interfaces: Object Process Diagram (OPD) and Object-Process language (OPL). An OPD is composed of objects, processes, states, and links between them. An OPL segment defines a set of keywords that can be used to represent the various types of relations objects and interactions can have. An OPL text can be automatically transformed to an OPD and vice versa. Thus, OPM can represent the structure and behavior of a system with either of its modeling interfaces.

Object Process Diagrams allow for a graphical representation of the OPM elements. Objects are represented by rectangles, processes by ellipses, and States by rectangles with rounded edges inside an object. These elements can be connected by two classes of links: Structural Links and Procedural Links. OPDs can have different levels of detail on certain objects and processes. An object in an OPD can be composed of several object-s/processes and links in another OPD. This enables the model developer to create different views of the same system, with many OPDs detailing different parts of the system and others showcasing it on a more abstract level.

Object Process Language segments allow for a textual representation of the OPM elements, represented by a distinct set of English keywords. When modeling in OPM, an OPL segment always has an equivalent OPD and vice-versa. Thus, parts of the system can be modeled in OPL segments and others with OPDs based on which is easier (for the person modeling) to express the desired structure and behavior. Hereafter in this text, OPL segments will be represented in this way: object will be represented using this Color/Font Combination and processes will be represented using this Color/Font Combination. Lastly, the tag name of tagged associations will be represented using *italics* to differentiate it from the native keywords of OPL.

Table 1 has the specification of all structural links OPM provides, with their meaning and OPD/OPL representation. These links indicate how elements relate to one another in structure. With the exception of *Exhibition*, the structural links can only connect an object to other objects and a process to other processes.

Table 2 contains some of the procedural links OPM provides. These links indicate how elements relate to one another in behavior. All of the listed procedural links may only occur between objects and processes. The consumption and results links use the same

Structural Link	OPD	OPL	Description
Aggregation/Participation		A consists of B and C.	B and C are parts of the whole A.
Exhibition/Characterization		A exhibits B and C.	B and C are attributes of A.
Generalization/Specification		B and C are As.	B and C are of type A.
Classification/Instantiation		B and C are instances of A.	B and C are unique objects of class A.
Tagged Association	tag	A *TagName* B	A relates to B according to what is written as the tag name.

Table 1 – OPD and OPL Representation of Structural Links.

graphical representation. What differentiates them is to which element the arrow points; if it points to a process, it is a consumption link; if it points to an object, it is a result link. The full list of the building blocks can be found in the OPM specification book (DORI, 2002).

Procedural Link	OPD	OPL	Description
Agent		Obj handles Proc.	The object (human/human operated) is responsible for the operation of the process.
Instrument		Proc requires Obj.	The process needs the object in order to occur. The object is not human nor is it human operated.
Consumption		Proc consumes Obj.	The process completely uses the object during its operation.
Result		Proc yields Obj.	The process creates a new object during its operation.
Effect		Proc affects Obj.	The process affects the object in an unspecified manner.

Table 2 – OPD and OPL Representation of Procedural Links.

Figure 1 shows one OPD. The green rectangles are objects, and the blue ellipsis is a process. Some of the structural links present in this model are Aggregation-Participation between Report and Header, Footer and Line, indicating that a header footer and line make up a Report; Exhibition-Characterization between Report and ID, indicating that reports have identifications; And Generalization-Specialization between Report and Travel Report,

indicating that a travel report is a type of report. There are also procedural links, such as the instrument link between Header/Footer/Line and Printing, indicating that the printing process requires all of those objects, And the result/consumption link between Printing and Printed Line, indicating that the printing process results in a printed line.

Figure 2 showcases the equivalent OPL representation of the OPD from Figure 1. When this model was made, it could have used either of the representations. The various links between model elements are present in this OPL segment as sentences, following the specification from tables 1 and 2. One such example is the sentence “Printing requires Line, Footer and Header,” which translates the meaning of the instrument link between the objects “Header,” “Footer,” and “Line” with the process “Printing.” In this example, a reader does not need to memorize the meaning of the link connecting “Header” and “Printing”; they can understand it by reading the OPL translation.

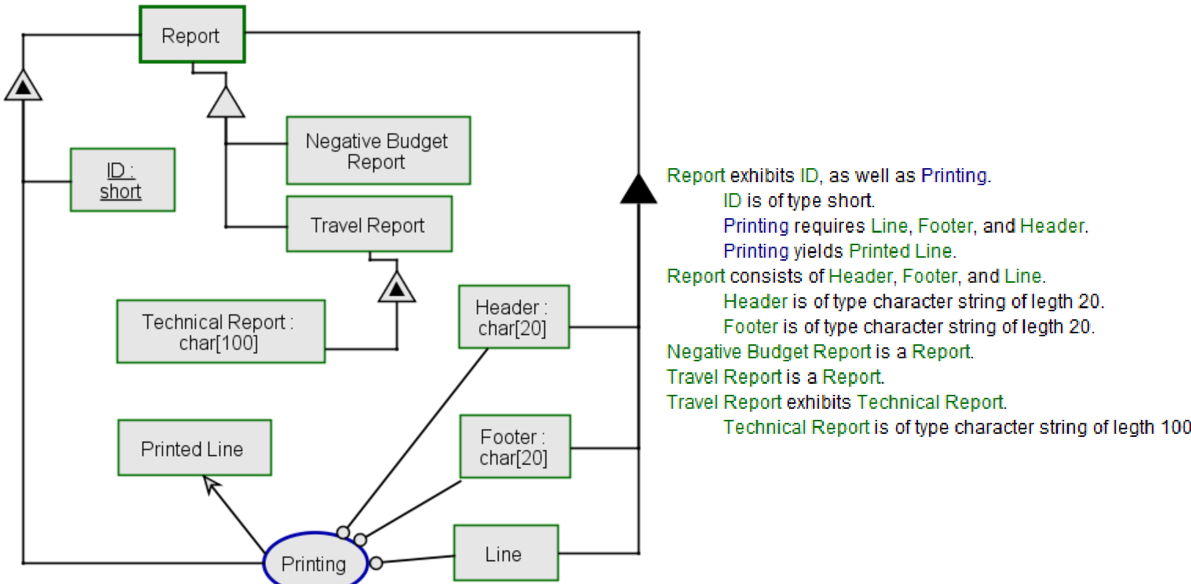


Figure 1 – Example Object Process Diagram.

Figure 2 – Example Object Process Language Segment.

3.4.1 OPCat

The OPCat tool (DORI et al., 2010), utilized for developing OPM models, has a graphical interface from which the user can create and manage OPDs. The OPD being developed is automatically translated to OPL in real time as it is being built. OPCat also allows the user to prepare and execute simulations with OPM models; these simulations ’activate’ certain objects or processes and, following the model’s links, ’activate’ other parts of the system, showcasing the modeled behavior.

OPCat also offers the option to convert an OPM model into a UML one. This feature

allows the user to select what types of UML diagrams to include in the converted model. Figure 3 shows the different types of UML generation options. They include Class Diagrams, Sequence Diagrams (for specific processes), and Use Case Diagrams, among others. It is important to note that not all OPM elements can be properly converted into UML elements. For the purposes of this work, in which we only want UML class diagrams, this conversion feature is enough.

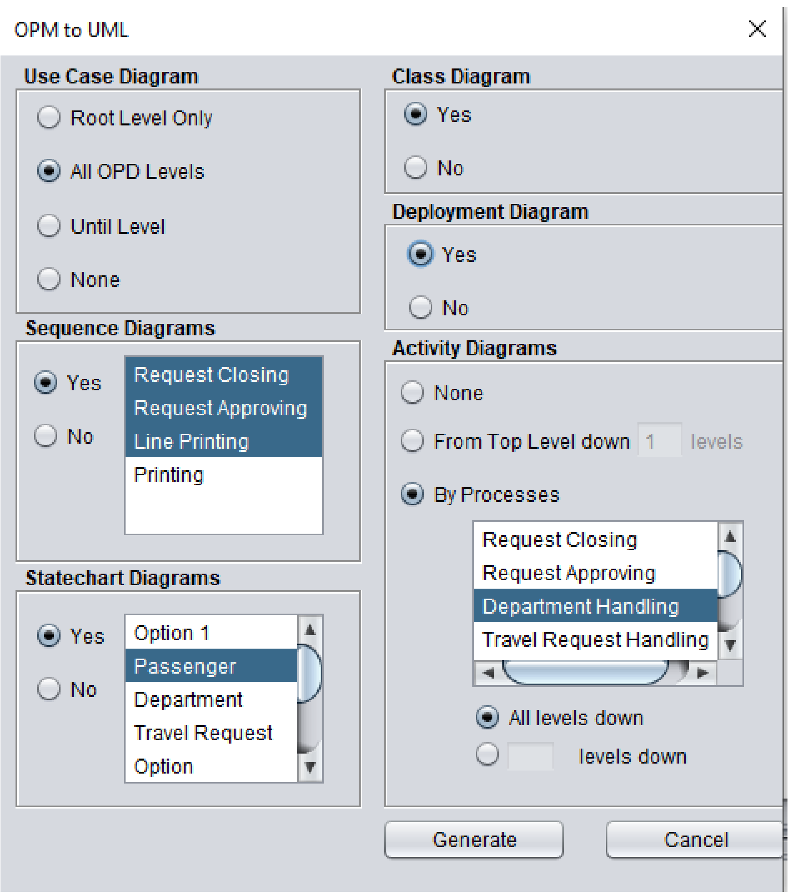


Figure 3 – OPCat UML generation from OPM model.

4 RELATED WORKS

This chapter presents other works that develop similar approaches to this work and explains the key differences.

This work extends the work developed by Santos and Nunes (SANTOS; NUNES, 2022), expanding the scope of the methodology to use the DSEEP steps for a more complete simulation development approach. The expanded methodology was based on Bocciarelli's work (BOCCIARELLI, P. et al, 2019).

Using the FOM file (formatted with XML) as an input, Santos and Nunes (SANTOS; NUNES, 2022) propose an HLA source code generator that generates low-level code, mainly encoders, decoders, and class managers. The present work expands this code generator to use a template engine for more modular and flexible code generation, adds support to many RTI services, such as Time Management and Ownership Management, and integrates this code generator in an MDA approach to generate code based on a FOM file that was made from model to model transformations starting from a conceptual model.

In (BOCCIARELLI, P. et al, 2019), the authors present a distributed simulation development approach that unites DSEEP and MDA, called Model-Driven DSEEP (MoDSEEP). In practical terms, MoDSEEP adds automatic transformations between the models involved in the DSEEP steps until becoming the simulation source code in the 'Develop Simulation Environment' step. This work's first conceptual model is made in SysML following the PIM model format; then, it is annotated with HLA-specific terms, like "Federation," "ObjectClass," etc., becoming a PSM. The PSM is then transformed into the simulation's FOM and source code. Our work differs from (BOCCIARELLI, P. et al, 2019) in key parts, such as the choice of modeling language(s) and the number of model transformations. Our approach explores two modeling languages, the OPM for the initial modeling as high-level PIM and UML for representing the system in more detail, including adding HLA concepts. Since we are using more modeling languages, we also adapted the number of model transformations from Bocciarelli's definition of MoDSEEP by adding one more model-to-model transformation from OPM to UML. Another difference is how to handle UML/SysML profiles. Bocciarelli's approach used profiles to annotate HLA concepts into the models. Our approach uses Ecore meta-models instead of profiles to validate and annotate our models.

In (DERE; GÖRÜR; OĞUZTÜZÜN, 2020), the authors propose a framework for developing agent-based simulations using SysML as the conceptual modeling language, more specifically, the Block Definition Diagrams (BDD) and Internal Block Diagrams (IBD) alongside the Acceleo tool for code generation. SysML was chosen due to its popularity and its purpose of being specialized in systems modeling. However, unlike our work, it focuses on generating non-distributed simulation's source code and solely uses SysML, justifying its use due to its popularity and adequacy in describing systems.

In (ARRONATEGUI; BAÑARES; COLOM, 2020), the authors propose a Model Driven Engineering (MDE) method to develop distributed simulations for healthcare systems. They use discrete event system (DES) type simulations to determine disease propagation, with parameters such as available resources (human and non-human), transport systems, and individual behaviors. The development method starts with the creation of UML Activity diagrams, which are then transformed into Petri net hierarchical models. Afterward, the Petri net models go through structural analysis and are finally transformed into source code. The work differs from ours mainly for having a greater focus on the healthcare area and the development of DES-type simulations, while our methodology focuses on making distributed simulations for any area. One interesting parallel between these works is that, much like we use OPM as a more abstract, more readable model for people with expertise in other areas, the aforementioned work uses UML Activity diagrams because they have structural similarities to Petri Net models (VLADIMIRIOVICH; ALEXANDROVICH; OLEGOVICH, 2015) and is assumed easier for doctors to understand, helping them better define the simulation's behavior compared to working directly with the Petri Net models.

Table 3 summarizes some key characteristics of our and the related works.

Work	HLA	Domain	Modeling Languages	Generates
(SANTOS; NUNES, 2022)	✓	General Use	HLA FOM	C++ code
(DERE; GÖRÜR; OĞUZTÜZÜN, 2020)		Agent-Based Simulations	SysML (BDD,IBD)	C++ code
(ARRONATEGUI; BAÑARES; COLOM, 2020)	✓	DES Simulations	UML(Activity) Petri Net	Java code
(BOCCIARELLI, P. et al, 2019)	✓	General Use	SysML	Java code
This work	✓	General Use	OPM, UML(Class)	C++* code

Table 3 – Comparative table between related works.

In Table 3: The second column characteristic checks if the work generates HLA source code; The third column has the simulation domain of each work. Meaning for what types of simulations the works are best used for; The fourth column tells what modeling languages the work uses for their Model Driven Engineering processes. Using more than one modeling language gives the development team more flexibility on how they can present conceptual models for eventual reviews with people who are not familiar with modeling languages. The fifth column shows what are the languages the resulting code generator creates for each work.

Analyzing table 3. All but the second work creates HLA code. The first and the latter two works are not domain-specific code generators, being able to generate HLA code for any type of simulation, whereas the second and third were made with a specific domain in mind. The third and the last works use multiple modeling languages. All works

either generate Java or C++ code. One thing to note is that since our code generator is template-based, generating code for different languages is just a matter of creating a new set of templates. Santos and Nunes's work (SANTOS; NUNES, 2022) generates domain-independent HLA C++ code. However, it uses just the FOM file to model the simulation, providing very little abstraction from which non-technical people can understand the model before it is implemented. Dere's work (DERE; GÖRÜR; OĞUZTÜZÜN, 2020) does not generate HLA code, so guaranteeing compatibility with different agent-based simulators may be an arduous task. Arronategui's work (DERE; GÖRÜR; OĞUZTÜZÜN, 2020) has a very specific domain, that being DES simulations for the medical field. It uses multiple modeling languages to help doctors with lesser knowledge of that subject correctly model behaviors using UML Activity Diagrams instead of the less abstract Petri Net models. Bocciarelli's work (BOCCIARELLI, P. et al, 2019) has the advantage of generating simulations for general use but is restricted to only using SysML as a modeling language, which can be quite difficult to understand for nontechnical people (ONG; JABBARI, 2019), making the development process have more risk requirement changes. Finally, our work has the benefit of generating simulations for general use while having multiple modeling languages to help with mutual understanding with stakeholders. On top of that, it has a template-based code generator, allowing for greater flexibility in case the programming language of the low-level HLA code needs to change.

5 OPM-UML BASED MODEL-DRIVEN DSEEP

This chapter presents a new methodology proposed for developing HLA distributed simulations. The methodology uses the DSEEP steps and adopts Model-Driven Architecture, and it explores OPM and UML in the conceptual modeling phase. The chapter is divided into three sections as follows. Section 5.1 gives greater insight into how the methodology came to be. Section 5.2 provides an overview of the methodology as a whole. Finally, Section 5.3 further explains the use of OPM.

5.1 PLANNING THE METHODOLOGY'S ACTIVITIES

As explained in Chapter 2, the problem of coordinating the stakeholders' and project developers' understanding of the distributed simulation to be developed via creating a conceptual model is still partially unsolved. Thus, a methodology needed to be created to attempt to solve this issue.

The overall objective of a new methodology is to help develop HLA distributed simulations with automatic transformations. Therefore, a model-to-text transformation in which an abstract model is transformed into code is necessary. For the transformation to generate usable code, this model needs to contain all of the HLA relevant information while still being readable to non-technical people. To circumvent this necessity, it was decided that an MDA approach was convenient so that an initial (PIM) model could be developed to be human-friendly and then gradually transformed into a more granular (PSM) model with all the information needed to become code.

While researching appropriate modeling languages to make the initial model, two main options were found: the industry standard UML (or a variation of it, like SysML) and OPM. After thorough consideration and analysis, such as the evaluation of the works (BOCCIARELLI, P. et al, 2019),(CERQUEIRA; AMBROSIO; KIRNER, 2016), (HAUSE; DAY, 2019), (BASNET et al., 2020) and (ONG; JABBARI, 2019). OPM was elected as the modeling language from which the PIM would be developed. UML would be used to transform the model into a PSM, and the FOM file would be transformed into code. Therefore, this methodology would entail an initial OPM modeling (which would be derived from the stakeholders' requirements), a model-to-model transformation into UML, a model annotation to specify HLA information, a model-to-model transformation into the FOM, and finally, a model-to-text transformation into code.

The decision to use OPM is further explained in Section 5.3.1. UML, specifically UML class diagrams, was chosen as an intermediary language because it would make the annotation process easier to perform as it would involve simply changing the class type to

match the object's HLA-specific role. The decision to include UML also allows development teams that prefer to use UML to communicate concepts with their stakeholders to still use our proposed methodology by skipping the initial OPM modeling.

The decision to use the FOM for code generation stems from the existence of many HLA code generators that use the FOM file as the input, such as *Pitch Development Studio* (Pitch Technologies, 2022) and *MAK VR-Link* (MAK Technologies, 2023).

Lastly, since our methodology has a greater focus on conceptual modeling, two review activities were also included to allow for model modifications before code development started. The first review happens after the initial PIM is developed (be it OPM or UML); it is attended by development team leads and stakeholders to discuss how the requirements were mapped into the conceptual model. The second review happens after the model is fully turned into a PSM; it is held internally by the development team to confirm that the annotation process was performed correctly.

With all of these planned activities, they were linked together by intermediary output documents, and thus, the proposed methodology was built. It can be seen in Figure 4 and will be explained in more detail in the next section.

5.2 PROPOSED METHODOLOGY

This section presents the proposed methodology providing an overview of all the methodology activities. Figure 4 contains all of these activities while also dividing them into the first few steps of DSEEP. This methodology does not specify activities in all DSEEP steps because it is focused on proposing a better conceptual modeling step.

The first two activities of the methodology are the "Objective Defining Meeting" and the "Simulation Specification Meeting". Both of these meetings work towards establishing the Simulation's Requirements but from different perspectives. The first meeting aims to create the high-level specification of the simulation, while the second meeting defines all of the lower-level requirements of the simulation. The high-level meeting is attended by the stakeholders and development team leaders. The low-level meeting is attended by more members of the development team alongside people affiliated with the stakeholders who have more technical knowledge to be able to discuss lower-level details.

Once the meetings are concluded and a "Simulation's Requirements" document is finished, the "OPM Model Development" activity can occur. This activity will use the requirements document in order to create the "Simulation's OPM Model" that accurately describes all of the high-level and low-level requirements in OPM. The process developed in this activity is further explained in Section 5.3.

After an OPM model of the simulation is created, it undergoes the "Modeling Review" activity. This activity involves evaluating the OPM model against what is expected of

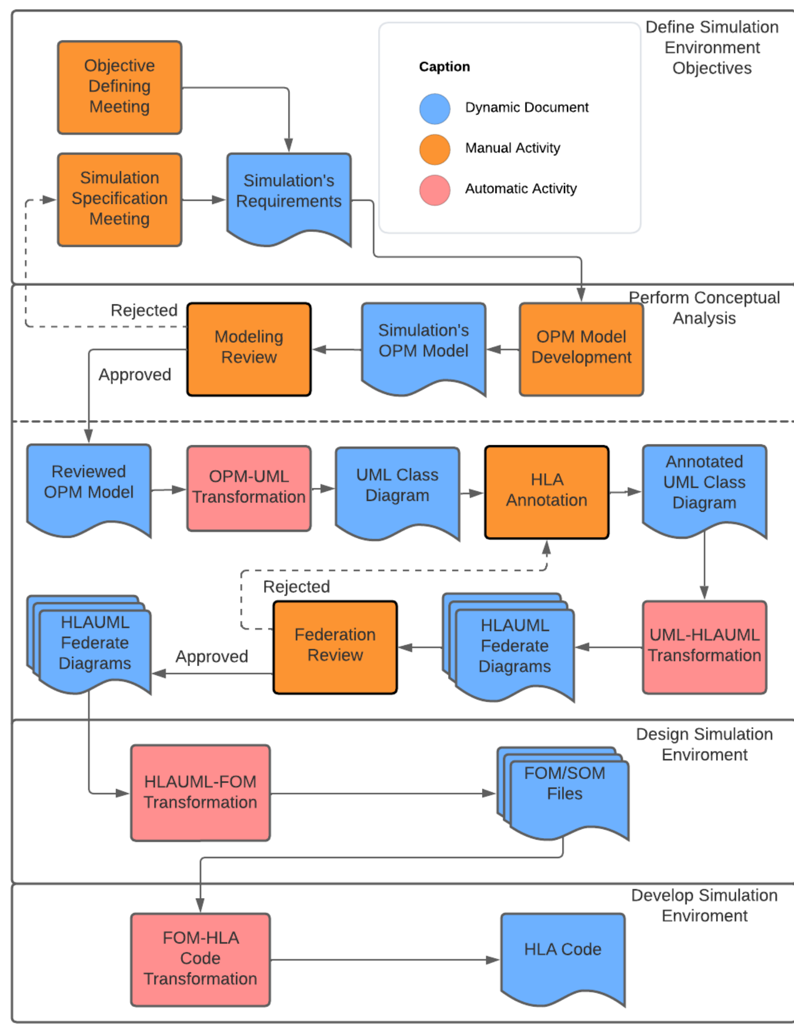


Figure 4 – Proposed Development Methodology

the simulation. This review is attended by the development team leads and the stakeholders to discuss if the presented model accurately attends to the expectations of the simulation. This is also the moment when the development team can ask questions regarding possible missing or ambiguous requirements. If the model requires many changes, it is deemed "Rejected," and the "Simulation Specification Meeting" is held once again to better define the misinterpreted requirements and possibly create new requirements from the development team's questions. If the model only requires few changes to its low-level views, such as class attributes, these small adjustments should be made, and the model is deemed "Approved".

