Defence Research and Development Canada
Recherche et développement pour la défense Canada

VRAPP system overview

KARMA

N. Harrison
C. Belhumeur
M. Lambert
J.-F. Lepage
DRDC Valcartier

E. Rouleau
E. Boivin
M.-A. Labrie
LTI Inc.

Defence R&D Canada – Valcartier
Technical Memorandum
DRDC Valcartier TM 2010-208
October 2013

Canada

VRAPP system overview

KARMA

N. Harrison
C. Belhumeur
M. Lambert
J.-F. Lepage
DRDC Valcartier

E. Rouleau
E. Boivin
M.-A. Labrie
LTI Inc.

Defence R&D Canada – Valcartier

Technical Memorandum
DRDC Valcartier TM 2010-208
October 2013

© Her Majesty the Queen in Right of Canada, as represented by the Minister of National Defence, 2013

© Sa Majesté la Reine (en droit du Canada), telle que représentée par le ministre de la Défense nationale, 2013

Abstract

Electro-optical (EO) guided weapons represent a significant and rapidly proliferating threat to Canadian Forces (CF). Therefore, developing and maintaining appropriate countermeasures is critical to improve platform survivability. The CF have a need for a comprehensive and thorough approach to provide the best reliable answer to EO warfare problems, timely and adapted to the needs. In this perspective, the Virtual Range for Advanced Platform Protection (VRAPP) is a system of systems providing technology for an EO virtual proving ground for the purpose of achieving a more robust, adaptable and agile force protection and, ultimately, to improve combat effectiveness in high threat environments. The project documentation includes guidelines, system overviews, a technical data package, a statement of requirements and technical specifications. This report presents the system overview of the KARMA simulation framework. The system overview summarizes the KARMA facility and its capabilities with references to existing technical documentation. The KARMA system overview will be the starting point for the design of the VRAPP simulation framework. The KARMA system will be updated and integrated throughout the spiral development phases and will be documented in a final VRAPP system overview.

Résumé

Les armes guidées par électro-optique (EO) représentent une menace appréciable et qui prolifère rapidement pour les Forces canadiennes (FC). Par conséquent, il est critique de développer et d'entretenir des contre-mesures appropriées pour améliorer la survie des plates-formes. Les FC ont besoin d'une approche globale et approfondie pour fournir la meilleure réponse aux questions de guerre EO, dans un délai opportun et d'une façon adaptée aux besoins. Dans cette optique, le polygone virtuel pour la protection évoluée de plates-formes, ou VRAPP, est un système de systèmes offrant la technologie requise pour développer un polygone virtuel d'essais EO visant à assurer une protection de la force plus fiable, adaptable et souple et, ultimement, à améliorer l'efficacité au combat dans des environnements où le degré de menace est élevé. La documentation de projet comprend des lignes directrices, des vues d'ensemble de systèmes, un jeu de documents techniques, un énoncé des besoins et des spécifications techniques. Ce rapport présente la vue d'ensemble du cadriciel de simulation KARMA. La vue d'ensemble du système résume l'installation KARMA et ses capacités tout en faisant référence à la documentation technique existante. La vue d'ensemble du système KARMA servira de point de départ à la conception du cadriciel de simulation de VRAPP. Le système KARMA sera mis à jour et intégré tout au long du développement en spiral et sera documenté dans un document final de vue d'ensemble du système VRAPP.

This page intentionally left blank.

Executive summary

VRAPP system overview: KARMA

N. Harrison; C. Belhumeur; M. Lambert; J.-F. Lepage; E. Rouleau; E. Boivin; M.-A. Labrie; DRDC Valcartier TM 2010-208; Defence R&D Canada – Valcartier; October 2013.

Introduction or background: Electro-optical (EO) guided weapons represent a significant and rapidly proliferating threat to Canadian Forces (CF). Therefore, developing and maintaining appropriate countermeasures is critical to improve platform survivability. The CF have a need for a comprehensive and thorough approach to provide the best reliable answer to EO warfare (EOW) problems, timely and adapted to the needs.

In this perspective, the Virtual Range for Advanced Platform Protection (VRAPP) Technology Demonstration Project (TDP) aims at demonstrating an EO virtual proving ground as the basis of a robust capability in support of EOW engineering and training. VRAPP includes the development of a synthetic environment framework using three levels of engagement simulation services integrating constructive, virtual and live simulation systems for the purpose of conducting engagement analysis between EO threats, platforms and countermeasures. VRAPP will also demonstrate processes driving the engineering of EO countermeasure techniques. A verification and validation methodology will be defined to ensure that the analysis output is reliable and that the level of details is fit for purpose.

VRAPP integrates the KARMA, SEMAC with IR jammer, SAMSARA, HARFANG and MAWS facilities developed through several applied research projects by the EOW and the Precision Weapon Sections at Defence R&D Canada – Valcartier. In preparation for VRAPP's specification, all the existing documentation on each facility was gathered in a technical data package and system overview documents were written to summarize each facility.

Results: This report presents the system overview of the KARMA simulation framework. The system overview summarizes the big picture of the system and its capabilities with references to existing technical documentation.

Significance: The KARMA system overview will be the starting point for the design of the VRAPP simulation framework.

Future plans: The KARMA system will be updated and integrated throughout the spiral development phases and will be documented in a final VRAPP system overview.

Sommaire

VRAPP system overview: KARMA

N. Harrison; C. Belhumeur; M. Lambert; J.-F. Lepage; E. Rouleau; E. Boivin; M.-A. Labrie; DRDC Valcartier TM 2010-208; R & D pour la défense Canada – Valcartier; octobre 2013.

Introduction ou contexte: Les armes guidées par électro-optique (EO) représentent une menace appréciable et qui prolifère rapidement pour les Forces canadiennes (FC). Par conséquent, il est critique de développer et d'entretenir des contre-mesures appropriées pour améliorer la survie des plates-formes. Les FC ont besoin d'une approche globale et approfondie pour fournir la meilleure réponse aux questions de guerre EO, dans un délai opportun et d'une façon adaptée aux besoins.

Dans cette optique, le Projet de Démonstration Technologique (PDT) Polygone virtuel pour la protection évoluée de plates-formes, ou VRAPP, vise à faire la démonstration d'un polygone virtuel d'essais EO servant à développer une capacité fiable en appui à l'ingénierie et à l'entraînement pour la guerre EO (GEO). VRAPP comprend le développement d'un cadriciel d'environnement synthétique faisant appel à trois niveaux de services de simulation d'engagement qui intègrent des systèmes de simulation constructifs, virtuels et réels afin d'analyser efficacement des engagements entre des systèmes EO menaçants, des plates-formes et des contre-mesures. VRAPP fera la démonstration de processus d'ingénierie qui appuient le développement de techniques de contre-mesures EO. Une méthodologie de vérification et de validation sera définie afin d'assurer la fiabilité des conclusions de l'analyse et d'atteindre le niveau de détails approprié au problème à l'étude.

VRAPP intègre les installations KARMA, SEMAC avec brouilleur IR, SAMSARA, HARFANG et MAWS développées dans le cadre de plusieurs projets de recherche appliquée par les sections GEO et Armes de précision du centre de R & D pour la défense Canada – Valcartier. En préparation à la spécification de VRAPP, toute la documentation existante à propos de chacune des installations fut rassemblée dans un jeu de documents techniques et des documents de vue d'ensemble de systèmes furent écrits pour résumer chaque installation.

Résultats: Ce rapport présente la vue d'ensemble du système KARMA. La vue d'ensemble résume le système global et ses capacités tout en faisant référence à la documentation technique existante.

Importance: La vue d'ensemble du système KARMA servira de point de départ à la conception du cadriciel de simulation de VRAPP.

Perspectives: Le système KARMA sera mis à jour et intégré tout au long du développement en spiral et sera documenté dans un document final de vue d'ensemble du système VRAPP.

Table of contents

List of figures

List of tables

1 Introduction

1.1 Background

Electro-optical (EO) guided weapons represent a significant and rapidly proliferating threat to Canadian Forces (CF). Therefore, developing and maintaining appropriate countermeasures is critical to improve platform survivability. The CF have a need for a comprehensive and thorough approach to provide the best reliable answer to EO warfare (EOW) problems, timely and adapted to the needs.

In this perspective, the Virtual Range for Advanced Platform Protection (VRAPP) Technology Demonstration Project (TDP) integrates the five following facilities of the EOW and the Precision Weapon Sections developed at Defence R&D Canada – Valcartier (DRDC Valcartier) through several applied research projects:

- KARMA simulation framework [1];
- Simulator of Engagement for Missiles, Aircraft and Countermeasures (SEMAC) with infrared (IR) jammer;
- SAMSARA hybrid hardware-in-the-loop IR tracking and guidance loop simulator;
- HARFANG mobile IR countermeasure field testing facility; and
- Missile Approach Warning System (MAWS) hardware-in-the-loop (HWIL) simulator.

VRAPP is a System of Systems (SoS) providing technology for an EO virtual proving ground for the purpose of achieving a more robust, adaptable and agile force protection and, ultimately, to improve combat effectiveness in high threat environments.

1.2 VRAPP components

As illustrated in Figure 1, the main components of the VRAPP SoS will include three levels of engagement simulation services linked through a collaborative network for data sharing and distribution.

The Tier 1 workstations will offer to external customers a direct access to EO engagement services for distributed simulation exercises or for standalone usage. They will include all required components (computer-generated forces, model repository, etc.) to carry out EO engagement simulations with medium-to-low level-of-details physics-based models of threats, platforms, countermeasure systems and environment.

Although very similar in nature, the Tier 2 workstations will serve as the development, test and evaluation platform for software components that are subsequently migrated in their validated form to the Tier 1 for operational usage. Models will be developed and validated within this tier before they are made available to external customers.

The Tier 3 workstations will offer high-level-of-details EO engagement services. The Tier 3 services comprise three independent HWIL simulators providing high-level-of-details representations of specific elements of the self-protection problem (threat, countermeasure, MAWS, etc.). These simulators will be used either on a standalone basis for specific investigations or could be linked to participate in other simulations as high-level-of-details components.

VRAPP will also include a comprehensive experimental database of results for different uses, including component validation. The engagement simulation services of the VRAPP SoS will be managed through a series of engineering processes to ensure consistency in usage and the validity of the results.

The VRAPP components will be accessible to the users through a web portal providing collaborative tools for posting and viewing requests, updating and upgrading versions, browsing and searching the database and model repositories, sharing documentation and lessons learned, and stepping through processes.

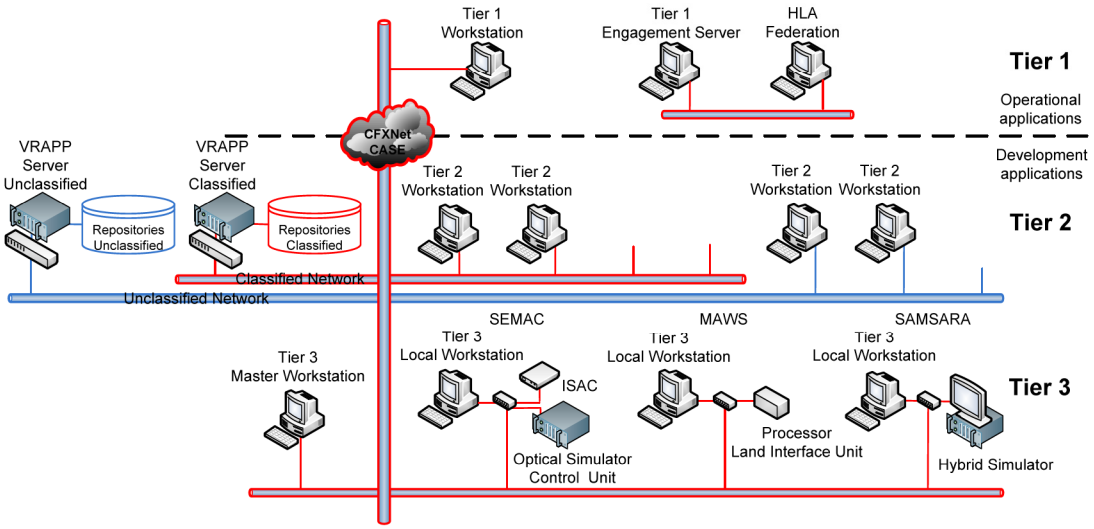

Figure 1: VRAPP envisioned architecture.

1.3 VRAPP documentation

The VRAPP documentation tree is depicted in Figure 2. The system overview documents summarize each facility integrated under VRAPP. The technical data package gathers the existing documentation on each facility. The guidelines documents present existing software development best practices. The Statement of Requirements (SOR) document introduces the VRAPP overarching requirements while the technical specification documents detail the requirements for each of the main functionalities: collaborative environment, simulation tools, simulation framework, software development, network, SEMAC Tier 3, MAWS Tier 3 and SAMSARA Tier 3.

This document presents the system overview of the KARMA simulation framework, a component of the VRAPP SoS referred as the VRAPP simulation framework. Information on the technical specifications and its development history can be found respectively in the Simulation framework technical specifications V0 document and in the VRAPP technical data package. This document refers specifically to files in the VRAPP Technical Data Package [2] (file names in blue underlined font).

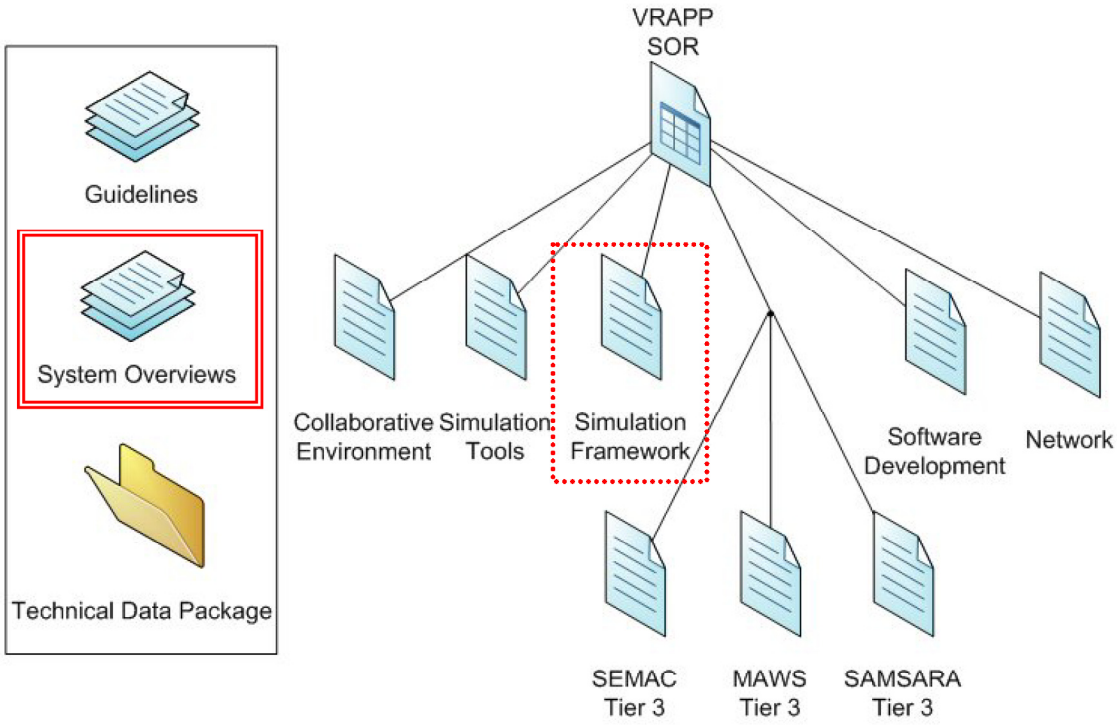

Figure 2: VRAPP documentation tree.

1.4 System description and purpose

The KARMA facility is primarily a generic Modelling and Simulation (M&S) process implemented in a specific integrated suite of tools. KARMA is inspired from the Model-Driven Architecture® (MDA®) to capture the knowledge of the experts into persistent M&S assets. KARMA is not an acronym; it refers to the incarnation of the knowledge of real-world systems into different forms of models [3].

DRDC Valcartier has fostered the teaming of software engineers with experts from different Communities Of Interest (COI), namely the COI for weapons and EO phenomenology, to transition their expertise into Synthetic Environments (SEs). Due to the proliferating use of M&S to solve a broad range of problems, these COI are challenged to deliver their knowledge in models for different applications and different clients. To maximize the return on the investment and to remain responsive to the various clients, the team has developed solid foundations to offer persistent weapon engagement M&S services. Please refer to the document KARMA Team Structure.pdf for more details.

KARMA was scoped to help focusing its development and ensure the production of useful deliverables because of its several potential applications and very broad reach. The underlying idea was to develop a M&S process of wide applicability and extensibility (growth potential) to any engagement simulation, but to limit its demonstration to a specific problem. This ensured that the efforts devoted to the development of KARMA were focused on achieving realistic objectives and on providing deliverables that were agreed upon. Consequently, it was decided to limit the extent of KARMA to engagements between EO-guided weapon systems, aircrafts and countermeasures (CMs). However, KARMA was designed without any limitations with respect to the type of environment to be modelled and simulated. Indeed, KARMA offers the flexibility for potential extension to other engagement environments (land and sea platforms) or different weapon technologies. It is expected that KARMA will eventually include Radio Frequency (RF) weapon system engagements.

In addition to flexibility, KARMA was designed to optimize models reusability. In fact, reusability is maximized through the uncoupling of the scenario modelling from the physics-based modelling. A system is divided into small models that are likely to be part of a more complex system. These fine-grained models have clear responsibilities. Therefore, this approach reduces the number of models and data files required for different simulation engagements. Without systematic reuse of components and configuration files, all the information would be duplicated, which could result in an unmanageable number of desynchronized versions of the models.

The KARMA process, its associated suite of tools and the simulation framework are detailed in Section 2; its usage is presented in Section 3; and finally, its development is presented in Section 4.

2 KARMA M&S

2.1 Modelling process

The KARMA modelling process is based on iterative micro-engineering development cycles that could be part of a larger process, such as the Federation Development and Execution Process (FEDEP). This approach proposes a concrete day-to-day working process for modellers. The generic KARMA modelling process is shown on Figure 3 and comprises three main aspects: modelling, simulation execution and analysis. KARMA mostly focuses on the simulation modelling phase to capitalize on experts' knowledge. In line with the MDA® philosophy, the conceptual model is independent from the implementation of the model and they are all linked through automatic code generation. The modelling process is based on software engineering concepts, tools and best practices to guide modellers in the development of models. The component-based development environment of KARMA allows reusable and interoperable model configurations. These components are defined at the conceptual modeling level, which becomes the reference for design.

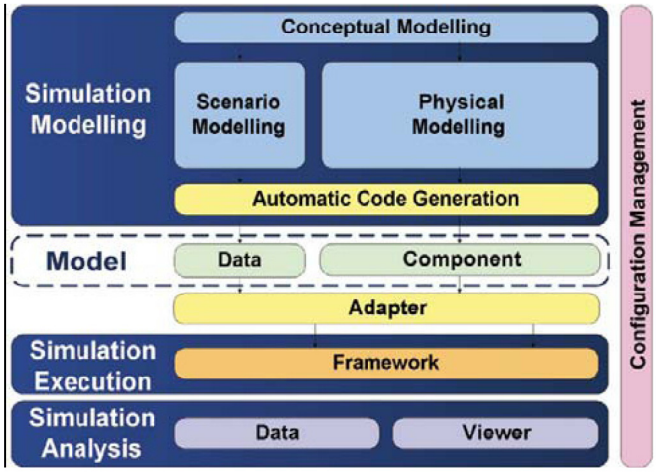

Figure 3: The generic KARMA M&S process.

The outcome of the modelling phase is a software model that is referred as a component and its associated data defined separately to foster reusing the same model into different simulation engagements. To maximize reusability, the models are adapted (from the *Adapter* design pattern) to specific simulation frameworks. Adapters reduce the dependence on a particular software product or environment. Moreover, they contribute to improve the reusability of generic models, the modularity of the dependence to simulation execution and the extensibility of the simulation framework. Theoretically, the adaptation is possible with any Object-Oriented (OO) and component-based simulation framework that supports plug-ins.

An option analysis originally led to the selection of Commercial-Off-The-Shelf (COTS) and custom tools to support the generic KARMA M&S process for the specific purpose of the KARMA team. The suite of tools is presented on Figure 4 and forms a collaborative integrated development environment.

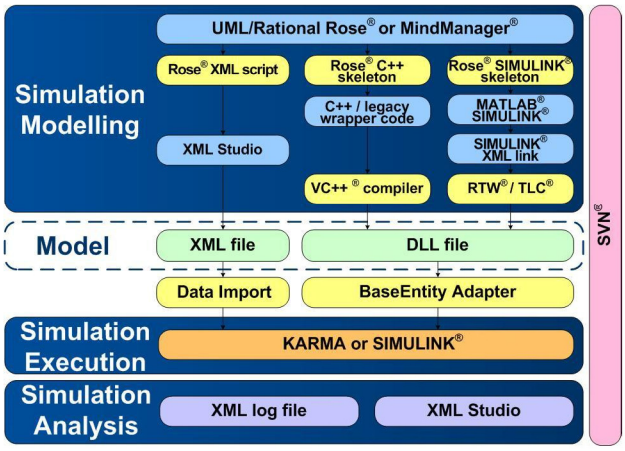

Figure 4: The KARMA integrated suite of tools.

The following subsections present different phases of the process that are detailed in the document A M&S Process to Achieve Reusability and Interoperability (NMSG 2002).pdf [4].

2.1.1 Modelling

Figure 5 presents the Level of Detail (LOD) pyramid where the engineering level is the most realistic representation for a simulation up to the campaign level which often trades fidelity for a broader analysis scope. A trade-off between output accuracy and computational time is made during the modelling stage to produce results as realistic as possible while maintaining the execution within the required time constraints.

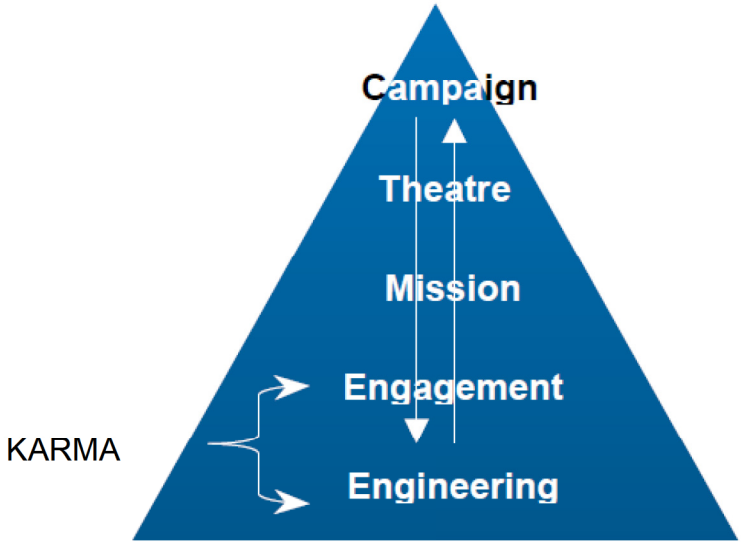

Figure 5: The M&S LOD pyramid.

The KARMA simulation framework is intended to allow for a varying LOD. Indeed, it is possible to use the KARMA simulation framework in closed loop for simulation execution having real-

time constraints or to perform engineering level analysis where real-time constraints are no longer required. However, as shown in Figure 5, the application of KARMA is intentionally focused on the engineering and engagement LODs. This approach was selected to establish a bottom-up integration of the M&S, which is believed to be the most effective way of developing models. The distinction between these two levels is not particularly well defined, even in the literature. This is made even fuzzier by the fact that there are variations within each category which, therefore, can overlap each other. It should be noted that the key element to discriminate these levels is the purpose of the model or the simulation. For instance, an engineering model will have a focus different from an engagement model depending on the particular needs of the simulation.

- Engineering level – at this level, the focus is essentially on modeling and simulating the physics of single entity parts over the duration of a physical phenomenon in order to assess system performance issues for design, systems/subsystems evaluation and test support.
- Engagement level – at this level, the focus is on modelling and simulating the behaviour of one or a few entities as a whole, or with some detailed subsystems of interest, over the duration of an interaction in order to assess system effectiveness issues.

Simulation modelling includes three main activities for each element to be simulated: conceptual modelling, physical modelling and scenario modelling which describes the interactions between the various simulated elements.

2.1.1.1 Conceptual modelling

Conceptual modelling is the first step of any structured M&S process. It is a high level representation of the simulation requirements for a model which includes its behaviour, properties and interactions. A conceptual model is presented in a form that allows everyone involved in the project to agree on and is the reference for the design stage. Modellers always go back to the conceptual model when any changes are required for a model. As a standard for representing these concepts, the Unified Modeling Language® (UML®) was selected to support the conceptual modelling. Therefore, applications of the simulation are described using *use case* diagrams while the static and dynamic aspects of the simulated elements are represented using *class* and *sequence* diagrams, respectively. Finally, the implementation of the elements is conceptualized using *component* diagrams.

2.1.1.2 Physical modelling

Once the stakeholders and the modellers agree on the conceptual model, each specialist can model the physics that is under its responsibility. This activity is the physical modelling which aims at producing the mathematical representation of the real-world behaviour. Parameters are used whenever it is possible since they increase the reusability potential by specifying what is left generic in the model; each component having customizable parameters and initial conditions. Parameters are the model data that remain constant over the simulation while initial conditions change over time. At the end of the physical modelling stage, the model is implemented into a software format that is independent from any COTS simulation framework and compiled as a Dynamic Link Library (DLL) which allows loading and instantiating the model dynamically at run-time. The software model can be directly written in an OO programming language (C++ was selected for KARMA) while legacy models are encapsulated into a class. However, since

modellers are not necessarily programmers, it is often impossible to require structured and standardized code from them. Therefore, automatic code generation tools are used for providing a standardized code skeleton and reducing the code to be written. Visual programming and simulation tools such as MATLAB®/Simulink® are also favoured for physical modelling without specific programming skills. Indeed, these tools were especially created for specific domain engineering-level rapid prototyping, test and validation. They often allow the reuse of functionalities through common libraries and the interoperability with other specialists. The only constraint is then to design physical models compliant with the OO and component-oriented conceptual model. The main challenge resides in switching from the block and wire representation of visual programming to the OO paradigm. Additionally, using component-based simulation, the model execution order is critical and the time steps between the integration schemes must be synchronized.

2.1.1.3 Scenario modelling

The last activity of the modelling phase is the scenario modelling which produces configurations of entity models with their associated parameters. The first two modelling activities (conceptual and physical) are dealing with generic objects while scenario modelling refers to instances of these objects. A scenario includes entities with their sub-models assemblies, the parameters associated to each model component (including initial conditions for the outputs) and the log settings for the simulation. The scenario modelling is based on the Extensible Markup Language (XML) standard that fosters the exchange, the modularity and the portability of the scenarios. This approach allows selecting different outputs to be logged or reusing the same model with different parameters.