Once the OPM model has been approved, the methodology advances to the second half of the conceptual analysis step. This part focuses on transforming the Platform Independent Model (PIM) into a Platform Specific Model (PSM). The first activity of this part is the automatic OPM-UML model-to-model transformation. This activity shall turn the OPM model into a "UML Class Diagram" document. It is essential to highlight that in our methodology, only UML class diagrams are used to model the PSM. This is because the

code generation receives FOM files, which are strictly structural. So, there is no need for the intermediate models to have diagram types containing behavioral information. Behavioral information may be used by the OPM model in order to further clarify intended simulation behavior to the stakeholders, but it will not be used by the code generator. Furthermore, treating all of the OPM objects as classes with their own types makes it easier to change the class type to fulfill a specific HLA role while maintaining the rest of the class without changes.

With the generated UML class diagram, the “HLA Annotation” activity can occur to produce the “Annotated UML Class Diagram”. The HLA annotation is a manual process that consists of further specializing certain classes in the UML diagram to represent HLA-specific roles, such as “Federate,” “Object,” or “Interaction.”. This activity shall be done by a member of the development team with a good understanding of HLA and the developed OPM model. This step starts to transform the UML model into a PSM since it now contains platform-specific (HLA) information.

The next activity after HLA model annotation is another automatic model-to-model transformation, named “UML-HLAUML Transformation”. Its goal is to transform the annotated model into an HLA-specific model by creating the publish/subscribe associations of the federates and the objects/interactions, creating a class diagram for each federate (containing only classes that the federate either publishes or subscribes to), and removing the elements of the previous model that are not relevant for the HLA simulation. The resulting diagram of this transformation is termed “HLAUML” to represent its complete transformation into a PSM.

The HLAUML diagrams can now be used for the “Federation Review” activity. This activity involves evaluating if the federates, objects, interactions, and publish/subscribe associations of the HLAUML model fit the proposed requirements of the distributed simulation. If the model needs adjustments, the process returns to the “HLA Annotation” activity; otherwise, the model is considered complete and approved. This review meeting is not attended by the stakeholders since the overall model was already reviewed; it is only attended by the development team leads to confirm the modeled federation that will be created.

The next activity is the last automatic model-to-model transformation of the methodology. The “HLAUML-FOM Transformation” receives the approved HLAUML model and generates the Federation Object Model (FOM) file alongside a set of Simulation Object Model (SOM) files for each federate that is in the input model.

After the generation of the FOM/SOM files, the model-to-text “FOM-HLA Code Transformation” can be executed. This transformation will use one of the available files depending on which of the federates has its code generated or whether it is desirable to generate HLA code containing the whole FOM. The generated “HLA Code” shall contain a usable API from which developers can use to implement your business logic and interact with the RTI to manage the various objects and interactions.

5.3 OPM MODELING

This section explains various topics regarding the use of OPM by this methodology, such as why OPM is used instead of already well-accepted and used languages like UML and SysML and how to generate OPM diagrams from written requirements.

5.3.1 Why use OPM?

This section explains why the OPM language was used for the initial conceptual model of this work's methodology instead of more usual and known languages like UML and SysML.

As explained before (Section 5.2), the Modeling Review activity is attended by the project's stakeholders and development team leads. Stakeholders are often non-technical people. As such, their requirements are described with a high level of abstraction. In contrast, modeling languages like UML and SysML require more than high-level specifications to correctly model the system's structure and behavior.

OPM, on the other hand, is capable of modeling a system's structure and behavior with high-level specifications with the use of different model views containing distinct levels of abstraction. For instance, if only the high-level requirements are well established, only high-level views of the system can be developed. The missing lower-level details are then pointed out and discussed in the review activity with the aid of having a high-level model from which to better visualize where the missing information fits. Another benefit of OPM is having only one diagram type that can showcase both structural and behavioral aspects, unlike UML and SysML which require different diagram types to show structure and behavior. Studies have shown that making models with many diagram types can have many problems, both for development and user understanding (ONG; JABBARI, 2019). Furthermore, OPM has a patented mapping of model elements into English sentences (Object-Process Language - OPL). OPL is perhaps the biggest factor in allowing its diagrams to be readable by non-technical people. As will be explained on Section 5.3.2, the OPL mapping allows for an ease of understanding of the many different types of links connecting processes and objects in a diagram. Thus increasing the reader's comprehension of the model.

The use of OPM, a more abstract model than UML and SysML, in the initial steps of our development methodology for distributed simulations aims to bridge the gap in abstraction between the stakeholder's high-level requirements and the need for more information on the lower-level simulation specifications, leading to a better understanding of the needs and wants of both parties involved in the simulation's development (stakeholders and developers). With greater mutual understanding, it is much less likely that major requirements will shift during the development of the simulation, reducing the project's cost, both in time

and money spent.

5.3.2 Modeling OPM from Requirements

This section explores, through a series of examples, how to create OPM diagrams from written requirements, specifically requirements related to an HLA distributed simulation. It showcases how writing requirements in certain ways has consequences for the resulting conceptual model. The examples are based on a distributed simulation with three fictional simulators called Alpha, Beta, and Delta.

The first example has the following requirements:

RQ1: During integrated simulations, Alpha, Beta, and Delta shall communicate between themselves using the HLA 1516 standard.

Considering that HLA uses the publish/subscribe architecture with a central RTI component, a middleware object, and processes relating to the publish/subscribe of each simulator are required. Each simulator is responsible for handling its respective publishing process and sending data to the middleware. The middleware object is required for the subscribing processes of each simulator, as it is the component that sends data to each simulator according to their preferences. The following OPL segment was constructed to follow the RQ1 specification.

Alpha handles Alpha Publishing.
Beta handles Beta Publishing.
Delta handles Delta Publishing.
Alpha Publishing affects Middleware.
Beta Publishing affects Middleware.
Delta Publishing affects Middleware.
Alpha Subscribing requires Middleware.
Beta Subscribing requires Middleware.
Delta Subscribing requires Middleware.
Alpha Subscribing affects Alpha.
Beta Subscribing affects Beta.
Delta Subscribing affects Delta.

This OPL segment, alongside its visual representation in Figure 5, contains behavior associated with a publish-subscribe distributed simulation that connects to a main middleware component (RQ1). Each simulator handles its publishing process; The middleware receives data from (is affected by) each publishing process; The middleware handles all subscribing processes; Each simulator receives data from (is affected by) their respective

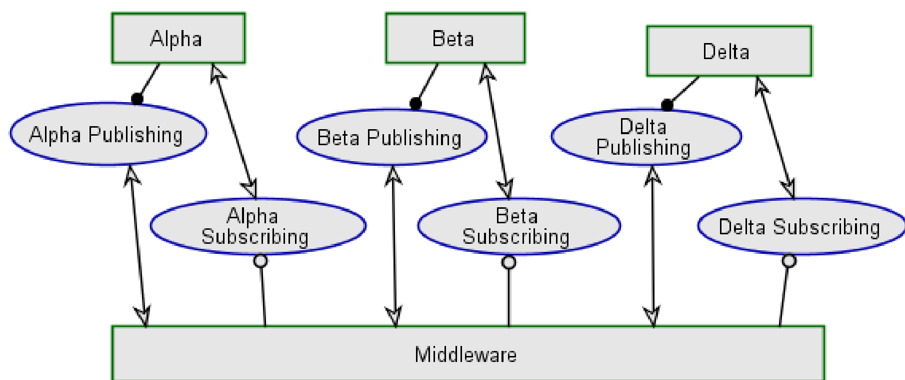


Figure 5 – OPD that implements RQ1

subscribing process. This segment can also be easily expanded if more simulators are planned to join the federation, given its modular design using the central middleware object.

It is important to note that the use of the agent link (line with a filled black circle on one of the ends) in this model deviates slightly from the guidelines in the original OPM specification (DORI, 2002). The specification defines agents as “An intelligent enabler, which can control the process it enables by exercising common sense or goal-oriented considerations” and explains that it refers to humans or human organizations. In the context of this work's modeling of distributed simulations, we consider all simulators connected to the distributed simulation to be OPM agents, whether they will have humans directly operating them or if their simulation will be strictly operated indirectly by the subscribing of objects and interactions from the other simulators.

The second set of requirements relates to the Alpha simulator's responsibilities in the federation.

RQ2: During integrated simulations, Alpha shall publish its aerial vehicles using the *AirVehicle* object class.

RQ3: During integrated simulations, Alpha shall subscribe to *WaterVehicle* and *GroundVehicle* objects and display them on it's simulation.

RQ4: During integrated simulations, Alpha shall publish a *RefuelRequest* Interaction when an *AirVehicle* owned by Alpha needs to refuel.

RQ5: During integrated simulations, Alpha shall subscribe to the *RefuelResponse* interaction to known where to send it's *AirVehicle* for the refueling process.

RQ6: During integrated simulations, Alpha shall have its simulation's date/time and speed regulated by the *Time* and *TimeScale* interactions, respectively.

This set of requirements clearly states the various classes that the *Alpha* simulator shall publish/subscribe to. Thus, the OPL segment will reflect that by adding the mentioned classes, with their relation to *Alpha*, and introducing consumption links between the published classes and *Alpha Publishing* and resulting links between the subscribed classes

and *Alpha Subscribing*. The following OPL segment fulfills the requirements following this specification:

> Alpha handles Alpha Publishing.
> Alpha Publishing consumes AirVehicle and RefuelRequest.
> Alpha Subscribing yields GroundVehicle, WaterVehicle, Time, TimeScale and RefuelResponse .
> Alpha *publishes* AirVehicle.
> Alpha *publishes* RefuelRequest.
> GroundVehicle *updates* Alpha.
> WaterVehicle *updates* Alpha.
> Time *updates* Alpha.
> TimeScale *updates* Alpha.
> RefuelResponse *updates* Alpha.

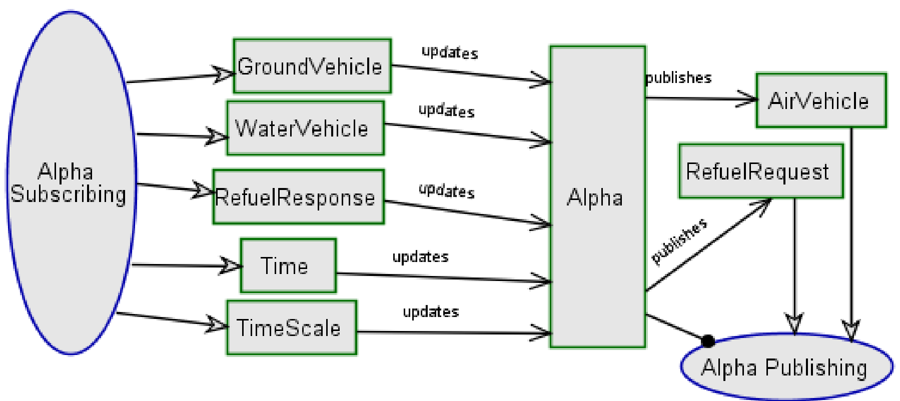


Figure 6 – Alpha Simulator OPD (RQ2 to RQ6).

This OPL segment, alongside its OPD equivalent representation in Figure 6, contains all requirements related to the Alpha simulator in the distributed simulation, meaning all classes it publishes or subscribes to. Much like in Figure 5, *Alpha* still handles its publishing process. However, in this OPD, which contains *Alpha*-specific information, objects *RefuelRequest* (from RQ4) and *AirVehicle* (from RQ2) have a tagged *publishes* association with *Alpha* to indicate that the simulator publishes those classes; In turn, the publishing process for alpha consumes said classes (and sends them to the middleware component, as seen in Figure 5). The subscribing process is slightly different from the one presented in the OPD for RQ1. Instead of a singular “affects” link between it and the simulator, the actual classes *Alpha* subscribes to, *GroundVehicle* and *WaterVehicle* from RQ3, *RefuelResponse* from RQ5, *Time* and *TimeScale* from RQ6, are yielded from the process, connecting to the simulator object with a tagged “updates” association.

The third set of requirements relates to the Beta simulator’s responsibilities while connected to the federation.

RQ7: During integrated simulations, *Beta* shall publish it's fluvial vehicles using the *WaterVehicle* object class.

RQ8: During integrated simulations, *Beta* shall subscribe to *AirVehicle* and *GroundVehicle* objects and display them on it's simulation.

RQ9: During integrated simulations, Beta shall subscribe to the *RefuelRequest* Interaction and publish a corresponding a *RefuelResponse* interaction, accepting the request and informing which vehicle can perform it or denying the request.

RQ10: During integrated simulations, Beta shall have its simulation's date/time and speed regulated by the *Time* and *TimeScale* interactions, respectively.

This set of requirements clearly states the various classes that the *Beta* simulator shall publish/subscribe to. Thus, the OPL segment will reflect that by adding the mentioned classes with their relation to *Beta* and introducing consumption links between the published classes and *Beta Publishing* and resulting links between the subscribed classes and *Beta Subscribing*. The following OPL segment was made to follow this specification.

Beta handles Beta Publishing.
Beta Publishing consumes WaterVehicle and RefuelResponse.
Beta Subscribing yields GroundVehicle, AirVehicle, Time, TimeScale and RefuelRequest.
Beta *publishes* WaterVehicle.
Beta *publishes* RefuelResponse.
GroundVehicle *updates* Beta.
AirVehicle *updates* Beta.
Time *updates* Beta.
TimeScale *updates* Beta.
RefuelRequest *updates* Beta.

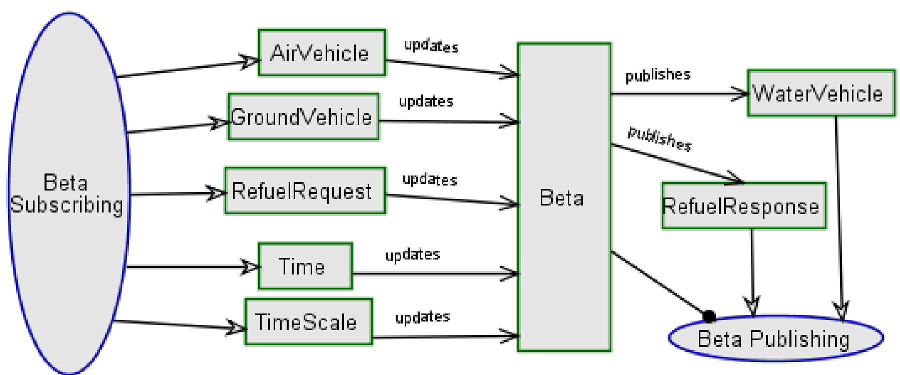


Figure 7 – Beta Simulator OPD (RQ7 to RQ10).

The resulting OPD from the *Beta* related requirements can be viewed in Figure 7. This OPD follows the same structure as the one for *Alpha*, with the only differences be-

ing which classes are published/subscribed to, which were adjusted to align with the *Beta* requirements.

The fourth set of requirements relates to the Delta simulator's responsibilities while connected to the federation.

RQ11: During integrated simulations, *Delta* shall publish it's ground vehicles using the *GroundVehicle* object class.

RQ12: During integrated simulations, *Delta* shall subscribe to *AirVehicle* and *WaterVehicle* objects and display them on it's simulation.

RQ13: During integrated simulations, Delta shall publish its simulation's date/time and speed with the *Time* and *TimeScale* interactions, respectively. Regulating the flow of time of the federation.

This set of requirements clearly states the various classes that the *Delta* simulator shall publish/subscribe to. Thus, the OPL segment will reflect that by adding the mentioned classes with their relation to *Delta* and introducing consumption links between the published classes and *Delta Publishing* and resulting links between the subscribed classes and *Delta Subscribing*. The following OPL segment reflects this specification.

Delta handles Delta Publishing.
Delta Publishing consumes GroundVehicle, Time and TimeScale.
Delta Subscribing yields AirVehicle and WaterVehicle.
Delta *publishes* GroundVehicle.
Delta *publishes* Time.
Delta *publishes* TimeScale.
AirVehicle *updates* Delta.
WaterVehicle *updates* Delta.

The resulting OPD from the direct translation of the *Delta* related requirements can be viewed in Figure 8. The structure of the OPM is visually different from the other simulator's OPD due to the lack of refuel request/response requirements and the fact that *Delta* publishes more classes than it subscribes to.

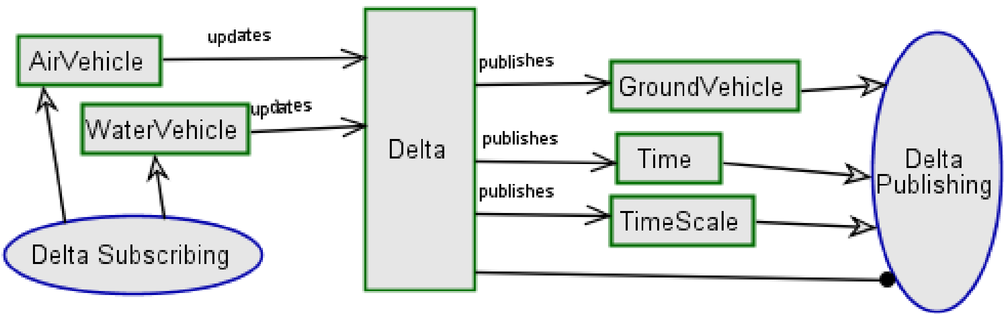


Figure 8 – Delta Simulator OPD (RQ11, RQ12 and RQ13).

Figures 6, 7 and 8 show that modeling publish/subscribe behavior in OPM can be both simple to develop, easily reproducible for different simulators and effective in showing the expected simulator's behavior.

The last set of inputs to finalize this distributed simulation's OPM model are the class specifications, which contain the attributes of the object/interaction classes. This type of specification is often not proposed by the stakeholders, rather, it is mostly defined by the people that will model the simulation. The next set of boxes contains specifications for all of the classes mentioned in the requirements.

VehicleObject: An object class related to the vehicle entities involved in the simulation between Alpha, Beta, and Delta. Instances of this class shall have the following attributes:

- Name: String Type. Contains the name of the vehicle.
- Team: Unsigned Integer Type. Contains the ID of the team that the vehicle is currently in.
- Damage: Double Type. Contains the current damage value of the vehicle. 0 is to be understood as no damage and 100 as completely destroyed.
- Position: Vector3 Type. It contains the current world location of the vehicle in the lat/lon/alt format.
- Velocity: Vector3 Type. It contains the current velocity vector of the vehicle in the lat/lon/alt format.

AirVehicle: An object class representing a vehicle operating mainly in the air. This object class inherits all attributes from the VehicleObject class and the following attributes:

- FuelLevel: Double Type. Contains the current fuel levels of the vehicle, measured in liters.
- FuelCapacity: Double Type. Contains the maximum capacity of fuel for this vehicle.

GroundVehicle: An object class representing a vehicle operating mainly on the ground. This object class inherits all attributes from the VehicleObject class.

WaterVehicle: An object class representing a vehicle operating on the sea. This object class inherits all attributes from the VehicleObject class and the following attribute:

- StoredFuelAmount: Double Type. Contains the current amount of fuel stored in this vehicle (available for other vehicles to refuel with). Measured in liters.

Figure 9 contains the OPD relating to the specification of the object classes. The *generalization/specialization* object-object relation was used to represent the hereditary

structure of *VehicleObject* and its child classes *AirVehicle*, *GroundVehicle* and *WaterVehicle*. The listed attributes were mapped with the *exhibits* object-object relation, and the auxiliary “Vector3” type was also added for completeness. Note that String types were implemented as “char[50]”.

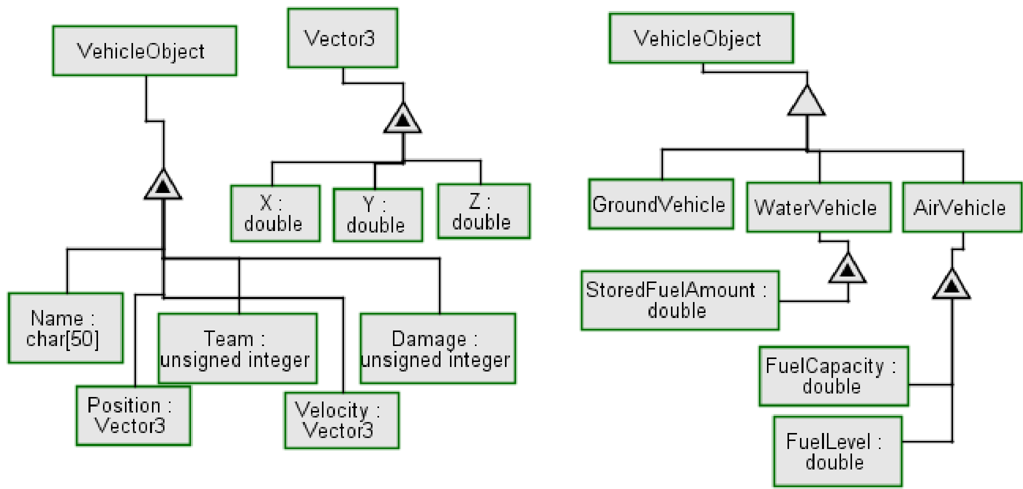


Figure 9 – Object Classes OPD

The next set of boxes specifies the various interaction classes present in the previous requirements.

Time: An interaction class that informs the current simulation Date/Time. Instances of this interaction shall have the following attributes:

- ClockTime: Date Type: Contains the current date of the simulation.

TimeScale: An interaction class that regulates the speed at which time increases. Instances of this interaction shall have the following attributes:

- SpeedFactor: unsigned integer type. The TimeScale value.

RefuelRequest: An interaction class that represents a request for refueling. Instances of this interaction shall have the following attributes:

- RequestingVehicleName: String Type. Contains the name of the vehicle requesting fuel.
- RequestID: Integer Type. Identifier of the request operation.

RefuelResponse: An interaction class that represents a response of a refuel request. Instances of this interaction shall have the following attributes:

- RespondingVehicleName: String Type. Contains the name of the vehicle that is available for refueling.
- ResponseID: Integer Type. Identifier of the request-response operation.
- ResponseResult: Enum Type. The result of the request can be “Accept” or “Deny”.

Figure 10 contains the OPD with the interaction classes of the distributed simulation. The listed attributes were mapped with the *exhibits* object-object relation.

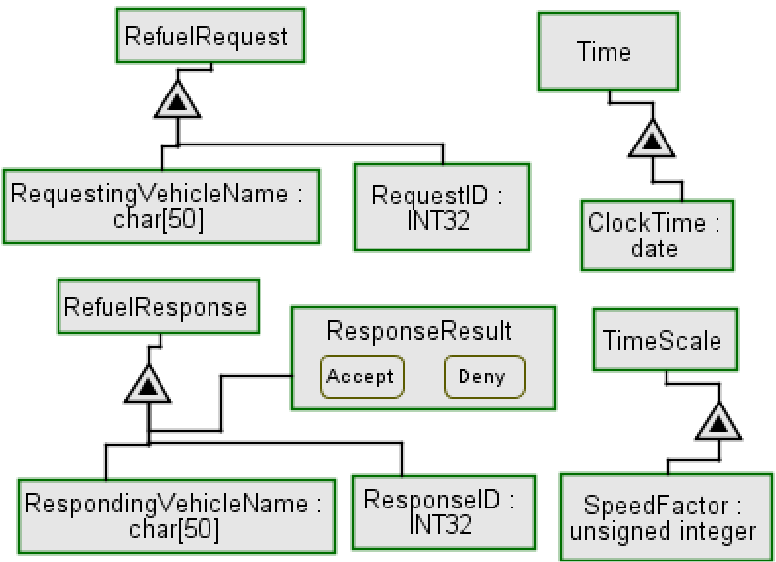


Figure 10 – Interaction Classes OPD

It is important to note that due to the direct equivalence of OPD and OPL. The modeller can choose which format they find the easiest to express the requirements with. They can even choose a hybrid approach, modeling some parts first with OPD and others first with OPL, if that is deemed to be the best option.

This section showed how to translate publish/subscribe distributed simulation requirements into OPM, whether OPL or OPD, with relative ease. The resulting model is quite scalable to fit more requirements. For example, adding new classes to be published/-subscribed means adding more exhibition links for the class attributes, more result/consumption links for the input/output of the simulator publish/subscribe processes, and more object-object associations to relate the class to each simulator. Notably, no more OPDs need to be created unless a new simulator/federate is added.

5.3.3 OPM Modeling Review

This section further explains the Model review process, a key moment where this methodology is benefited by using OPM.

Because OPM can support high-level abstractions and OPL is present, the model evaluators can be the stakeholders themselves. This increases the accuracy of the review activity because the model is analyzed by the same people who proposed the requirements. Since this methodology focuses more on conceptual modeling, the review was split into two activities. One to review the OPM model (the subject of this section) and another to review the HLA model annotation (which will not be discussed in this section). Both reviews were placed on the DSEEP's "Perform Conceptual Analysis" step. During the OPM modeling review, the model evaluators (stakeholders and developers) should view the developed OPM model (OPDs and OPL segments) alongside the requirements and discuss missing information or incorrect behavior.

The requirements presented in Section 5.3.2 were the final result of a modeling review. An example of how and why some of the requirements were changed will now be presented.

> **RQ11-Old:** During integrated simulations, *Delta* shall make its vehicles available for the other simulators to see.

The older version of RQ11 does not clarify that the way that *Delta* should make its vehicles available is by publishing them to the middleware. Additionally, the requirement did not specify the use of the *GroundVehicle* class for this purpose because the *GroundVehicle* class was not yet conceptualized; only after the review were the class specifications realized. Thus, the requirement was changed to the one that was shown in Section 5.3.2. Without the review, the expected behavior, which is for *Delta* to publish *GroundVehicle*, could have been modeled in an incorrect manner, leading to erroneous behavior in the final simulation.

The modeling review can not only change requirements but add new requirements as well. For example, take a look at the first version of RQ2:

> **RQ2-Old:** As an Alpha operator, I want Alpha to display remote vehicles on the terrain.

This requirement can be interpreted as the Alpha simulator subscribing to the vehicle object classes published by the other simulators. The requirement was written from a user's perspective. These types of requirements are called User Stories; they are fairly common and recommended when writing requirements in agile development methods (SCHöN; THOMASCHEWSKI; ESCALONA, 2017). When writing requirements for a publish/subscribe distributed simulation, we advise not to use user stories when describing the data to be exchanged between simulators. A simulator operator can see what their simulator receives from the others but does not always see what their simulator sends to the others;

as such, the requirements relating to the publishing of certain objects and interactions may be missing from the requirements list if they are mostly user stories, meaning that some of the simulators could be missing publishing behavior if the requirements are not changed. Thus, another requirement needed to be added so that the publishing behavior could be fully specified, resulting in the following two requirements, which are the finalized versions of RQ2 and RQ3:

RQ2: During integrated simulations, Alpha shall publish its aerial vehicles using the *AirVehicle* object class.

RQ3: During integrated simulations, Alpha shall subscribe to *WaterVehicle* and *GroundVehicle* objects and display them on it's simulation.

This section explained the importance of the OPM modeling review activity in correcting requirements that have incomplete information or are worded in such a way that critical information might be implicit. In turn, it guarantees that the distributed simulation behaves as the stakeholders expect.

6 AN IMPLEMENTATION FOR THE PROPOSED METHODOLOGY

This chapter explains an implementation of the proposed development methodology and the reasoning behind the technologies used to perform each part. Figure 11 shows the order of operations to generate HLA source code of our methodology, excluding the initial meetings. The figure splits the workflow into four parts, identified by dotted lines. Each part corresponds to a goal derived from MDA directives.

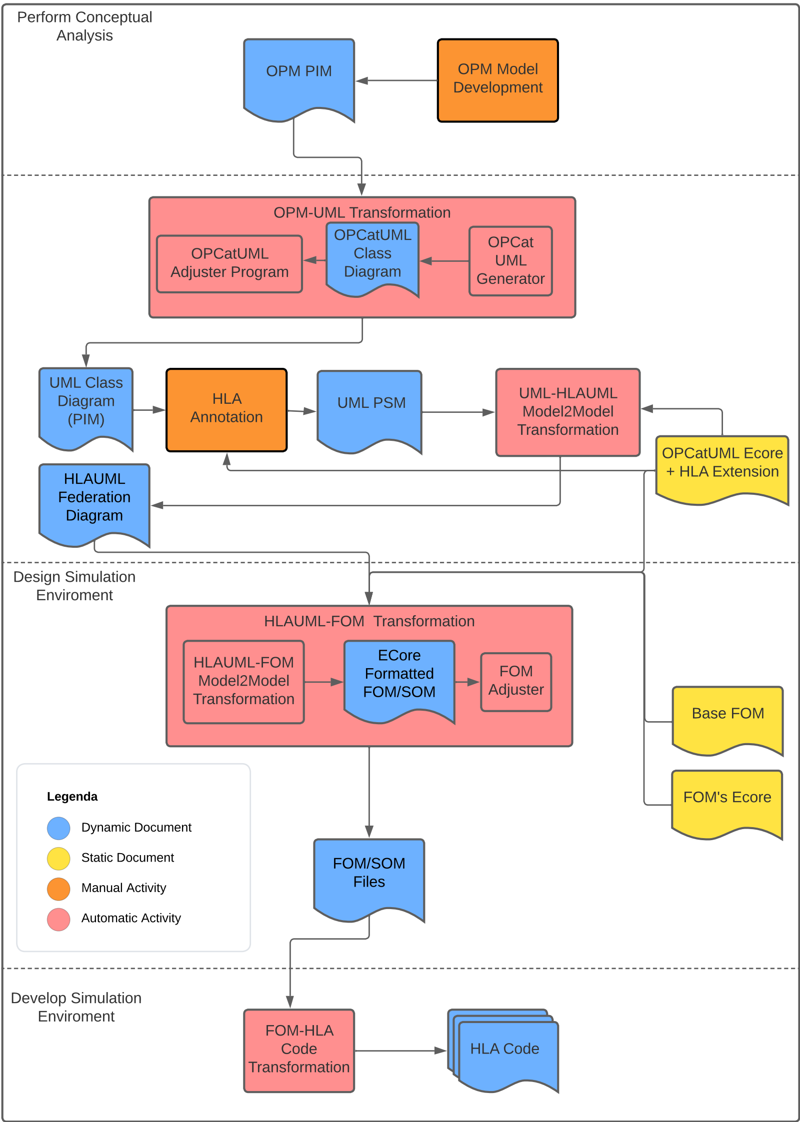


Figure 11 – Implementation of the Proposed Methodology

From MDA, the methodology implementation was separated into four main goals.

1 Develop a PIM (Platform Independent Model) using OPM.

2 Transform the OPM PIM into a UML PSM (Platform Specific Model).

2.1 Transform the OPM PIM into a UML PIM.

2.2 Transform the UML PIM into a UML PSM.

3 Transform the UML PSM into the FOM file.

4 Transform the FOM into the final HLA source code.

This chapter goes over these four objectives, explaining what needs to be done for each of them, what tools are available to perform the task, which of the tools were chosen and how the tool was used to accomplish the goal. Sections 6.1, 6.2, 6.3 and 6.4 explain each one of the goals in order.

6.1 DEVELOPING A PLATFORM INDEPENDENT MODEL USING OPM

To perform the conceptual analysis (see Figure 11), and to achieve the first goal (developing a PIM using OPM), the first step of the implementation is to solve the activity to create an OPM PIM model to represent the desired distributed simulation. As such, a modeling tool for using OPM is required.

We found two candidate tools to perform this task. OPCloud (OPCloud Ltd, 2024) and OPCat (DORI et al., 2010). Both tools allow the user to create OPM models using their respective GUIs. OPCloud has a paid subscription plan whereas OPCat is free to use. On top of that, OPCat has a UML export feature that allows for the conversion of an OPM model into a UML model (further explained in 3.4.1). For these reasons, OPCat was chosen for this implementation's OPM model development activity.

The manual OPM modeling activity creates the "Simulation's OPM Model" document, which is the goal of this step and it is showed as OPM PIM on Figure 11.

6.2 TRANSFORMING THE OPM PIM INTO A UML PSM

This section explains how the second objective, the transformation of the OPM PIM into a UML PSM, was achieved by our implementation.

To turn the OPM PIM into a UML PSM, two activities are needed. Firstly, the OPM PIM needs to be transformed into a UML PIM. Secondly, the UML PIM needs to be annotated into a UML PSM.

6.2.1 Transforming the OPM PIM into a UML PIM

To transform the OPM PIM into a UML PIM we explore the OPM-UML export feature of OPCat to generate a UML class diagram containing all of the objects and associations of the OPM model. This UML class diagram is named “OPCatUML Class Diagram” since it is a UML class diagram generated in OPCat. The OPM-UML export feature is the first activity inside the greater OPM-UML transformation activity indicated in Figure 11.

It is important to note that the OPCatUML file adheres to UML version 1.3. Therefore, it does not have any of the new features added since it is not directly compatible with some modern UML modeling tools. However, if some users want to update the UML version used in their implementation of the methodology, they can implement the model-to-model transformation in a different way. In other words, it’s not a limitation of the actual implementation. For this work it fills the goal to transport the OPM PIM information to a UML representation.

6.2.2 Transforming the UML PIM into a UML PSM

Once we have a UML PIM, the next task is to turn the UML PIM into a UML PSM. Since we already know that model-to-model transformations will occur with the UML PSM, we need to choose an environment that allows for the development and execution of model-to-model transformations and the definition of meta-models to perform said transformation.

To perform model to model transformations, a variety of languages exist. Three prominent languages are ATL (Atlas Transformation Language) (ATL Development Team, 2024), QVT (Query View Transform) (OMG, 2024), and ETL (Epsilon Transformation Language) (Epsilion Development Team, 2024). These three languages can be used in the Eclipse IDE (Eclipse Foundation, 2023) with ECore (EclipseECoreTools Project, 2023) meta models, so they shall be the IDE and meta-model format of choice for this implementation.

The language choice is challenging due to its dependence on the user’s programming preferences. It is important to highlight some possibilities. ATL and ETL offer a hybrid approach, mixing imperative and declarative structures. The QVT language is divided into three sublanguages: QVT-Relations, QVT-Core, and QVT-Operacional. The latter of which uses an imperative approach, while the other two are declarative. From the previous author’s background, QVT-Operational was chosen as the preferred language for model-to-model transformations in this work. It has a strictly imperative nature, and it was easier for the developers of this work to learn.

With a development environment, meta-model format, and model-to-model transformation language chosen, the next step is to create a meta-model that fits all UML PIM models that can be created from the OPCat OPM-UML export feature (considering only class diagrams).

An Ecore meta-model that can represent all of the possible OPCatUML models was created manually using the ECoreTools plugin for Eclipse by analyzing a series of OPCatUML models with different compositions. This Ecore meta-model can be reused to develop any distributed simulation using this implementation. Many elements of the meta-model are not used by any transformation, but since the models have them, they must be contained within the meta-model.

Figure 12 shows a part of the developed OPCatUML meta-model. The model begins with the `Content` root element, which contains a reference to the `General Model` element. The general model has one or many references to `Model` elements that represent the different generated diagrams. Each diagram has a `NamespaceOwnedElement` element that contains the diagram's various associations, classes, datatypes and generalization/specialization relationships. The model-to-model transformations that make use of this core have to follow this element structure to be able to access (or generate) information about the classes of the diagram.

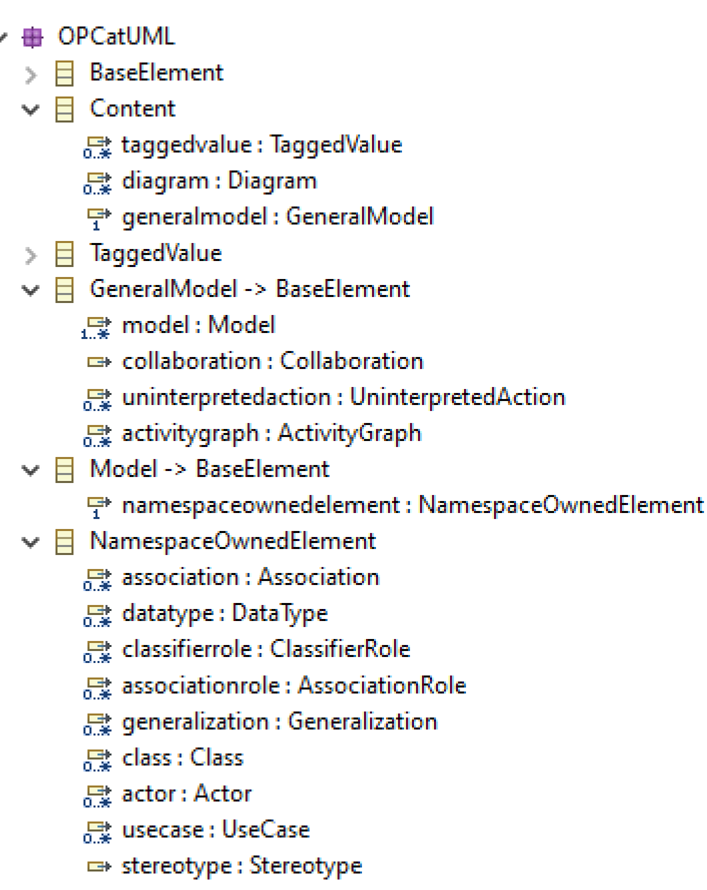


Figure 12 – Part of the OPCatUML Meta-Model

Even while trying to model as closely as possible to the OPCatUML file format, some parts of it could not be properly expressed with Ecore. Thus, an adjuster program was developed in the Eclipse IDE to adjust the OPCatUML files to conform with the developed meta-model. This program is the last activity inside the greater OPM-UML transformation

activity, as can be seen in Figure 11.

The most important adjustments that the program makes are as follows:

- To remove dots from the element names because ECoreTools only accepts element names if they are valid Java identifiers.
- To remove colons in the names. All elements have a colon in their name, indicating UML 1.3 packages. This could be replicated in ECoreTools by creating many sub-packets, but it was decided that it would be simpler to just remove the colons from the element's names.
- To adjust the file header. The header of the OPCatUML file is updated from XMI 1.2 to XMI 2.0 to enable some compatibility with Eclipse's XMI editors. This header update maintains the file integrity since XMI 2.0 maintains backward compatibility with XMI 1.2.
- The attribute "xmi.id" present in many of the elements is changed to "id", because the "xmi.id" attribute cannot not be viewed in Eclipse XMI editors.
- All of the non-root elements had their names altered to lowercase since that is how Ecore interprets element names.
- To adjust the final names to maintain the integrity. After removing the dots and colons and converting the names to lowercase, some elements happened to have the same name. To resolve this, the elements were merged. This did not cause any problems due to how these pairs of elements were arranged in the file, one being the direct and single child element to the other.

Now that we have an Ecore-compliant UML PIM, the next step is to annotate the model, turning it into a UML PSM. To allow for the annotation of HLA concepts to the OP-CatUML model, the Ecore meta-model was extended with the required classes by creating a sub-package. The HLA sub-package can be viewed in Figure 13. This extension contains elements representing many of the HLA concepts, like `ObjectClass`, `InteractionClass`, `Federate`, `Publish`, `Subscribe`. All of these elements are derived from elements of the main package, allowing for the substitution of the elements for their more specific HLA variants.

With the HLA-extended Ecore meta model, model annotation can occur. The annotation process entails specifying the original classes within the diagram into new classes that accurately reflect their roles in the proposed distributed simulation. The model annotation is split into two activities. The first is a manual process where the developer assigns new types to existing classes. The second one is a model-to-model transformation that

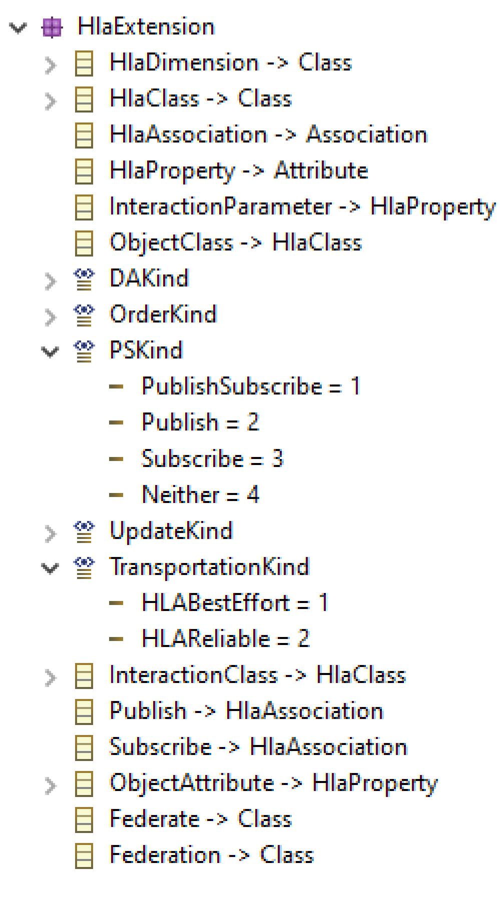


Figure 13 – HLA Extension of the OPCatUML meta-model

uses newly annotated classes to generate, among other things, the publish/subscribe associations between the federates and the objects/interactions.