2.1.2 Simulation

Some part of the simulation execution can be delegated to an existing COTS framework that provides functionalities such as scenario creation, execution control, doctrines, trajectories, viewers, etc. Some frameworks may even include built-in models and data. This approach originates from the fact that the expertise of the process initiators mainly resides in simulation modelling not in time management, distributed simulation, terrain database, visualization, etc.

The use of a recognized simulation framework contributes to the interoperability between the modellers by providing a common infrastructure. Within the defence community, such interoperability may also be improved if the chosen framework is compliant with High-Level Architecture (HLA).

2.1.3 Analysis

Results produced during a simulation can be used to validate the models created in the KARMA simulation framework or to study EO-guided weapon engagements. KARMA users must know if the models for a simulation represent an accurate picture of the reality and their validity for that simulation. In this context, some analysis tools, presented in Subsection 3.4, are offered to the KARMA users in order to validate the models.

Simulation analysis often requires a defined set of variables for each type of viewer. Thus, predefined log configurations can be reused between similar simulation analysis activities. As an example, the same entities states must generally be logged for KARMA Viewer3D replay and the same Line of Sight (LOS) angles must be logged for typical countermeasure effectiveness analysis. Refer to Subsection 3.5.9 for more details about how simulation results are analyzed.

2.2 Simulation framework

Originally, the KARMA framework was designed to provide services like a scenario loader and a data logger but not an execution scheduler. Indeed, one of the requirements of the KARMA framework was that it could run with many simulation frameworks easily in order to benefit from the scheduling and scenario edition functionalities. STRIVE®, developed by CAE, was the first simulation framework used to perform simulation engagements using KARMA models. This framework is described in Subsection 4.6. Additionally, a model execution scheduler was developed for the KARMA framework in order to demonstrate that KARMA can be used with many frameworks. STRIVE® is no longer used since it is more convenient to use the KARMA scheduler and no commercial licence is required.

The following subsections present the main components of the KARMA simulation framework and the implementation of this framework.

2.2.1 Main components

The KARMA simulation framework includes a simulation manager, named *SimulationEnvironment*, which calls all the entities in the simulation throughout the simulation execution. This component is also responsible for instantiating components having only one instance during a simulation (singletons) like the *Theatre* and the *Factory* that play a key role inside the simulation. The *Theatre* contains all the entities of the simulation while the *Factory* is in charge of instantiating the *Loader* and the *Logger*. The *Loader* loads all the models used in the simulation and the *Logger* logs the variables of the simulation according to the selection made by the user (log settings). Also, during a simulation, the *SimulationEnvironment* has the capacity to instantiate dynamically and destroy entities.

2.2.2 Implementation

The KARMA simulation framework is developed in C++ using Microsoft Visual Studio 2005®. The main components like the *Theatre*, the *SimulationEnvironment*, the *Factory*, the *Logger* and the *Loader* are created in independent projects in order to be compiled as a separate DLL. The models developed for the KARMA simulation framework are implemented using the C++ language or Simulink®. A DLL is created for each model. Thus, it is possible to modify a model without the need of recompiling the main components and all other models.

The development of the KARMA simulation framework is also based on various third party libraries; the majority being implemented in C++. Table 1 shows the libraries that are used in KARMA.

Table 1: External libraries used in KARMA.

Library	Application
Standard Template Library (STL)	Used to provide containers that store a collection of objects. For example, the *Theatre* owns a map container that contains reference to the objects in the simulation.
DataTypes	Used as the data types standard. It was created in order to have a common base class when data of different types need to be set into a container that needs data with same type.
Xercesc 2.8	Used to read/write the XML files and to keep the simulation data in the XML format.
MODTRAN	Used to generate atmospheric transmittances.
OpenSceneGraph (OSG) 2.2.0	Used to produce the rendering of IR signatures based on 3D models.
OSMesa 7.0.1	
OpenDynamicsEngine (ODE) 0.9	Used for collision detection based on 3D models.

3 Using KARMA

The user, as its name suggests, is someone who uses KARMA to perform a task like simulate an engagement and analyze simulation results. This section aims at providing the information required to perform these tasks properly. It targets typical end users but not necessarily experts in M&S of weapon system engagements neither in information technologies. The role of the user is defined in Subsection 3.1. The terminology and conventions used in KARMA are presented in Subsection 3.2. Subsection 3.3 presents the models repository while Subsection 3.4 presents the various tools available to the user. Finally, the basic principles that a user needs to know are covered in Subsection 3.5.

3.1 Role

A KARMA user performs any kind of simulations using the models created by the developers that is presented in Section 4. In addition, the user is able to analyse the results produced by the execution of the simulations and is able to export simulation results to other simulation or analysis tools. In the KARMA context, the user is able to simulate the engagement between a weapon system and a target according to the following steps:

- simulation preparation (planning);
- simulation execution; and
- simulation analysis (results).

3.2 Terminology and conventions

KARMA users must be familiar with the terminology and the conventions used in KARMA in order to understand the functionalities. The KARMA terminology is presented in Subsection 3.2.1. The coordinate systems used are presented in Subsection 3.2.2. The specification for the interface of guided weapon models is introduced in Subsection 3.2.3. Finally, the library of the data types developed for KARMA is presented in Subsection 3.2.4.

3.2.1 KARMA terminology

The use of KARMA requires being familiar with the terminology presented below.

- *Theatre* – finished 3D volume containing all the entities that take part in the simulation.
- *BaseEntity* – entity representing any autonomous object of the real-world (e.g. aircraft, missile, flare, etc.) having a role in a scenario. Each entity has an attitude (yaw, pitch, and roll), a location (x, y, and z), a geometry and a signature.
- *Part* – subsystem of an entity (e.g. sensor, fuze, motor, seeker, autopilot, etc.) that cannot exist by itself.
- *Characteristic* – represents abstract information of an entity that people cannot touch (e.g. IR signature, geometry, etc.).

- Parent – element (*BaseEntity* or *Part*) that encompasses a given element. A *BaseEntity* or a *Part* is the parent of the elements of its direct composition.
- Composition – children elements of a given element.
- *SimulationEnvironment* – manager that calls models during simulation execution and manages the creation/deletion of the entities.

Some acronyms are frequently used in typical KARMA EO-guided weapon engagement simulations. These acronyms are presented in Table 2.

Table 2: List of common acronyms.

Acronym	Description
FOV	Field of View
LOS	Line of Sight
MAWS	Missile Approach Warning System
CM	Countermeasure
DIRCM	Directed IR Countermeasure
ManPADS	Man Portable Air Defence System
IR	Infrared
UV	Ultraviolet
RF	Radio Frequency
DOF	Degree of Freedom
NED	North East Down
CMDS	Countermeasure Dispenser System

3.2.2 Coordinates

Entities that take part of the *Theatre* have attitude and location properties defined with respect to an earth-fixed coordinate system using the North East Down (NED) convention. Figure 6 shows

the earth-fixed coordinate system (E) and the aircraft body coordinate system (B). The northern axis, eastern axis, down axis corresponds respectively to the x-axis, y-axis and z-axis. The down axis is chosen instead of the up axis in order to comply with the right-hand rule.

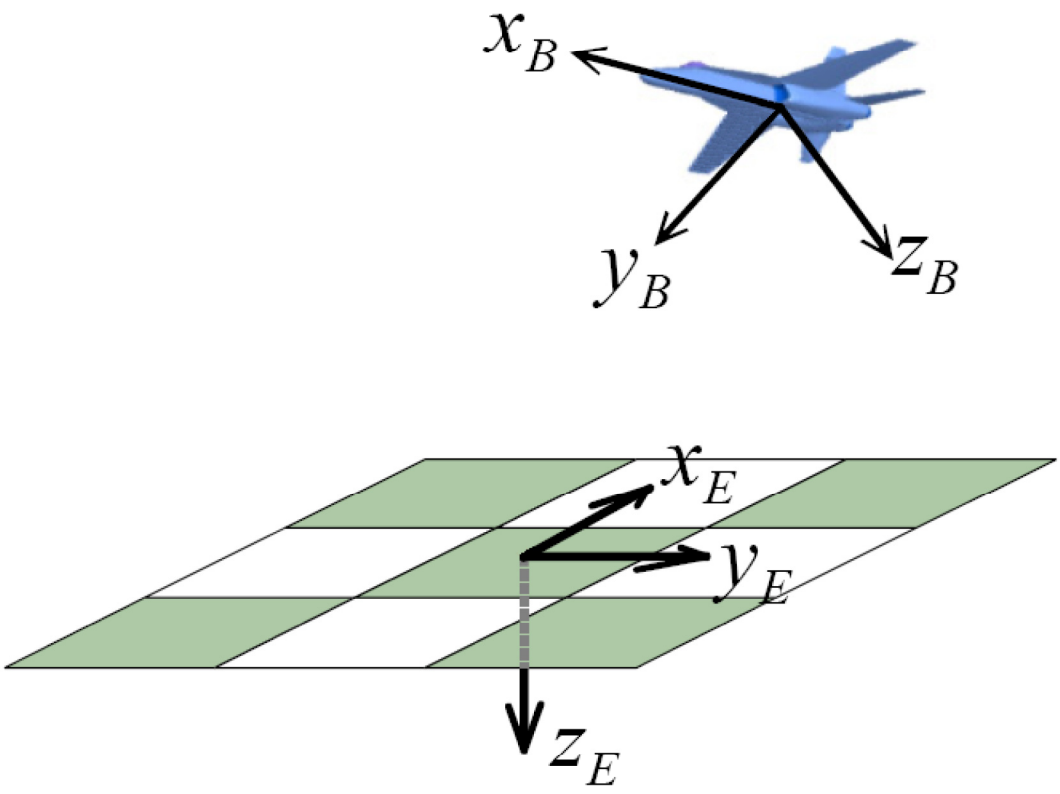

Figure 6: Earth-fixed (E) and the body (B) coordinate systems.

The orientation of an entity in relation to the NED coordinate system is given using the Tait-Bryan convention for Euler angles rotations. The first Euler rotation, yaw, is around the z-axis and the rotation angle is denoted by φ. The second Euler rotation, pitch, is around the y-axis and the rotation angle is denoted by θ. The last Euler rotation, roll, is around the x-axis and the rotation angle is denoted by ψ. The Figure 7 presents the rotations on the aircraft body coordinate system.

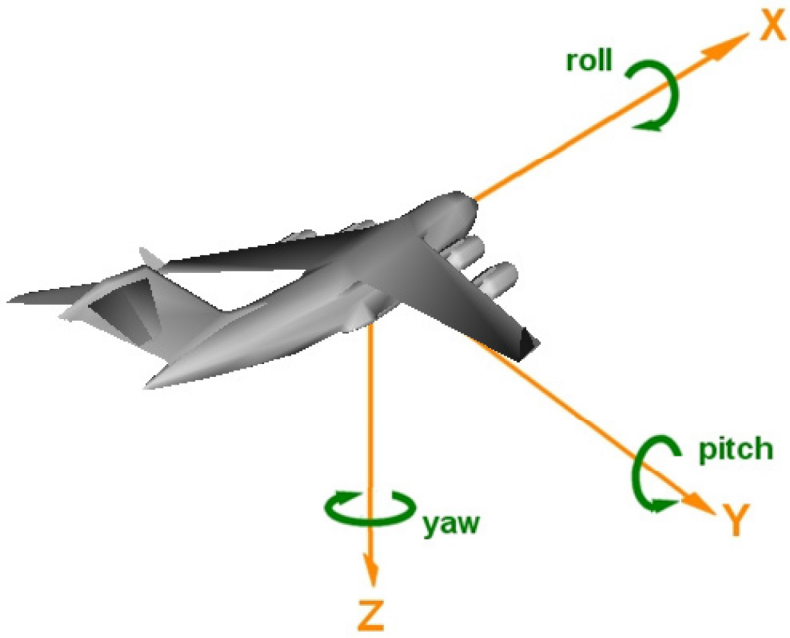

Figure 7: Rotations on the aircraft body coordinate system.

It is really important, in a simulation framework, to be able to switch from one system of reference to another. Sometimes it is also useful to get the data in the polar, the spherical, the cylindrical and the rectangular coordinate format. Thus, in order to be able to do those transformations into KARMA, a coordinate package has been created. Refer to the document COORDINATE.pdf for more details.

3.2.3 MIST

The Munition Interface Specification for the TTCP (MIST) is an interface specification for 6DOF guided weapon models [6]. It has been created to standardize the M&S technology applied to guided weapon/munition models and to establish both a baseline and techniques for reusing such models in a large range of applications and simulation environments through portability. Included is a variable naming convention, a functional decomposition for a baseline guided weapon model (including several variants), a signal specification for the data exchanged by weapon sub-component models and a blueprint for developing both modelling architectures and models that are independent of simulation architectures. This specification is not strictly applied in the modelling for KARMA, but when the names of the MIST standard are adequate and do not conflict with the KARMA naming convention, the MIST names are used.

3.2.4 Data types

The *DataTypes* library came from the necessity to have a common base class for all data types used in the development of the KARMA simulation framework. It is useful to have a common

base class when data of different types need to be set into a container that manages data with the same type: integer, floating-point, boolean, character, string, array, vector3 and matrice.

Because other development projects than KARMA could need to use these data types, the *DataTypes* library was done independently from KARMA, so it is possible to reuse it in other applications.

Refer to the documents DataTypes.pdf and DataTypesPackage.pdf for more details.

3.3 Model repository

Over the years, the KARMA simulation framework has achieved a level of maturity that allows every new application to contribute to a repository of models (referred as the *Model Repository*) that is shared across DRDC stakeholders for future reuse. KARMA users benefit of the models contained in this repository to create their different scenarios required for a specific simulation execution.

The models used in KARMA are located in the *ModelRepository* folder which is structured using three important subfolders for the users: *doc*, *Models* and *xml*. All documentation related to the modelling (model documentation template, Verification and Validation (V&V), etc.) are gathered under the *doc* folder. The source code of the models and the documentation pertaining to these models are in the *Models* folder while the scenarios (including parameter and composition files) are located in the *xml* folder.

The models available in the *Model Repository* are sorted by categories.

- *BaseEntity* – contains Aircraft, Bomb, Expendable (Flare), Ground Vehicle, Lifeform (Soldier) and Missile.
- *Characteristic* – contains Behaviour, Manoeuvre, Obscuration, Operation, Signature and Structure.
- *Other* – contains Environment and Transmittance.
- *Part* – contains Airframe, Autopilot, CMDS, Control Surface, DIRCM, Entity Dispensor, Fuze, Gyroscope, Jammer, Laser Designator, MANPADS launcher, MAWS, Motor, Navigation, Seeker, Sensor and Warhead.
- Utilities – contains Joystick.

In general, a folder is created for each subcategory in the folder of the corresponding category (e.g. Aircraft for a *BaseEntity* category). If the subcategory can be further divided into subcategories, then other folders are created, and so on. Each model is then stored in the folder corresponding to its specific subcategory (e.g. Aircraft3DOF model for the Aircraft subcategory). Every model must have an associated documentation: usually a Microsoft Word® document having the same name that the model and included in the model folder.

For example, the *BaseEntity* category contains 12 subcategories: Bomb, Expendable, FixedWing, GroundFixed, Lifeform, Missile, Other, RotaryWing, Ship, Tracked, Virtual and Wheeled. The

FixedWing subcategory contains four models: Aircraft, Aircraft3DOF, Aircraft747 and CommandedEntity.

It shall be noted that less than 25% of the models in the *Model Repository* are documented because model documentation for existing models was not required at the beginning of KARMA. For those undocumented models, sometimes documentation is contained in the mask of the Simulink® models. However, every new model added to the *Model Repository* is fully documented. There is no document that gives an overall picture on an object that aggregates other objects (model composition).

3.4 Tools

3.4.1 KARMA Studio

KARMA Studio is a simulation management tool that allows the planning, the execution and the analysis of engagement simulations. The various parameters required for the simulation are stored in XML format and KARMA Studio contains several generic functionalities for editing the XML files. The development phases of KARMA Studio were finished around 2005 and, except for specific features, no development effort was dedicated since then. Development efforts have been initiated in 2009 to replace this tool by other custom tools dedicated to specific functionalities that are required for the preparation, execution and analysis of simulations (see Subsections 3.4.2 to 3.4.5).

KARMA Studio consists in a dynamic adaptive interface [5]. Thus, the Graphical User Interface (GUI) that is presented to the user adapts to the content of XML files that are processed by the application. This means that the data contained in parameter files or other XML files used in KARMA (see Subsection 3.5.2) are interpreted and are presented to the user with appropriate graphical components. As a result, KARMA Studio hides the XML layer that is not user-friendly to users and developers. It contains six tabs, which are *Planning*, *Scenario Configuration*, *Log Configuration*, *Batch Run Configuration*, *Simulation* and *Analysis*.

It should be noted that the documents mentioned in this section are not up to date. Some modifications to the GUI were carried out since the last version of the documents. However, the operational principles and the main functionalities are the same. Although the documents mainly contain information for the developers, there are useful details for the users.

3.4.1.1 Planning

The *Planning* tab, shown in Figure 8, allows the user to view and edit XML parameter files and to create, view and edit XML composition files used to specify the composition of a model (*BaseEntity*, *Part* or *Characteristic*). The XML parameter files contain the parameters of a model and the XML composition files generated within the *Planning* tab contain the parts that compose a specific *BaseEntity*.

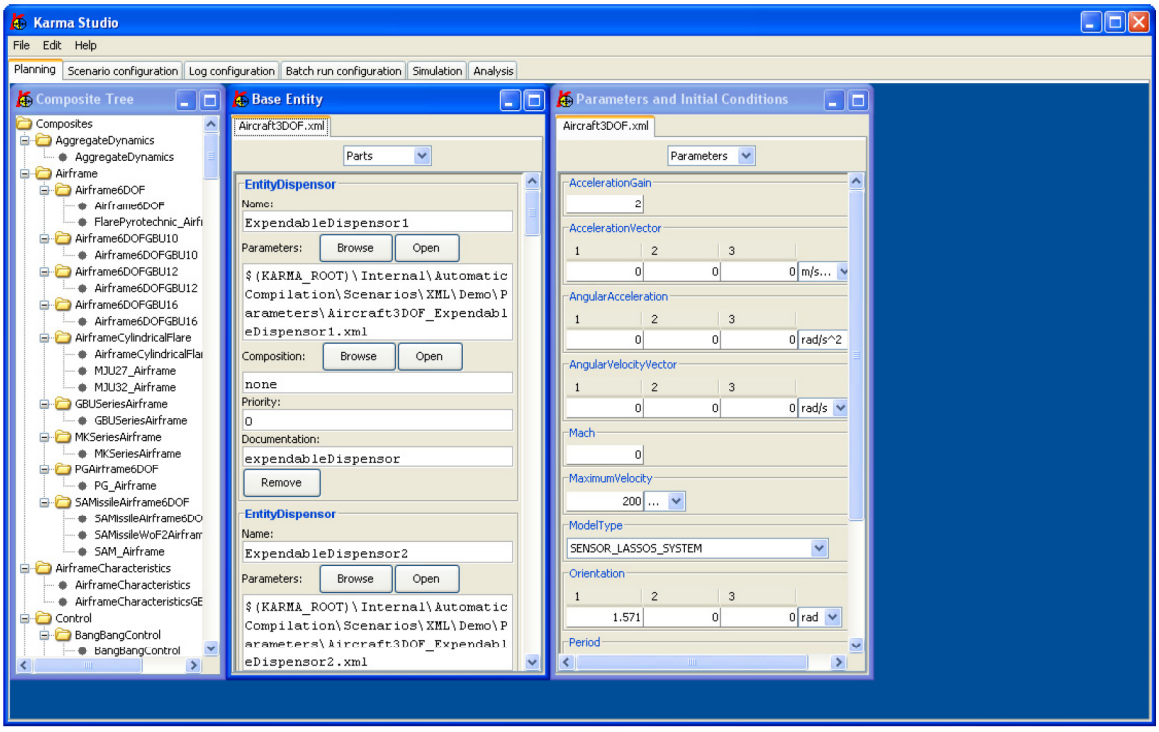

Figure 8: Planning tab in KARMA Studio.

The XML *BaseEntity* composition file is used to assemble several *Parts* and *Characteristics* that compose a *BaseEntity*. The user can add any combination of components (*Part*, *BaseEntity* or Characteristics) to a *BaseEntity* composition file by dragging the desired component into the *Composite tree* and dropping it into the *Part* section of the composition file.

The XML parameter file contains all the parameters that the user could eventually be interested to change for a specific instance of a model. The user can change some of these parameters and use the resulting XML file without having to recompile the model.

It is possible to create a composition XML files using KARMA Studio but not parameters XML files. However, each object has a default XML parameter file that can be copied and edited. Once the parameters and composition files are created, the user is ready to create a KARMA scenario composition file into the *Scenario configuration* tab.

Refer to the document Planning.pdf for more details.

3.4.1.2 Scenario configuration

The *Scenario configuration* tab, shown in Figure 9, allows the user to create, view and edit XML composition files that correspond to a scenario composition. Scenario creation is detailed in Subsection 3.5.7. The XML composition files generated within the *Scenario configuration* tab contain the entities that compose a specific scenario. A scenario also contains the environment and the log configuration for data logging. The log configuration file is selected under the *Log*

information section and is used to determine the log format and which data to log during a simulation.

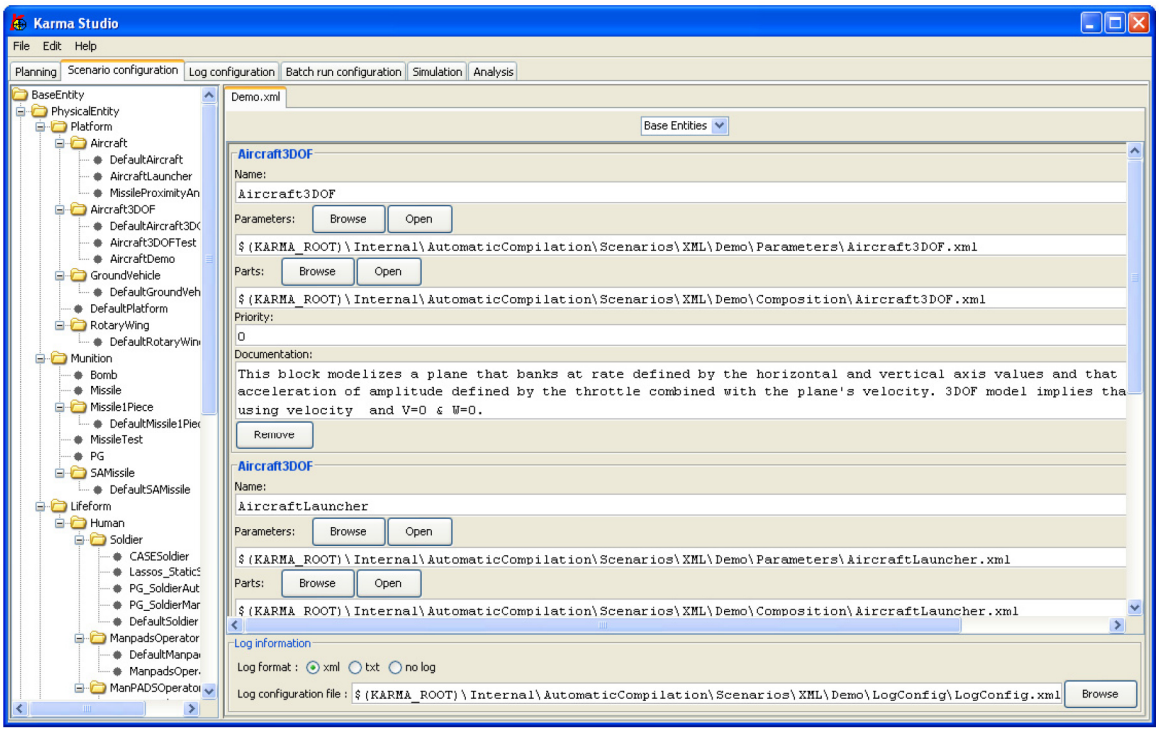

Figure 9: Scenario configuration tab in KARMA Studio.

Refer to the document Scenario.pdf for more details.

3.4.1.3 Log configuration

The *Log configuration* feature, shown in Figure 10, allows the user to choose the variables to log during a simulation. The *Log configuration* tab contains two sections. The first one contains fields that specify the directory and name of the destination file used to save the simulation results. The second section contains an adaptive area that modifies its content according to the variables selected. To select a given variable, one has to drag the variable from the tree and drop it in the *Selected variables* area. For each variable it is possible to specify the log frequency by entering the desired value in the frequency field or by selecting the final value option to log only the initial and final values of the variable.

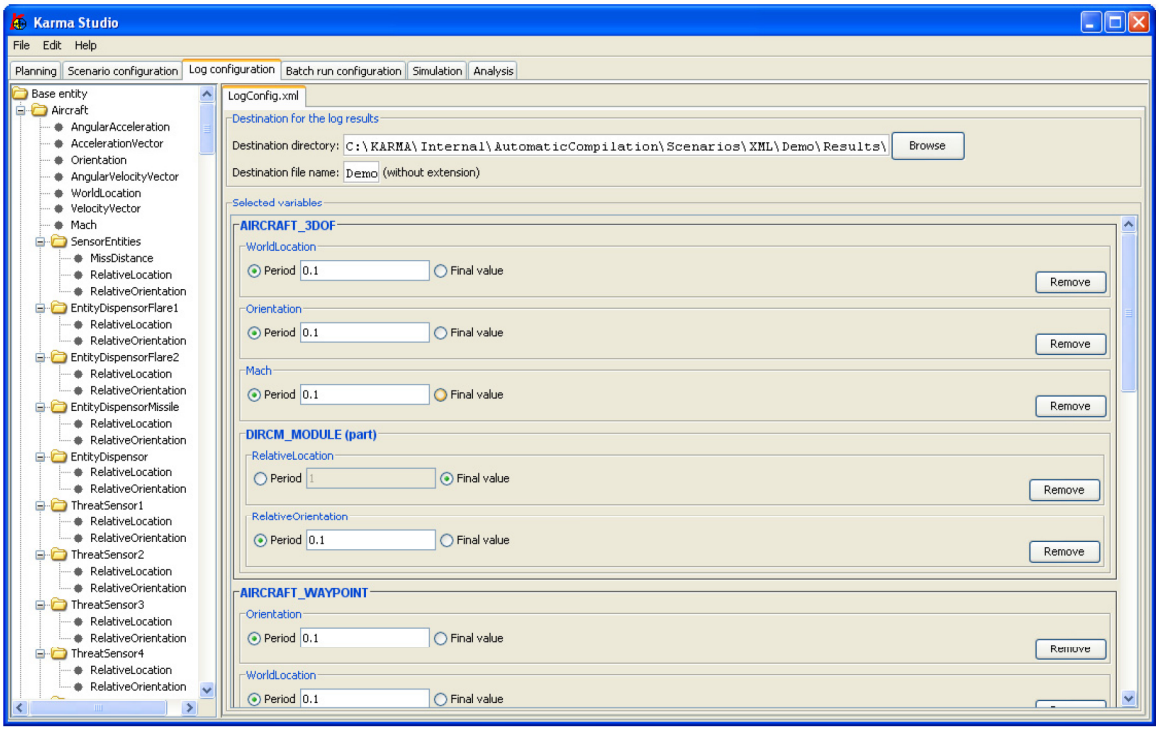

Figure 10: Log configuration tab in KARMA Studio.

Refer to the document LogConfiguration.pdf for more details.

3.4.1.4 Batch run configuration

The *Batch run configuration* feature, shown in Figure 11, allows the user to make a selection of parameters that can be used for parametric studies which are detailed in Subsection 3.5.6. When a parameter is selected, a graphical component containing its batch run details is added to the drop area. Each parameter is linked to a XML parameter file that will be used for the parametric study. For each parameter value, the user can specify the starting and ending values and the step value that will be used for the batch run simulation.

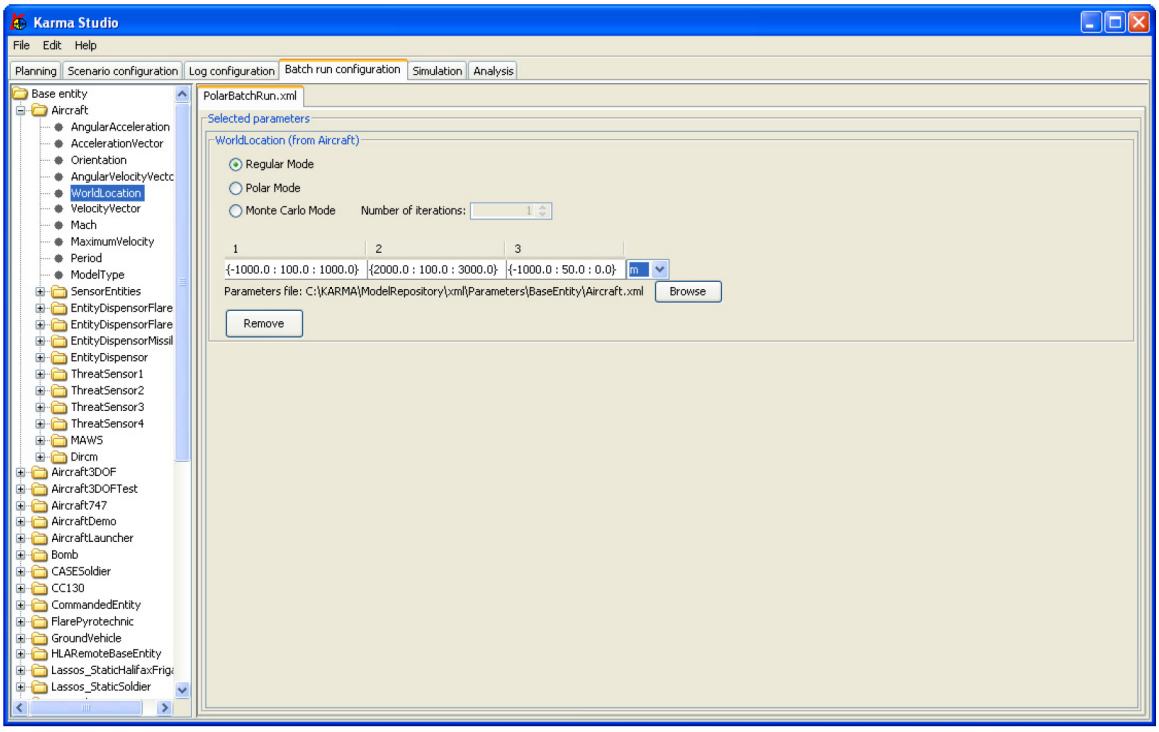

Figure 11: Batch run configuration tab in KARMA Studio.

Refer to the document BatchRunConfiguration.pdf for more details.

3.4.1.5 Simulation

The main purpose of the *Simulation* tab, shown in Figure 12, is to execute KARMA simulations. This tab is separated into four sections. The first section is *Simulation loading* allowing specifying the scenario for the simulation. The second section is the *Batch run information*. When the *Batch run simulation* box is checked, a batch run configuration file is used and many simulations are executed using the same scenario but varying a parameter between each run. The third section contains a hierarchy of entities and three tabs: *Runtime option*, *Matlab plot* and *Information*. The hierarchy of entities is a tree that includes all the outputs available in the *BaseEntity* of the scenario and theirs parts. It is possible to drag and drop the desired output in the *Runtime option* tab to view the value or in the *Matlab plot* tab to plot the value during the simulation. Once the simulation is started, the *Simulation information* section, in *Information* tab, shows real-time information relative to the execution of the simulation. The last section is the *Simulation control*, which contains the elements to control simulation execution.

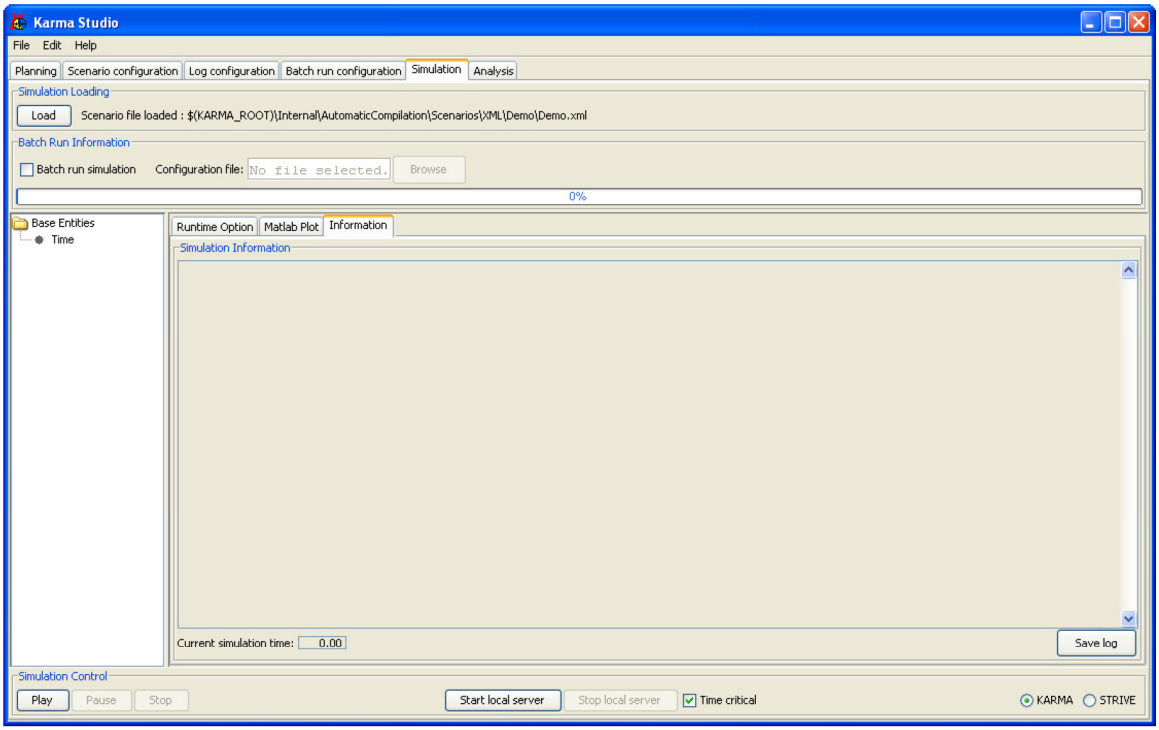

Figure 12: Simulation tab in KARMA Studio.

Refer to the document SimulationExecution.pdf for more details.

3.4.1.6 Analysis

The *Analysis* functionality allows the user to visualize and analyze results previously logged during simulation. The *Analysis* tab, shown in Figure 13, contains six sections: *Results file load*, *Scene generation visualization*, *Viewer selection*, *Analysis configuration*, *Analysis control* and *Analysis variables*. The XML file that contains simulation results is selected using the *Load* button. Any variables logged during a simulation are presented in the *Analysis configuration* as a composition tree for each entity. A viewer must be selected into the *Viewer selection* section and then, variables to analyze are dragged from the tree and dropped into the *Analysis variables* area. Selected variables can be exported to Excel® or presented graphically using MATLAB®. The Simplay 3D viewer is not supported anymore. Finally, the miss distance between two entities can be computed using the *Miss distance* option.

When the results of a scenario that contains at least one sensor using the IRSG are loaded, the *Scene generation visualization* section allows to create an .AVI file from images generated by these sensors, when images have been saved on disk. The options for the .AVI file to be created are selected using the *AVI configuration* button and the window shown in Figure 14.

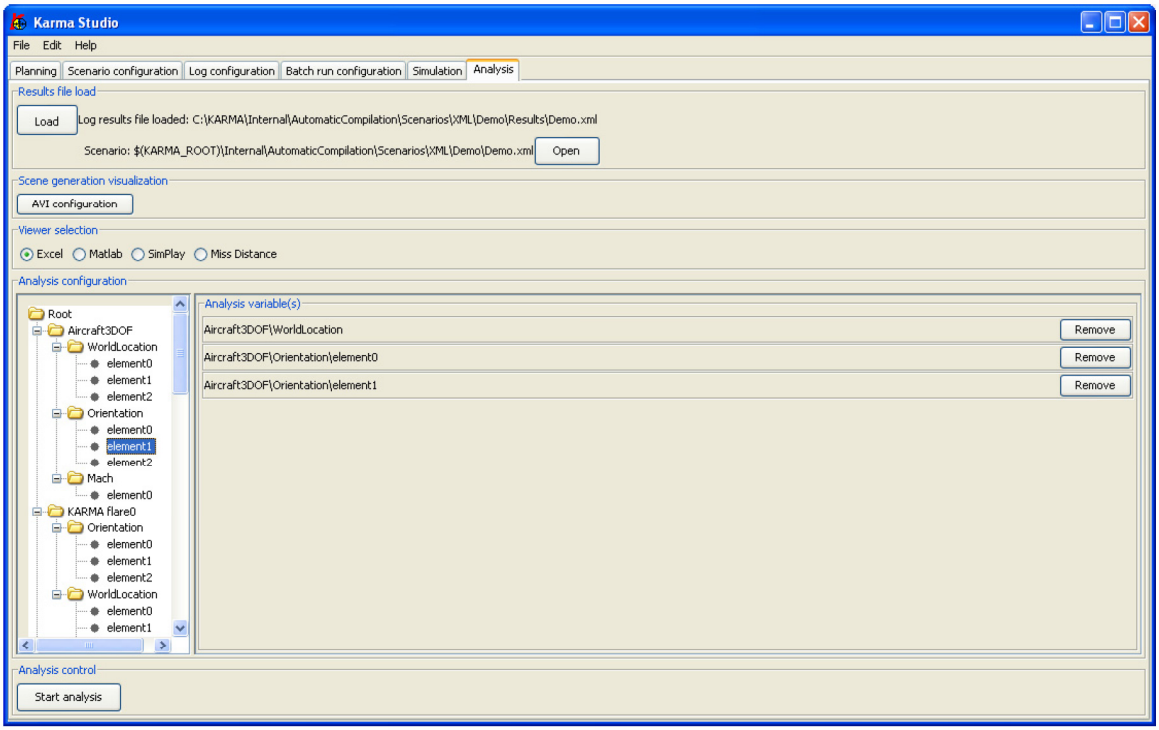

Figure 13: Analysis tab in KARMA Studio.

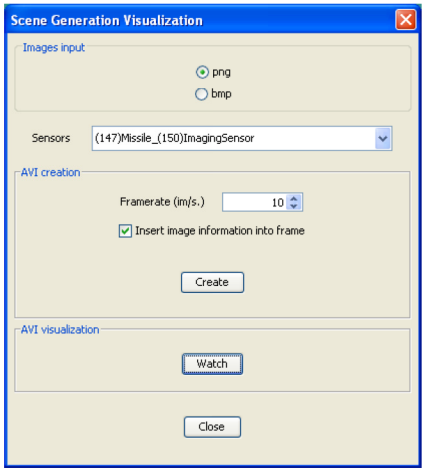

Figure 14: Scene generation visualization window for AVI configuration.

Refer to the document Analysis.pdf for more details.

3.4.2 KARMA Viewer3D

The KARMA Viewer3D is an analysis tool for KARMA simulations that allows visualizing a simulation at runtime or in post analysis. The viewer allows analysing an engagement simulation

in 3D by selecting the point of view from any entity. It is also possible to show trajectory of the entities and visualize the maximum range of the sensors/lasers. At runtime, all the entities in the scenario are displayed in the viewer while only the logged entities are displayed in post analysis. The commands of the KARMA Viewer3D are presented in Table 3.

Table 3: KARMA Viewer3D commands.

Key sequence	Command
Spacebar	Change the point of view from one entity to another.
C	Change the point of view from one camera to another: - following camera; - free camera; - first person camera; and - bird-eye view camera.
F	Fit all entities in camera FOV (bird-eye camera view only).
L	Lock/unlock camera so that camera will follow the entity or stay in place (bird-eye camera view only).
V	Toggle the IR camera (post analysis mode only).
Up arrow	Increase the speed of the simulation.
Down arrow	Decrease the speed of the simulation.
0	Return to the normal simulation speed.
P	Pause/resume the simulation.
R	Rewind the simulation (post analysis mode only).
=	Increase mouse scroll speed.
-	Decrease mouse scroll speed.

Key sequence	Command
S	Toggle sensors detection range display.
T	Toggle entity trajectories display.
F12	Take a screenshot of the viewer.
Esc	Quit.

3.4.3 SMAT

The Signature Modelling and Analysis Tool (SMAT) is a computer program designed to associate physical properties to a 3D model in order to perform IR analysis using the scene generation module of KARMA [7] [8]. The physical properties are defined and indexed in a temperature and material database. The indexes that are used in the database refer to the polygon *IR Color Code* and *IR Material Code* attributes of the OpenFlight 3D model that can be set for each polygon using a tool like Remo3D. The database also supports the use of a scaling varying in time. The supported IR analyses in SMAT are the IR image, the contrast intensity polar plot, the intensity spectrum plot, the contrast intensity range plot and the contrast intensity time plot. All the details are available in the document SMAT User's Guide.pdf [9].

3.4.4 KARMA Designer

The KARMA Designer tool is a computer program that provides functionalities for creating and editing a scenario using entities for which the composition has been pre-defined outside of KARMA Designer. It allows selecting entities of the scenario, their properties and their behaviour used for a specific simulation while keeping the internal representation of a scenario, including XML files management, transparent to the user. The KARMA Designer tool is shown in Figure 15 and contains the following graphical elements:

- Toolbar (1) for the most commonly used functionalities;
- *Scenario* section (2);
- *Entities* section (3);
- *Views* section (4);
- *Parameters* section (5); and
- *Parameter documentation* section (6).

The KARMA Designer tool is configurable through a XML configuration file. It allows setting different elements of the tool including the list of all available entities for the *Entities* section, a list of all available environments and a list of available collision detection techniques.

The KARMA Designer tool allows to import and export KARMA formatted scenarios; open, create, delete, save and rename (save as) a scenario file. It allows to visualize a scenario using 3D views and supports different 3D formats for entities and terrain:

- OpenFlight (.flt);
- OpenSceneGraph Binary (.ive); and
- OpenSceneGraph ASCII (.osg).

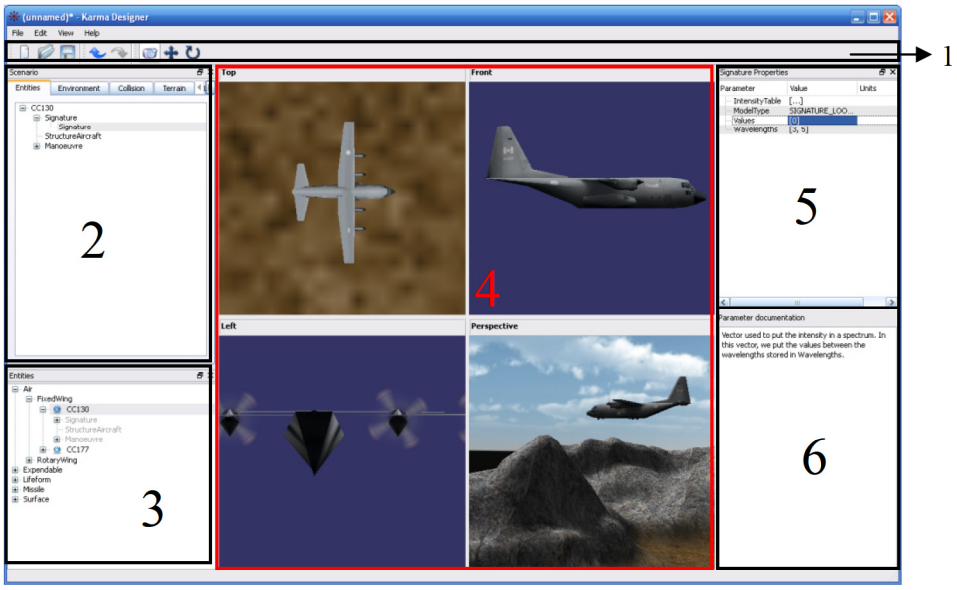

Figure 15: KARMA Designer tool.

The user can control the camera in the 3D views using the *Camera* mode while the user can control graphically the location and orientation of the entities in the *Translation* and *Rotation* modes, respectively. An icon is shown at the entity location when the 3D model of an entity is not visible in a view due to the zoom level.

Once a scenario is open, the *Scenario* section allows browsing the different elements of the scenario using separate tabs: the entities in the scenario (including their composition), the environment, the collision detection technique, the terrain, the log configuration and the simulation parameters. The scenario editor *Entities* section displays a list of all available entities grouped under logical categories. An entity is added to the scenario by dragging the associated item from this list and dropping it into the entities list of the *Scenario* section or in one of the four views of the *Views* section.

Each entity can be edited (position/orientation/velocity, renamed, deleted, duplicated, translated and rotated) from the entities list of the *Scenario* section. It is also possible to perform some of these actions graphically within one of the four views of the *Views* section. While browsing the entities and their composition in the *Scenario* section, the *Parameters* section displays a parameters list with the parameter name, value and units. This section allows to edit the

parameters that are not set as read only. The documentation of the selected parameter is shown in the *Parameter documentation* section.

3.4.5 KARMA Executor

The KARMA Executor tool, shown in Figure 16, is a computer program being developed to ease simulation execution. It allows selecting KARMA scenarios already defined, choosing their execution order, setting their batch run parameters and their execution mode. Scenarios are selected from a tree comprising XML scenario files located under the KARMA_ROOT folder or by browsing other locations. A scenario is selected by dragging the associated item into the tree and dropping it into the scenario list. A XML batch run configuration file can be set for a given scenario in the list by double-clicking on the *Batch parameters* cell of the scenario. Its execution mode is set similarly. Two modes are available As Fast As Possible (AFAP) and Real Time (RT). The list of scenarios can be saved on disk with their properties and be loaded later. Finally, once the simulation is started, each scenario is executed one after another according to the execution order and the status of a scenario is set to *Completed* as soon as its execution is completed. Note that a console window is open when a scenario is being executed.

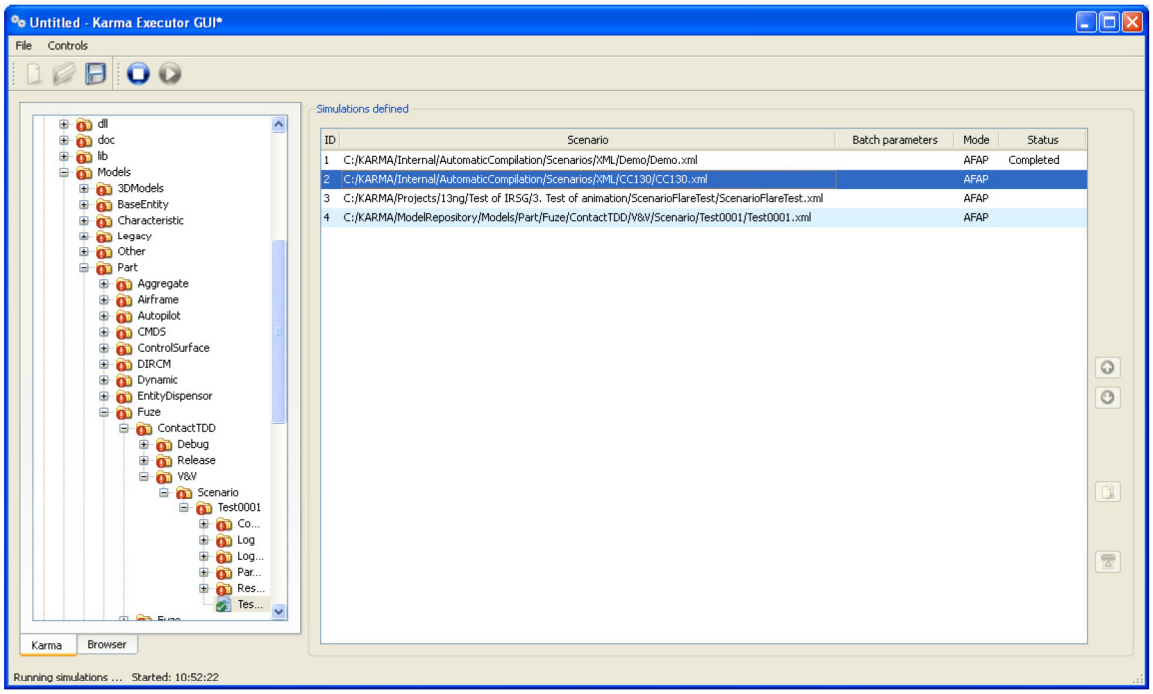

Figure 16: KARMA Executor tool.

3.5 KARMA basics

3.5.1 Composition

In KARMA, like almost each simulator, a system of composition is needed. The composition is divided into three levels. The first level of composition is for the scenario. As mentioned previously, a scenario is composed of several entities and an environment. The second level represents the entities, which can be composed of other entities and parts. Finally, the last level of composition represents the parts, which can contain other parts. Because KARMA has a flexible system of composition, the users of KARMA can create complex objects that represent the reality with several levels of composition if required.

3.5.2 XML

To support the application of syntactic composition, XML was identified as a technology that provides the necessary modularity, extensibility and composition to script the various M&S constituents. The different types of XML files used in KARMA are presented below [5].

- Composition – the scenario files as well as the entity files are based on this scheme. The scenario composition files contain the list of *BaseEntity* (e.g. aircraft or missile) that are involved in a scenario while the entity composition files contain each subsystems of an entity.
- Parameters – the parameter files contain the values of the parameters and initial conditions of a *BaseEntity*, *Part* or *Characteristic*.
- Batch run configuration – the batch run configuration files contain settings applied for parameters variation from one simulation execution to another.
- Log configuration – the log configuration files contain the settings applied for data logging during simulation execution.
- Log results – the log result files contain the data logged during a simulation execution.

3.5.3 Model inputs and outputs

A KARMA model might have inputs, parameters and outputs. The parameters are invariant during the execution of a simulation. The inputs of a given model are connected to the outputs of one or more models during the initialization stage and the search is carried out according to a specific order as presented below. The inputs and the outputs are used to make the exchange of information between the subsystems of the same *BaseEntity*. The exchange of information can also be carried out between a *BaseEntity* and one of its subsystems (*Part*, *BaseEntity* or *Characteristic*).

In the case of a *Part*, the search of the outputs is done in the following order:

- composition of the *Part*;
- composition (*Part* only) of the parent;

- composition (*Part* only) of the *BaseEntity*;
- outputs of the parent;
- outputs of the *BaseEntity*;
- outputs of the environment; and
- parameters of the *Part*.

In the case of a *BaseEntity*, the search of the outputs is done in the following order:

- composition of the *BaseEntity* (the composition of each composite is searched when the output is not found in the composite); and
- outputs of the environment.

Thus, if an output name is employed by several objects, the input might not be connected to the right output. It should be noted that in the case of a *Part*, if the output is not found and if a parameter of the same name exists, then the value of the parameter is used for all the duration of the simulation. The simulation can begin if and only if all the inputs are connected to outputs or parameters. Some improvements have to be done in KARMA in order to better manage some cases with multiple levels of composition.

3.5.4 Priorities

The priority is an attribute being used to establish the order of execution of the entities and the components when at least two objects must be called at time t. The priority attribute is useful when at least one input of an object depends on outputs of another object. It is then necessary to guarantee that an output is updated before the object gets its value.

In the composition files, it is possible to specify the order of execution of the objects by two methods. For example, if *object 1* must be executed before *object 2*, *object 1* can be declared before *object 2* in the composition file. Alternatively, the user can specify the value in the *priority* label with a smaller integer. If *object 1* and *object 2* have a respective priority of 2 and 0, then the second object will be called before the first object at each period. In this case, the order in the composition file does not matter.

A modeller must be aware that an inappropriate execution order might introduce a delay of one simulation step and produce a different behaviour.

3.5.5 Initialization

One of the first steps, before the execution of a scenario, is to read the contents of the composition files to determine which models shall be used and load the corresponding DLL in memory. Thereafter, parameters and initial outputs of each model are initialized according to the parameter files. Finally, the input-output association is performed: inputs of each model are connected to the outputs of one or more models.

During the initialization stage, a *BaseEntity* that is used as equipment (e.g. missile weapon for an aircraft), being in the composition of another entity instead of being in the scenario, will be activated under specific conditions. KARMA activates a *BaseEntity* at launch time when this entity is not in the scenario initially. However, it is possible to activate an entity at a specific time using an *Activate* behaviour (*Characteristic*) combined with a *TimeCondition* behaviour condition.

3.5.6 Batch run

The batch run feature allows the user to make a parametric study of parameters or initial outputs for a given scenario using KARMA Studio. The parametric study is a set of simulation where the initial value of one of the studied parameters changes from one simulation execution to another. For example, a typical study is to change the initial position of a *BaseEntity* at each simulation execution to determine a detection range prior to an engagement. The study can be done using three modes by setting the starting, final and step values for each parameter being studied. A batch run configuration file must be created to specify the studied parameters.