The manual annotation consists of analyzing the class diagram's `Class` elements and giving them their appropriate HLA roles according to what they represent. For instance, a class in the model may be modified to represent a `Federate`, setting that class as a role of participant in the simulation. Listing 6.1 shows how the `xsi:type` attribute of the class was given the value `HLA:Federate` to give the role of a federate to the class. Note that this process could be done by directly editing the file or abstracting it using an auxiliary program.

Listing 6.1: Example of a class annotated as a Federate

```
<class
    xsi:type="HLA:Federate"
    name="SimulatorA"
    isRoot="true"
    isLeaf="true"
    isAbstract="false"
    id="S.456"
    isActive="false">
    <classifierfeature/>
</class>
```

Not every class in the UML diagram undergoes a type change during the annotation process. This is because the OPM modeling may encompass components that do not pertain to the network communication between simulators but serve to enhance the overall clarity of the simulation procedures. Furthermore, the annotation process assumes responsibility for augmenting the UML diagram with additional details, such as attributes and types that might have been omitted from the initial OPM model.

After the manual annotation process, the first QVT-Operational transformation, UML-HLAUML, is executed. This transformation receives the OPCatUML annotated model alongside the HLA extended OPCatUML ecore and generates a UML PSM class diagram for the federation (named “HLAUML”), containing all the objects, interactions, data types and publish/subscribe associations of every federate. The publish/subscribe associations are created using the following logic: If a federate has an association with an object/interaction class, then a publish/subscribe association is created based on the direction of the association. If the association is from the federate to the class, a “publish” association is created; if the opposite is true, then a “subscribe” association is created.

After the publish/subscribe associations are created, one additional class diagram is built for each federate class found in the input model. These class diagrams are still contained in the same file. A given federate’s class diagram only contains classes with which the federate has some association. Lastly, even though this transformation creates publish/subscribe associations, the development team still can adjust these associations if they consider it necessary.

Beyond these operations, this transformation also controls which elements from the model will be passed onto the output model. Many elements generated by the OPM-UML transformation are not used by the succeeding operations, like classes that are not part of the distributed simulation, associations that became redundant with the addition of the publish/subscribe associations, and elements that indicate how to visually build the model.

The UML PSM class diagram for the federation (HLAUML) modeling activity creates the “HLAUML Federate Diagrams” document, as it is showed on Figure 11, finishing the

second methodology goal, to transform the OPM PIM into a UML PSM (Platform specific model).

6.3 TRANSFORMING THE UML PSM INTO THE FOM

This section explains how our implementation achieved the third objective, transforming the UML PSM (HLAUML model) into the FOM.

To transform the HLAUML model into a FOM, another model-to-model transformation is required. To do that, the realised methodology implementation continues to use the Eclipse IDE, Ecore meta-models, and QVT-Operational to perform this task.

To generate a FOM from a QVT Operational transformation, a meta model for the FOM is required. This meta-model was not created manually for this work. The FOM file has a publicly available XSD (XML Schema Definition), defined in IEEE1516 (IEEE, 2010). With the FOM XSD, a feature of the ECoreTools plugin was used to transform the XSD schema into an Ecore meta-model.

The “HLAUML-FOM Model2Model Transformation” model to model transformation (see Figure 11) has four inputs: the “HLAUML Federate Diagrams”, the HLAUML model generated by the previous transformation; a “Base FOM” file, offered by HLA standard or by the community of simulation domain; a “OPCatUML Ecore + HLA Extension”, the ecore metamodel for the HLAUML model input; and a “FOM’s Ecore”, the ecore metamodel for the FOM model output.

The base FOM file contains the base structure of a FOM, which is utilized as a skeleton that will receive many fields from the HLAUML model throughout the transformation. The base FOM contains the identification element `HLAObjectRoot` and `HLAInteractionRoot` and data types that are common to any FOM, such as `HLAinteger64BE`, `HLAfloat64BE`, `HLAASCIIchar`, and `HLAboolean`.

Normally, a FOM file has one `ObjectModelType` element, which contains all of the information that the FOM holds. However, this transformation generates more than one `ObjectModelType` element. The first of these elements is meant to represent the whole FOM, while the others represent each federate’s SOM, containing only the objects/interactions/data types that the federate associates with.

The Table 4 showcases the source for all the information mapped onto the output FOM. Some general fields, like `BasicDataRepresentations`, come directly from the base FOM, and more specific fields, like the FOM’s name and object/interaction classes, come from the input HLAUML model.

For the creation of the FOM’s data types, the following logic was applied to the data types from the HLAUML model.

Element	Source
Identification	
Name	“HLAUML”
Objects	
Name	“HLAUML”
Class Hierarchy	“HLAUML” + Base FOM
Attribute Name	“HLAUML”
Attrbute Type	“HLAUML” + Base FOM
Interactions	
Name	“HLAUML”
Class Hierarchy	“HLAUML” + Base FOM
Parameter Name	“HLAUML”
Parameter Type	“HLAUML” + Base FOM
Switches	Base FOM
Data Types	
BasicDataRepresentations	Base FOM
SimpleDatatypes	“HLAUML” + Base FOM
EnumeratedDatatypes	“HLAUML” + Base FOM
ArrayDatatypes	“HLAUML” + Base FOM
FixedRecordDatatypes	“HLAUML”

Table 4 – Transformation Information Source for Generating a FOM.

- `BasicDataRepresentations`: None are added since all of them are listed in the base FOM.
- `SimpleDatatypes`: If the data type is called `Integer`, `INT`, `float`, `double`, `long`, or `short`, they are added to the `SimpleDataType` list. The values of the `resolution`, `accuracy`, and `units` are left as blank.
- `ArrayDatatypes`: If the data type name is one of the `SimpleDataType` values followed by “[]”, a corresponding array type is created. The exceptions to this are char arrays and the string data type; with these cases, the `HLAunicodeString` type from the base FOM is used.
- `EnumeratedDatatypes`: When a type with the format “{Name1, Name2, Name3..}” is found in the model, it is considered an `EnumeratedDatatype`. This data type should contain a list of names and numbers representing the possible values of the enumeration. The names are extracted from the type’s name format, and the values are attributed in the same order they are listed.
- `FixedRecordDatatypes`: If a data type is found with the same name as a `Class` element in the HLAUML, it is considered a `FixedRecordDataType`. The `field` attribute list is filled with the class elements attributes.

Creating FOM’s interactions and objects requires further explanation because the

structure between the input (HLAUML) and output (FOM) models differs. In the FOM structure, objects and interactions are organized as trees, where classes contain other classes inside them, indicating heredity, where all objects inherit from `HLAObjectRoot` and all interactions inherit from `HLAInteractionRoot`. However, the HLAUML model has all the classes in a list. The heredity is represented by two class attributes, `Generalization` and `Specialization`; they contain class IDs. Listing 6.2 shows a simplified QVT-Operational recursive algorithm that uses the class IDs from these attributes to transform the class structure from the list format to the tree format.

Listing 6.2: Simplified QVT-O Algorithm for transforming the class structure between HLAUMl and FOM.

```
helper AjustUMLClasses(in classes:Set(Class)):Set(Class)
{
    var newClasses : Set(Class);
    //Adds all root classes to this list.
    classes->forEach(_class){
        //Empty generalization means root class
        if(_class.generalization = ""){
           var newClass := _class.ComposeClass(classes);
           newClasses += newClass;
        };
    };
    return newClasses;
}
helper Class::ComposeClass(in allClass:Set(Class)):Class
{
    // if is a leaf class, returns itself
    if(self.specialization = ""){
       return self;
    };
    var classChildren := allClass->select(class | class.generalization = class.id);
    // Adds child classes to its owned element.
    // Executes threcursive call of this method.
    // Effectively, this transforms the
    // class structure to a proper tree
    self.namespaceownedelement.class +=classChildren.ComposeClass(allClass);
    return self;
}
```

Much like the OPCatUML model, the Ecore conforming FOM/SOM model that is created from the HLAUML-FOM model to model transformation needs to be adjusted in order to be compatible with programs that read and validate FOM files. As such, a FOM adjuster program was developed to align this file with the expected format.

The FOM Adjuster program receives the FOM-Like file generated by the model to model transformation and performs two operations to it in order to make the file accurate to what is expected by many FOM readers.

The first operation consists of instancing attributes as elements, transforming the attributes of an element into child elements containing the attribute's value. The following example illustrates this. Consider an interaction class with a `name` attribute with the value `HLAinteractionRoot` as follows.

```
<interactionClass name="HLAinteractionRoot"/>
```

This interaction class is transformed into an element that contains a child `name` element, with the value `HLAinteractionRoot`.

```
<interactionClass>
  <name>HLAinteractionRoot</name>
</interactionClass>
```

A notable exception to this rule is the child elements of the `switches` element of the FOM because these elements are already in the desired format. With child elements instead of attributes.

The second operation done by the FOM Adjuster is the separation of the many `ObjectModelType` elements generated by the HLAUML-FOM transformation. The adjuster creates one FOM file for each `ObjectModelType` element, effectively generating both the FOM file and the SOM files for each of the federates in the model and ending the “Design Simulation Environment” step and third objective (to transform the UML PSM into the FOM file).

6.4 TRANSFORMING THE FOM INTO HLA SOURCE CODE.

This section explains how our implementation achieved the fourth and last objective, transforming the FOM into HLA source code.

There are many code generator options to be used in order to perform this task. This implementation’s code generator used what as developed in a previous work (SANTOS; NUNES, 2022) as a basis, expanding the capabilities of the code generation with the use of the *StringTemplates* library for C# to enable easier changes to the generated code and allowing for the possibility of generating code for multiple languages.

This source code generation program can receive any FOM file and generate code according to its contents. To handle inputs, the XSD schema of the FOM was used. To generate code, a series of C++ class templates were created to allow for the creation of all of the encoders/decoders for each datatype, a manager class for each object/interaction class, and an RTI manager (called federate manager) to allow for connecting and disconnecting to/from a federation.

The developed RTI manager also allows the use of the HLA time management service. The class managers contain a sub-manager for executing HLA ownership management calls.

In summary, using a third party code generation tool from FOM files, this last “FOM-HLA Code Transformation” produces the final expected HLA Code and ends the DSEEP Develop Simulation Environment step over our OPM-UML based MoDSEEP methodology.

7 VALIDATION

This section will present a series of experiments made to help validate this work's contributions. The first two experiments help validate the first contribution, that being the new conceptual modeling step that is more adequate for non technical readers, which is often the case with project stakeholders. The first experiment, presented in Section 7.1, contains a case study showing how someone with no prior knowledge of OPM was able to translate distributed simulation requirements into OPDs. The second experiment, presented in Section 7.2, aims to justify using OPM against the field standard UML by comparing models made in each language to describe the same set of requirements.

The remaining two experiments help validate the second contribution, which relates to the implementation of the new modeling step in a full MDA development methodology for HLA simulations. Section 7.3 contains the third experiment, in which an example federation goes through all of the steps of the methodology until it becomes the simulation's source code. Section 7.4 presents the fourth experiment, in which analysis and tests are performed on the source code generated by the previous experiment.

7.1 EXPERIMENT 1: MILITARY CASE STUDY

This case study aims to showcase how OPM is easier to comprehend and model for people unfamiliar with conceptual modeling languages. In it, a military software maintainer with only basic notions of conceptual modeling languages (and no prior knowledge of OPM) was tasked with developing an OPM model to represent a federation connecting two real simulators used by the Brazilian Army to educate two distinct sectors. This model would then be analyzed to determine if it accurately depicts the simulation to see what someone unfamiliar with conceptual modeling languages can do it with OPM.

The rest of this case study is described in sections dedicated to each of the initial activities of the development methodology. Section 7.1.1 pertains to the first activity, the "Objective Defining Meeting"; Section 7.1.2 pertains to the second activity, the "Simulation Specification Meeting"; Section 7.1.3 pertains to the third activity, "OPM Model Development"; Section 7.1.4 pertains to the last activity of this case study, the "OPM Model Review."

7.1.1 Objective Defining Meeting

The first activity, the "Objective Defining Meeting," aims to create a high-level specification of the distributed simulation that will be developed. It is attended by stakeholders

and development team leaders.

The first meeting decision was to use two real simulators that in this case study are called *STat* and *SCon*. They have the following characteristics.

STat Simulator:

- A virtual tactical simulator developed to educate the people in charge of giving orders to the operators of a specific artillery battery. In this experiment, we will call it *STBattery.*
- It represents a virtual environment where *STBattery* can be controlled and have its specific operations be performed in order to execute missions.
- It offers a visual and interactive representation of the artillery systems, containing a 2D view of the terrain where units are represented by 2D symbols and a 3D view of the terrain with a movable camera that represents the vehicles of *STBattery* with detailed 3D models.
- Trainees use this simulator to learn the doctrine required to use *STBattery* in order to perform missions and evaluate the damage done to the targets.

SCon Simulator:

- Represents the constructive environment of the military operation, including allied and enemy forces, marked areas, movement routes, and other relevant factors.
- Contains information regarding the terrain's topography and updates the positions and statuses of the deployed units to reflect their actions during the operation.
- Does not have a personalized doctrine for the operation of *STBattery* (since it contains various other types of units). Those details are abstracted in favor of only representing the final effects on the targets that *STBattery* shot.

From this meeting, the main objective of the simulation federation was established:

> The main objective is to create an education environment that enables training exercises involving *STBattery* in the simulators *SCon* and *STat* operating in an integrated way by HLA (distributed simulation).

Furthermore, the following directives were solidified:

- Doctrine-specific actions regarding *STBattery* will be executed by *STat* to provide the artillery commander trainees with a more immersive and realistic experience.

- Relevant data regarding the state of the mission, like target coordinates and current battery position and where the launched missiles landed, shall be shared between the simulators.
- Both simulators shall use the Real-Time Platform Reference (RPR) FOM (SISO, 2015) as a basis for the distributed simulation communication. Extending the FOM when it is deemed necessary to fulfill a requirement.

7.1.2 Simulation Specification Meeting

The second activity, the "Simulation Specification Meeting," has the goal of defining less abstract details about the distributed simulation that will be developed. When this meeting took place in this case study, requirements were separated into two categories. Requirements relating to what the simulators need to configure before connecting to the federation and requirements relating to functionalities that the simulators must display during the simulation.

The first category of requirements, called prerequisites (PQ), can be seen in Table 5. The second category of requirements (RQ) can be seen in Table 6. Since this work's development methodology centers around creating run-time code for distributed simulators, the requirements that will most influence the conceptual model are the ones from the second category.

PQ	**Description**
PQ1	*SCon* and *STat* must be able to operate with geographical terrains with significant overlap.
PQ2	*SCon* and *STat* shall have an agreed-upon *EntityType* attribute mappings, correlating all possible units and munitions to an enumeration. The *EntityType* attribute should follow the specifications from RPR FOM.

Table 5 – Prerequisites of the distributed simulation with *SCon* and *STat*.

RQ	Description
RQ1	*SCon* should be "Time Regulating." This means publishing the *Time* and *TimeScale* interactions.
RQ2	*SCon* shall publish the allied and enemy units involved in the exercise with the *AggregateEntity* RPR ObjectClass.
RQ4	*SCon* shall publish the *Drawings* and *DrawingLayers* that are present in its exercise.
RQ5	*SCon* shall publish the impact of munitions from units owned by it with the *MunitionDetonation RPR InteractionClass.*
RQ6	*SCon* shall subscribe to *MunitionDetonation* interactions sent by *STat*. Damaging any of its entities that are within the radius of the detonation.
RQ7	*SCon* shall subscribe to the *AggregateEntity* objects published by *STat*, representing them in the 2D terrain view with symbols corresponding to its *EntityType*.
RQ8	*SCon* shall be able to acquire and release the ownership of units (*AggregateEntity* RPR object) from/to *STat* using the RPR *TransferControl* interaction class.
RQ9	*STat* shall be "Time Constrained," meaning it shall subscribe to the *Time* and *TimeScale* interactions and change its date/time and simulation speed accordingly.
RQ10	*STat* shall subscribe to the *AggregateEntity* objects published by *SCon*, representing them in the 2D terrain view with symbols corresponding to its *EntityType* and in the 3D view as either blue or red objects depending on whether they are a friend or an enemy.
RQ11	*STat* shall publish updates to the attributes of *AggregateEntity* objects it has ownership of.
RQ12	*STat* shall subscribe to the *Drawing* and *DrawingLayer* interactions, displaying the drawings in the 2D view of the terrain and organizing them into their respective layers.
RQ13	*STat* shall be able to acquire and release the ownership of *AggregateEntity* objects of type *SBattery* from/to *SCon*; Enabling *STat* to control the *SBattery* and perform the doctrine actions to execute the artillery mission and release the control of it when the mission is complete.
RQ14	*STat* shall publish *MunitionDetonation* interactions to indicate all detonation points of munitions launched by *SBattery*.
RQ15	*STat* shall subscribe to *MunitionDetonation* interactions and be capable of calculating and applying damage to vehicles of a given *SBattery* (that it has ownership over) in the radius of the detonations.

Table 6 – Summarized requirements of the distributed simulation with *SCon* and *STat*.

Most classes in the requirements follow their RPR FOM specification. Some, like "Time" and "Drawing," needed to have their specifications created. Those specifications were made, but their details are out of the scope of this case study and will not be shown.

7.1.3 OPM Model Development

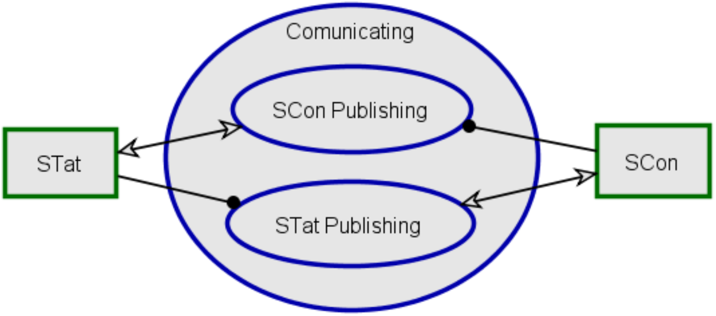
This section discusses how the subject was able to model the desired distributed simulation using OPM and what their experience was while making it.

As discussed before, the conceptual model that needs to be developed should contain the specifications from the run-time requirements of Table 6. Thus, the subject was instructed to learn OPM and model the following behavior.

- STat and SCon need to be able to send and receive data between them.
- SCon needs to send objects to STat of types *AggregateEntity*, *Drawing* and *DrawingLayer*.
- STat needs to receive objects from SCon of types *AggregateEntity*, *Drawing* and *DrawingLayer*.
- SCon needs to send interactions to STat of types *Time*, *TimeScale*, *MunitionDetonation* and *TransferControl*.
- STat needs to receive interactions from SCon of types *Time*, *TimeScale*, *MunitionDetonation* and *TransferControl*.
- STat needs to send objects to SCon of type *AggregateEntity*.
- SCon needs to receive objects from STat of type *AggregateEntity*.
- STat needs to send interactions to SCon of types *TransferControl* and *MunitonDetonation*.
- SCon needs to receive interactions from STat of types *TransferControl* and *MunitonDetonation*.

As is normal when learning a modeling language, the subject had some initial difficulties trying to come up with the OPM model of the federation after looking at example OPM models and the language specifications. However, not much later, the first OPD of the simulator was created. The highest level OPD (and accompanying OPL segment) can be seen in Figure 14; It has the initial definitions of data exchange between the simulators (represented each as an object), receiving and sending information to one another through the *Communicating* process, more specifically, through sub-processes of *Communicating* regarding their publishing behavior. The *SCon Publish* is handled by *SCon* and affects (sends data to) *STat*, and the *STat Publishing* is handled by *STat* and affects (sends data to) *SCon*.

After defining the high-level view shown in Figure 14. The subject decided to expand on each of the elements of the main OPD (except *Communicating*) with their own specific

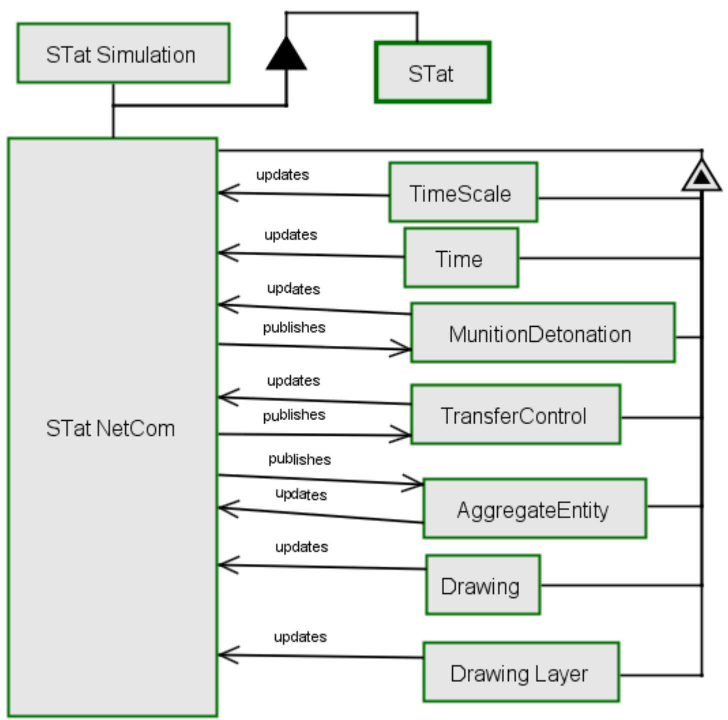


STat handles STat Publishing.
SCon handles SCon Publishing.
Communicating consists of STat Publishing and SCon Publishing.
SCon Publishing affects STat .
STat Publishing affects SCon.