- Regular mode – a value is varied between an initial value and a final value between each simulation execution.
- Polar mode – a vector is varied horizontally (X-Y plane) within a polar region, by setting the starting, final and step values of polar coordinates (r and θ), between each simulation execution. The r and θ values are set using the X and Y elements respectively. Figure 17 shows how the location of an aircraft can be varied, for a total of 15 locations marked using a black dot, using the polar coordinates.
- Monte Carlo mode – a value is varied between a minimum and maximum according to a predefined probability distribution. A global parameter allows defining the number of iterations required for the parametric study. The current implementation makes it possible to perform a study with many values varying at the same time during the same Monte Carlo simulation.

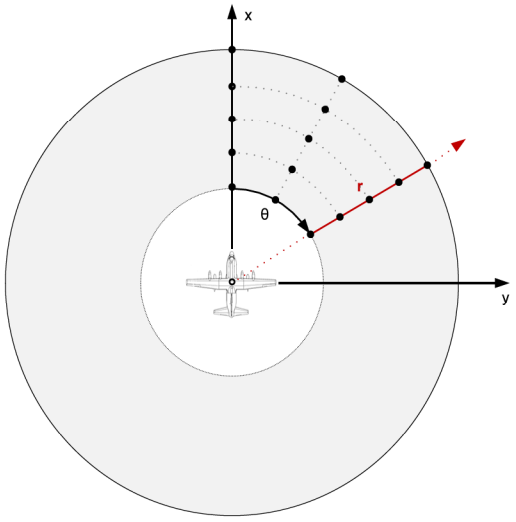

Figure 17: Polar coordinates for a polar batch run.

3.5.7 How to create a scenario

The easiest way to create a scenario is by using KARMA Studio. Indeed, the GUI of KARMA Studio simplifies the creation and edition of XML files for users that are not familiar with the XML format. KARMA Studio also avoids browsing for the files referenced in a scenario or a composition. Although an alternative method is to create or edit the XML files using any text editor, it is not recommended for inexperienced users.

Scenario creation is divided into three main stages. First the user defines the entities that belong to the scenario. In turn, composition and parameter files are created or selected. Then, the user specifies the variables that will be logged during the simulation by creating or selecting a log configuration file. Finally, when a parametric study is required for that scenario, the user defines a batch run configuration file.

To create a scenario with KARMA Studio, it is necessary to open an existing scenario or to create a new scenario file. The user can add entities to the XML scenario file by dragging the desired *BaseEntity* into the entity tree and dropping it into the *Base Entities* section of the *Scenario configuration* tab. It is possible to modify the parameter file or composition file of a *BaseEntity* by clicking on the corresponding *Browse* button. It is also possible to edit the contents of the parameter file or composition file by clicking on the corresponding *Open* button. The user is then led up to modify the file in the *Planning* tab.

In the *Scenario* tab, it is also possible to select the log format between no log, txt and xml and the log configuration file, if appropriate. The text format produces a text file (.txt) that contains the logged results for each time step in a sequential fashion. The XML format creates a XML file with the results regrouped in a hierarchical way representing the *BaseEntity* composition for each logged variable. The log configuration file contains the variables name of *BaseEntity* and *Part* to log. The variables can be logged at a specific period or only for the initial and final value.

The document KarmaStudioGuide.pdf details the steps to create a scenario. As for other documents related to KARMA Studio, this document is not up-to-date.

3.5.8 How to run a simulation

The KARMA simulator is used to perform the simulation execution. A scenario can run in *real-time* or *as fast as possible*. The *real-time* mode means that the simulation framework tries to call the entities and their parts in real time. If the entities cannot be called in a real-time fashion, the user is informed at the end of the simulation with the warning message "*The engagement is unable to simulate in real time*". In the second mode, the simulator will call entities as fast as possible. Sometimes the simulation framework will run faster than real time and sometimes it will be slower. In this mode, the simulation can run quickly during a simulation while being slower when many entities must be called at the same time. The simulator is also in charge of sending the results to the user at the end of the simulation. The final location, orientation, velocity, angular velocity and acceleration of all the entities in the scenario are shown to the user.

Three methods can be used to perform a simulation. The first method is to use KARMA Studio. In this case, the user must load the scenario in the *Simulation* tab. It is possible to run a scenario in batch if the *Batch run simulation* box is checked and if the batch run file is specified. Before

starting the scenario, with KARMA Studio, it is necessary to start the server. When the simulation is ready to start, a window appears to specify the duration of the simulation and if the simulation must be simulated in real time.

The second method simply consists in executing the *.bat* file located in the bin directory of KARMA. With a text editor, it is possible to specify the scenario, the duration of the simulation and if the simulation must be done in real time. During the simulation, some information is displayed in a DOS Command Prompt window.

The third method is similar to the second, except that the simulation is shown into the KARMA Viewer3D. This tool is located in the directory *utilities\ViewerDelta3D* of KARMA. The .bat file is located in the bin directory of the Viewer3D.

3.5.9 Results analysis

There are two main approaches to analyze simulation results in KARMA. There is an analytical approach using KARMA Studio which allows the user to visualize and analyze results previously logged during a simulation. Another way to analyze a simulation is using the KARMA Viewer3D tool presented in Subsection 3.4.2 which generates a 3D representation of a simulation.

Using KARMA Studio, under the *Analysis* tab presented in Subsection 3.4.1.6, simulation results are loaded by clicking on the *Load* button. The user selects the appropriate file then the logged variables are summarized in the *Analysis configuration* as a composition tree for each entity. The user is able to select any variable of interest and perform three kinds of analysis using the corresponding viewer type: export to Excel®, generate a plot using MATLAB® or compute the miss distance between two entities.

When exporting variables to Excel®, the user selects an item (variable or one of its elements) using drag and drop. Each item is added in the *Analysis variables* area. As soon as the user clicks on the *Start analysis* button, a Comma Separated Values (CSV) file is generated and then opened with Excel®. The results are displayed with the time steps in the first column and the selected items in the subsequent columns.

When generating a plot using MATLAB®, two types of graphs are available: XY plot and polar plot. Both types of plot are generated by clicking on the *Start analysis* button. Using the first type, the user has to drag and drop a variable element for the X-Axis and one for the Y-Axis. Using the polar type, the user has to drag and drop the variables of interest; all the variable elements are needed to compute the azimuth and elevation values. Before the polar plot is generated, the user selects either a static plot (continuous lines) or a dynamic plot (values shown with a moving point for each variable). A MATLAB® script file (m-file) is created and executed in a new MATLAB® command window each time a plot is generated.

Finally, the 3D approach allows runtime analysis and post analysis. A batch file is used to start the KARMA Viewer3D using specific settings stored in a XML file. Runtime analysis is achieved by executing the *Karma3d.bat* file in the *Utilities\ViewerDelta3d\bin* directory, which starts the scenario selected in the XML file and displays a 3D representation accordingly. Similarly, the post analysis of results previously logged during a simulation is performed by executing the *ViewerDelta3DPostAnalysis.bat* file.

3.5.10 Limitations

3.5.10.1 General limitations

The requirements identified to develop a process for carrying out engagement-level M&S of weapon systems are presented in the technical memorandum TM 2001-271 KARMA Definition and Requirement.pdf [1]. However, the requirements below, amongst others, are not currently addressed in KARMA.

- KARMA does not advise the user if a combination of components with various LOD sub-categories is inappropriate.
- It is not possible to perform multiple simulation runs based on various series of input parameters (batch runs).
- The user is not able to attach analysis comments to the output data and monitored parameters. The user can edit the graphs, but no formal tool is available to comment the analysis and generate report.
- The user cannot import scenarios from other test or simulation tools.
- The user is not able to import scenario parameters (altitudes, aspects, weather conditions, manoeuvres, etc.) from external sources (COTS spreadsheets, word processor, etc.).

3.5.10.2 KARMA Studio

Currently, there are several issues with KARMA Studio, but development is kept to a minimum since it is expected to develop another tool in a near future. However, KARMA Studio is mostly used to analyse simulation results as presented earlier.

4 Developing KARMA

4.1 Role

The former section presented how KARMA can be used to generate simulations results. This section presents the developer's role in KARMA. At this point, it is worth defining developer and modeller terms. A modeller is someone who creates new models by using existing functionalities or elements (e.g. modelling classes, data logging, analysis tools, etc.) of the KARMA simulation framework while the developer is someone who modifies or creates new ones. This development is performed at the architectural level of KARMA. Although the current document focuses on the developer, the most part of the description also applies to a modeller.

The KARMA project was started in 2000; its modelling process and simulation framework has been developed during five years by introducing new modelling concepts and models. At the beginning of the year 2006, KARMA development was strictly done through projects based on KARMA. For the past two years, it has been clear that KARMA needed to be constantly updated to accommodate the needs of different projects as new concepts are introduced, level of details of models is increased and new analysis needs appear.

The developer must adhere to the KARMA philosophy, document every development effort and make sure that every model used by the KARMA team remains compatible with the new or modified code. It is the developer's responsibility to report any problem or error to the delegate architect. In order to minimize conflicts and leverage the development process, standardized tools have been selected for the KARMA team.

4.2 Model repository

As mentioned earlier in Subsection 3.3, KARMA models are gathered into a database named *Model Repository*. This repository contains only models that can be used into KARMA simulations while any architectural or modelling elements are located into a separate section.

As mentioned in Section 3, the KARMA modelling process produces model data in XML parameter file and model component in DLL. Moreover, scenarios are built using XML files. The different versions of these files are tracked using TortoiseSVN®, a Subversion (SVN) client implemented as a Windows® shell extension. Practically, SVN automates the sharing and the version control of the UML® conceptual models, the C++ source code, the Simulink® models and many other files.

The KARMA team has elaborated an approach that ensures that models are stable enough before being available in the repository. Each project has a development project where models are gathered in the early stages of the modelling. When models are documented, tested and validated, they are moved into the *Model Repository* section of the database. Once a model is available in the *Model Repository*, one can assume that the model is stable and usable within the validation bounds.

KARMA models are stored into the *Model Repository* using a specific hierarchy. Each model is derived from one of the KARMA object types (*BaseEntity*, *Part*, *Characteristic*) and gathered according to its category and behaviour. For example, an aircraft model named *AircraftModel* that is a *BaseEntity* object type and classified under the *FixedWing* category is derived from *BaseEntity/PhysicalEntity/Platform*. Table 4 lists the current model categories. Models that are not derived from a KARMA object type are gathered under the *Other* category. This is the case for the atmospheric transmittances. There is also a *Legacy* category for existing models not compatible with KARMA and a *Utilities* category for elements that can be used by different models.

Table 4: Model categories.

Object Type	Category
BaseEntity	Bomb
	Expendable
	FixedWing
	GroundFixed
	Lifeform
	Missile
	Other
	RotaryWing
	Ship
	Tracked
	Virtual
	Wheeled
Part	Aggregate
	Airframe
	Autopilot
	CMDS
	ControlSurface
	DIRCM
	Dynamic
	EntityDispensor
	Fuze
	Gyroscope
	Jammer
	LaserDesignator
	Manoeuvre
	MANPADSLauncher
	MAWS
	Motor
	Navigation
	Seeker
	Sensor
	Warhead

Object Type	Category
Characteristic	Behavior
	Manoeuvre
	Obscuration
	Operation
	Signature
	Structure
Other	Environment
	Transmittance
	Wind

KARMA supports two types of models: C++ and Simulink® models. Those models are gathered under KARMA types and model categories as show in Subsection 3.3. Each model has its own folder where its associated information can be found (e.g. project/model files, documentation, etc.). Figure 18 shows an *Airframe* library (Simulink®) with two airframe models that can be either C++ or Simulink® models.

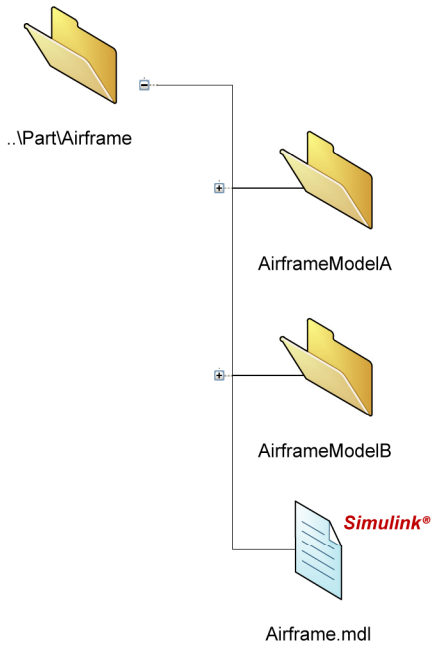

Figure 18: Example of models library.

4.2.1 C++ models

Those models are developed using Microsoft Visual Studio®. A project is used to define how source files (*.h* and *.cpp*) are compiled and linked in order to generate a DLL. Every properties of a project are set using the environment variable *KARMA_ROOT*. Typical settings are shown in Figure 19. This project is included into the KARMA solution (*KARMA_ROOT\src\karma.sln*) which encompasses architectural and model projects.

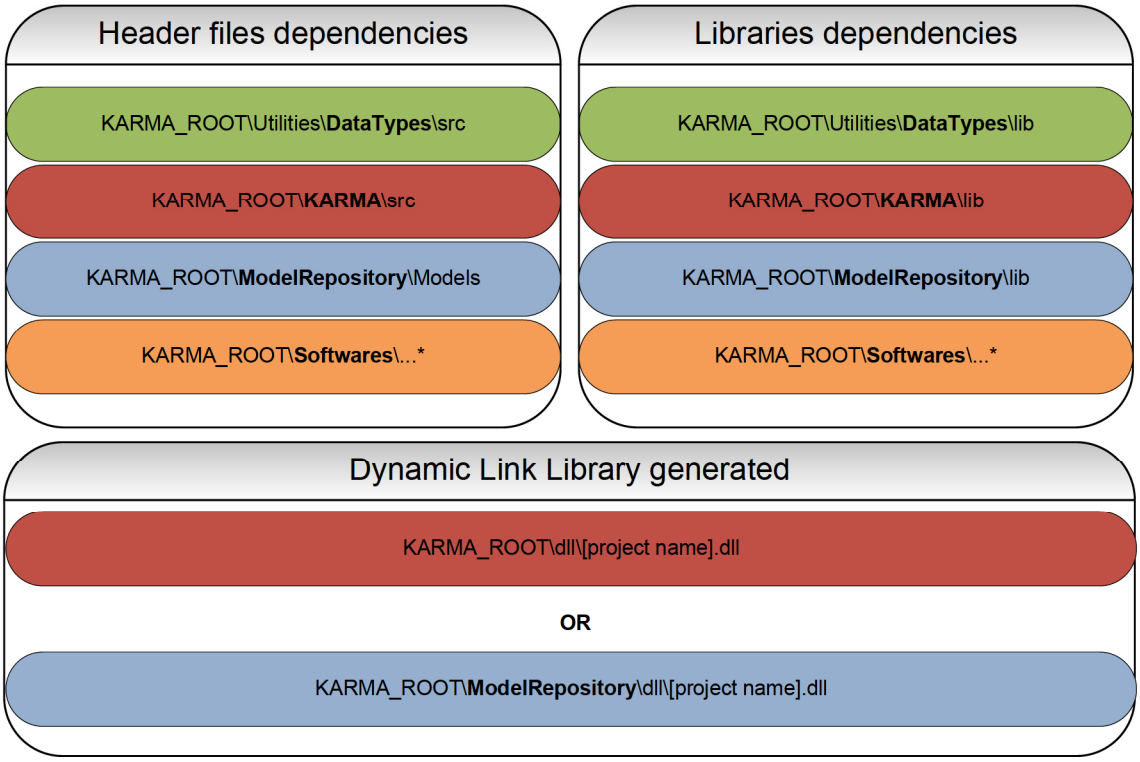

Figure 19: Typical settings of C++ models.

All third party libraries used in KARMA are located into the *Software* folder and dependencies are defined using *KARMA_ROOT*. Custom utility libraries are located into the *Utilities* folder (e.g. KARMA *DataTypes*). Architectural classes are gathered into packages under *KARMA_ROOT\src* folder and its package is specified (e.g. #include "*BaseEntity\Root.h*") when a header file is included into another class. Finally, when the C++ model is generated, the project name is generally used and the letter *d* is added at the end of the name to specify a debug version of the DLL. Please note that dependencies to specific models must be limited as much as possible, but when the header file of a model is included into another class, a detailed path must be supplied (e.g. #include "*Part\Airframe\AirframeModelA.h*" for models shown in Figure 18).

4.2.2 Simulink® models

There are two ways to define Simulink® models: using a model or library file. KARMA uses libraries to gather Simulink® models of a given category. This approach allows reusing the model into a different context or using the model inside a Simulink® scenario without having to worry about modifications in the original model. Models referencing to a model inside a library are kept linked to the library to allow for automatic update when the model is modified in the library. Every model is made available into the Simulink® *Library Browser* shown in Figure 20 to ease model access. Moreover, utility blocks that can be used in many models are gathered into the *ModelRepository\Models\Utilities* folder. Simulink® *SFunctions* are also located into this folder.

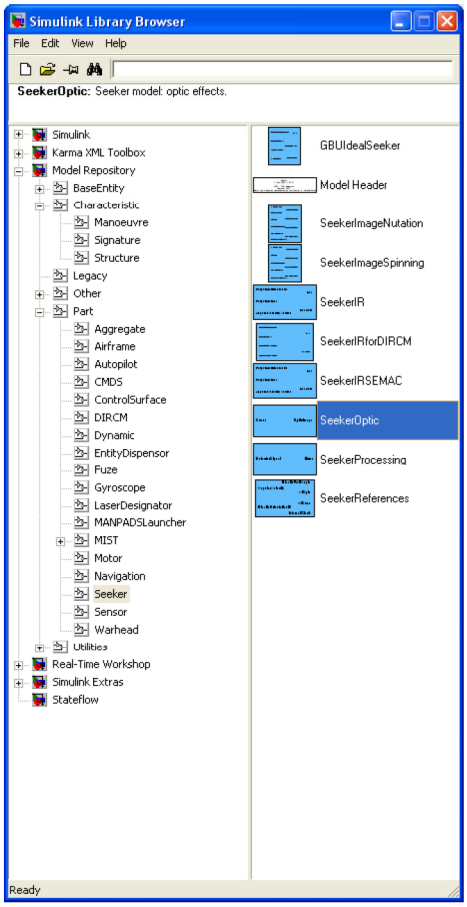

Figure 20: Simulink® models available into the Simulink® Library Browser.

A Simulink® model is converted into a KARMA model (DLL) by defining a separate model that is linked to the library. Inputs and outputs names are defined at this stage, and the GUI shown in Figure 21 is used to generate the DLL. This approach uses a script that generates a C++ KARMA model from a Simulink® model and is based on Real-Time Workshop® (RTW) that is presented in more details in Subsection 4.5.2.3. Dependencies are defined as for C++ models (see Figure 19), but this aspect is taken into account in RTW scripts.

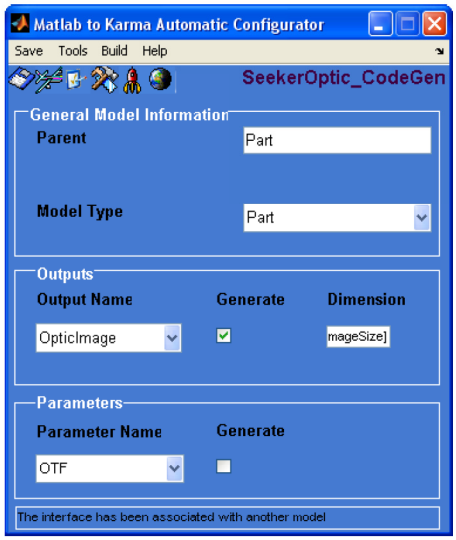

Figure 21: MATLAB® to KARMA Automatic Configurator GUI.

Sometimes different Simulink® models use the same library reference in order to maximize reusability. For example, airframe models are generated with aerodynamic coefficients hardcoded into the DLL. Therefore it is possible to create two different airframe configuration models (e.g. *AirframeA* and *AirframeB*) that are linked to the same model inside the *Airframe* library. When a modification is made to the *Airframe* model inside the *Airframe* library, airframe configurations will reflect these changes when regenerated.

4.3 How it works

This section presents the details about the main elements of the KARMA simulation framework that allows data exchange between models.

4.3.1 Hierarchy

First of all, there are three object types in KARMA: *BaseEntity*, *Part* and *Characteristic*. These were defined previously in Subsection 3.2.1. Hierarchies are made to tailor the model's behaviour using different interfaces that are shown in Figure 22 along with other basic types. When a similar concept exists in the standard Real-time Platform Reference Federation Object Model (RPR FOM), the RPR FOM hierarchy and names have been used for compliance. Models are used through their interface (e.g. *AbstractMAWS*) in order to minimize dependencies and allow for switching from a model to another seamlessly. All interfaces are located into the architectural section of KARMA and are referenced by models located into the *Model Repository*. Typically, interfaces are named using the *Abstract* prefix or using the letter *I* as prefix for pure interfaces.

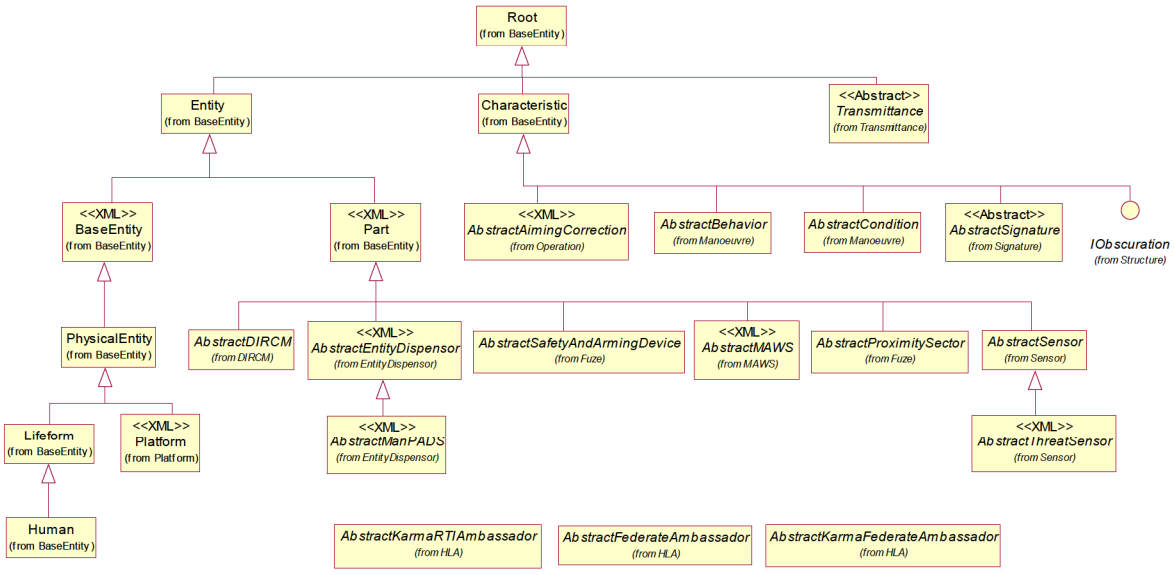

Figure 22: General hierarchy of KARMA models.

4.3.2 Data exchange

All KARMA models have a common base class that is responsible for data exchange between models (parameters and outputs) and for storing subsystems composition: the *Root* class. This class allows a model to be composed from many models that are derived from the *Root* class. The composition is recursive meaning that a sub-system can be composed of subsystems. For example, an aircraft (*BaseEntity*) that is composed of a protection suite (*Part*), an infrared signature (*Characteristic*) and missile and/or flares (*BaseEntity*). Furthermore, the protection suite can be composed of other subsystems (e.g. sensors, CMDS, etc.).

Figure 23 shows a class diagram for the three object types in KARMA, including the *Root* class. At this point it is important to distinguish between parameters, inputs and outputs. Parameters are assumed to remain unchanged for the whole simulation while outputs will probably change according to internal states and inputs. Finally, inputs are not buffered into each models, they are linked to another model output instead. This association is performed during simulation initialization and is detailed in Subsection 4.3.8.2. Therefore, as soon as an output is updated in a model, other models depending on this value see the new output value.

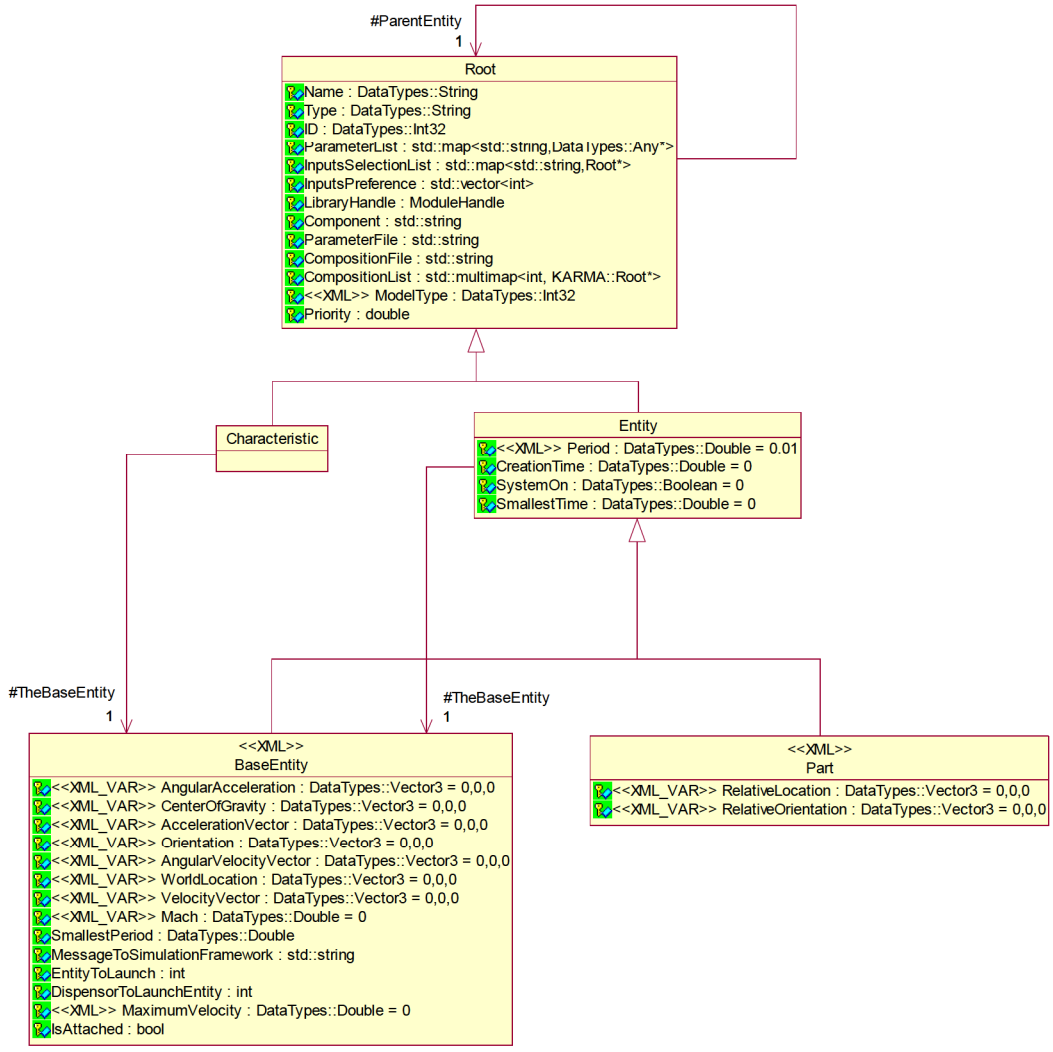

Figure 23: KARMA object types UML® class diagram.