Figure 14 – Main OPD and OPL of the Subject Model.

OPDs in order to properly model the requirements. The expansion of the simulator OPD will serve as an isolated view of how each simulator is structured and its relation to each of the classes that will be communicated between them. The expansion of the publishing processes will provide a complete view of the publishing behavior of one simulator and the subscribing behavior of the other.

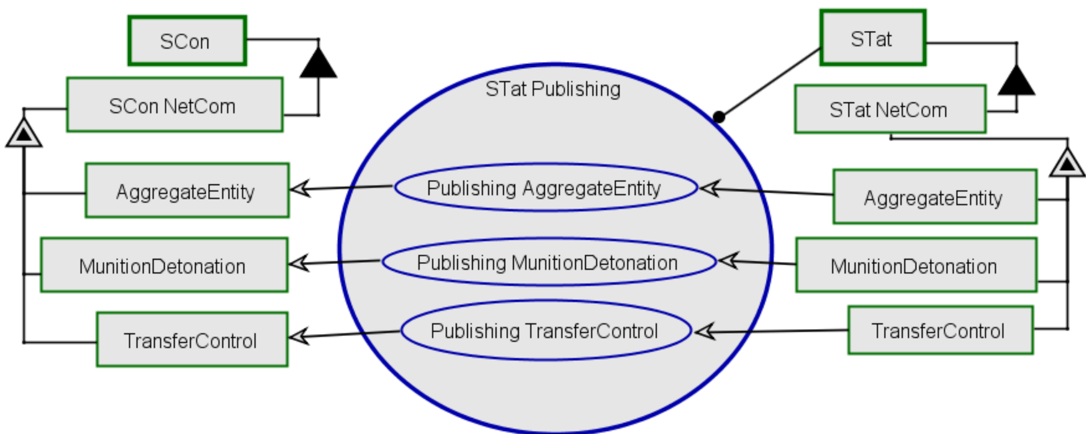
Starting with the expansion of *STat*, Figure 15 shows the subjects STat centered OPD. It has *STat* consisting of two objects, *STat Simulation* representing the internal simulation of STat and *STat NetCom* representing the network communication scripts of STat. The network communication object exhibits the definitions of all network objects/interactions, as well as STat's relation to each class using tagged associations indicating the *STat* either *publishes* the class or the class *updates STat*. This diagram/segment relates to requirements 9 to 15, which contain all behavior related to *STat*.



STat consists of STat NetCom and STat Simulation.
STat NetCom exhibits MunitionDetonation, TransferControl, AggregateEntity, Drawing, Drawing Layer, Time and TimeScale.
MunitionDetonation updates STat NetCom.
TransferControl updates STat NetCom.
AggregateEntity updates STat NetCom.
Drawing updates STat NetCom.
DrawingLayer updates STat NetCom.
Time updates STat NetCom.
TimeScale updates STat NetCom.
STat NetCom publishes AggregateEntity.
STat NetCom publishes MunitionDetonation.
STat NetCom publishes TransferControl.

Figure 15 – STat Centered OPD and OPL of the Subject Model.

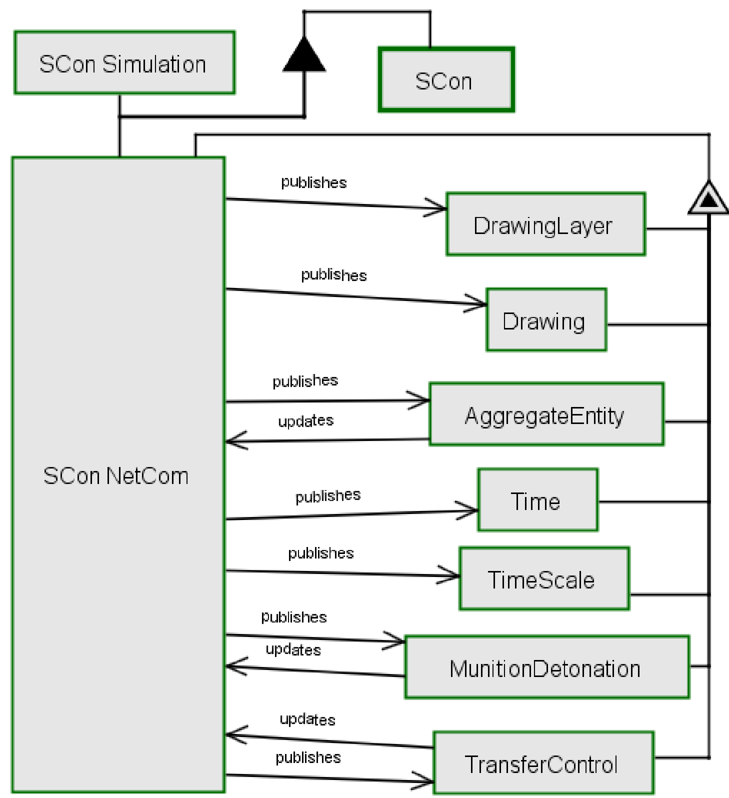
The next part of the model that will be discussed is the expanded *STat Publishing* process OPD/OPL segment in Figure 16. The main process, *STat Publishing*, contains three sub-processes responsible for receiving the published data from *STat* and creating/updating its corresponding class in *SCon* using the result/consumption link. This diagram/segment relates to RQ6, RQ7, RQ8, RQ11, RQ13, and RQ14; these requirements contain all publishing behavior of *STat* and all subscribing behavior of *SCon*.



STat consists of STat NetCom.
STat NetCom exhibits AggregateEntity, MunitionDetonation and TransferControl.
STat handles STat Publishing.
SCon consists of SCon NetCom.
SCon NetCom exhibits AggregateEntity, MunitionDetonation and TransferControl.
STat Publishing consists of Publishing AggregateEntity, Publishing MunitionDetonation and Publishing TransferControl.
Publishing AggregateEntity consumes AggregateEntity.
Publishing AggregateEntity yields AggregateEntity.
Publishing MunitionDetonation consumes MunitionDetonation.
Publishing MunitionDetonation yields MunitionDetonation.
Publishing TransferControl consumes TransferControl.
Publishing TransferControl yields TransferControl.

Figure 16 – STat Publishing centered OPD and OPL of the Subject Model.

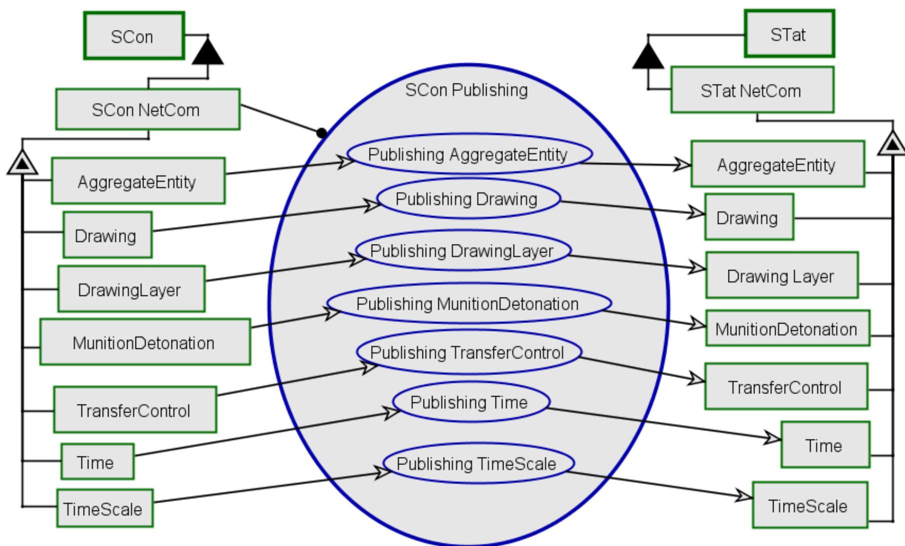
Similarly to how *STat* was expanded, the *SCon* centered OPD from Figure 17 contains the same overall structure as the one from Figure 15, with the main difference being changing *STat* to *SCon* and adjusting the tagged associations of *SCon NetCom* and the classes to reflect RQ1-RQ7.



SCon consists of SCon NetCom and SCon Simulation.
SCon NetCom exhibits MunitionDetonation, TransferControl, AggregateEntity, Drawing, Drawing Layer, textObjectFont and TimeScale.
MunitionDetonation updates SCon NetCom.
TransferControl updates SCon NetCom.
AggregateEntity updates SCon NetCom.
SCon NetCom publishes AggregateEntity.
SCon NetCom publishes Drawing.
SCon NetCom publishes DrawingLayer.
SCon NetCom publishes MunitionDetonation.
SCon NetCom publishes TransferControl.
SCon NetCom publishes Time.
SCon NetCom publishes TimeScale.

Figure 17 – SCon Centered OPD and OPL of the Subject Model.

The last diagram/segment of the subject's conceptual model, the expanded *SCon Publish* diagram, can be seen in Figure 18. It features the same overall structure as the one in Figure 16, featuring the classes *SCon* published and *STat* subscribes to. This diagram/segment relates to RQ1, RQ2, RQ3, RQ8, RQ9, RQ12 and RQ15; these requirements contain all publishing behavior of *STat* and all subscribing behavior of *SCon*.



SCon consists of SCon NetCom.
SCon handles SCon Publishing.
STat consists of STat NetCom.
STat NetCom exhibits AggregateEntity, Drawing, DrawingLayer, MunitionDetonation,TransferControl,Time and TimeScale.
SCon NetCom exhibits AggregateEntity, Drawing, DrawingLayer, MunitionDetonation,TransferControl,Time and TimeScale.
Publishing AggregateEntity consumes AggregateEntity.
Publishing AggregateEntity yields AggregateEntity.
Publishing Drawing consumes Drawing.
Publishing Drawing yields Drawing.
Publishing DrawingLayer consumes DrawingLayer.
Publishing DrawingLayer yields DrawingLayer.
Publishing MunitionDetonation consumes MunitionDetonation.
Publishing MunitionDetonation yields MunitionDetonation.
Publishing TransferControl consumes TransferControl.
Publishing TransferControl yields TransferControl.
Publishing Time consumes Time.
Publishing Time yields Time.
Publishing TimeScale consumes TimeScale.
Publishing TimeScale yields TimeScale.

Figure 18 – SCon Publishing centered OPD and OPL of the Subject Model.

7.1.4 OPM Model Review

This section reviews the subject's model to see if it correctly represents the requirements and how it would change if new requirements were added.

First off, regarding the completeness of the model. The simulator-specific diagrams from Figures 17 and 15 partially fulfill requirements R1 to RQ8 and RQ9 to RQ15, respectively, by providing an isolated view of the relation each simulator has with the objects/interactions. The missing information, how data goes from one simulator to another, is modeled at a high level in the diagram in Figure 14 and with higher granularity in Figures 16 and 18. Thus, the model the subject made correctly models the proposed simulation. The

subject noted that the immediate textual feedback of OPL explaining the semantics of the links between model elements greatly helped them confirm that they were modeling the structure/behavior they envisioned.

Now, judging the way the model was structured in terms of accurately describing the publish subscribing architecture, one characteristic is very apparent. When looking at the model in Figure 14 and considering publish/subscribe network architecture, it's clear that the model developed by the subject does not have explicit subscribing processes for the simulators. Rather, they receive information directly from the publishing process of the other simulator. When asked about this design choice, the subject reported that for every requirement requiring one simulator to publish a class, there was another for the other simulator to subscribe to. Thus, the subject found it appropriate to leave the subscribing implicit. This is a valid way to model a federation where everything published by one simulator is subscribed by the other. However, if a new simulator were to be added to the simulation and did not subscribe to all data published by the other two, or if any of the other two simulators did not subscribe to all data published by this new simulator, major changes would have to be made to the model.

Another point of judgment of the subject's model is how any requirement change would require changes in various diagrams. For example, to change some class behavior related to *STat*, many diagrams would need to be changed because of the behavior dictated by not having dedicated subscribing processes. If one class is no longer subscribed by *STat*, then its OPD needs to be changed, alongside *SCon's* publishing OPD. Since *SCOn* would no longer publish that class, its OPD would have to change too. These characteristics would likely be pointed out in an OPM modeling review activity, and adjustments would be made to better model the simulation. The resulting model, after adjustments, should likely look somewhat like the one presented in Subsection 5.3.2.

We want to reinforce that the subject's first OPM conceptual model succeeded in modeling the distributed simulation requirements, even if it needed some alterations in order to be more resilient to future requirement changes. Showing that it is possible and feasible for someone with no prior knowledge of OPM to conceptualize distributed simulations with it. In the next experiment, modeling with OPM will be compared with modeling the same distributed simulation with UML.

7.2 EXPERIMENT 2: MODELING COMPARISONS BETWEEN OPM AND UML

This experiment will compare conceptual models made with OPM against similarly made models in UML to showcase how the differences in structure lead to different levels of readability. This aspect is very important for the first model, as it will be seen/approved by the stakeholders alongside the annotated model in the “Modeling Review” activity in this

work's proposed methodology.

It is important to acknowledge that the readability of a conceptual model is context-dependent. This experiment aims to explore the readability of models in distributed simulations when the main reader is a non-technical person.

The readability of a model can be derived from a series of criteria. For this experiment, we will consider the following:

- Clarity of Structure: The way the model portrays the distributed simulations structure. The most important structures to be evaluated are the various ObjectClasses, InteractionClasses, and Federates of the Federation.
- Clarity of Behavior: The way the model portrays the distributed simulation's behavior. The most important behaviors to be considered are the connections of the simulators to the RTI and the publishing and subscribing of the objects and interactions.
- Scalability: The way the model grows in size when different types of elements are added. A more scalable model requires fewer changes to existing diagrams when a new element is added and can have fewer diagrams while still clearly representing the distributed simulation's structure and behavior.

As mentioned before, OPM only has one type of model, whereas modeling languages like UML have many types of diagrams that can be classified into larger groups: Behavioral Diagrams and Structural Diagrams. For this experiment, the UML modeling will consider only one diagram type for each group. The Class Diagram will represent the Structural group, and the Activity Diagram will represent the Behavioral group.

The federation that will be used for this experiment is the same as the one developed in Subsection 5.3.2. Meaning that the UML model aims to satisfy the same set of requirements as the OPM.

Before analyzing the UML model, the OPM model will be briefly presented again. A more in-depth analysis was presented in Subsection 5.3.2. This model consists of:

- A main diagram containing the connections between simulators and the middleware (Figure 19). Each of the simulators has a dedicated publishing and subscription process.
- A diagram for each simulator element of the main diagram. The simulator-specific diagrams contain all classes that the simulator publishes or subscribes to. Furthermore, these classes have tagged associations with the simulators to clarify their relation with it. These classes are either consumed by the simulator's publishing process or yielded from the subscribing process. An example of the *Alpha* simulator diagram can be seen in Figure 20.

- A diagram containing all object class definitions, including all of their attributes and relations with one another.
- A diagram containing all interaction class definitions, including all of their attributes and relations with one another.

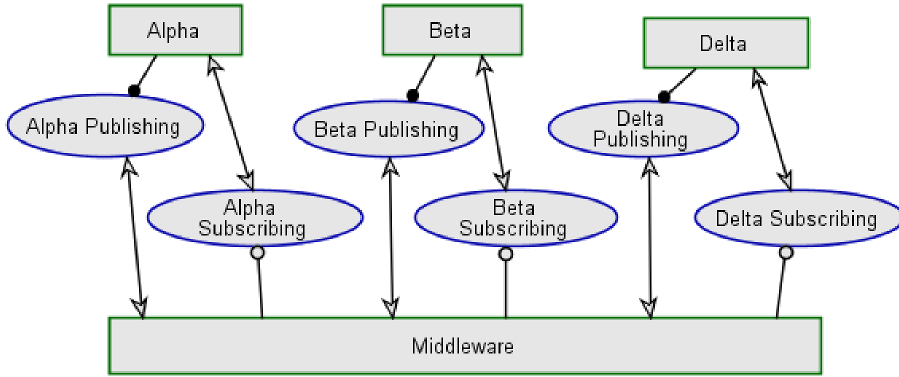


Figure 19 – Distributed Simulation OPM Diagram.

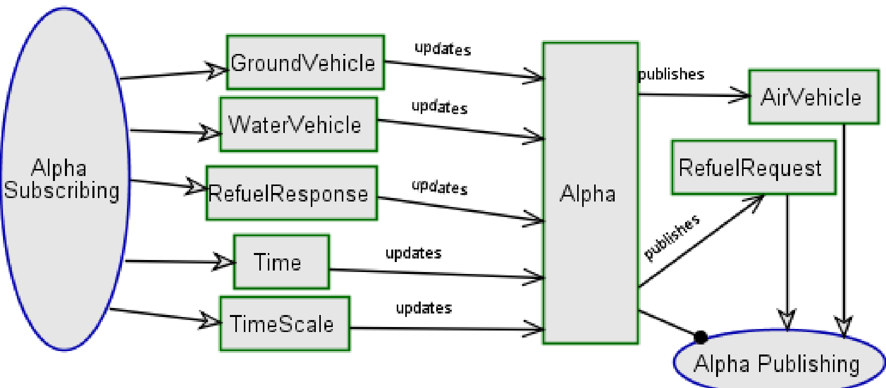


Figure 20 – Alpha Simulator OPD.

Figure 21 contains the UML class diagram. It has classes for each simulator, each object/interaction class, and a class for the middleware component. Each simulator has an association with the middleware classes to represent their connection to the RTI during integrated simulations, and the specific vehicle classes have the generalization relation with the parent *VehicleObject* class.

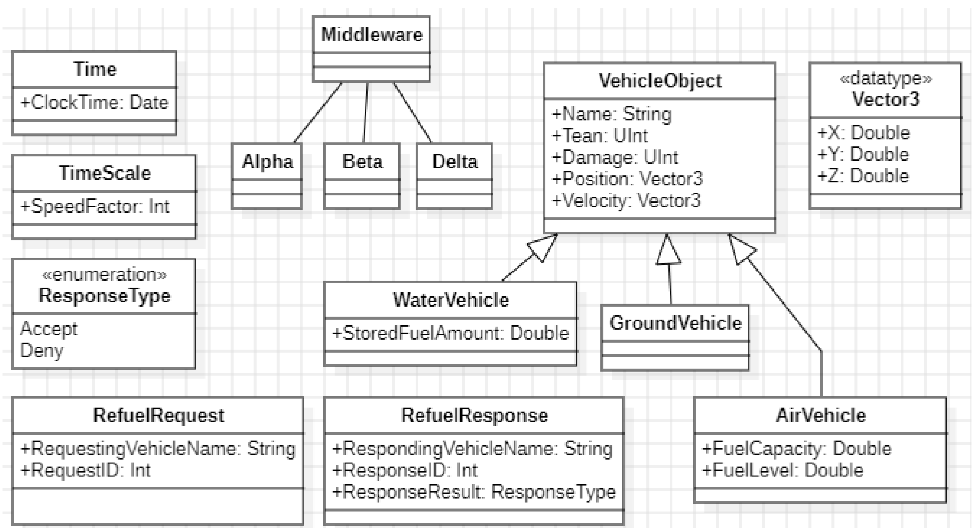


Figure 21 – UML Class Diagram for the Simulation.

To model simulation behavior, six UML Activity diagrams were made. They represent the following:

1 Publishing/Subscribing of the *AirVehicle* Object.

2 Publishing/Subscribing of the *GroundVehicle* Object.

3 Publishing/Subscribing of the *WaterVehicle* Object.

4 Publishing/Subscribing of the *Time* Interaction.

5 Publishing/Subscribing of the *TimeScale* Interaction.

6 Publishing/Subscribing of the *RefuelRequest* and *RefuelResponse* Interactions.

The first five follow a similar structure. For example, in Figure 22, an activity diagram represents the publishing of the *AirVehicle* object class. The diagram contains swimlanes to indicate what component does what actions; it also contains a fork node to represent that the Middleware sends the object class to both of the other simulators and finally, both flows join to end the diagram flow. The same idea is applied to the activity diagrams representing the publishing of the other vehicle classes and the *Time* and *TimeScale* interactions.

The last activity diagram, presented by Figure 23, covers the publishing and subscribing of the *RefuelRequest* and *RefuelResponse* interactions. The diagram does not have a swimlane for *Delta* because it neither publishes nor subscribes to any of the related

classes of the diagram. This diagram starts with the publishing of a refuel request by *Alpha*, then, that request is relayed to *Beta* by the middleware, who, in turn, responds with a refuel response, which is received by *Alpha* through the middleware. This diagram could have been split into two and followed the same structure as the others, but since these two interactions are more closely related, it does not make sense to publish a *RefuelResponse* without having received a *RefuelRequest*, we found it appropriate to join the two classes into a single diagram.