Model parameters and outputs are defined as attributes in the different classes of the hierarchy and a reference is kept to allow other model accessing those values. A map container is used for parameters and outputs. Methods are available to set a reference, shown in Figure 24, and retrieve a value.

```
void KARMA::Root::setParameterReference(const char* name, DataTypes::Any*
value)
{
    typedef std::pair <std::string, DataTypes::Any*> map_Pair;
    m_parameterList.insert(map_Pair(name, value));
}
```

Figure 24: Source code of the setParameterReference method.

KARMA *DataTypes* are used to store the values. Data exchange is accomplished using a map container that stores references on parameters and outputs (*Any* objects at the *Root* level). *DataTypes* are presented in Subsection 4.3.4. Inputs are associated to a model providing the output and a *getInput* method, shown below, is used in the processing to access outputs of another model.

```
KARMA::Root* KARMA::Root::getInput(std::string inputName)
{
    return m_inputsSelectionList[inputName];
}
```

Figure 25: Source code of the getInput method.

There are some outputs available for all *BaseEntity* and *Part* models. Outputs giving the entity position, orientation, velocity, acceleration and angular acceleration are available for every *BaseEntity* models while the outputs giving the relative position and orientation of a *Part* are available for every *Part* models.

4.3.3 Composition

One of the main features of KARMA is its model composition. As mentioned previously in Subsection 3.5.1, models are created according to a specific type and are based on different interfaces to represent correctly their behaviour. These models are used into many configuration through composition based on XML files. This way KARMA models are assembled to build a high-level model. For a particular model, a composition XML file contains a list of models having their associated parameter and composition files. Composition is recursive and information is stored at the *Root* class level (composition multi-map) once a composition file is loaded in memory for a given scenario. The composition map is filled during the initialization stage of a simulation execution and is presented in Subsection 4.3.8. *EntityType* is used to associate a model to a type and many models of the same type are allowed in the composition. An example of a composition file is presented in Annex A. It shows an example of an aircraft that is composed of many parts (Entity dispensor, Threat system and Expendable dispensor), characteristics (Structure aircraft, Manoeuvre and Signature) and base entities (KARMA missile, KARMA flare).

4.3.4 DataTypes

KARMA *DataTypes* are used for every XML parameter and output defined into models and their interfaces. This approach is explained by the unique way to manage all types through a common type called *Any*. The UML® class diagram is presented in Figure 26.

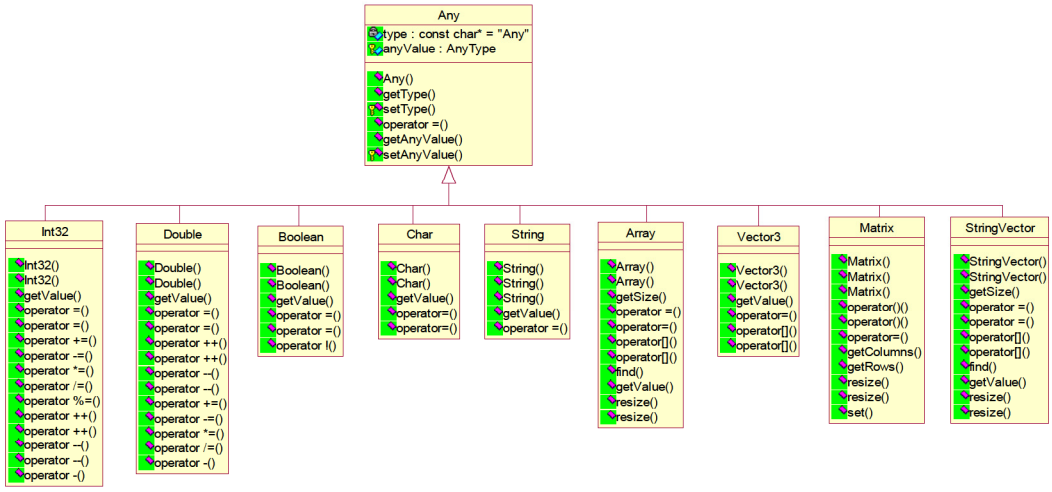

Figure 26: DataTypes UML® class diagram.

When a parameter is retrieved from a model (the outputs-parameters map), its type is assumed and a static cast is used (e.g. *WorldLocation* is casted into *DataTypes::Vector3*). If the *DataTypes* are not casted properly, then an error is generated and the simulation is aborted.

More details about the *DataTypes* implementation are presented into the document DataTypes.pdf.

4.3.5 Coordinates

Position and orientation are defined at two levels in KARMA: *BaseEntity* and *Part*. Any entity in the *Theatre* has a position and an orientation defined in the earth reference system (also called inertial system). Every *Part* (subsystem) of an entity has a position and an orientation relative to its parent, the *BaseEntity*. Conversely, every *Part* of a *Part* is relative to its parent, but this time the parent is a *Part*. When multiple levels of composition are used, transformations are performed to convert a relative value into another reference system. Earth and *BaseEntity* (also called body) reference systems are commonly used.

The KARMA *Coordinate* library offers the following transformations and the UML® class diagram is shown in Figure 27:

- translation;
- rotation;
- transformation from one system of reference to another;
- conversion from Euler angles to quaternions and vice versa;
- transformation from cartesian to polar coordinates and vice versa;
- transformation from cylindrical to rectangular coordinates and vice versa;

- transformation from spherical to rectangular coordinates and vice versa; and
- transformation from spherical to cylindrical coordinates and vice versa.

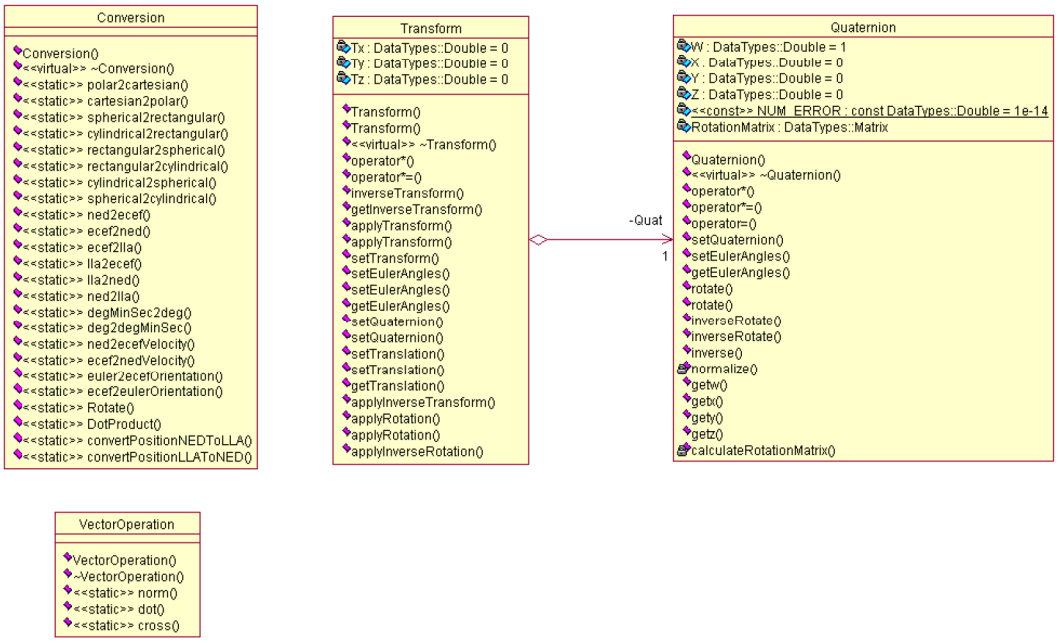

Figure 27: Coordinate UML® class diagram.

Formulas of the *Coordinate* library and how to use transformations are presented into the document CoordinatePackage.pdf.

4.3.6 Simulation

The main steps involved during a simulation execution are presented below and shown in Figure 28.

- Initialization – the simulation modules (e.g. *CollisionManager*, *Logger*, and *Theatre*) are initialized as well as the entities (and their associated models) used in the scenario.
- Execution – models are executed by calling their *run* method until the end of a simulation and a summary of the simulation execution is updated.
- Termination – the data logged during a simulation execution is saved in a log result file and memory released.
- The *SimulationEnvironment* class is the base class of the scenario execution. As presented in Subsection 2.2.1, this class calls several modules according to the scenario needs. The main steps and modules are presented in following sections.

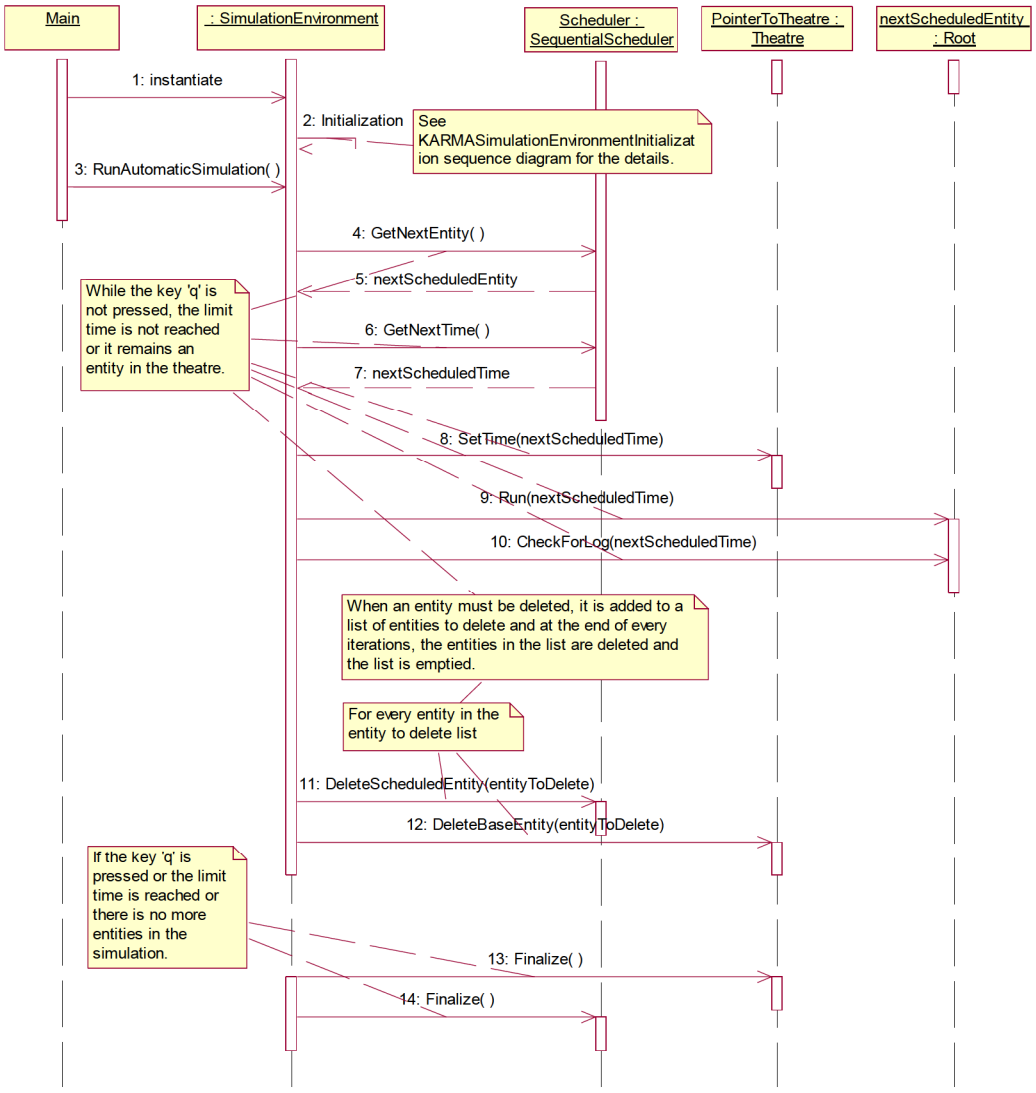

Figure 28: Simulation (main steps) UML® sequence diagram.

4.3.7 Execution

Simulation execution is controlled by the *SimulationEnvironment* class that offers methods for controlling execution as listed in Table 5.

Table 5: Main methods of the SimulationEnvironment class.

Method	Description
initializeSimulation	Prepares a simulation by specifying a scenario file, a duration and a simulation mode (*real-time* or *as fast as possible*). Then, loads the scenario file and add entities to the *Theatre*.
initializeEntities	Initializes all entities in the *Theatre*.
Update	Runs entities and their parts until a specific simulation time is reached.
terminateAction	Terminates the simulation, shows entity states and saves the simulation log results file.

SimulationEnvironment is used by different programs such as the KARMA simulator (*karma.exe*), the KARMA Viewer3D and any adapter for a COTS simulation environment. The KARMA simulator is a program used to execute a simulation locally or using a client-server architecture. This simulator is developed using the *Main* project of the KARMA Visual Studio® solution and uses the *SimulationEnvironment* class to control simulation execution. The KARMA simulator is launched from a command prompt or using a batch file. KARMA Studio performs simulations using this program.

The *Server* class implements the User Datagram Protocol (UDP). The maximum length for the messages is 512 bytes. When a simulation is executed from KARMA Studio, the KARMA simulator is responsible for activating a server port and calling *SimulationEnvironment* methods.

Refer to the document Communication.pdf for more details on the communication architecture.

4.3.8 Initialization

An initialization stage is performed before beginning a simulation execution. The *SimulationEnvironment* initializes the simulation modules (e.g. *Theatre*) as well as the entities and their subsystems. Figure 29 presents the initialization sequence diagram.

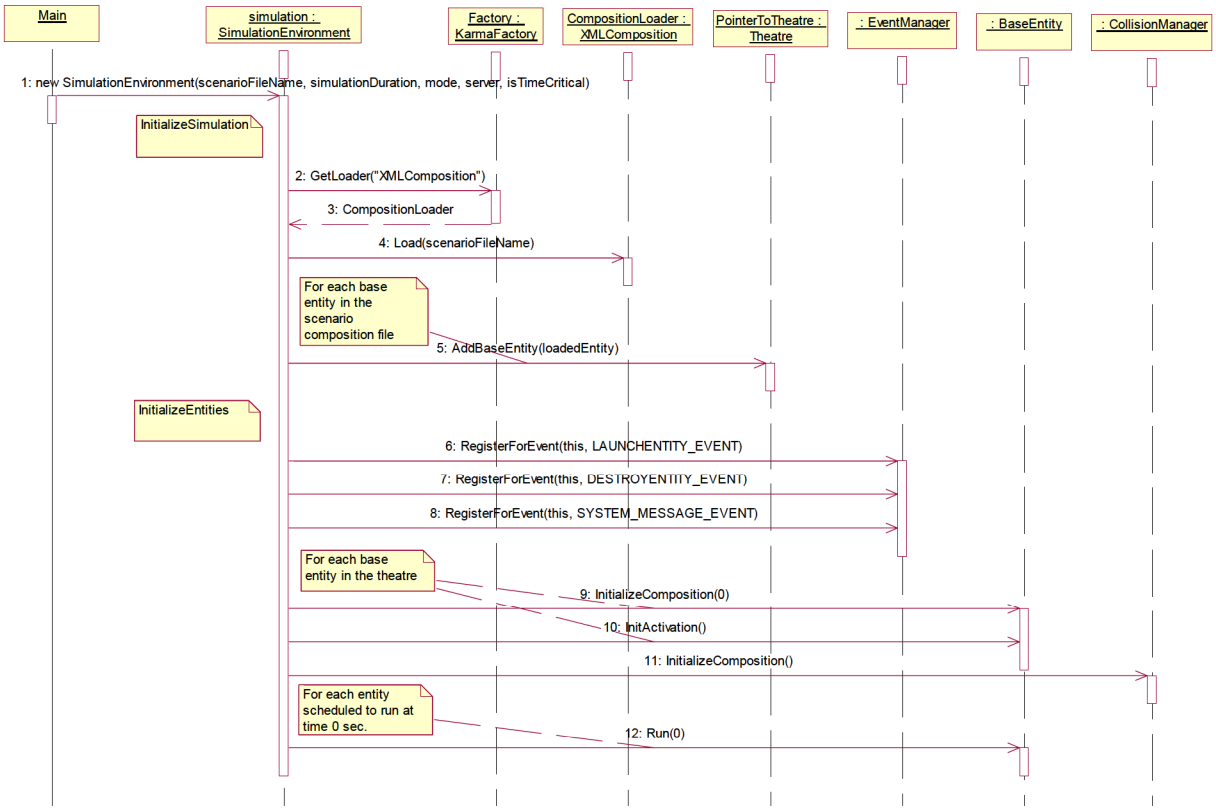

Figure 29: Initialization (main steps) UML® sequence diagram.

When the scenario being simulated is loaded, XML files are parsed to extract models used in the scenario, along with their parameters and their respective composition. This process is accomplished using the *Loader* library; its static class diagram is shown in Figure 30. Among the different kinds of XML loaders, the *XMLComposition* and *XMLParameters* loaders are used during the initialization stage.

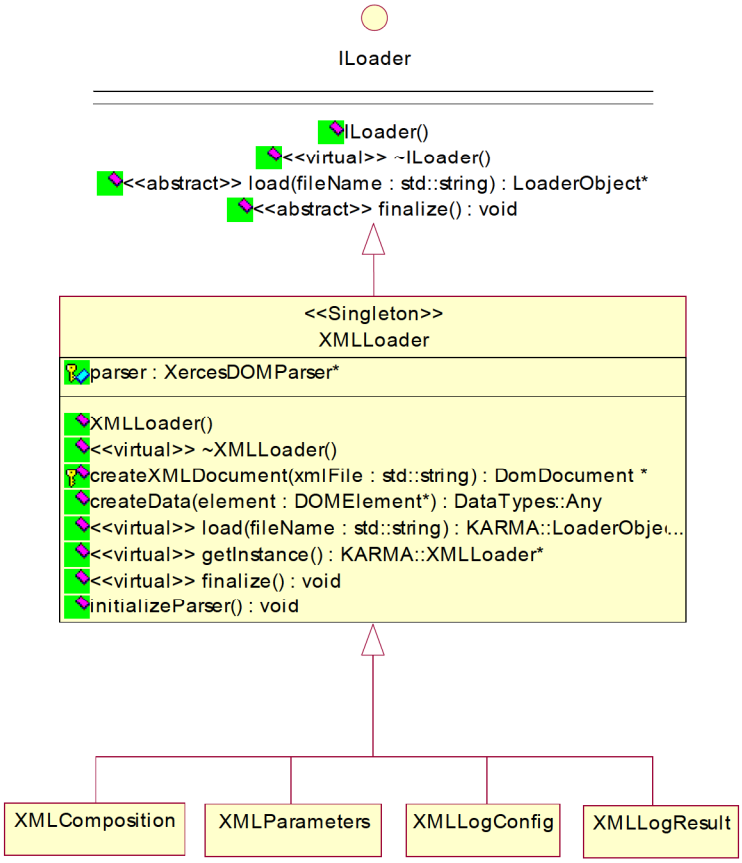

Figure 30: Loader UML® class diagram.

Figure 31 presents the sequence diagram for loading a scenario. *SimulationEnvironment* loads a scenario by calling *XMLComposition::Load*. For each model defined in the scenario, this method loads the model DLL into memory then creates an instance using a generic function named *CompositionFactory*. This function is implemented for each model by the way of a macro. Indeed, the *COMPOSITION_FACTORY_MACRO* shown in Figure 32 calls the model constructor with three arguments: instance name, model parameter file and model composition file. These arguments are used in the *InitCreation* call, allowing model parameters to be loaded using the *XMLParameters* loader and then its composition to be loaded using *XMLComposition* loader. This is a recursive process that allows any composition levels. Other initialization activities that are independent from the composition (other models) are performed into the constructor.

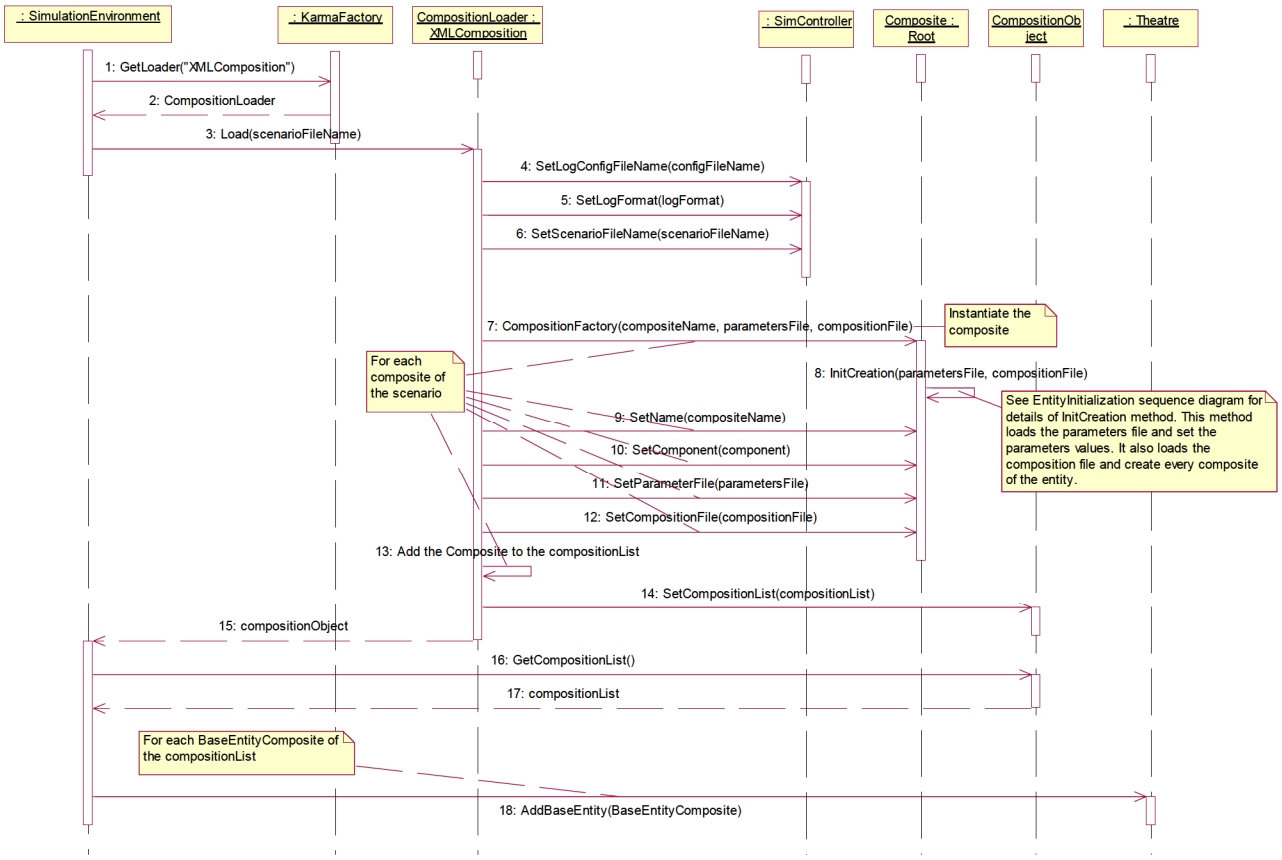

Figure 31: Scenario loading UML® sequence diagram.

```
#define COMPOSITION_FACTORY_MACRO(CHILD)\
    extern "C" __declspec(dllexport) KARMA::Root* CompositionFactory
(std::string name, std::string parametersFile, std::string compositionFile);\
KARMA::Root* CompositionFactory (std::string name, std::string parametersFile,
std::string compositionFile)\
{\
  KARMA::Root *object = new CHILD(name, parametersFile, compositionFile);\
  return object;\
}
```

Figure 32: Source code of the COMPOSITION_FACTORY_MACRO macro.

Each class registers its inputs, parameters and outputs during the call of a model constructor. The *SetParameterReference* method is a service implemented in the *Root* class and associates a name to a reference on an attribute. The *ParameterList* is used to gather parameters and outputs. Model inputs are stored into a separate list named *InputsSelectionList* by associating a name to a reference on a *Root* object. This reference is NULL initially until the model providing the input is found (see Subsection 4.3.8.2).

4.3.8.1 Model initialization

The model initialization is divided into three steps: creation, initialization and activation.

- Creation – the *InitCreation* method is called when a model is created and is responsible for setting model parameters (and initial outputs) and populating composition. Figure 33 presents the sequence diagram for this initialization step.
- Initialization – the *InitComposite* method is called once all models have been created and is responsible for inputs location as presented in Subsection 4.3.8.2. This step is useful to perform any initialization activities that depend on other models (e.g. explicit link to a model from the composition) but that are independent of run-time conditions. Everything that is performed during this step reduces the simulation execution time since simulation is not started yet.
- Activation – the *InitActivation* method is called when the *BaseEntity* associated to a model is inserted in the *Theatre*, ready to be executed. Usually, activation is made when starting the simulation execution or when an entity is launched during a simulation as presented in Subsection 4.3.8.3. However, it is possible to activate an entity before launching using an *Activate* behaviour (*Characteristic*).