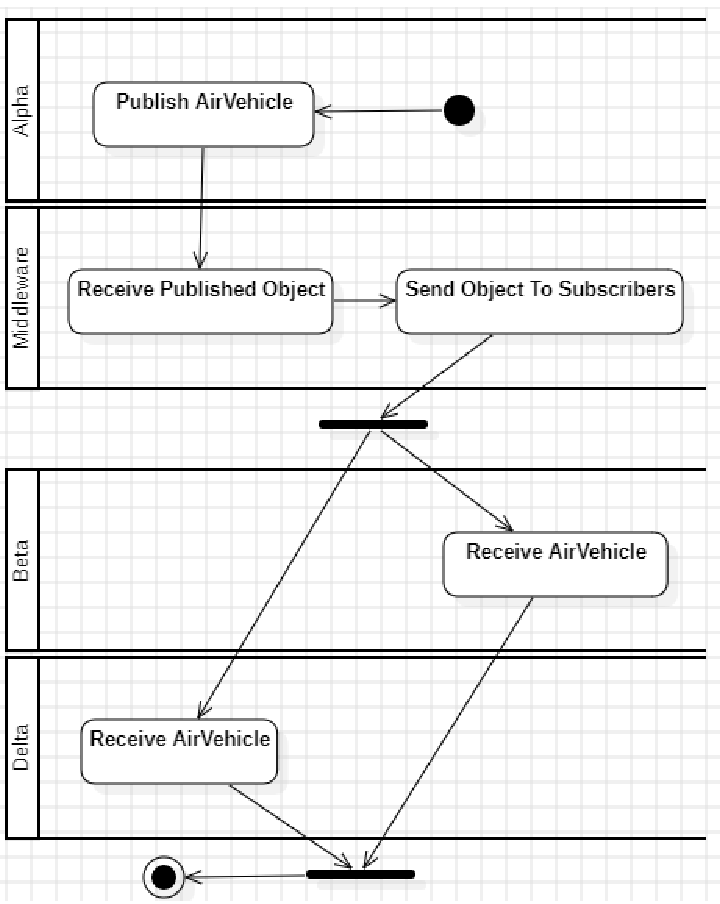


Figure 22 – UML Activity Diagram for publishing AirVehicle.

In terms of structural clarity, both models are able to portray the structures needed to solve the requirements. Both models contain all of the Object/Interaction classes and their attributes, the simulator classes, and the middleware class. However, they both also have downsides when compared to the other. UML Class diagrams do not contain textual representation of the semantical relations between the classes; OPM diagrams take up more space to represent class attributes because of the need to use the “exhibits” structural link. When considering an initial conceptual model, the textual representation of the structure of the simulation is more important than the amount of space the model takes up, so OPM has the advantage in structural clarity.

In terms of behavioral clarity, the models are able to convey the requirements correctly but do so in different ways. Analyzing Figures 22 and 23, we see that the activity

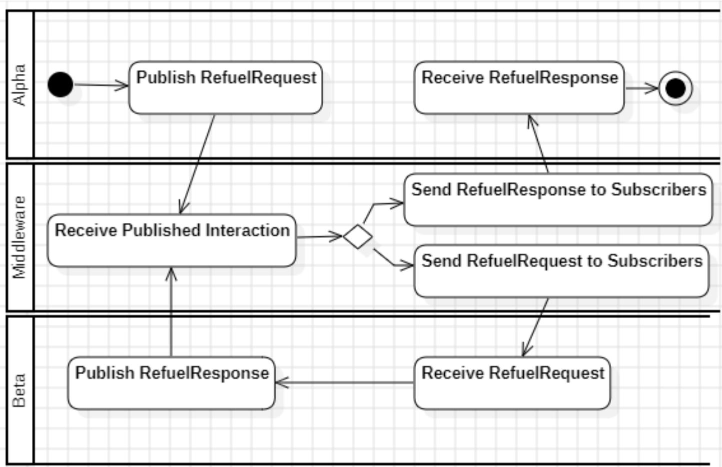


Figure 23 – UML Activity Diagram for publishing RefuelRequest/RefuelResponse.

diagrams focus on how a published class goes from one simulator to the other, with the use of the middleware swimlane. Observing Figure 20, it becomes clear that the OPM diagrams focus on the simulators themselves and what information goes in and out of them. From a stakeholder perspective, the overall process is more important than the fine details. The view presented by the simulator-specific OPDs better facilitates the understanding of the full picture of the simulation when compared to the data flow-focused view of the UML Activity diagrams. For example, to check what information *Alpha* is receiving during the simulation, it would be necessary to check all of the activity diagrams instead of just the *Alpha* OPD. On top of that, the UML activity diagrams have their own semantics (activities, control flows, swimlanes, fork/join nodes), which differ from the other UML diagrams, making it more difficult for readers not familiar with UML Activity diagrams to fully understand them. This is not an issue with OPM because it has only one diagram type and the textual representation with OPL.

To analyze the scalability of models, we consider two cases: adding a new class to the federation (meaning either an interaction or an object) and adding a new federate to the federation. In both cases, a brief description of what needs to change in each model will be given, along with an estimate of how many existing diagrams need to be changed and how many diagrams need to be created to accommodate the change. This estimate will use the Big O notation to classify the growth of the models in each case. In addition, the following two variables will be used (FedAssoc and ClassAssoc). These variables were chosen because they were the main factors in determining the number of changes required in each model in both situations that will be analyzed.

- *FedAssoc*: The number of class associations a federate has. Meaning the number of classes a federate either publishes or subscribes to.
- *ClassAssoc*: The number of federate associations a class has. Meaning the number of federates that either publish or subscribe to a class.

Adding a new Object/Interaction class to the OPM model means:

1 Creating new objects in the "Object Classes" or "Interaction Classes" OPDs to represent the class and their attributes.

2 Creating new "exhibits" links between the class and their attributes.

3 Adding the class to the simulator OPDs that have an association with it.

The number of changed diagrams equals *ClassAssoc* + 1 because *ClassAssoc* diagrams are changed in item 3, and 1 diagram is changed in items 1 and 2, resulting in O(*ClassAssoc*). The number of new diagrams that need to be created equals zero, resulting in O(0).

Adding a new Object/Interaction class to the UML model means:

1 Creating the new class and class attributes in the class diagram.

2 Creating a new Activity Diagram to model its publishing behavior, following the same structure as the others.

In this case, the number of changed diagrams is equal to 1, in accordance with item 1, resulting in O(1). The number of new diagrams that need to be created is also equal to 1, in accordance with item 2, resulting in O(1).

Adding a new simulator to the OPM model means:

1 Creating a new object in the "Federation" OPD to represent the new simulators.

2 Creating new processes to represent the new simulator's publishing/subscribing in the "Federation" OPD, with links to the middleware and the new simulator object.

3 Creating a new OPD for the simulator. Following the same structure as the others.

The number of existing diagrams that need to be changed is equal to 1, in accordance with items 1 and 2. Only the "Federation" OPD needs to be changed, resulting in O(1). The number of new diagrams that need to be created is also equal to one, according to item 3, resulting in O(1).

Adding a new simulator to the UML model means:

1 Creating a new class in the class diagram to represent it.

2 Creating new swimlanes in the activity diagram of classes that the simulator associates with to model the new simulator's publish/subscribe behavior.

The number of existing diagrams that need to be changed is equal to *FedAssoc* + 1 because *FedAssoc* are changed in item 2, and 1 diagram is changed in item 1, resulting in O(*FedAssoc*). The number of diagrams that need to be created is equal to zero, resulting in O(0).

To better visualize the differences in scalability, we will add the following class to the federation to see how many changes must be made to accommodate it.

New class specification:

> **HybridVehicle:** An object class representing a vehicle that can operate on the ground and the water. This object class inherits all attributes from the VehicleObject class and has the following attribute
>
> - CurrentTerrainType: Enum Type. Contains what terrain the vehicle is currently over. Can be “SturdyGround”, “MuddyGround” or “Water”.

Simulators *Beta* and *Delta* will publish the *HybridVehicle* object class, and all three simulators will subscribe to it. This means that the *ClassAssoc* variable is equal to three.

For OPM, we estimate *ClassAssoc* + 1 diagrams to be changed and none to be created. In practice, all three of the simulator OPDs were altered to add the new object alongside the object classes OPD, resulting in the change of four existing diagrams, which is *ClassAssoc* + 1 and can be simplified to O(*ClassAssoc*). No new diagrams were created, so the estimate was correct as well. Figure 24 shows the altered *Beta* simulator OPD after the addition of *HybridVehicle*. This new class appears as two objects in the model; One of them is yielded from the *Beta Subscribing* process and has a tagged “updates” association with the simulator object, and the other is consumed by the *Beta Publishing* process and has a tagged “publishes” association directed at it from the Beta simulator object.

Figure 25 shows part of the OPD containing object classes after the addition of *HybridVehicle*. It features the new class as one object, which has a generalization structural link with its parent object *VehicleObject*, an “exhibits” structural link with an object representing the *CurrentTerrainType* attribute of the class. Since the *CurrentTerrainType* attribute features a new type (*TerrainTypeEnum*), another object with the enum’s name and possible values (modeled as different object states) was also added to the diagram.

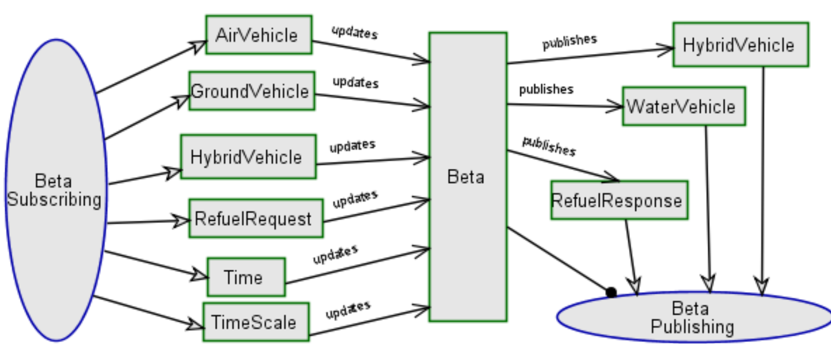


Figure 24 – Beta Simulator OPD after the addition of the *HybridVehicle* Class.

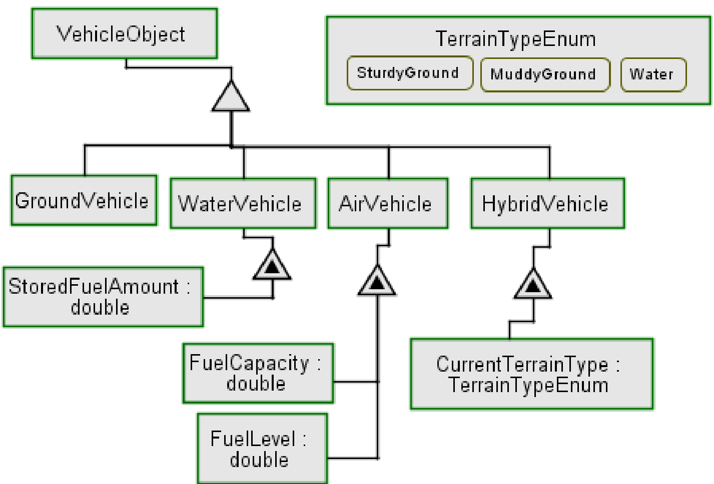


Figure 25 – Part of the Object Classes OPD after the addition of the *HybridVehicle* Class.

For UML, we estimate O(1) diagrams to be changed and O(1) diagrams to be created. The class diagram must have the new object class, with its attributes and the new enum type, resulting in a change in one diagram. A new activity diagram must be created to showcase the publishing/subscribing of the new object class. The first half of this new diagram can be seen in Figure 26; it contains swimlanes for every simulator that publishes the class and two Decision/Merge nodes indicating that either of the simulators can publish the class at any given time. The second half of the diagram was not shown since it follows the same structure as the second half of the activity diagram in Figure 22.

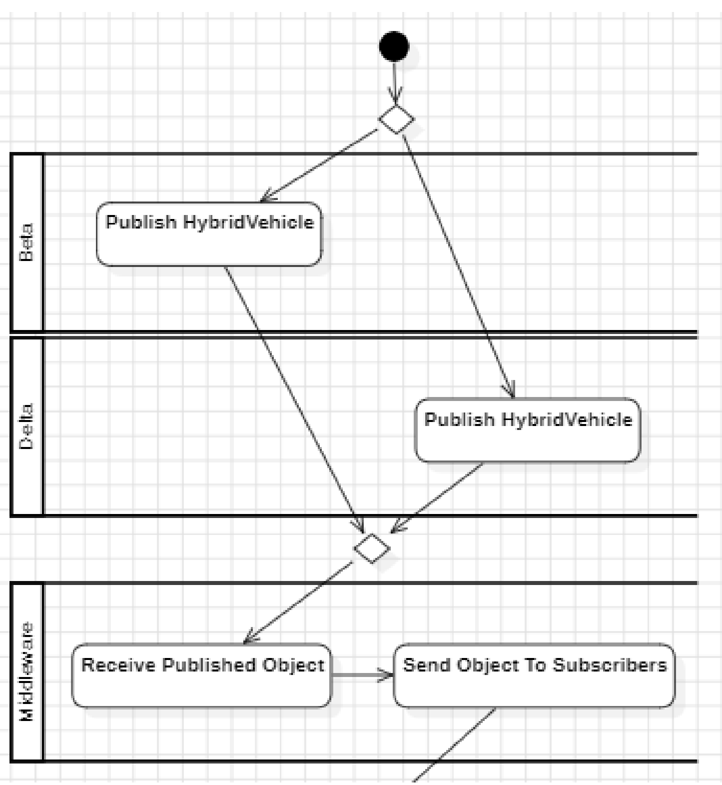


Figure 26 – First Half of the Activity Diagram for *HybridVehicle*.

Now, we will add a new federate to both models to see how each has to grow. Simulator *Gamma* shall:

- Publish *AirVehicle*.
- Publish *Time* and *TimeScale*
- Subscribe to *AirVehicle*,*GroundVehicle* and *HybridVehicle*.

In total, *Gamma* associates with six different classes, resulting in *FedAssoc* to be equal to six.

In OPM, we estimate only one diagram to be altered and one to be created. This is correct since only the "federation" OPD was changed, and a new OPD for *Gamma* was added. This new OPD can be seen in Figure 27. It follows the same structure as the other simulator OPDs, having its corresponding publishing and subscribing respectively consuming and yielding the classes that it publishes/subscribes to.

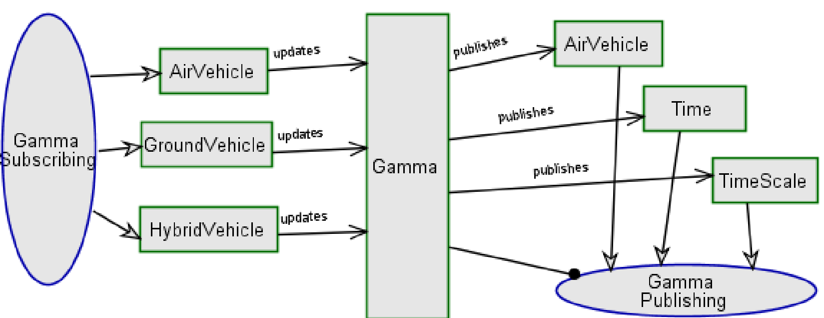


Figure 27 – Gamma Simulator OPD.

In UML, we estimate *FedAssoc*+1 diagrams to be changed and none to be created. For this example, the activity diagrams for *AirVehicle*, *Time*, *TimeScale*, *GroundVehicle* and *HybridVehicle* were changed. Additionally, the class diagram had one new class added to represent *Gamma*, resulting in seven diagram changes, which is accurate to the estimate. Figure 28 shows the second half of the *GroundVehicle* activity diagram. It contains the changes that were necessary to represent Gamma's subscribing of it, these changes involve the addition of a new swimlane and actions for the subscribing of the class.

Analyzing these statements about how the models grow. It is clear that in both languages, the models will have to grow. Frame 7 summarizes the comparisons in both cases. When adding new classes, OPM requires changing *ClassAssoc* diagrams and creating no new ones; UML requires one diagram to be created and one to be changed. Thus, OPM requires more diagrams to be changed than UML but requires no new ones to be added, unlike UML, which needs fewer changes to existing diagrams but adds one new diagram for each class.

When adding new simulators, the situation flips; OPM requires fewer changes to existing diagrams but needs a new one to be created, whereas UML requires *FedAssoc*

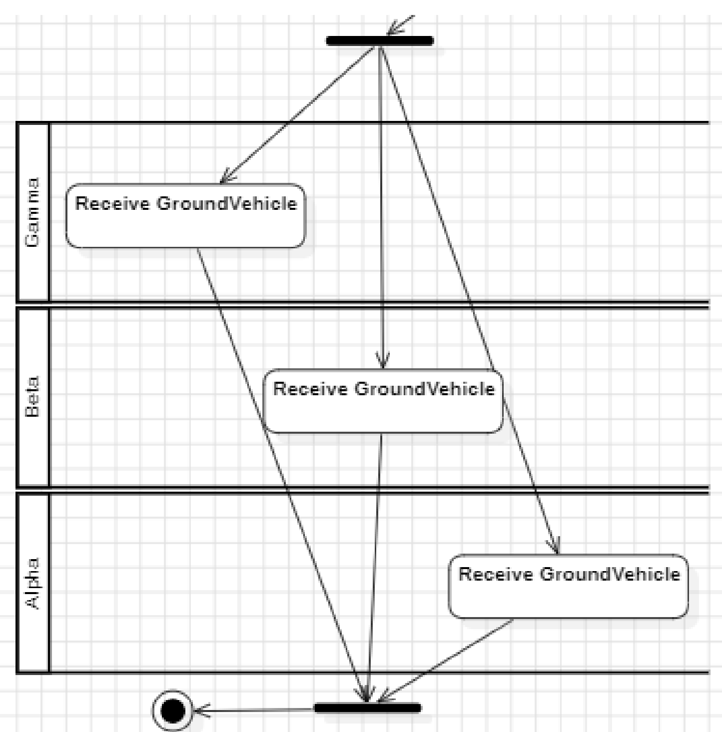


Figure 28 – Second Half of the GroundVehicle Activity Diagram after adding Gamma.

existing diagrams to be changed and no new ones. This means that in a simulation with many federates, UML has the advantage in terms of ease of finding information.

	UML	OPM
Adding a new Class (Changes to existing Diagrams)	O(1)	O(*ClassAssoc*)
Adding a new Class (New Diagrams)	O(1)	O(0)
Adding a new Federate (Changes to existing Diagrams)	O(*FedAssoc*)	O(1)
Adding a new Federate (New Diagrams)	O(0)	O(1)

Table 7 – Comparative frame between the scalability of the OPM and UML models.

Considering the fact that in HLA simulations, the number of classes is much higher than the number of federates, meaning that the average *FedAssoc* is significantly higher than the average *ClassAssoc*. OPM models following the presented structure will have fewer total diagrams when compared to the presented UML models because the number of OPM diagrams depends on the number of federates. Furthermore, in the case that a new federate is added, the cost of adjusting the UML model will be significantly higher than when a new class is added to the OPM model. Taking all of this into account, the developed OPM structure has better scalability than the developed UML structure when modeling publish/subscribe distributed simulations.

In conclusion to the experiment, both languages can represent the structure of the

simulation. Similarly, OPM can represent the behavior of a distributed simulation in a more readable way due to how the diagrams are organized, and OPM also has better scalability when it comes to modeling a bigger federation. Thus, OPM is more appropriate for the initial conceptual modeling language that is presented to stakeholders and iterated upon in modeling reviews.

7.3 EXPERIMENT 3: PROPOSED MDA METHODOLOGY APPLIED TO AN EXAMPLE

This experiment involves using the developed implementation of the methodology (explained in further detail in Chapter 6) to verify if the proposed methodology contains all of the necessary steps in order to transform an OPM model into usable and efficient HLA code. The federation used for this experiment is the same as the one developed in Section 5.3.2.

A GUI program was developed to facilitate the use of the developed implementation for transforming OPM into HLA Code. The main screen of the GUI can be seen in Figure 29. It features buttons to perform the various operations of the methodology, a button to set configurations, and a button that briefly explains the function of every other button.

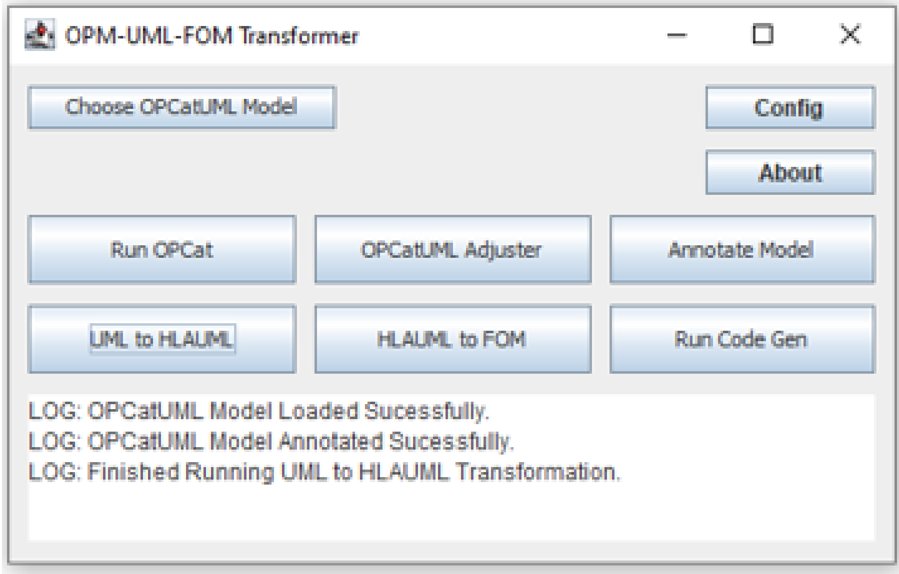


Figure 29 – Main screen of developed GUI Program.

Figure 30 contains a representation of the buttons of the developed GUI on its left side and arrows coming out of the buttons and into activities to indicate that the button performs that set of activities.

The first step before all of the transformations is to develop the OPM model that represents the desired simulation requirements. To perform OPM modeling, the user should press the “Run OPCAT” button to launch the OPCat program. When the model is com-

Figure 30 – UI Button to Proposed Process equivalence.

pleted, the user can execute the OPM-UML transformation already integrated into OPCat to generate a UML model that reflects the one modeled. Figure 31 shows the OPCat modeling environment with the OPD, OPL, UML generation, and OPD building blocks.

The UML model can be imported into the GUI program using the “Choose OPCatUML model” button. After selecting the input model, the user can run the OPCatUML Adjuster program by pressing its designated button. Figure 32 shows the model file after it has been adjusted. It features the various classes, datatypes, and associations present in the OPM model.

Executing the adjuster enables the user to annotate the model. When the respective button is pressed, a different screen is shown that enables the user to annotate the model. This screen can be seen in Figure 34. It features two drop-down elements, the first to select which class to annotate and the second to select what HLA role to give to the class. With both selected, the user may press the “Annotate Class” button to confirm the assignment of the selected HLA role to the selected class. After annotating all desired classes. The user can press the “Save and Confirm” button to return to the main screen.

Figure 33 shows the model file after the manual annotation activity. The classes

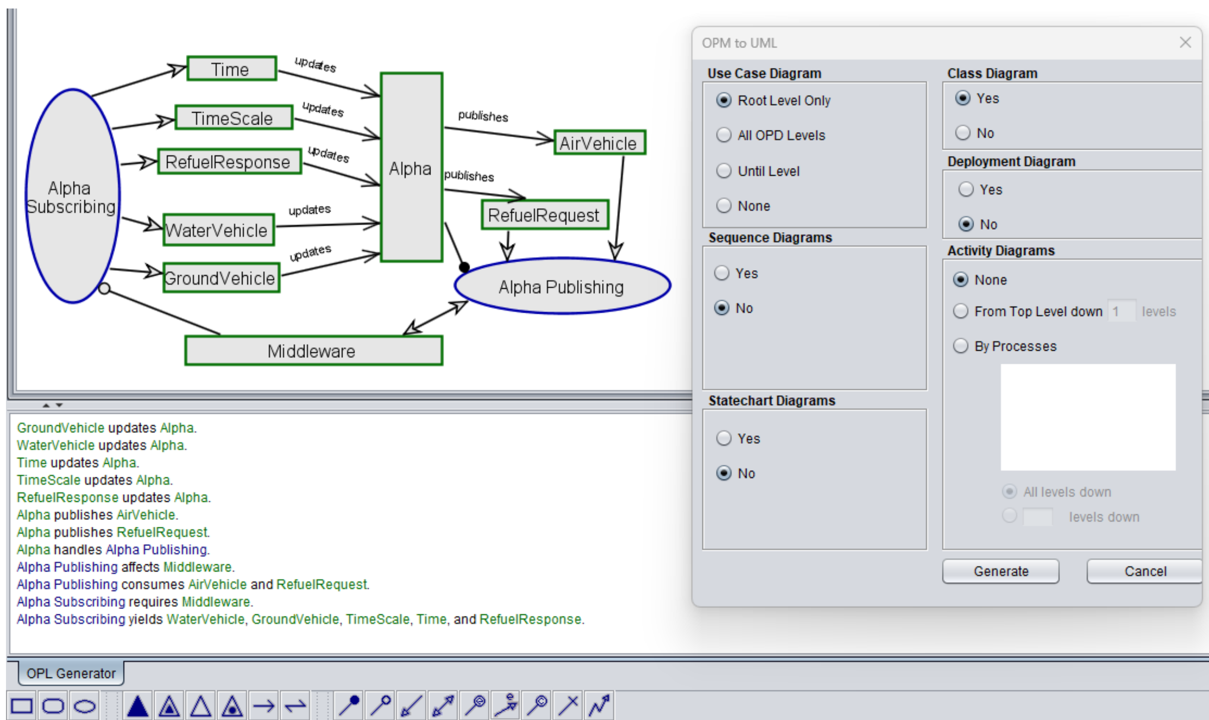


Figure 31 – OPCat Environment and the integrated OPM-UML transformer.

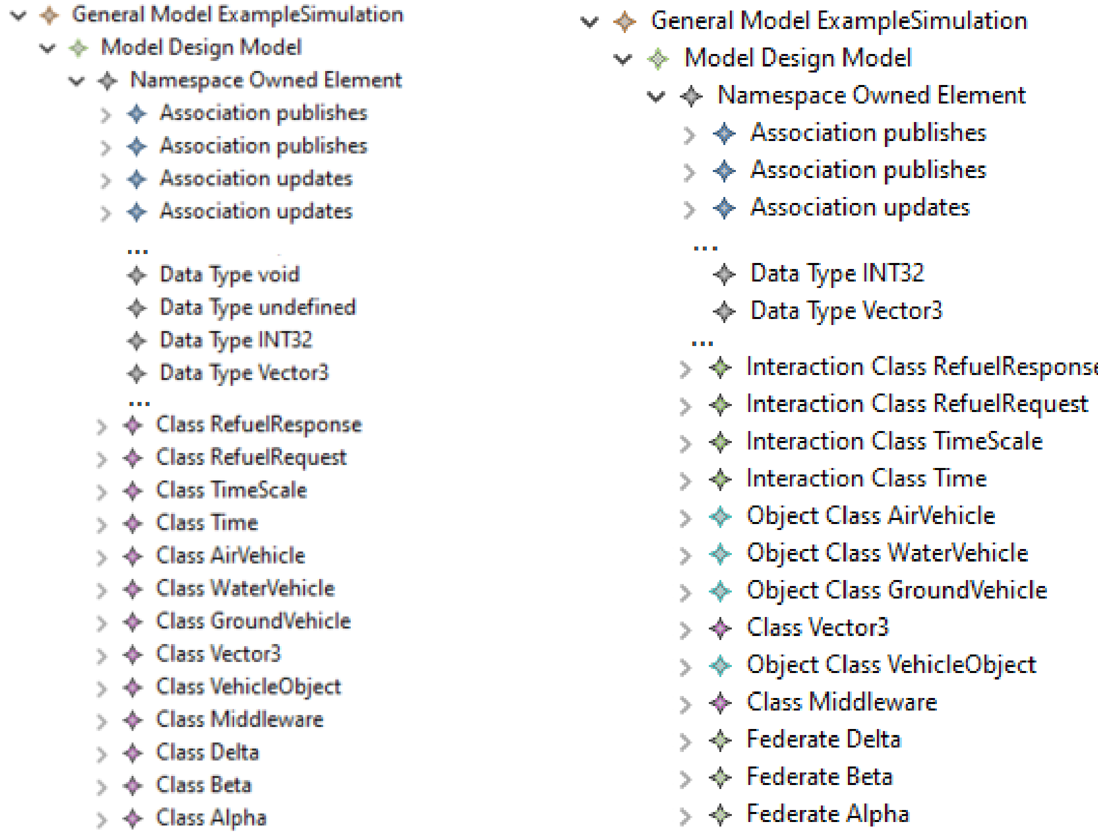


Figure 32 – UML model after the OPCatUML Adjuster program.

Figure 33 – UML Model after the Annotation Process.

have new types to specify their role in the federation according to what was inputted in

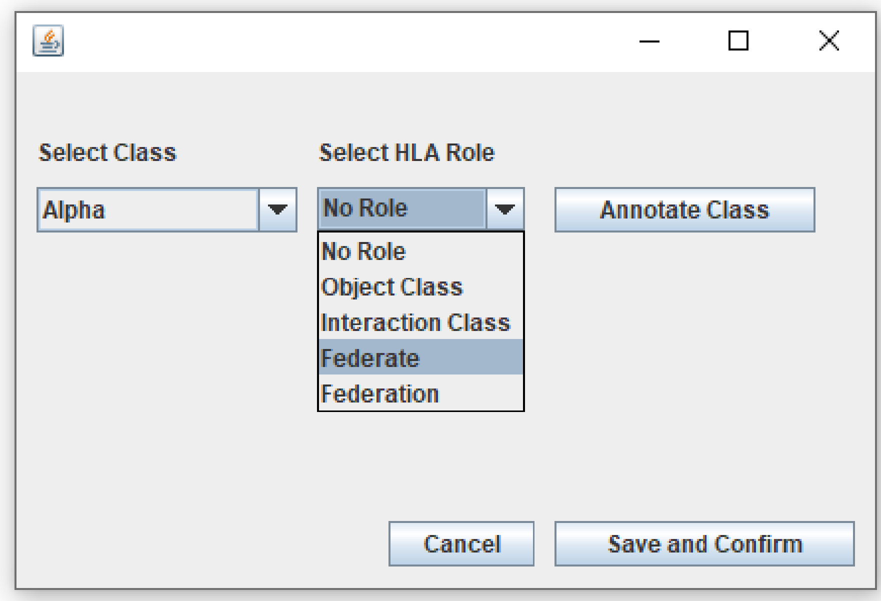


Figure 34 – Annotation Helper Screen of GUI Program.

the annotation screen. In this example, the classes representing the simulators were annotated as *Federate*; the classes representing the proposed objects were annotated as *Object Class*; and the classes representing interactions were annotated as *Interaction Class*.

Having completed the manual annotation process, other operations of the main screen of the OPM-UML-FOM Transformer can be executed. The next one is the "UML to HLAUML" button that performs the "UML-HLAUML Transformation" over the annotated UML model, resulting in a set of HLAUML federate diagrams. Next, the

The "HLAUML to FOM" button performs the HLAUML-FOM model-to-model transformation, resulting in a set of HLA object models for the federation and its federates.

Finally, the "Run Code Gen" button opens the GUI of the code generator, as shown in Figure 35. There, the user can decide, among other things, which classes of the FOM to include in the generated code.

In the last step, the Code Generator can use the FOM file to generate the final C++ code. The contents of the generated code for this example are discussed in Section 7.4.

In conclusion, this part of the experiment demonstrated the methodology's integrity in achieving its objectives, the transformation of an OPM model into usable HLA code, and the capacity to incorporate all of its activities into an efficient and easy-to-use tool.

7.4 EXPERIMENT 4: ANALYSING AND TESTING GENERATED CODE

This experiment is separated into two parts. The first part, in Subsection 7.4.1 uses the source code generated by the previous experiment to showcase how to use it to con-

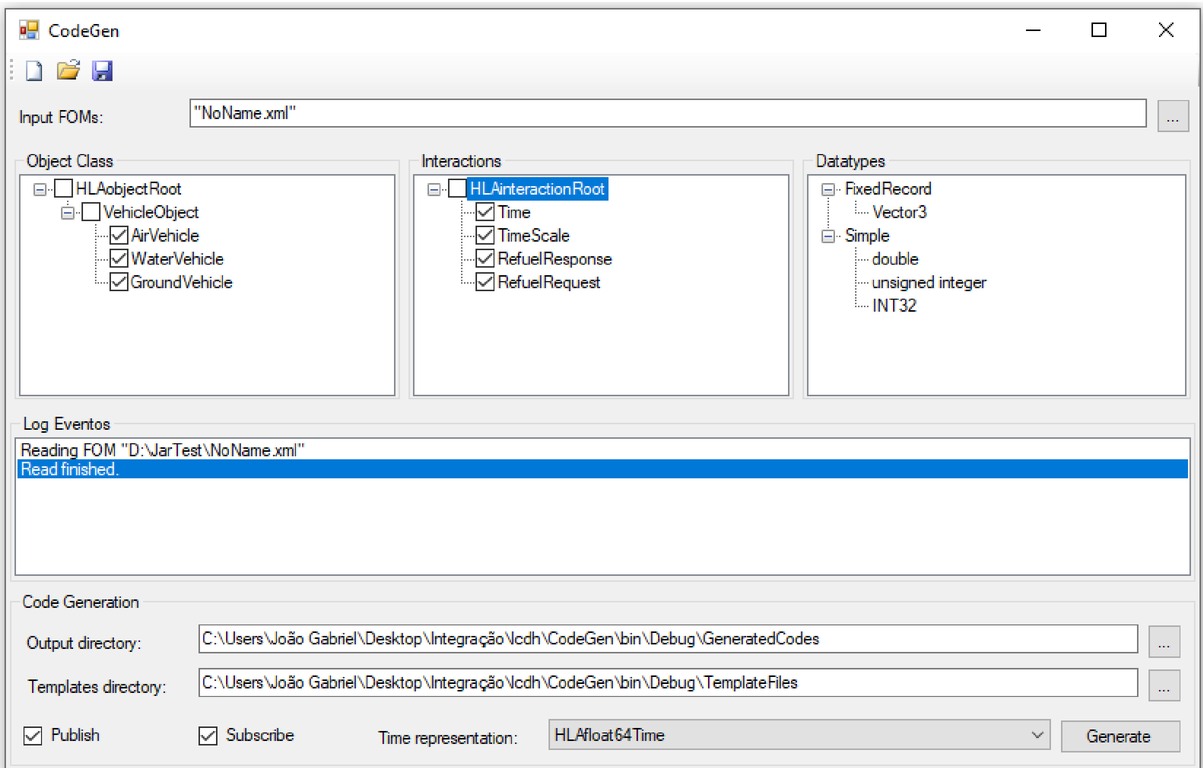


Figure 35 – Code Generator UI after FOM insertion.

nect to an RTI, create objects, and listen for remote object creations. The second part, in Subsection 7.4.2, compares the code generated from a real distributed simulation's FOM from this work's code generator against a commercial code generator using the Halstead code metrics.

7.4.1 HLA Code Tests

Firstly, a program will be made to connect to an RTI. Then, multiple RTI implementations will be used in order to check if the code can connect to them. The RTIs used will be use Pitch RTI and MAK RTI.

Using the automatically created *FederateManager* and *HlaSettings* classes, connecting to an RTI with the output code is just a matter of creating an instance of *FederateManager* and *HlaSettings* with appropriate parameters and calling *FederateManager.Connect()*. Listing 7.1 shows an example of how to do that. It connects to the RTI and then stays on a while loop to remain connected.

Listing 7.1: Main Function that uses the generated code to connect to a RTI

```
int main()
{
    FederateManager fedManager;
    std::string ip = "localhost"; std::string port = "8989";
    std::string federationName = "MyFederation"; std::string fomName = "ExampleFederation.xml";
    std::string federateName = "Alpha"; std::string federateType = "Example";
    bool isTimeRegulating = false;  bool isTimeConstrained = true;
    double lookAhead = 1;   double stepSize = 1;
    HlaSettingsPtr rtiSettings =
    std::make_shared<HlaSettings>(
        HlaSettings(
            "crcAddress=" + ip + ":" + port,
            federationName,{fomName},
            federateName, federateType,
            isTimeRegulating, isTimeConstrained,
            lookAhead, stepSize
        ));
    fedManager.Connect(rtiSettings);
    while (true)
    {
    }
}
```

Figures 36 and 37 show the result in the mentioned RTI implementations. The program can successfully create and join a federation.

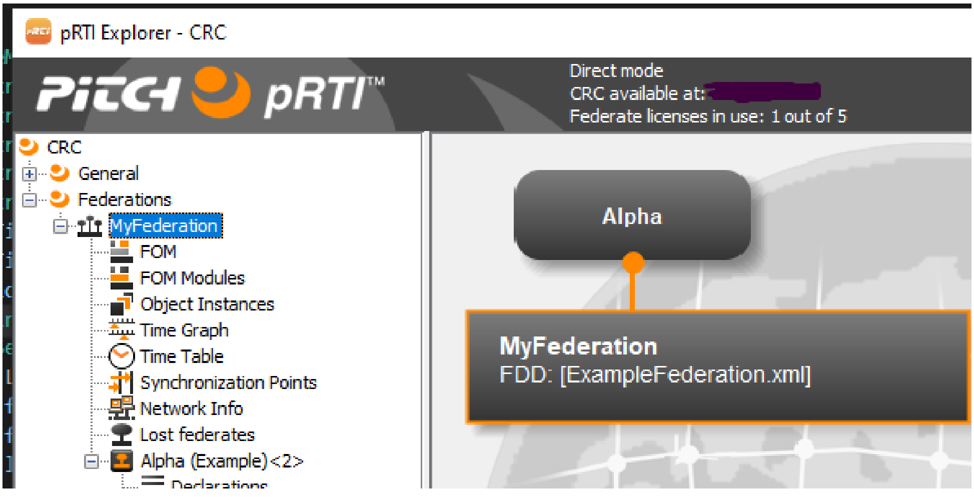


Figure 36 – Generated Code Connecting to Pitch RTI.

The second test will check if the federates can publish and subscribe to objects. To execute this test, three instances of the code will be executed simultaneously, representing the three simulators of the proposed federation. Each instance will have a different federate name and will publish different vehicles. Listing 7.2 shows the creation of vehicle object classes based on which the simulator is being executed. The code for the Alpha simulator publishes two *AirVehicle* objects, the code for Beta publishes one *WaterVehicle*, and the one for Delta publishes three *GroundVehicles*. Note that *HlaAirVehiclePtr*, *HlaAirVehicleManager*, *HlaGroundVehiclePtr*, *HlaGroundVehicleManager*, *HlaWaterVehiclePtr* and *HlaWaterVehicleManager* were automatically created by the code generator.

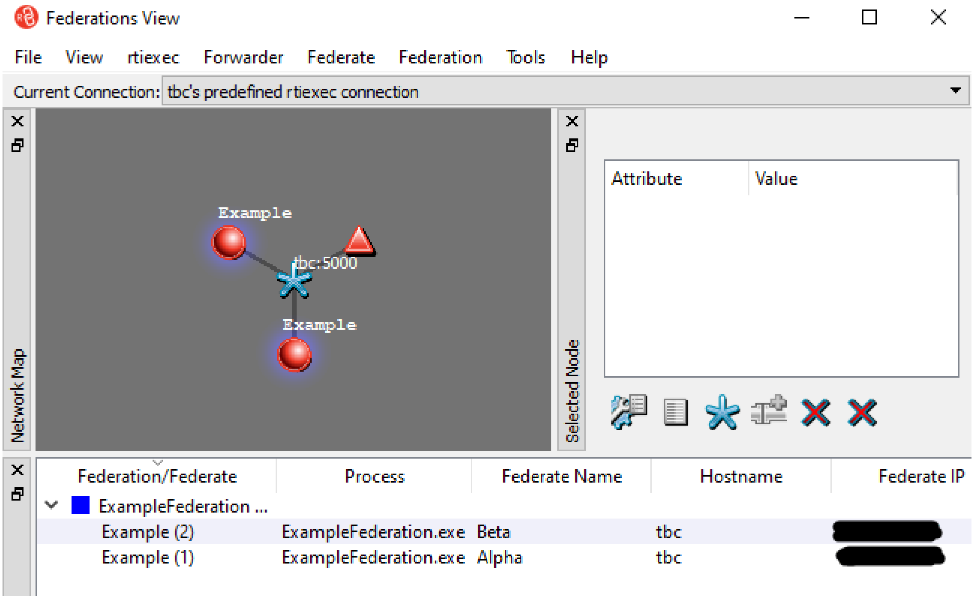


Figure 37 – Generated Code Connecting to MAK RTI.

Listing 7.2: Generated code that publishes objects based on what simulator is being executed.

```
//------------------Alpha Simulator------------------------//
    HlaAirVehiclePtr av1 = HlaAirVehicleManager.CreateLocalObject("Alpha_AirVehicle_1");
    HlaAirVehiclePtr av2 = HlaAirVehicleManager.CreateLocalObject("Alpha_AirVehicle_2");
//------------------Beta Simulator--------------------------//
    HlaWaterVehiclePtr wv1 = HlaWaterVehicleManager.CreateLocalObject("Beta_WaterVehicle_1");
//------------------Delta Simulator-------------------------//
    HlaGroundVehiclePtr gv1 = HlaGroundVehicleManager.CreateLocalObject("Delta_GroundVehicle_1");
    HlaGroundVehiclePtr gv2 = HlaGroundVehicleManager.CreateLocalObject("Delta_GroundVehicle_2");
    HlaGroundVehiclePtr gv3 = HlaGroundVehicleManager.CreateLocalObject("Delta_GroundVehicle_3");
```

To subscribe to these published objects, the developer must create classes extending the object listener class (created by the code generator). Listing 7.3 shows the header file of a manually created class that extends *HlaWaterVehicleListener* and overrides the methods provided by it to have its own implementation. The implementations of the methods simply print the information they receive.

Listing 7.3: Header file of extended HlaWaterVehicleListener Class.

```
class MyWaterVehicleListener: public HlaWaterVehicleListener{
  void WaterVehicleDiscovered(HlaWaterVehiclePtr instance)   override;

  void WaterVehicleRemoved(HlaWaterVehiclePtr instance)  override;

  void WaterVehicleUpdated(HlaWaterVehiclePtr instance, std::set<HlaWaterVehicleAttribute>
    const& attributes) override;

  void NameUpdated(HlaWaterVehiclePtr instance, std::string value) override;

  void PositionUpdated(HlaWaterVehiclePtr instance, Vector3 value) override;

  void DamageUpdated(HlaWaterVehiclePtr instance, unsigned_integer value) override;

  void SpeedUpdated(HlaWaterVehiclePtr instance, Vector3 value) override;

  void TeamUpdated(HlaWaterVehiclePtr instance, unsigned_integer value) override;

  void StoredFuelAmountUpdated(HlaWaterVehiclePtr instance, double value) override;
};
```

Having implemented listener classes for all three object classes, they must be assigned to their respective object class managers. Listing 7.4 shows how to assign the extended *WaterVehicle* listener class to the *HlaWaterVehicleManager*.

Listing 7.4: Assigning the extended listener to the manager class.

```
HlaWaterVehicleManager& wvManager = fedManager.GetObjectManager().GetWaterVehicleManager();

wvManager.SetWaterVehicleListener(std::make_shared<MyWaterVehicleListener>());
wvManager.SetEnabled(true);
```

Considering that every federate has all three vehicle object classes implemented and assigned to their managers. Figure 38 shows the result from executing the publishing code. It shows that:

- *Alpha* received one *WaterVehicle* instance and three *GroundVehicle* instances.
- *Beta* received two *AirVehicle* instances and three *GroundVehicle* instances.
- *Delta* received two *AirVehicle* instances and one *WaterVehicle* instance.

This result is exactly what was expected when considering which classes each simulator subscribes to and the code from Listing 7.2.

In conclusion, this part of the experiment presented how to use the generated code to connect to an RTI, create objects, and listen to updates about objectsshowing that the automatically generated code can be easily used to start federate development.

```
Hello World
I'm Alpha
Discovered New WaterVehicle Instance Named Beta_WaterVehicle_1
Discovered New GroundVehicle Instance Named Delta_GroundVehicle_1
Discovered New GroundVehicle Instance Named Delta_GroundVehicle_2
Discovered New GroundVehicle Instance Named Delta_GroundVehicle_3

Hello World
I'm Beta
Discovered New AirVehicle Instance Named Alpha_AirVehicle_1
Discovered New AirVehicle Instance Named Alpha_AirVehicle_2
Discovered New GroundVehicle Instance Named Delta_GroundVehicle_1
Discovered New GroundVehicle Instance Named Delta_GroundVehicle_2
Discovered New GroundVehicle Instance Named Delta_GroundVehicle_3

Hello World
I'm Delta
Discovered New WaterVehicle Instance Named Beta_WaterVehicle_1
Discovered New AirVehicle Instance Named Alpha_AirVehicle_1
Discovered New AirVehicle Instance Named Alpha_AirVehicle_2
```

Figure 38 – Publish/Subscribe ObjectClass test for generated code.

7.4.2 HLA Code Metrics Analysis

Now that the code generated from the federation that had this works methodology applied to it (in Sections 5.3.2 and 7.3). We will compare the HLA code generated by this implementation for a real simulator with the code generated by Pitch Developer Studio. Both of them received the same FOM.

Many metrics exist to evaluate generated code, such as total lines of code (LOC), number of methods, number of method parameters, number of types, etc. The metrics that will be used for this experiment are the Halstead metrics (HALSTEAD, 1977). These metrics use the number of operands and operators from the source code to evaluate various code complexity characteristics, such as Volume, Difficulty, Effort, and Time spent. These metrics are language and paradigm-independent. These metrics have been criticized for the ambiguous definition of operand and operator. But, currently, they are the most complete software complexity analysis metrics and correlate highly with other software metrics (COIMBRA; RESENDE; TERRA, 2018).

Halstead metrics can be separated into two groups: the core parameters and the derived measures. The Halstead core parameters are as follows:

- n_1: Number of distinct operators in a software.
- n_2: Number of distinct operands in a software.
- N_1: Number of total operators in a software.
- N_2: Number of total operands in a software.

The Halstead derived measures are calculated based on the core parameters:

Measure	Formula	Description
n	$n_1 + n_2$	Software Vocabulary.
N	$N_1 + N_2$	Software Length.
V	$N * \log_2 n$	Software Volume.
D	$\frac{n_1}{2} * \frac{N_2}{n_2}$	Difficulty/Error proneness of the Software.
E	$V * D$	Effort required to understand or implement the Software.
B	$E^{2/3}/3000$	Estimate of the number of errors (Bugs) in the implementation.
T	$E/18$	Approximation of the time required to implement the code (in seconds).

Table 8 – Halstead Derived Measures.

Since both codes were automatically created, some Halstead metrics, like the number of bugs (B) and the approximation of the code's implementation time (T), are less relevant. The actual number of bugs in the code would be much lower than the estimate due to the lack of human error during code generation (bugs can still happen when developing templates for the generator, however). The implementation time is irrelevant because the code is generated in seconds automatically. Metrics like the effort (E) required to understand and the difficulty of the code (D) become more important as the original developers who made the code generation templates leave and new ones take their place even when said code was generated automatically.

Tables 9 and 10 contain the total, mean, median, and standard deviation of each Halstead metric per generated file from our own Code Generator solution and Pitch Developer Studio, respectively. Our solution generated 293 files, and Pitch's generated 282. The values for the difficulty and effort metrics are highlighted as we consider them to be the most important. Comparing their values from our generator to Pitch's, in terms of difficulty (D), both generators have similar results, although Pitch's deviation is three times the value of ours, indicating that some files may have much greater difficulty/error proneness than others. In terms of effort (E), the total, mean, and deviation are one order of magnitude greater in Pitch's code, indicating that our code requires much less effort to maintain in case some refactoring or optimization is needed. These results demonstrated that our methodology implementation is able to produce a very good quality source code.

Figure 39 contains a histogram comparison of the derived Halstead metrics of both code generators. From the figure it is possible to see that even with the difference in the number of files, our generator had lower values in every derived metric. This means we generated code with less volume, lower difficulty, and less effort to understand/maintain.

In conclusion, this experiment showed that the code produced by this work's HLA code generator is less voluminous, error-prone, and requires less effort to understand and

Metric	Total	Mean	Median	Deviation
n1	54	24	23	4.00341
n2	8483	74.9522	43	124.375
N1	134602	459.392	186	958.369
N2	102775	350.768	133	746.669
N	237377	810.16	318	1704.12
n	28993	98.9522	66	125.789
V	1.79563e+06	6128.43	1869.97	15895.7
D	**14323.2**	**48.8845**	**37.2727**	**26.8715**
E	**1.69455e+08**	**578344**	**76095.4**	**2.50616e+06**
T	9.41415e+06	32130.2	4227.52	139231
B	440.296	1.50272	0.598574	3.13568

Table 9 – Halstead metrics for our code generator.

Metric	Total	Mean	Median	Deviation
n1	74	26.8227	26	7.14915
n2	11682	85.4078	29	198.197
N1	275579	977.23	199	3356.05
N2	198526	703.993	121.5	2516.72
N	474105	1681.22	317.5	5866.04
n	31649	112.23	54	201.381
V	4.18669e+06	14846.4	1844.25	59801.9
D	**19545**	**69.3085**	**56.2277**	**63.4735**
E	**1.11013e+09**	**3.93664e+06**	**105467**	**2.52415e+07**
T	6.16741e+07	218702	5859.3	1.40231e+06
B	1061.65	3.76473	0.744088	14.2143

Table 10 – Halstead metrics for Pitch Developer Studio code generator.

maintain than code from the commercial code generator it was compared to.

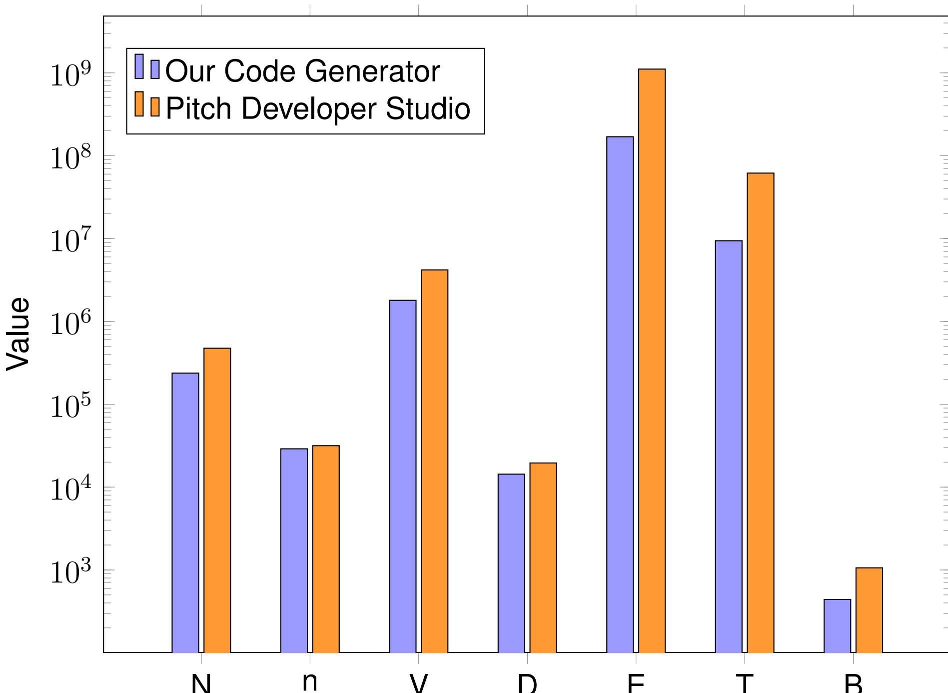


Figure 39 – Histogram of the total values of each derived metric.

8 CONCLUSIONS

This work has presented a new distributed simulation development methodology based on Model-Driven DSEEP using OPM and UML. This methodology makes use of a new “Perform Conceptual Analysis” step to increase cooperation between non-technical stakeholders and the development team that uses the OPM modeling language. OPM has a textual and graphical representation of the model, consists of a single diagram type, and mixes model structure and behavior to provide a more understandable conceptual model.

The methodology explores an automatic model-to-model transformation to derive a UML class diagram from the OPM model, which then undergoes a manual annotation step to attain HLA specific information. This transformation can be seen as an effective method for get a natural and comprehensible way to support conceptual modeling by non-technical stakeholders. In the “Design Simulation Environment” step, the annotated UML model is transformed into the FOM using another automatic model-to-model transformation. From the FOM, automatic HLA code generation can occur.

This work has also presented a possible implementation for the proposed methodology using OPCat for OPM modeling and automatic model transformation to UML. A standalone Java program to execute UML model annotation and to trigger QVT-Operacional model-to-model transformations to get the FOM from UML and a standalone C# template-based code generator to create, from the FOM, a usable HLA API containing serialization and deserialization of all of the FOM datatypes, a manager class for the federate, a manager class for each type of object, and a manager class for the interactions.

Finally, a series of experiments were developed to help validate what was proposed.

1 A case study in which the subject, a military software maintainer with no prior knowledge, was tasked with developing an OPM conceptual model of a distributed simulation involving two real military simulators. The subject was able to learn OPM and develop the conceptual model. Having noted that OPL greatly aided in learning the meaning of the various links that can connect OPM model elements. This means that for someone without prior knowledge of modeling languages, OPM was intuitive and easy to learn and use to make a conceptual model.

2 Two distributed simulation initial conceptual models, one with OPM and another with UML, were made following the same requirements and underwent a series of comparisons, such as how they portray the simulation structure and behavior and how they scale when adding new elements to them. The structural and behavioral comparisons allow us to conclude in favor of OPM when considering that the conceptual model uses a single diagram type and is overall more understandable for the initial modeling stage. The scalability comparison allows us to conclude in favor of OPM in a

DS with more classes (that need to be communicated between the simulators) and in favor of UML in a DS with more simulators. Since it is the case that a DS usually has a lot of classes that are communicated between a few simulators, we conclude that OPM is a better conceptual model for the initial modeling of distributed simulations.

3 The implementation of the methodology was done and was applied to an example in which all of the proposed activities (after OPM model development) were performed. The implementation demonstrated the viability of the methodology in achieving its objectives and the capacity to incorporate all of its activities into an efficient and easy-to-use tool.

4 The HLA code generated by the previous experiment was given an introduction on how it should be used and the overall code was compared, using the Halstead metrics, to the output code from the same FOM given to a commercial code generator. Resulting in better results in all metrics. Highlighting that the code from our code generator is less voluminous, error-prone, and requires less effort to understand and maintain.

In conclusion, our developed methodology and accompanying implementation show that not only is MDA development of HLA distributed simulations possible with the use of the Object-Process Methodology, but it is also preferable to the use of UML in order to better align what the stakeholders want the project to be and what the developers interpreted due to OPM being more readable to people with less knowledge of modeling languages.

For future works, implementing an OPM-UML model to model transformation that is detached from OPCat may be interesting in order to expand its capacity to generate more complex UML structures. Furthermore, the automatic transformation of written requirements into OPL segments could make this methodology even easier to use.

REFERENCES

ARRONATEGUI, U.; BAÑARES, J. Á.; COLOM, J. M. A mde approach for modelling and distributed simulation of health systems. In: SPRINGER. **International Conference on the Economics of Grids, Clouds, Systems, and Services**. [S.l.], 2020. p. 89–103.

ATL Development Team. **Atlas Transformation Language main page**. 2024. Accessed: 2024-22-03. Disponível em: <https://eclipse.dev/atl/>.

BASNET, S. et al. Comparison of system modelling techniques for autonomous ship systems. In: ____. [S.l.: s.n.], 2020. p. 125–139. ISBN 9788395669606.

BOCCIARELLI, P. et al. A model-driven approach to enable the simulation of complex systems on distributed architectures. **SIMULATION**, v. 95, n. 12, p. 1185–1211, 2019. Disponível em: <https://doi.org/10.1177/0037549719829828>.

BOCCIARELLI, P. et al. Model-driven distributed simulation engineering. In: **2019 Winter Simulation Conference (WSC)**. [S.l.: s.n.], 2019. p. 75–89.

CERQUEIRA, C. S.; AMBROSIO, A. M.; KIRNER, C. A model based concurrent engineering framework using iso-19450 standard 7th international conference on systems & concurrent engineering for space applications. 2016.

COIMBRA, R. T.; RESENDE, A.; TERRA, R. A correlation analysis between halstead complexity measures and other software measures. In: **2018 XLIV Latin American Computer Conference (CLEI)**. [S.l.: s.n.], 2018. p. 31–39.

CRUES, E. Z. et al. Spacefom-a robust standard for enabling a-priori interoperability of hla-based space systems simulations. **Journal of Simulation**, Taylor & Francis, v. 16, n. 6, p. 624–644, 2022.

DERE, B. E.; GÖRÜR, B. K.; OĞUZTÜZÜN, H. A framework for constructing agent-based aerospace simulations using model to text transformation. In: **Proceedings of the 2020 Summer Simulation Conference**. [S.l.: s.n.], 2020. p. 1–12.

DORI, D. **Object-Process Methodology**. [S.l.]: Springer Berlin, Heidelberg, 2002. XXV-455 p.

DORI, D. et al. Opcat–an object-process case tool for opm-based conceptual modelling. In: UNIVERSITY OF CAMBRIDGE CAMBRIDGE, UK. **1st International Conference on Modelling and Management of Engineering Processes**. [S.l.], 2010. p. 1–30.

DAMBROGIO, A. et al. Enabling reactive streams in hla-based simulations through a model-driven solution. In: **2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)**. [S.l.: s.n.], 2019. p. 1–8.

Eclipse Foundation. **Eclipse Foundation website**. Eclipse Foundation, 2023. Accessed: 2023-12-04. Disponível em: <https://www.eclipse.org/>.

EclipseECoreTools Project. **ECoreTools plugin website**. 2023. Accessed: 2023-12-04. Disponível em: <https://projects.eclipse.org/projects/modeling.emft.ecoretools>.

Epsilion Development Team. **Epsilion Scripting Languages main page**. 2024. Accessed: 2024-22-03. Disponível em: <https://eclipse.dev/epsilon/>.

GRAHAM, J. Creating an hla 1516 data encoding library using c++ template metaprogramming techniques. In: **2007 Spring Simulation Interoperability Workshop Proceedings, 07S-SIW-035**. [S.l.: s.n.], 2007.

HALSTEAD, M. H. **Elements of Software Science (Operating and programming systems series)**. [S.l.]: Elsevier Science Inc., 1977.

HAUSE, M. C.; DAY, R. L. Frenemies: Opm and sysml together in an mbse model. **INCOSE International Symposium**, v. 29, n. 1, p. 691–706, 2019. Disponível em: <https://incose-.onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2019.00629.x>.

IEEE. Ieee standard for modeling and simulation (m&s) high level architecture (hla)– federate interface specification. **IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)**, p. 1–378, Aug 2010.

IEEE. Ieee standard for modeling and simulation (m&s) high level architecture (hla)– framework and rules. **IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)**, p. 1–38, Aug 2010.

IEEE. Ieee standard for modeling and simulation (m&s) high level architecture (hla)– object model template (omt) specification. **IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)**, p. 1–112, Aug 2010.

IEEE. Ieee standard for distributed interactive simulation (dis) – communication services and profiles. **IEEE Std 1278.2-2015 (Revision of IEEE Std 1278.2-1995)**, p. 1–42, 2015.

IEEE. Ieee recommended practice for distributed simulation engineering and execution process (dseep). **IEEE Std 1730-2022 (Revision of IEEE Std 1730-2010)**, p. 1–74, 2022.

LEE, T.-D.; YOO, S.-H.; JEONG, C.-S. Hla-based object-oriented modeling/simulation for military system. In: SPRINGER. **Systems Modeling and Simulation: Theory and Applications: Third Asian Simulation Conference, AsianSim 2004, Jeju Island, Korea, October 4-6, 2004, Revised Selected Papers 3**. [S.l.], 2005. p. 122–130.

MAK Technologies. **MAK VR-Link**. MAK Technologies, 2023. Accessed: 2023-02-04. Disponível em: <https://www.mak.com/mak-one/tools/vr-link/>.

MALL, R. **Fundamentals of software engineering**. [S.l.]: PHI Learning Pvt. Ltd., 2018.

MöLLER, B.; KARLSSON, M.; LöFSTRAND, B. Reducing integration time and risk with the hla evolved encoding helpers. In: . [S.l.: s.n.], 2006.

OMG. **OMG organization website**. 2023. Accessed: 2023-7-04. Disponível em: <https:/-/www.omg.org/>.

OMG. **OMG QVT Specification website**. 2024. Accessed: 2024-22-03. Disponível em: <https://www.omg.org/spec/QVT/>.

ONG, D.; JABBARI, A. A review of problems and challenges of using multiple conceptual models. In: . [S.l.: s.n.], 2019.

OPCloud Ltd. **OPCloud website**. OPCloud Ltd, 2024. Accessed: 2024-22-03. Disponível em: <https://www.opcloud.tech/>.

PETTY, M. D.; WINDYGA, P. S. A high level architecture-based medical simulation system. **Simulation**, Sage Publications Sage CA: Thousand Oaks, CA, v. 73, n. 5, p. 281–287, 1999.

Pitch Technologies. **Pitch Developer Studio**. Pitch Technologies, 2022. Accessed: 2022-02-09. Disponível em: <https://pitchtechnologies.com/developer-studio/>.

SANTOS, G. dos; NUNES, R. An approach to build source code for hla-based distributed simulations. In: **Anais Estendidos do XII Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. Porto Alegre, RS, Brasil: SBC, 2022. p. 98–103. ISSN 2763-9002. Disponível em: <https://sol.sbc.org.br/index.php/sbesc_estendido/article/view/22849>.

SCHöN, E.-M.; THOMASCHEWSKI, J.; ESCALONA, M. J. Agile requirements engineering: A systematic literature review. **Computer Standards Interfaces**, v. 49, p. 79–91, 2017. ISSN 0920-5489. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0920548916300708>.

ŠENKỲR, D.; KROHA, P. Problem of inconsistency in textual requirements specification. In: **Proceedings of the 16th international conference on evaluation of novel approaches to software engineering–ENASE**. [S.l.: s.n.], 2021. p. 213–220.

SISO. Standard for real-time platform reference federation object model version 2.0. Citeseer, 2015.

SysML Org. **SysML website**. Object Management Group, 2022. Accessed: 2022-02-09. Disponível em: <https://sysml.org/>.

TENA Software Development Activity. **TENA SDA website**: Home. Boulder, 2023. Accessed: 2023-7-04. Disponível em: <https://www.tena-sda.org/>.

TOPÇU, O. et al. Distributed simulation. **A Model Driven Engineering Approach. Springer**, Springer, p. 9, 2016.

VLADIMIRIOVICH, M. A.; ALEXANDROVICH, V. A.; OLEGOVICH, R. D. Automatic translation uml activity diagrams to petri net. In: **2015 International Siberian Conference on Control and Communications (SIBCON)**. [S.l.: s.n.], 2015. p. 1–4.