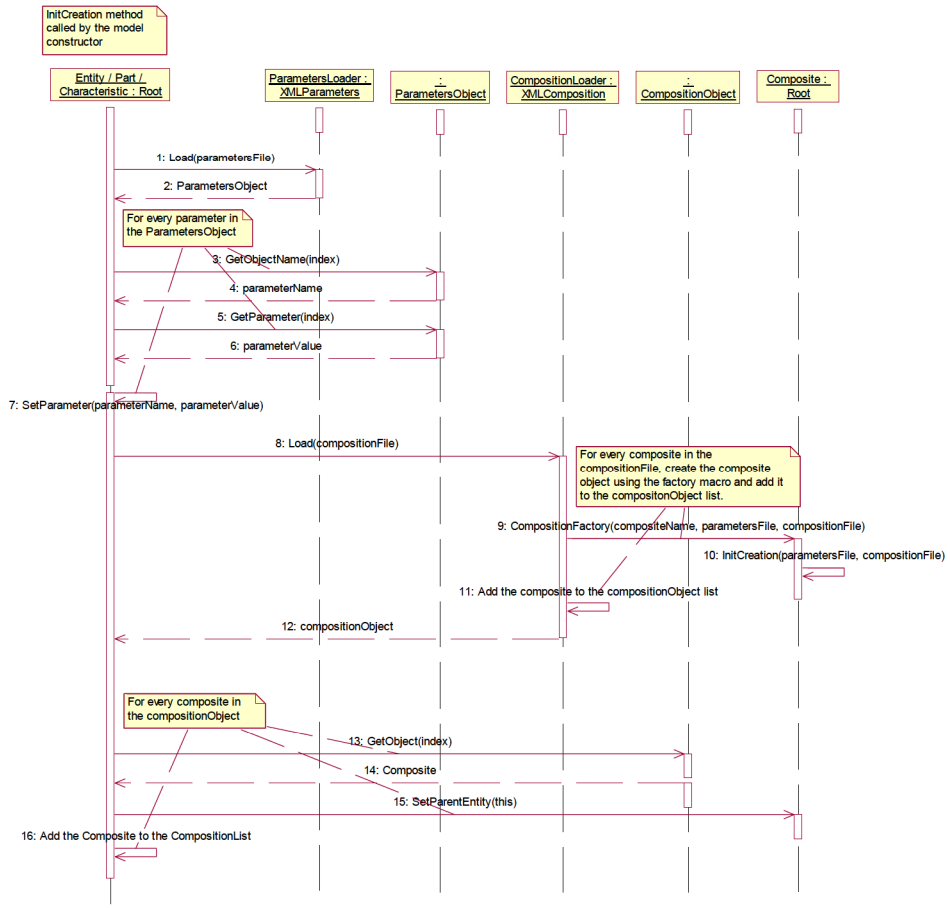

Figure 33: Model initialization (InitCreation) UML® sequence diagram.

4.3.8.2 Input initialization

Inputs are initialized during the initialization step (*InitComposite*) of a model initialization. The process for locating model inputs is performed using the *InitializeInputs* method implemented at the *BaseEntity* and *Part* levels. An input represents a reference to a model that generates the corresponding output. Therefore, during input initialization, every input of a model is associated to the output of another model. Outputs are searched using the exact name of the input, according to the search order described in Subsection 3.5.3, from any composition level of its associated *BaseEntity*. When there is a match, a reference is set into the *InputsSelectionList* for the input name. If an input is not found, simulation is aborted with an error message.

4.3.8.3 BaseEntity as equipment

Sometimes, *BaseEntity* are used into a model composition for equipment purpose (e.g. missiles and flares). These entities are created and initialized before beginning a simulation execution, during the initialization stage (corresponding to the first two model initialization steps). However, they are not inserted into the *Theatre* since these entities might be launched during simulation (see Subsection 4.3.12.1 for more details).

The attribute *IsAttached* is used to indicate that a *BaseEntity* is used as equipment. This attribute is set during the initialization step (*InitComposite* method). When an entity is activated before launching, while being attached to another *BaseEntity* (e.g. missile fixed onto a launch rail of an aircraft), only specific parts types from its composition can be activated. The *Activate* behaviour activates sensors, seekers and gyroscope types. Obviously, these components do not update the dynamic states of the attached entity. Indeed, while used as equipment, the dynamic states of a *BaseEntity* are updated using the *BaseEntity* that owns that equipment and assumes that the equipment is located directly into the *BaseEntity* composition.

The *Quantity* property, in the XML *BaseEntity* composition file, can be used to create many copies of the *BaseEntity*. Many copies of *Part* components can also be created using this property. Instances of the model are created and initialized but they will be activated only when the *BaseEntity* is inserted into the *Theatre*.

4.3.8.4 Log initialization

All the data that is logged during a simulation execution is saved into a log file. The sequence diagram of the log initialization in the XML format is shown in Figure 34. The log settings are applied for each model of the scenario and the log initialization is performed in the *initComposite* method of a model. The *XMLLogConfig* loader allows loading and/or retrieving log settings. Entries are created into the log file according to the log settings. Finally, every model that is going to log data during the simulation has its unique ID made negative.

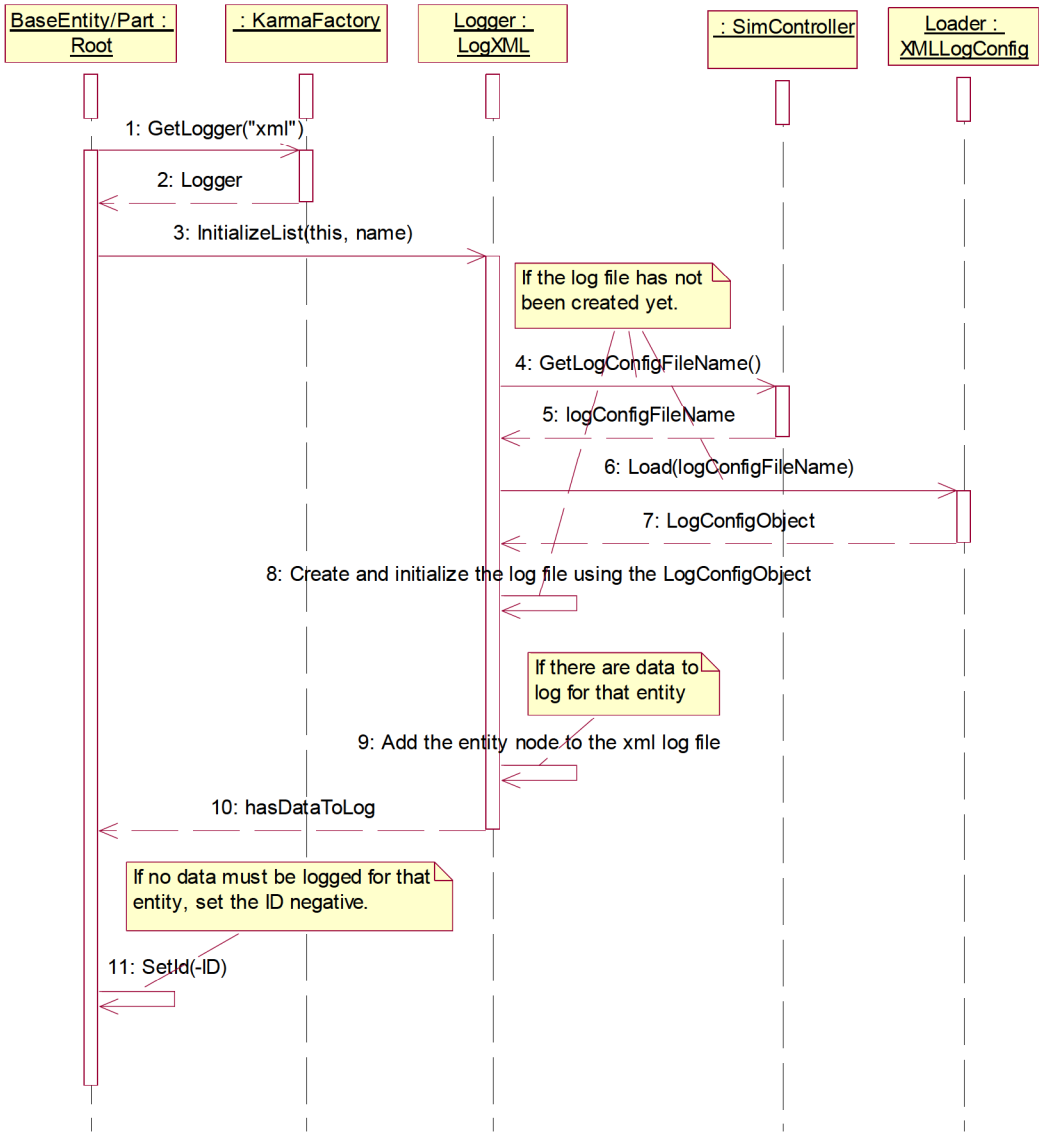

Figure 34: Log initialization (XML) UML® sequence diagram.

4.3.9 Sequencing

During simulation execution, models are executed by calling their *run* method until the end of simulation. These calls are made according to the sequencing mechanism presented below. Each model that is derived from *BaseEntity* or *Part* has a *Period* parameter. This parameter is used to schedule the next time when the model needs to be executed. During the initialization process, in

the *initActivation* method, the model schedules an execution call if the *Period* parameter is larger than 0 (in fact there is a limitation on the call frequency that is set to 10^{-8} sec.). The *run* method is called for the scheduled simulation time and the model can update its states and outputs.

Model sequencing is implemented into the *Scheduler* library. A multi-map for simulation time and *Root* reference is used to keep track of the models to be executed in time. A priority is attributed in the XML composition. This priority is added to the simulation time in order to sort models by priority. A *Priority* attribute is maintained into the *Entity* class. *Priority* is defined as an integer between 0 and 99 into XML, but is converted into a double value during initialization. This value is used by the scheduler for operating on the scheduling map. It is important to note at this point that for models being scheduled for the same time and having the same priority, the priority is only determined by the composition. That means the first model that is initialized and that performs its initial scheduling will be called before other models.

4.3.10 Data logging

KARMA allows logging model outputs, simulation events as well as a simulation summary that can be used to trace simulation execution. The latter is referred as the developer log. These three types of log are presented in the following subsections.

4.3.10.1 Output log

Simulation can be set to log outputs in two formats: XML and Text (TXT). For both formats, there is a XML configuration file called *LogConfig* that is used to setup outputs to be logged. This log is based on the model composition and uses model types. Figure 35 shows a typical XML log configuration file.

When a *BaseEntity* or a *Part* is executed, outputs are logged at the beginning of the *run* method, before updating outputs. The sequence diagram in Figure 36 presents typical calls involved into the data logging.

```
<fileType>LogConfig</fileType>
<destination><directory>C:\</directory><fileName>LogFile</fileName></destination>
<logEntity name="MISSILE">
	<logVariable frequency="0.1" name="WorldLocation" type="vector3"/>
	<logVariable frequency="0.1" name="Orientation" type="vector3"/>
	<logVariable frequency="0.1" name="Mach" type="double"/>
	<logPart name="SEEKER_IR_FOR_DIRCM">
		<logVariable frequency="0.1" name="LOS" type="vector3"/>
	</logPart>
</logEntity>
<logEntity name="AIRCRAFT_3DOF">
	<logVariable frequency="0.1" name="WorldLocation" type="vector3"/>
	<logVariable frequency="0.1" name="Orientation" type="vector3"/>
	<logVariable frequency="0.1" name="Mach" type="double"/>
</logEntity>
</data>
```

Figure 35: XML log configuration file example.

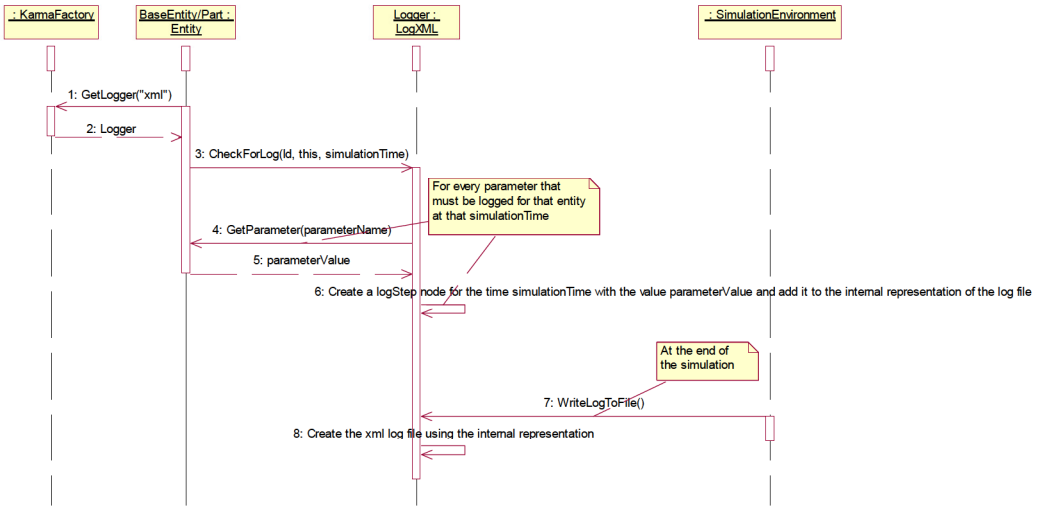

Figure 36: Data logging UML® sequence diagram.

4.3.10.2 Event log

Events are logged into the same log file as the outputs. The sequence diagram for event logging is shown in Figure 37. The method *logEvent* is responsible for this log. All events are logged if the flag to log the event is active. The method is called by the *EventManager* class when an event is received. Please refer to Subsection 4.3.11 for more details. Note that the events cannot be logged in a text format because the method is not implemented.

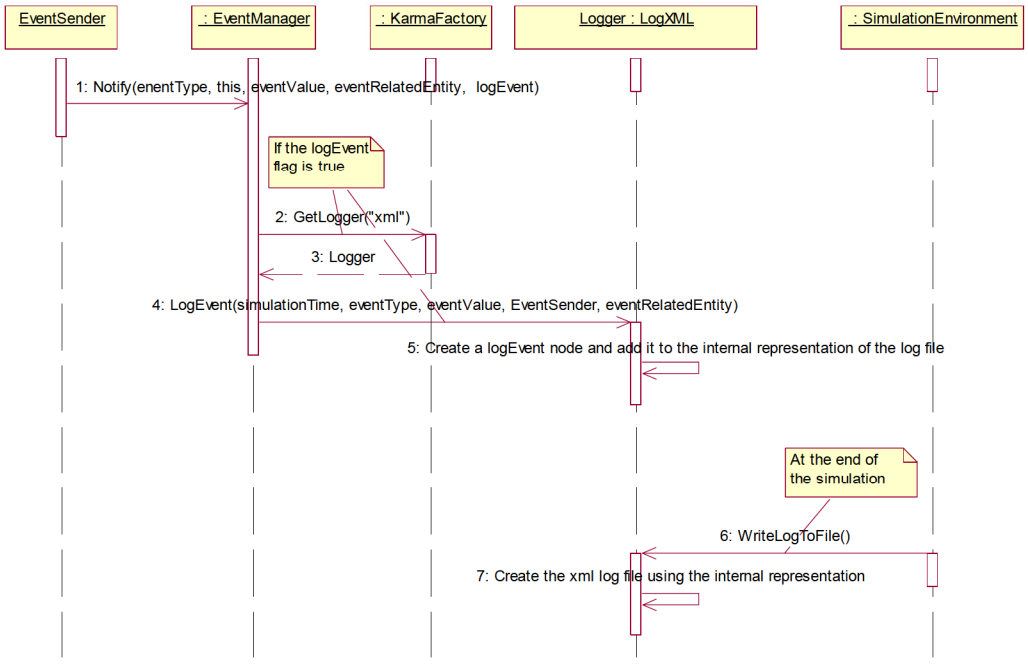

Figure 37: Event logging UML® sequence diagram.

4.3.10.3 Developer log

This functionality is only available for the debug version of KARMA. It is implemented by the *LogManager* class (*UtilityClass* library). This class is static, so it can be called in any class without having to instantiate it. There are some methods to write a string into a buffer and the buffer is written into a text file using *WriteLog* method. The log file is located into the *KARMA\src* directory and named *log.txt*. This log is used to monitor the main activities of the base services, namely the XML loading, initialization, and so on.

4.3.11 Events

Events can be used during simulation to notify other models that something occurred and might affect their behaviour or solely to inform the user. A parameter is used when notifying an event to specify whether or not the event needs to be logged (see Subsection 4.3.10). There are predefined events listed in Table 6, but other event types can be added in the *Events* header. Events are defined as an integer value and a string for display is associated. Informative events that do not change the simulation are *MESSAGE* events and are not handled by the models.

Table 6: Event types.

Event Type	Description
Any	Used to register for all event notifications.
Time	When a given simulation time is reached.
LaunchEntity	When an entity must be launched.
EntityLaunched	When an entity has been launched.
ObjectCreated	When an object is created.
SensorCreated	When a sensor is created.
Detection	When a model detects a threat.
MawsDetection	When a MAWS model detects a threat.
Declaration	When a model declares a threat.
MawsDeclaration	When a MAWS model declares a threat.
Dazzling	When a model performs a dazzling CM.
SendDircmCode	When a DIRCM model updates its code.
LaunchLaser	When a laser is being activated or deactivated.
Illumination On/Off	When a laser designator starts or stops target designation.
MissDistance	When the closest point of approach for two specific entities is reached.
Proximity	When a proximity is detected by a proximity target detection device.

Event Type	Description
Collision	When a collision between two entities or between an entity and the terrain is detected.
ObjectDestruction	When an object is about to be destroyed.
DestroyEntity	When an entity must be destroyed.
RunCompleted	When a model run is completed.
SimulationTerminated	When a simulation is completed.
SystemMessage	When an informative system message could be of interest to a user.
ChangeAnimation	When a 3D model animation of an entity must be changed in the 3D viewer.
Load2dImage	When a 2D image must be loaded in the 3D viewer.

There is an *EventManager* that is responsible for intercepting events and dispatching them to the models that has subscribed to the event. The *notify* method is used to send an event. A model that is interested in receiving events is derived from *IEventListener* class and implements the *eventResponse* method. The *Register* and *Unregister* methods are used to subscribe/unsubscribe to specific events. Figure 38 shows the UML® class diagram of the *Event* module.

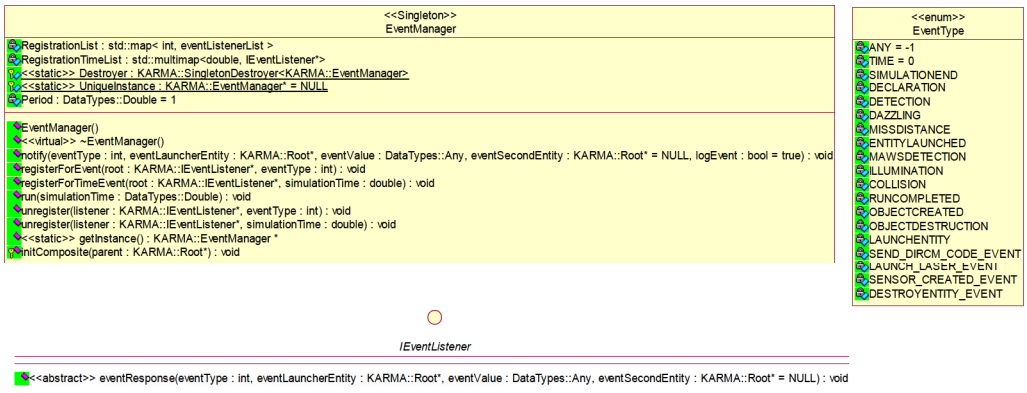

Figure 38: Class diagram of the Event library.

4.3.12 Dynamic behaviours

The outcome of a simulation execution is often dictated by the entities available in the *Theatre* and their interactions. In that sense, the following subsections present the main dynamic behaviour: launching entities, detecting entities, detecting collisions between entities and destroying entities.

4.3.12.1 Launching entities

As mentioned earlier, entities defined as equipment are created and initialized during the simulation initialization stage, but are not activated until launch time. As soon as an entity is launched, the following steps are performed. First the *BaseEntity* is removed from the composition, then inserted into the *Theatre*. Finally, the *BaseEntity* and its associated models are activated using the *InitActivation* method. Figure 39 presents the sequence diagram of entity launch.

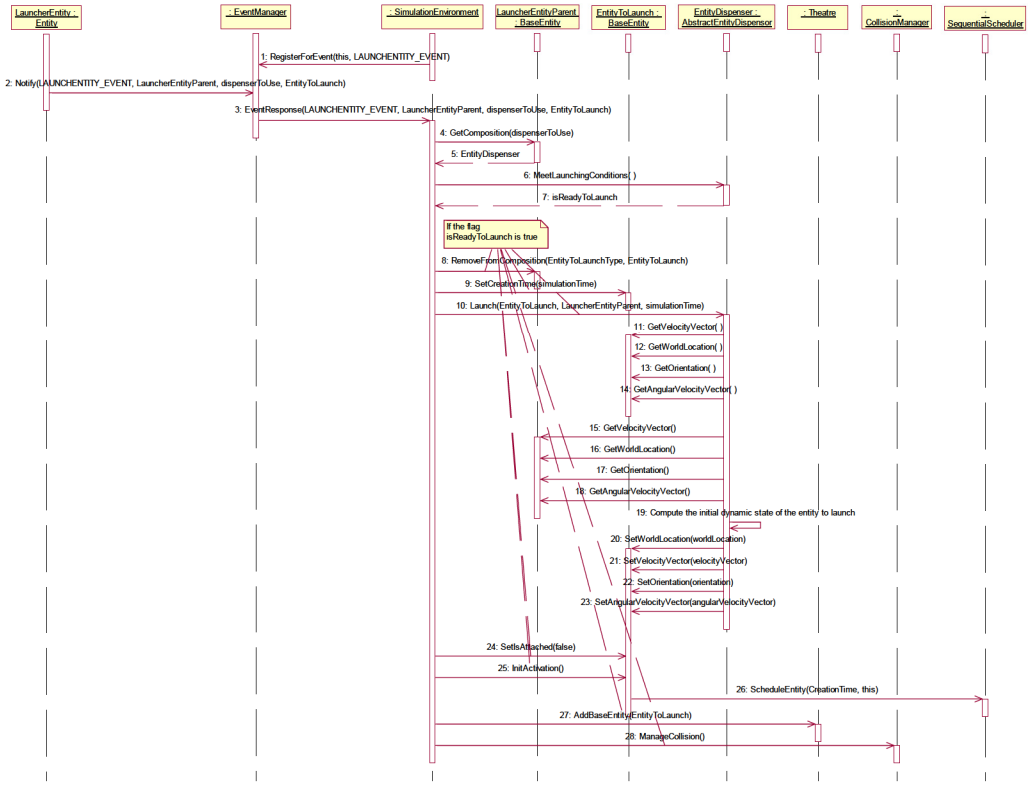

Figure 39: Entity launching UML® sequence diagram.

Dispensers are used to launch entities (e.g. launch flares or missiles). *AbstractEntityDispensor* is responsible for computing initial dynamic states when launch occurs. The inputs of the dispenser are the velocity, the location, the angular velocity and the orientation of the entity that wants to

launch flares or missiles. The outputs are the velocity, the location, the angular velocity and the orientation of the entity that will be launched. The parameters of the dispenser are the ejection velocity and the ejection angular velocity. These parameters will determine the velocity and the orientation of entity that will be launched.

4.3.12.2 Detecting entities

The *Intersection* class allows gathering all entities visible in a given FOV (e.g. a sensor or laser beam). An entity is considered visible in the FOV if its location (*WorldLocation*) is within the FOV region up to a specific detection range and is not hidden by another entity. The visible entities are added to the list of entities seen.

An entity is hidden if there is another visible entity closer to the sensor and its radius, assuming a sphere based on its largest dimension (width, height or depth), hides completely the entity. Figure 40 shows an example of a *Tested entity* that is hidden by *Entity 3*. *Entity 3* is considered seen by the sensor because it is not hidden by the three other entities.

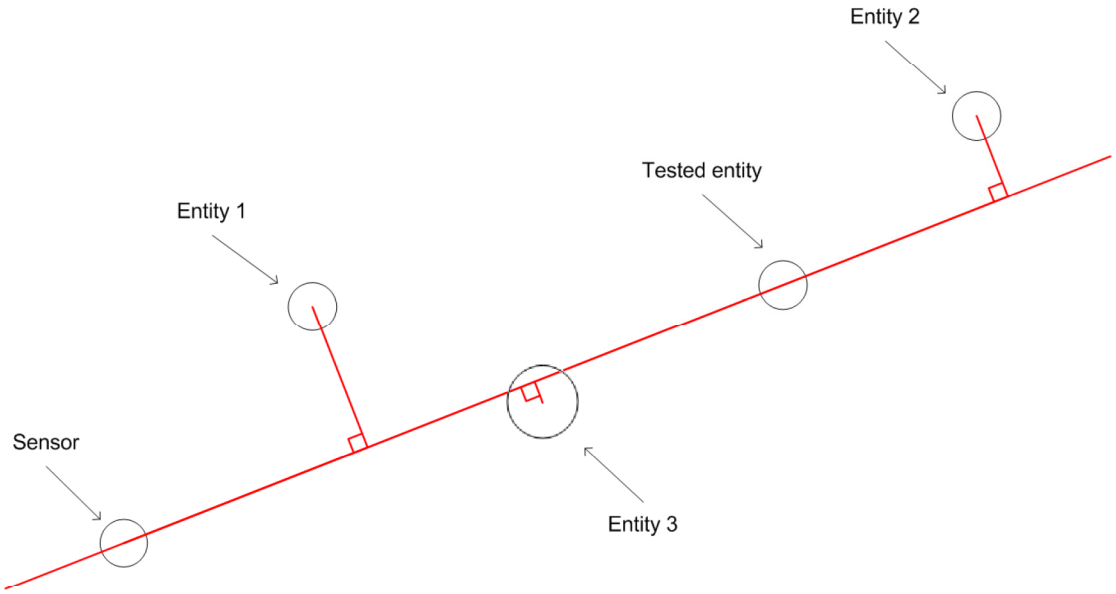

Figure 40: Intersection example.

In the actual implementation of KARMA, this class is used to declare a threat in the MAWS. Refer to the document Intersection.pdf for more details.

4.3.12.3 Detecting collision between entities

The collision detection between entities is activated into a KARMA scenario by using the collision model (*Collision* library). There are two types of collision mechanisms: *BoundingSphere* and *CollisionDetection3D*.

BoundingSphere – simple algorithm based on a sphere that is calculated for each entity using its largest dimension (width, height or depth) as a radius. Collision detection is called at a variable pace, according to the shortest period for which a collision between two entities might occur. Collision detection must be performed as soon as an entity is launched to update the time for the next collision detection call.

CollisionDetection3D – algorithm based on the intersection of 3D geometries defined in a 3D model associated to the *BaseEntity*. Collision detection is called at a fixed pace, according to the *Period* of the collision detection algorithm.

For both collision algorithms, collision detection is performed at the beginning of the simulation time, before advancing in time. The UML® class diagram shown in Figure 41 presents the main classes involved in collision detection.

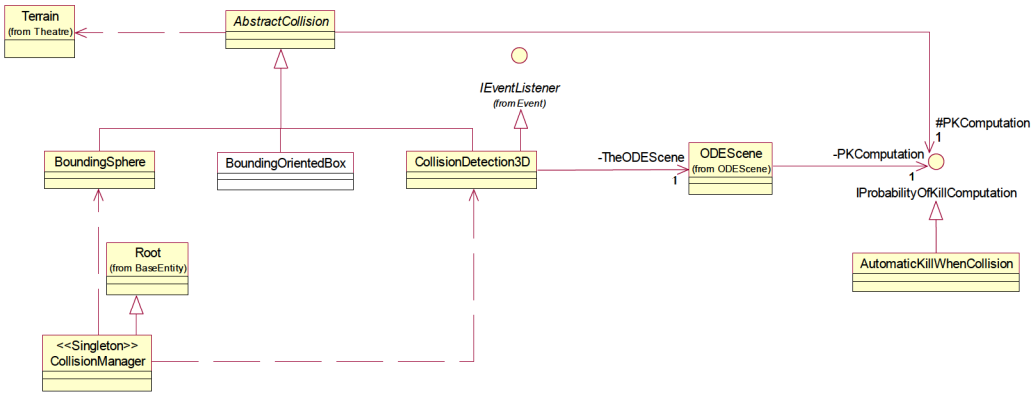

Figure 41: Collision detection UML® class diagram.

The sequence diagrams for each collision algorithm are shown in Figure 42 and Figure 43. The *CollisionManager* calls the appropriate collision algorithm and collision is checked between every entities. When a collision is detected, an event is launched and the *collide* method is called for both entities. A simple probability of kill method is implemented for the moment. The level of damage is 100% when there is an impact between two heavy entities (aircraft, ground vehicle and ship) or for the life form implicated in an impact, otherwise the level of damage is 0%. Thus, if a missile intercepts an aircraft, the aircraft will not be deleted, neither the missile. Additional behaviour must be implemented if the entity needs to be destroyed. This is the case for the missile in the preceding example. The *collide* method is simply overridden at the *Munition* level and detonation is activated before destroying the missile.

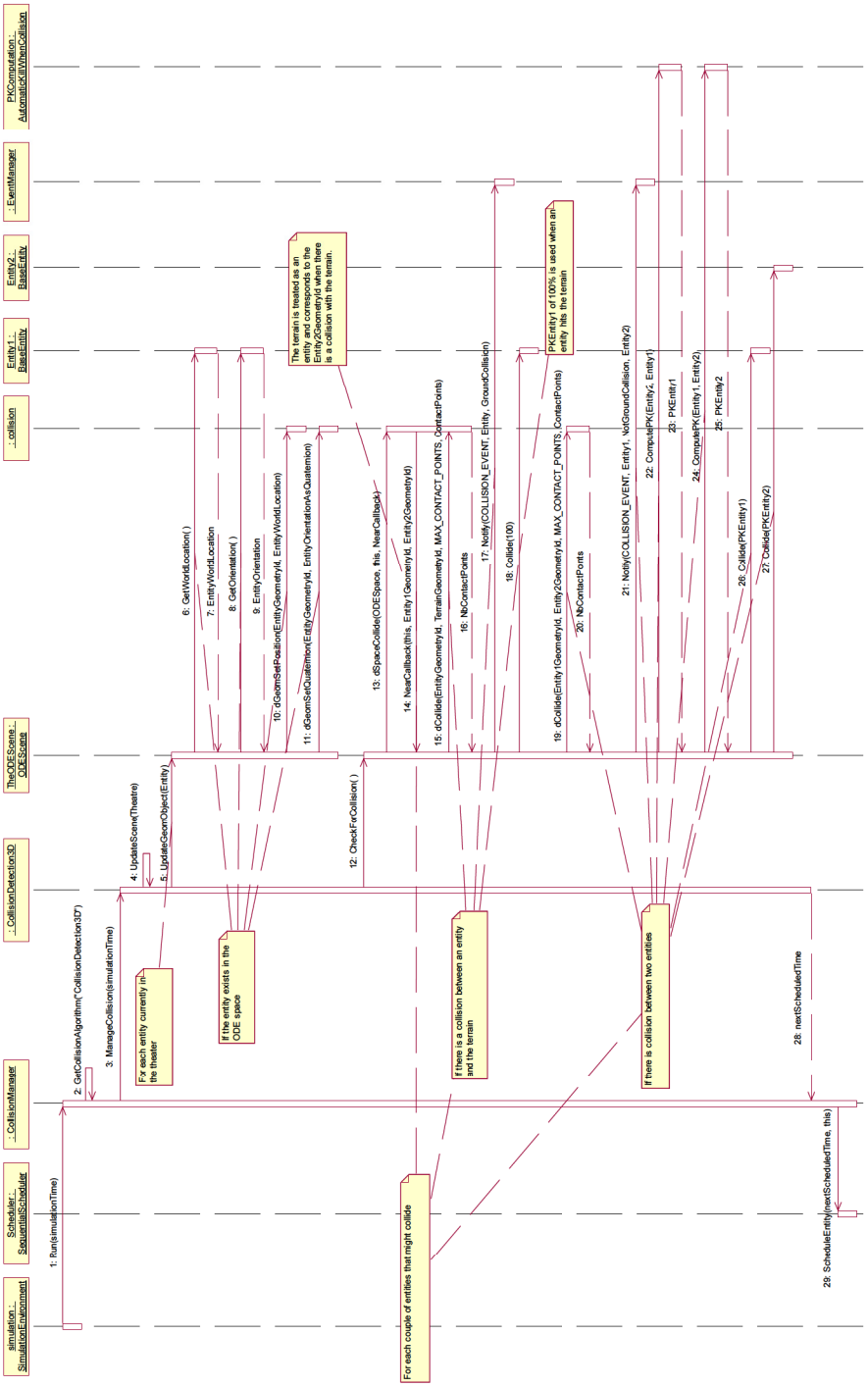

Figure 42: 3D collision detection UML® sequence diagram.

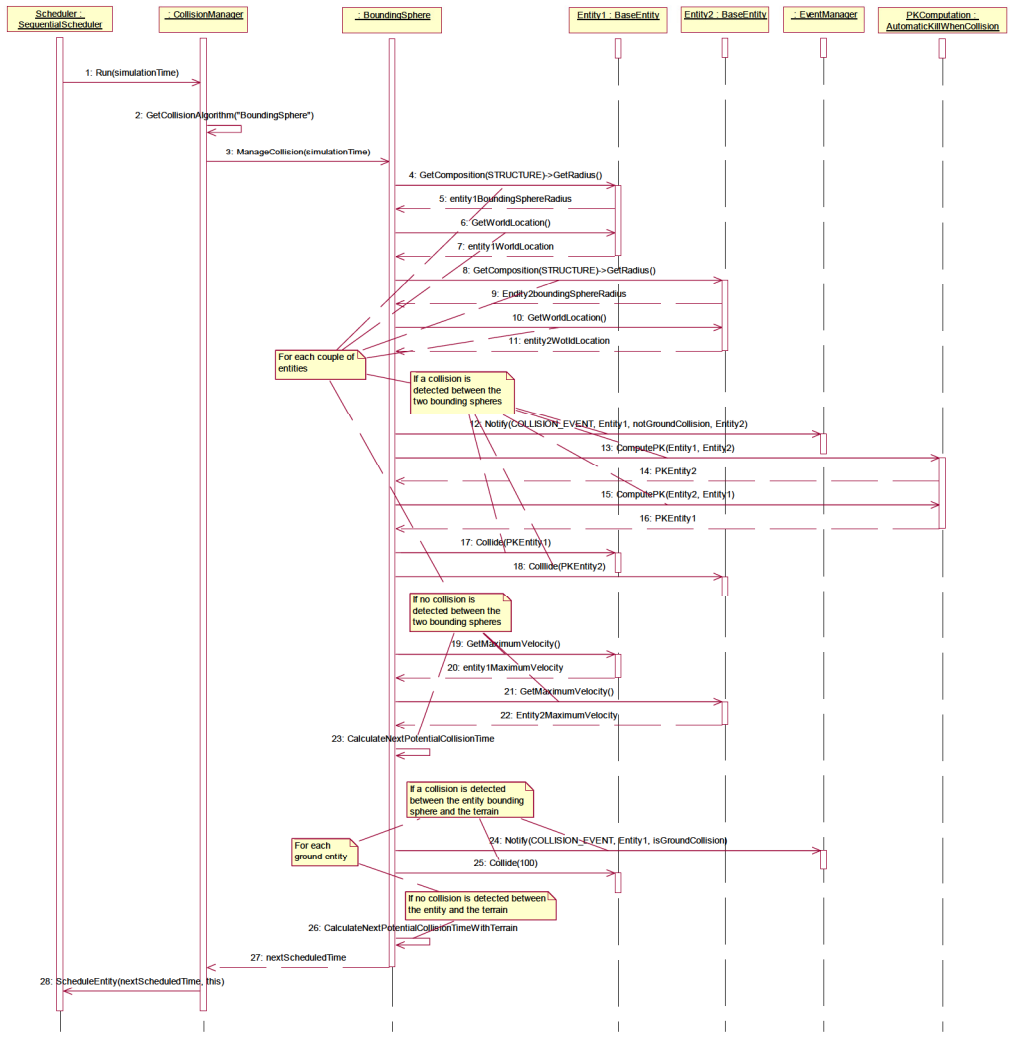

Figure 43: BoundingSphere collision detection UML® sequence diagram.

4.3.12.4 Destroying entities

When an entity is destroyed and removed from the *Theatre*, some steps must be followed, as shown in the sequence diagram in Figure 44. A *DESTROYENTITY_EVENT* event is sent to destroy the entity. The *SimulationEnvironment* subscribes to this event and is responsible for the entity destruction. Entity deletion is postponed until all models involved into the simulation are executed for the current simulation time (corresponding to the destruction time). To insure that invalid references will not occur, any model that keeps a reference to a model must subscribe to this event and perform appropriate actions prior to the entity destruction. This approach allows other models to perform action for the current simulation time, before the entity is effectively

destroyed. Once all models are executed and the simulation time is ready to be changed, entities are destroyed and removed from the *Theatre* and the DLL is freed.

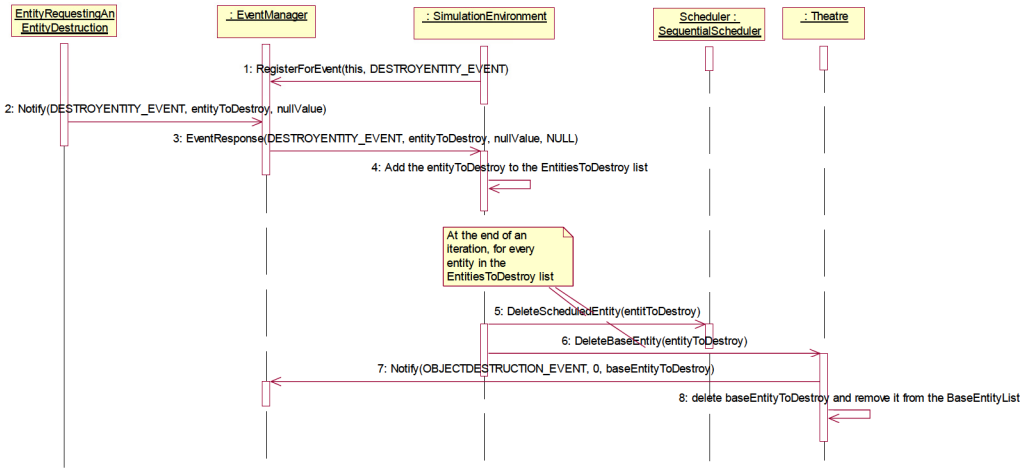

Figure 44: Entities destruction UML® sequence diagram.

4.3.13 HLA interface

The KARMA simulation framework has an HLA layer designed to allow a KARMA simulation to participate as a federate in an HLA federation. The KARMA HLA module is designed to be as generic as possible to efficiently support multiple RTIs and multiple FOMs. Although the HLA layer is not fully RTI and FOM-agile, it is an important step in that direction. Figure 45 illustrates how KARMA interfaces with two RTIs and two FOMs through an HLA Manager and a FOM Mapper. Figure 46 shows the KARMA HLA architecture that is described in detail in the document Support to CASE Griffon Mothership – DRDC Valcartier ECR 2006-424.pdf [11]. The KARMA HLA layer may not be fully functional any more since it has not been maintained recently.

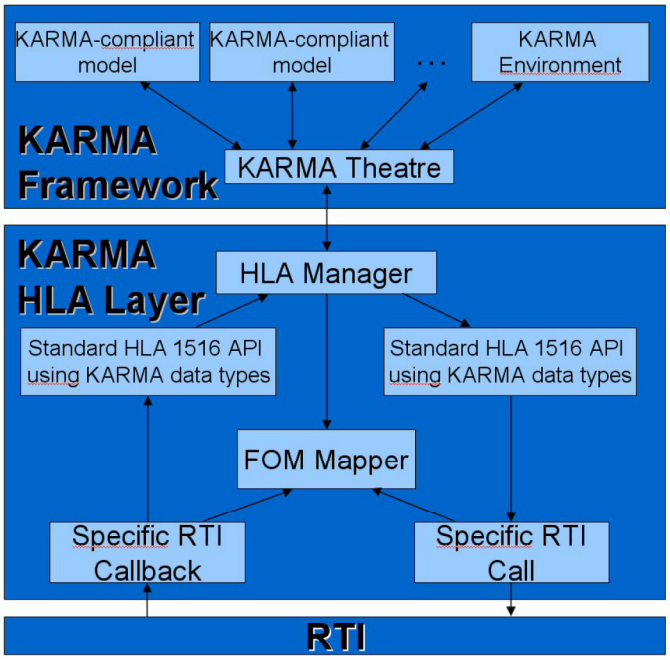

Figure 45: KARMA HLA layer.

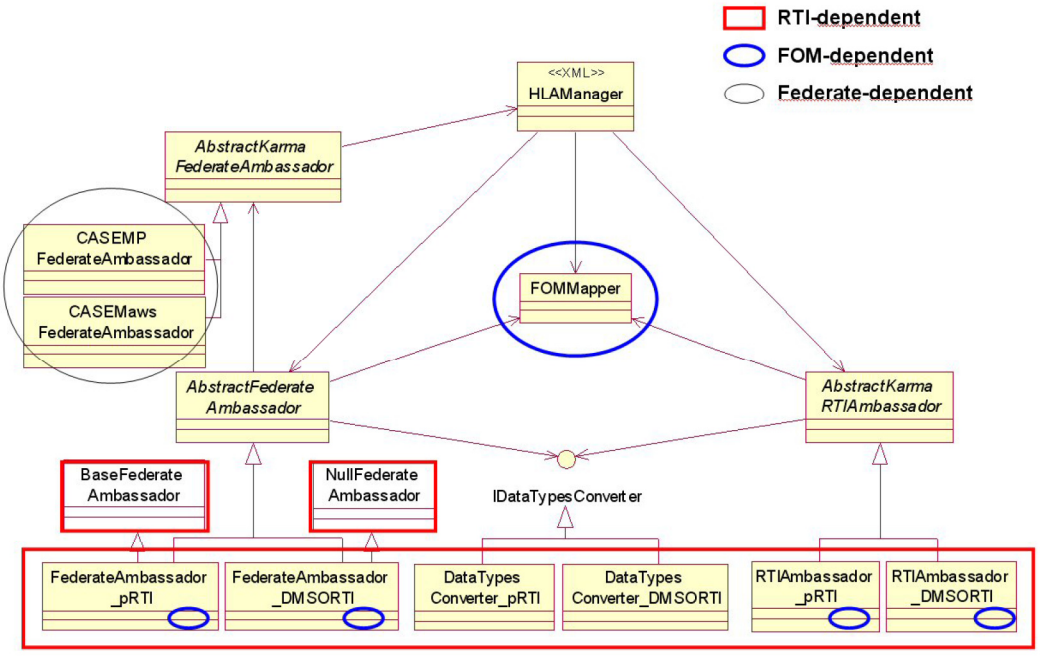

Figure 46: KARMA HLA architecture.

4.4 Work approach

The development guidelines are presented into the document KARMA Development Guidelines.pdf and the programming conventions are listed into the document KARMA Coding Standards.pdf.

The KARMA team has been structured to foster communication between projects by discussing project needs, past/current/future developments and also debating architectural matters since any changes might impact more or less other projects. The KARMA team development organization is presented in the document KARMA Team Structure.pdf. Each project has a delegate architect which attends weekly architectural meetings. This delegate architect is responsible for all development and modelling in its project; he must check that the KARMA philosophy is preserved and the standardized modelling and development practices are followed by every member of his team. The delegate architect is the first person contacted by the other team members of a project for any aspects related to KARMA. In addition to the architectural meetings, quarter meetings are held to inform members of different projects about approaches, tools, developments, needs or any subject related to KARMA.

Abstract classes or interfaces, like the one in the class diagram shown in Figure 22, shall be used whenever it is useful in order to reduce dependencies and to standardize the behaviour of the models. This is performed at modelling time. As one can see, the object types are specialized to meet subsystems requirements.

Sometimes bugs come up when several people work on a complex software like the KARMA simulation framework. To help the KARMA developers to locate problems in the code, KARMA and each model created in C++ or Simulink® must be generated in a debug version as well as in a release version. Obviously, the release version of KARMA and of the models is far more efficient than the debug version. Also, the KARMA developers have decided to create a package, *LogManager*, to help them to locate problems in the code. It is sometimes very useful to have some information logged for debugging purposes. The KARMA team has developed a central message system that logs information for developers only in debug mode.

4.5 Tools and libraries

KARMA development and modelling is based on some commercial tools, libraries and custom tools.

4.5.1 Commercial tools

Commercial tools can assist in the development and modelling of the KARMA simulation framework and of its models, as well as in the edition of 3D models and documentation.

4.5.1.1 Modelling

KARMA is designed according to the UML® standard and using the Rational Rose® (IBM) visual modelling tool. Every classes of the KARMA simulation framework, including C++ and

MATLAB® models are modeled using UML®. Use cases, sequence diagrams and class diagrams are mainly used. Classes are gathered into packages (e.g. *BaseEntity*, *Theatre*, *Signature*) to ease navigation through many classes and these packages are saved into separate units (*.cat* files) to minimize conflicts due to team working. Components (*.sub* files) are created for each dynamic library that is created. These components are used for code generation. KARMA development starts from a UML® specification of the models. Rational Rose® generates class skeletons and programming is performed into Visual Studio® as presented below. In addition to code generation, XML parameter file are generated automatically using scripting capabilities of Rational Rose® (see Subsection 4.5.2.3 below).

As mentioned earlier, scenarios are defined using XML files. Each type of XML file (composition, parameters, log configuration, log results and batch run) has an associated schema that is used for validation. These schema are edited using XMLSpy® (Altova). This tool offers a graphical interface that ease modelling of the XML schema and also offers schema validation.

4.5.1.2 Development

KARMA is developed using tools like Visual Studio® (Microsoft®), Eclipse and MATLAB® (Mathworks). The development is divided into different parts: simulation framework, simulation component (model) and GUI.

The simulation framework is developed using Visual Studio®. This tool is also used for the development of C++ models. An add-in named Visual Assist® (Visual Tomato) is used to enhance development features. Visual Studio® is configured straightforward and dependencies are located using the *KARMA_ROOT* environment variable according to the structure presented earlier into Subsections 3.3 and 4.2.

Additionally, KARMA relies on a configuration management tool, named SVN. There is only one SVN database, but the *Model Repository* is treated as a distinct database to ease the concept of repository in the current section. KARMA and its *Model Repository* are version-controlled to ease tracking the development process. This approach reduces recovery time when a configuration problem arises (e.g. hard drive broken) or an earlier version of a file (e.g. model executable, model source code, documentation, etc.) is needed. It is recommended to perform releases as often as possible to allow users to use a stable version of KARMA and available models. The release mechanism is also used to create a subset of the *Model Repository*. It is useful when some models are distributed to a KARMA user (e.g. client, analyst, R&D international exchange).

In addition to C++ models, KARMA supports MATLAB® models. MATLAB® is an engineering tool that allows performing computation tasks easily and that is widespread in the scientific community. This tool offers many specialized add-in to reduce development time. One of this add-in is Simulink® that is used for rapid prototyping and for modelling of dynamic systems. Several KARMA models are developed using Simulink®. These models are created graphically and do not require any programming skills. Another add-in, RTW, is used to generate a KARMA model from a Simulink® model. RTW generates stand-alone C code that is wrapped by a KARMA C++ model using scripting capabilities (see Subsection 4.5.2.3 for more details).

Besides the development of the simulation framework and model components, three GUIs were developed and are presented in the Subsection3.4. The first one is KARMA Studio that is developed in Java using Eclipse. The second GUI is SMAT that is developed in C++ using DialogBlocks which allows building GUI graphically using the wxWidgets® library that is presented in the Subsection 4.5.3.1 below. The last one is Matlab to Karma Automatic Configurator that was developed using MATLAB®.

4.5.1.3 3D Models

KARMA simulations are based on 3D models for IR scenes, collision and for 3D visualisation of the engagement. 3D models are mainly purchased from RealDB, but sometimes, minor modifications are required.

PolyTrans® (Okino) is used to convert and optimize 3D models. This tool supports many formats as import and export files. However, KARMA uses mainly OpenFlight® (*.flt*) models so an add-in is used to export OpenFlight® models. It is worthwhile to note that the conversion process might leave out some details about the 3D models.

Another tool is used to edit existing OpenFlight® models. Remo3D® (Remograph) allows for creating/editing models and support the OpenFlight® features almost completely. This tool is the starting point for IR signature modelling along with SMAT (SMAT Developer's Guide.pdf) [10].

4.5.1.4 Documentation

KARMA is documented at different levels in addition to the UML® modelling and the source code. High-level documentation is produced using Word® (Microsoft®) and Visio® (Microsoft®). Sometimes, MindManager® (Mindjet) is also used to structure ideas and concepts graphically. It is a mind mapping tool that can be used for brainstorming and even for conceptual modelling. Doxygen® (OpenSource) is used at the developer's level to capture the class hierarchy of KARMA.

4.5.2 Custom tools

The KARMA success story and its ease of use are explained by well suited tools and automatic code generation. The following aspects are targeted by custom tools developed as a part of KARMA: M&S, engagement simulation analysis and automatic code generation.

4.5.2.1 M&S

At the very beginning of the KARMA project, XML was selected to foster reusability of simulation components and in turn to maximize the benefits of M&S. KARMA Studio was created to help the user and the developer during scenario development as well as for the simulation execution and analysis. However, due to funding difficulties, maintenance and development of KARMA Studio have been neglected in the past years. Soon after the beginning of the KARMA project, it appears to the KARMA team that a tool to ease the automatic code

generation of Simulink® models would be very useful for modellers. The Karma Automatic Configurator GUI was then created using MATLAB.

More recently, in 2007, the implementation IR signatures in KARMA have been revisited to introduce spatial sources. These signatures are modeled using SMAT and Remo3D®. Basically, SMAT is a GUI based on wxWidgets® (GUI), OpenSceneGraph® (3D), OSMesa® (3D), ChartDirector® (graphics) and that uses KARMA's IR scene generation. More details about the use of SMAT were presented in Subsection 3.4.3 while SMAT development is described in the document SMAT Developer's Guide.pdf.

4.5.2.2 Analysis

The analysis of simulation results has a key role into M&S. In addition to KARMA Studio analysis capabilities, 3D playback is offered by a custom tool: 3D Viewer.

This 3D viewing tool was developed in 2007 by the KARMA team. This 3D Viewer exploits the benefits of OpenFlight® models but also offer other file formats. There is a run-time version that executes a KARMA simulation while displaying entities and there is a post-simulation version that is based on the simulation XML log file. An advantage is the possibility to display sensor viewing regions. This tool is based on the Delta3D® library. More details about 3D Viewer are presented in the document Karma 3DViewer.pdf.

4.5.2.3 Code generation

As mentioned earlier, automatic code generation is used in the KARMA development process, starting from UML® diagrams. Rational Rose® scripting capabilities are used to generate XML parameters as well as the Simulink® core model.

The Rose Script® language is used to convert UML® into XML parameters through the use of XML stereotype. Essentially, there are three scripts:

- XML Generation;
- UML® to MATLAB®; and
- MATLAB® to UML®.

These scripts can be launched via a Rational Rose® menu (*Tools* → *KARMA*). For each class having the XML stereotype, the script named XML Generation parses the model hierarchy and generates all attributes having the XML stereotype. See the following document to learn how these scripts are used: ROSE SCRIPT.pdf.

A similar script, UML® to MATLAB®, is used to generate a Simulink® core model: a location on disk is selected, inputs, outputs and parameters are created according to the UML®. Also the initial value of the outputs and the parameters are saved in a MATLAB® file (*.m*). The developer can do reverse engineering from the Simulink® model with the script MATLAB® to UML® to generate the UML® class. It should be noted that the parameter file must be a *.m* file.

The steps to generate a C++ KARMA version of a Simulink® model are presented in the document KARMA development Guidelines.pdf. There is a nice feature for Simulink® models, some code can be inserted at specific locations (between tags). This code will not be overwritten during the code generation process and allows to perform custom actions based on model's inputs/outputs or any other conditions.

A C++ KARMA version of Simulink® models is generated using RTW scripting. The Target Language Compiler (TLC) allows controlling code generation by the use of TLC scripts (*.tlc* files). These scripts are described in Table 7. There is a script file for different stages of the generation process, including one for a model header file and one for a model implementation file. A script has ASCII characters that are generated as is into a selected file and commands can be embedded to perform specific tasks (e.g. generate a code section for each parameters of the model).

Table 7: Description of the TLC scripts.

TLC file name	Description
KARMA_makefile_debug.tlc	This script generates a makefile template, for the debug mode, specific to a Simulink® model (*OBJECT* value only, which contains the KARMA model name without suffix) for code generation using RTW. It allows to retrieve the name of the Simulink® core model (generated model) with a *_CodeGen* suffix and the name of its wrapper for KARMA without this suffix.
KARMA_makefile_release.tlc	Same as *KARMA_makefile_debug.tlc* except for the release mode.
KARMA_malloc.tlc	KARMA C++ model in a DLL in release mode with a wrapper around Simulink® core model generated using RTW.
KARMA_malloc_debug.tlc	Same as *KARMA_malloc.tlc* except for the debug mode.
KARMA_objdoc.tlc	Creation of the documentation file.
KARMA_cppbody.tlc	KARMA model wrapper for Simulink® core model generated using RTW.
KARMA_cpphdr.tlc	KARMA model wrapper for Simulink® core model generated using RTW.

TLC file name	Description
KARMA_formatbody.tlc	This file is responsible for generating the *model.c* file for the various code formats. Currently supported code formats are: grt, grt_malloc, s-function (with accelerator).
KARMA_formathdr.tlc	This file formats header information into *CompiledModel.ModelFiles* fields.
KARMA_formatwide.tlc	This system file is the entry point for RTW RealTimeMalloc code format. The files *model.h*, *model.c*, *model_private.h*, *model_data.c* are produced.

It should be noted that the TLC scripts must be updated for each MATLAB® version. Also, the new dependencies with the other library (*.lib*) must be added to the makefile scripts.

Code generation is launched from a custom interface (*KARMA.m*) that calls different MATLAB® scripts (*.m* files) to execute appropriate tasks. Information is retrieved from the Simulink® model while the script is being executed (e.g. list of inputs). It is important to select the root block of the model to allow correct behaviour. The configuration for code generation is stored in the model's properties, under the description tab. A reserved section is inserted in the description. A Simulink® model is generated in two steps: generation and build. The first step generates the core model (C code) along with the model wrapper (C++ code). A makefile is also generated for the next step. The second step builds the model DLL by invoking the makefile. The DLL is generated into the DLL section of the *Model Repository*. The developer has the possibility to insert custom code between the two steps. The code must be placed in predefined sections of the model header and implementation files. This code is preserved at the beginning of the first step. It is imperative to keep these files into the configuration management repository (SVN) for persistency purpose (e.g. KARMA automatic compilation). Another key element for code generation is the way a Simulink® model is linked to a library. The path to each library must be listed into the MATLAB® paths. Thus, when a model is loaded, its dependencies are searched for inside each library found in the paths. The path to each model is also required for KARMA automatic compilation.

4.5.3 Libraries

The following libraries were used for the development of custom tools and for the development of the KARMA simulation framework.

4.5.3.1 GUI

SMAT relies on a multiple platforms windowing library named wxWidgets® (OpenSource) and a chart components library ChartDirector® (Advanced Software Engineering).

The library wxWidgets® offers a single, easy-to-use Application Programming Interface (API) for writing GUI applications on multiple platforms that still utilize the native platform's controls and

utilities; giving the look and feel appropriate to that platform. This library has many features such as:

- window components (e.g. file browser, button, tree control);
- online help;
- network programming;
- streams;
- clipboard and drag & drop;
- multithreading;
- image reader/writer (variety of popular formats);
- database support; and
- HTML viewing and printing.

Many chart types are available with ChartDirector® and user interactions are implemented through events replication allowing tooltips, zoom and mouse interactions.

The use of these libraries is presented in the document SMAT Developer's Guide.pdf.

4.5.3.2 XML

KARMA is based on XML files and two libraries are used to read/write XML files. The first one, Xerces C++® (OpenSource), is used by the loader and logger modules of KARMA. The other one, Jaxp® (Sun), is used by KARMA Studio. Jaxp® stands for Java API for XML Processing.

4.5.3.3 3D

KARMA uses 3D models into its simulations and for viewing purposes.

Simulations might use the IR scenes generation module (*SceneGenerator3D*) that is based on OpenSceneGraph® (OpenSource) and OSMesa® (OpenSource). The first one manages a scene of 3D models and offers many functionalities such as 3d model loader and attributes edition. The second one is another implementation of OpenGL that is responsible for graphics rendering. This library is used to allow for 16-bit rendering.

Simulations might also use collision detection based on 3D models. The implementation relies on an open source library, OpenDynamicsEngine®, which performs collision detection using 3D geometries. OSG is used to load 3D models that are converted into geometries for ODE.

Finally, the Delta3D® library is used by the 3D Viewer. This library already integrates other libraries, including OSG and ODE. The two versions of the viewer, run-time and post-analysis, as mentioned earlier are implemented using Delta3D®.

4.5.3.4 Atmospheric model

KARMA integrates two atmospheric models: the MODerate resolution atmospheric TRANsmission (MODTRAN) and the Suite for Multi-resolution Atmospheric Radiative Transmission (SMART). The first one is a recongized model in the scientific community, developed by the industry in collaboration with the US Air Force Research Laboratory while the second one was developed by DRDC Valcartier and is based on MODTRAN.

KARMA uses a custom version of the MODTRAN 4 software that has been modified to output transmittances in a correlated-k space. This custom version of MODTRAN is integrated in KARMA by the *ModtranAdapter* and is presented in the document ModtranK in KARMA.pdf.

SMART is integrated in KARMA by the *SmartAdapter* using the SMART Interface (SMARTI) C++ library. This library was developed to simplify the interfacing of DRDC's SMART C++ library into projects. The main objective of SMARTI is the calculation of atmospheric radiative quantities such as transmitted solar irradiance, atmospheric radiative fluxes, path and background radiances, and transmittance. Although the SMART library is much more versatile, that makes it by design quite complicated and time consuming to learn. SMARTI provides an interface to many of the more useful features of SMART, while taking care of the more complicated aspects of working with SMART internally. SMARTI is capable of calculating radiative quantities at moderate resolutions (1, 5 or 15 cm^{-1} wavenumber bins) or for an entire band using the wideband correlated-k theory. The use of the SMARTI library is presented in the document Smarti Library.pdf.

4.6 Adapting KARMA for a SE

KARMA offers its own simulation environment named *SimulationEnvironment* that is presented in details in the Subsection 2.2. However, it is possible to run KARMA models from an external SE. The feasibility was demonstrated in the past by adapting KARMA to STRIVE® (from CAE Inc.), but this SE is no longer used. This section presents how STRIVE® was adapted to give an overview of how any other SE that offers an API for plug-in addition can execute KARMA models.

STRIVE® is an HLA-compliant software package in which simulation entities can interact with each other and the system operator, and in which entities can be distributed among federates. The STRIVE Studio® offers the possibility to users to create their scenarios in a graphical way and to see their simulations in a 2D and a 3D viewers. STRIVE® is also responsible for time management and for collisions between entities.

The KARMA team has created a small tool that allows the conversion of a scenario that has been created inside KARMA Studio to a scenario that STRIVE® recognizes. So, a scenario that has been created inside KARMA Studio can run inside STRIVE®.

The role of STRIVE® in a KARMA simulation is to call the KARMA adapter at the right time during the simulation. Figure 47 shows the relation between KARMA and STRIVE® into a simulation. The adapter takes care of calling the entity inside KARMA. In fact, the adapter is responsible for many tasks during a simulation using STRIVE®. The adapter class is responsible for instantiating all the entities in the simulation, scheduling the entities and calling them at the

right time, calling the services that belong to STRIVE®, checking if new entities have to be created in STRIVE®.

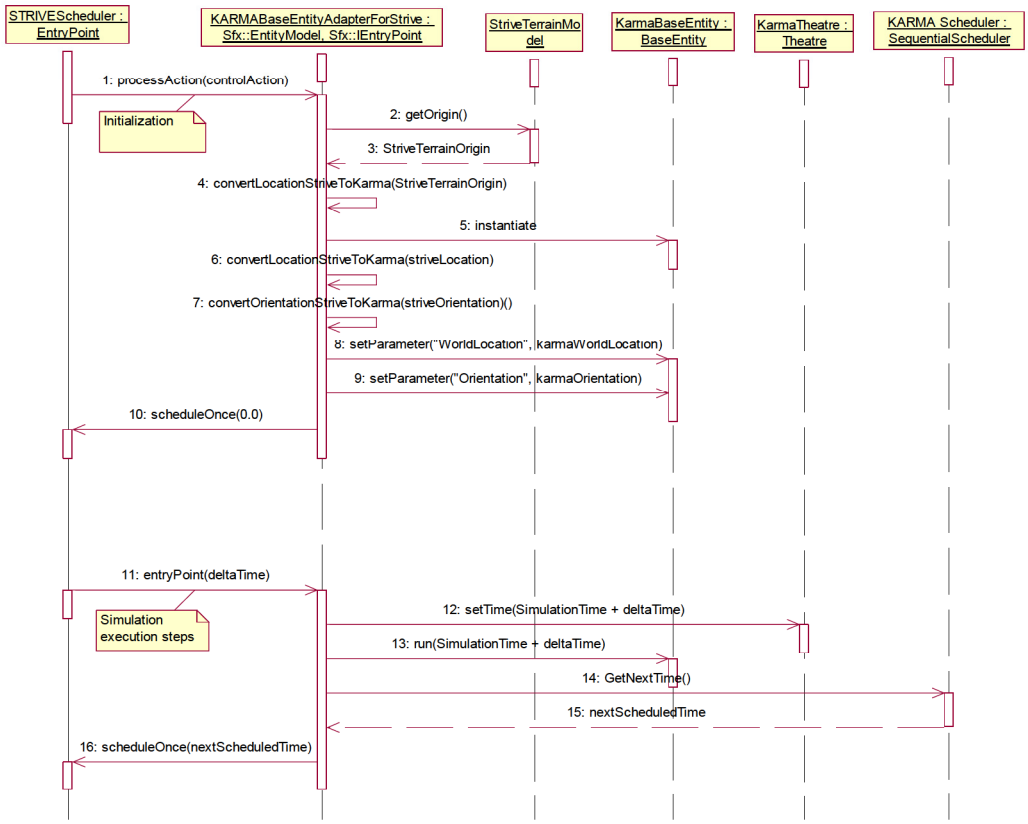

Figure 47: Simulation using STRIVE® UML® sequence diagram.

STRIVE® begins by calling the method *processAction* of the adapter for each entity in the simulation. This method indicates to the adapter which entities exist in the simulation. Inside this method, the adapter instantiates the right base entities. Next, the adapter schedules each base entity at its time step and the STRIVE® scheduler calls the adapter when the time step of a base entity is reached. When the adapter is called by the STRIVE® scheduler because the time step of a base entity is reached, the adapter calls this base entity and schedules the next time that this base entity must be called. The STRIVE® scheduler continues to call the adapter and the adapter continues to call the entities until the end of the simulation.

Refer to the document STRIVE.pdf for more details.

4.7 Adapting KARMA to HWIL

Sometimes, detailed engagement analyzes are required and the LOD of KARMA simulations is increased by using a HWIL component in a scenario. It thus allows the most realistic representation for a component instead of modelling the behaviour of that component.

A HWIL component is used in KARMA using a virtual entity, meaning that the processing is performed outside KARMA. Depending on the nature of the component, a *BaseEntity* or a *Part* model can be used to adapt it to a KARMA simulation. The virtual entities are used almost transparently in KARMA, having inputs, outputs and parameters as every other model. Indeed, these entities are only distinguished from other entities at the level of the *Theatre* class. A *BaseEntity* is gathered into the *m_baseEntityList* or the *m_virtualBaseEntitiesList*. The following subsections present how the SEMAC facility is used as *BaseEntity* and how the MAWS hardware is used as a *Part*.

4.7.1 SEMAC

SEMAC is a HWIL facility that generates an IR scene that is used to test a flare CM against IR-guided missiles. SEMAC is an open-loop system, but it has been demonstrated that with the help of KARMA, which simulates the flight dynamics of the missile and of the aircraft, the loop can be closed to increase the possibilities of SEMAC. SEMAC was developed in a distinct project and the reader should refer to the SEMAC system overview document for further details **Error! Reference source not found.**.

KARMA communicates with SEMAC via a virtual *BaseEntity* named *VirtualSceneGenerator*. The role of this *BaseEntity* is to supply all the information that is needed by the SEMAC controller. This information is:

- position of the aircraft;
- velocity of the aircraft;
- position of the flare;
- velocity of the flare;
- size of the aircraft;
- intensity of the aircraft;
- size of the flare; and
- intensity of the flare.

Thus, this *BaseEntity* is a kind of sensor that looks at the scene and then extracts some information. In fact, the *VirtualSceneGenerator* does not communicate directly with the SEMAC controller. The virtual *BaseEntity* communicates with an intermediate module (*SEMACLink*) that communicates with the SEMAC controller.

Refer to the document SEMACClosedLoop.pdf for more details.

4.7.2 MAWS

The AN/AAR-47 MAWS is a small lightweight passive EO threat warning device used to detect surface-to-air missiles fired at helicopters and low-flying fixed wing aircraft. It provides audio and visual-sector warning messages to the aircrew and automatically allocates CMs. This project consists of using the real hardware into the KARMA simulation environment but without its sensors. The real sensors are replaced by KARMA numerical sensors that provide the detected irradiance to the hardware. Like SEMAC, the project was developed in a distinct project and the reader should refer to the MAWS system overview document for further details **Error! Reference source not found.**.

Figure 48 shows the HWIL MAWS UML® sequence diagram. When the HWIL MAWS is used into a KARMA simulation, an instance of the class *MAWSAAR_47HWIL*, which is a virtual *Part*, is created to communicate with the hardware. During the simulation, the *MAWSAAR_47HWIL* sends the irradiance received from each simulated sensor to the HWIL MAWS. When the HWIL MAWS declares a threat, the information is sent to the *MAWSAAR_47HWIL* which calls the CMDS to trigger countermeasure sequence.

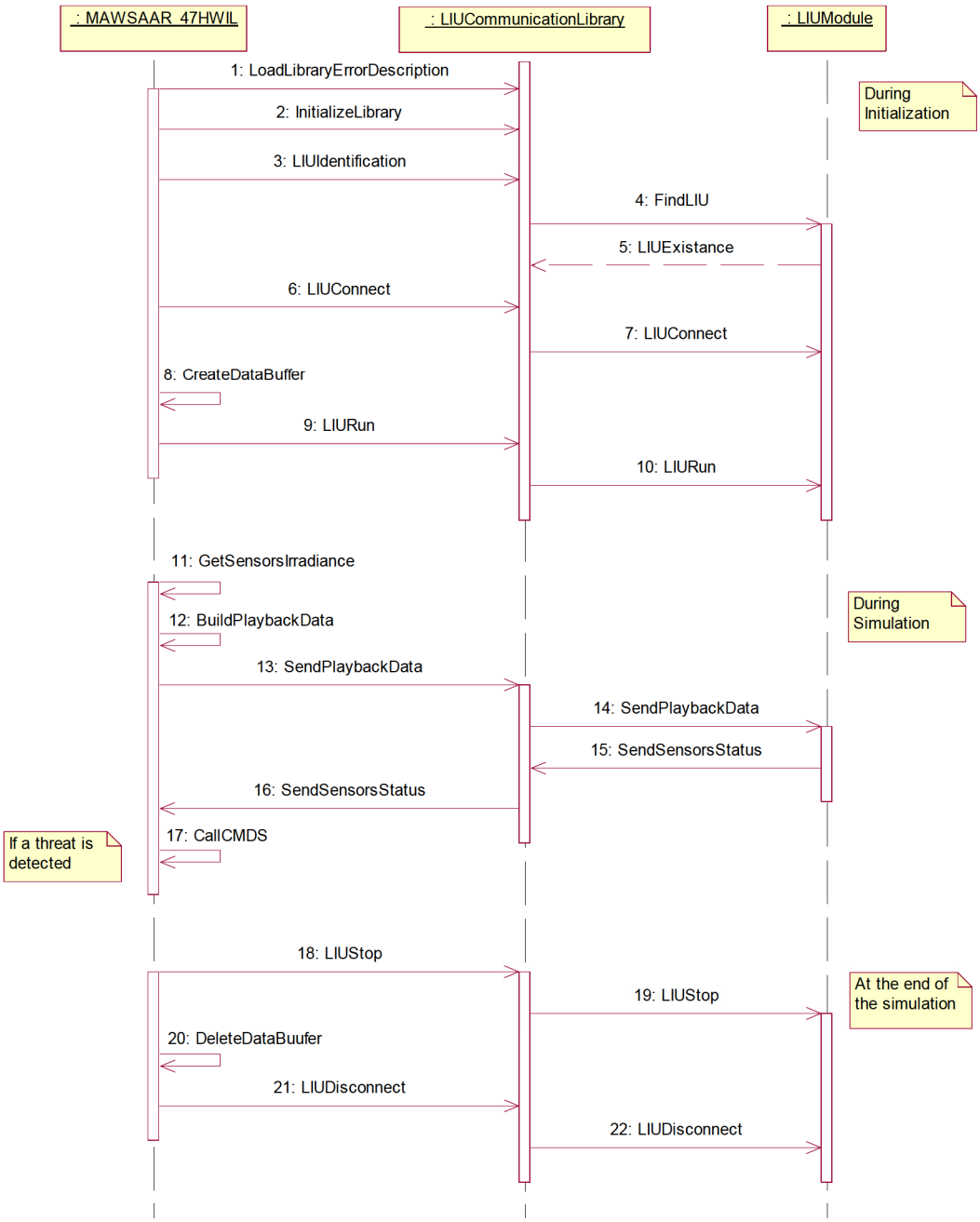

Figure 48: Sequence diagram of the HWIL MAWS.

5 Conclusion

The KARMA system overview summarized the KARMA facility and its capabilities with references to existing technical documentation. This document, along with the VRAPP simulation framework technical specifications V0, will be the starting point for the design of the VRAPP simulation framework V0. The simulation framework technical specifications will evolve jointly with the other VRAPP technical specifications throughout the spiral development phases to produce the technical specifications V1 and V2. The KARMA system will be updated and integrated into the VRAPP system of systems that will be documented in a final VRAPP system overview.

References

[1] Gilbert, B., Harrison, N., Lauzon, M., Pigeon, Maj. R., Morin, A., "Project KARMA – Definition and requirements", DREV TM 2001-271, December 2001.

[2] Harrison, N., Boily, P., Belhumeur, C., Alain, J., "VRAPP Technical Data Package", DVD, DRDC Valcartier SL 2011-269, PROTECTED A, August 2011.

[3] Harrison, N., Gilbert, B., Jeffrey, A., Lestage, R., Lauzon, M., Morin, A., "KARMA: Materializing the Soul of Technologies into Models", Paper 2256, Proceedings of the I/ITSEC 2005 Interservice/Industry Training, Simulation and Education Conference, Orlando, Florida, USA, 28 November-1 December 2005.

[4] Harrison, N., Gilbert, B., Lauzon, M., Jeffrey, A., Lalancette, C., Lestage, R., Morin, A., "A M&S Process to Achieve Reusability and Interoperability", Proceedings of the NATO M&S Conference 2002, RTO-MP-094-11, Paris, France, October 24-25 2002.

[5] Harrison, N., Gilbert, B., Jeffrey, A., Lauzon, M., Lestage, R., "Adaptive and Modular M&S Configuration for Increased Reusability", Paper 1864, Proceedings of the I/ITSEC 2004 Interservice/Industry Training, Simulation and Education Conference, Orlando, Florida, USA, 6-9 December 2004.

[6] Fletcher, D., Anderson, R., Lauzon, M., Harrison, N., Halsall, G., Everatt, E., Vanden-Heuval, M., "MIST Interface Specification Version 1.0", TR-WPN-7/1-2006, TTCP WPN TP-7 KTA 7-11 final report, May 2006.

[7] Lepage, J.-F., Rouleau, E., Richard, J., Harrison N., "Infrared scene generation for countermeasures simulations: Implementation in the KARMA framework, phase 1", DRDC Valcartier TR 2010-284, PROTECTED A, 2013.

[8] Rouleau, E., "Infrared Scene Generation (IRSG) Developer's Guide", LTI Report No. LTI2008DES-3, DRDC Valcartier CR 2008-258, September 2008.

[9] Richard, J., "Signature Modeling and Analysis Tool (SMAT) User's Guide", LTI Report No. LTI2008DES-1, DRDC Valcartier CR 2008-260, September 2008.

[10] Richard, J., "Signature Modeling and Analysis Tool (SMAT) Developer's Guide (U)", LTI Report No. LTI2008DES-2, DRDC Valcartier CR 2008-259, September 2008.

[11] Harrison, N., Belhumeur, C., Gilbert, B., Lemelin, C, Bérubé, N., Fraser, D., "Support to CASE Griffon Mothership – ManPADS and MAWS Federate", DRDC Valcartier ECR 2006-424, July 2008.

Annex A Aircraft XML composition file example

```
<fileType>BaseEntity</fileType>
<composite name="Structure aircraft">
	<component>Structure</component>
	<parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\Characteristic\Struc
tureAircraftWaypoint.xml</parameters>
	<composition>none</composition>
	<priority>0</priority>
	<documentation> </documentation>
</composite>
<composite name="Manoeuvre">
	<component>Manoeuvre</component>
	<parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\Part\Manoeuvre.xml</
parameters>
	<composition>$(KARMA_ROOT)\ModelRepository\xml\Compositions\ManoeuvreComposit
ion.xml</composition>
	<priority>0</priority>
	<documentation> </documentation>
</composite>
<composite name="Signature">
	<component>Signature</component>
	<parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\Characteristic\Signa
ture.xml</parameters>
	<composition>$(KARMA_ROOT)\ModelRepository\xml\Compositions\SignatureAircraft
.xml</composition>
	<priority>0</priority>
	<documentation> </documentation>
</composite>
<composite name="KARMA missile">
	<component>Munition</component>
	<parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\BaseEntity\Missile.x
ml</parameters>
	<composition>$(KARMA_ROOT)\ModelRepository\xml\Compositions\MissileCompositio
n.xml</composition>
	<quantity>1</quantity>
	<documentation> </documentation>
</composite>
<composite name="Entity dispensor">
	<component>EntityDispensor</component>
	<parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\Part\EntityDispensor
.xml</parameters>
	<composition>none</composition>
	<priority>0</priority>
	<documentation> </documentation>
</composite>
<composite name="Threat system">
	<component>EntityDispensor</component>
	<parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\Part\EntityDispensor
Missile.xml</parameters>
	<composition>none</composition>
	<priority>0</priority>
	<documentation>  </documentation>
</composite>
<composite name="KARMA flare">
	<component>Flare</component>
	<parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\BaseEntity\Flare.xml
```

```
</parameters>
	<composition>$(KARMA_ROOT)\ModelRepository\xml\Compositions\FlareComposition.
xml</composition>
	<quantity>2</quantity>
	<documentation> </documentation>
</composite>
<composite name="Expendable dispensor">
	<component>EntityDispensor</component>
	<parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\Part\EntityDispensor
Flare.xml</parameters>
	<composition>none</composition>
	<priority>0</priority>
	<documentation> </documentation>
</composite>
```

List of acronyms

2D	Two Dimensional
3D	Three Dimensional
API	Application Programming Interface
CF	Canadian Forces
CM	Countermeasure
CMDS	Countermeasure Dispenser System
COI	Communities of Interest
COTS	Commercial-Off-The-Shelf
CSV	Comma Separated Values
DIRCM	Directed IR Countermeasure
DLL	Dynamic Link Library
DND	Department of National Defence
DOF	Degree of Freedom
DRDC	Defence Research & Development Canada
DRDKIM	Director Research and Development Knowledge and Information Management
EO	Electro-Optical
EOW	Electro-Optical Warfare
FEDEP	Federation Development and Execution Process
FOM	Federation Object Model
FOV	Field Of View
GUI	Graphical User Interface
HLA	High-Level Architecture
HTML	HyperText Mark-up Language
HWIL	HardWare-In-the-Loop
ID	Identification
IR	Infrared
LOD	Level Of Detail
LOS	Line Of Sight
M&S	Modelling and Simulation

MANPADS	Man Portable Air Defence System
MAWS	Missile Approach Warning System
MDA	Model-Driven Architecture
MIST	Munition Interface Specification for the TTCP
MODTRAN	MODerate resolution atmospheric TRANsmission
NED	North East Down
ODE	OpenDynamicsEngine
OO	Object-Oriented
OSG	OpenSceneGraph
R&D	Research & Development
RF	Radio-Frequency
RPR FOM	Real-time Platform Reference Federation Object Model
RTI	Run Time Infrastructure
RTW	Real-Time Workshop
SE	Synthetic Environment
SEMAC	Simulator of Engagement for Missiles, Aircraft and Countermeasures
SMART	Suite for Multi-resolution Atmospheric Radiative Transmission
SMARTI	SMART Interface
SMAT	Signature Modelling and Analysis Tool
SOR	Statement Of Requirements
SoS	System of Systems
STL	Standard Template Library
SVN	Subversion
TDP	Technology Demonstration Project
TLC	Target Language Compiler
TXT	Text
UDP	User Datagram Protocol
UML	Unified Modeling Language
UV	Ultraviolet
V&V	Verification and Validation
VRAPP	Virtual Range for Advanced Platform Protection
XML	eXtensible Markup Language

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence R&D Canada – Valcartier
2459 de la Bravoure Road
Quebec (Quebec)
G3J 1X5 Canada

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED
(NON-CONTROLLED GOODS)
DMC A
REVIEW: GCEC April 2011

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

VRAPP system overview: KARMA (U)

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

Harrison, N.; Belhumeur, C.; Lambert, M.; Lepage, J.-F.; Rouleau , E.; Boivin, E.; Labrie, M.-A.

5. DATE OF PUBLICATION (Month and year of publication of document.)

October 2013

6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)

100

6b. NO. OF REFS (Total cited in document.)

11

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Valcartier
2459 de la Bravoure Road
Quebec (Quebec)
G3J 1X5 Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)

03sb

9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)

DRDC Valcartier TM 2010-208

10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

Electro-optical (EO) guided weapons represent a significant and rapidly proliferating threat to Canadian Forces (CF). Therefore, developing and maintaining appropriate countermeasures is critical to improve platform survivability. The CF have a need for a comprehensive and thorough approach to provide the best reliable answer to EO warfare problems, timely and adapted to the needs. In this perspective, the Virtual Range for Advanced Platform Protection (VRAPP) is a system of systems providing technology for an EO virtual proving ground for the purpose of achieving a more robust, adaptable and agile force protection and, ultimately, to improve combat effectiveness in high threat environments. The project documentation includes guidelines, system overviews, a technical data package, a statement of requirements and technical specifications. This report presents the system overview of the KARMA simulation framework. The system overview summarizes the KARMA facility and its capabilities with references to existing technical documentation. The KARMA system overview will be the starting point for the design of the VRAPP simulation framework. The KARMA system will be updated and integrated throughout the spiral development phases and will be documented in a final VRAPP system overview.

Les armes guidées par électro-optique (EO) représentent une menace appréciable et qui prolifère rapidement pour les Forces canadiennes (FC). Par conséquent, il est critique de développer et d'entretenir des contre-mesures appropriées pour améliorer la survie des plates-formes. Les FC ont besoin d'une approche globale et approfondie pour fournir la meilleure réponse aux questions de guerre EO, dans un délai opportun et d'une façon adaptée aux besoins. Dans cette optique, le polygone virtuel pour la protection évoluée de plates-formes, ou VRAPP, est un système de systèmes offrant la technologie requise pour développer un polygone virtuel d'essais EO visant à assurer une protection de la force plus fiable, adaptable et souple et, ultimement, à améliorer l'efficacité au combat dans des environnements où le degré de menace est élevé. La documentation de projet comprend des lignes directrices, des vues d'ensemble de systèmes, un jeu de documents techniques, un énoncé des besoins et des spécifications techniques. Ce rapport présente la vue d'ensemble du cadriciel de simulation KARMA. La vue d'ensemble du système résume l'installation KARMA et ses capacités tout en faisant référence à la documentation technique existante. La vue d'ensemble du système KARMA servira de point de départ à la conception du cadriciel de simulation de VRAPP. Le système KARMA sera mis à jour et intégré tout au long du développement en spiral et sera documenté dans un document final de vue d'ensemble du système VRAPP.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Simulation framework; modelling process; scenario composition; guided-weapon engagement

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca