D4.4 PRELIMINARY VERSION OF CPSOS SIMULATION TOOLS AND TRAINING DATA GENERATION

Authors IBM, UoP, I2Cat, ROBOTEC, CRF

Work Package WP4 CPSoSaware System Layer Design and adaptation of dependable CP(H)SoS

Abstract

This report constitutes the output of task T4.4 “CPSoS Simulation Tools and Integration” and describes a preliminary version of CPSoSaware simulation and training block (SAT).

Funded by the Horizon 2020 Framework Programme
of the European Union

Deliverable Information

Work Package	WP4 CPSoSaware System Layer Design and adaptation of dependable CP(H)SoS
Task	T4.4 CPSoS Simulation Tools and Integration
Deliverable title	D4.4 Preliminary Version of CPSoS Simulation Tools and Training Data Generation
Dissemination Level	PU
Status	Final
Version Number	1.00
Due date	M14

Project Information

Project start and duration	01/01/2020 – 31/12/2022, 36 months
Project Coordinator	Industrial Systems Institute, ATHENA Research and Innovation Center 26504, Rio-Patras, Greece
Partners	1. ATHINA-EREVNITIKO KENTRO KAINOTOMIAS STIS TECHNOLOGIES TIS PLIROFORIAS, TON EPIKOINONION KAI TIS GNOSIS (ISI) the Coordinator 2. FUNDACIO PRIVADA I2CAT, INTERNET I INNOVACIO DIGITAL A CATALUNYA (I2CAT), 3. IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD (IBM ISRAEL 4. ATOS SPAIN SA (ATOS), 5. PANASONIC AUTOMOTIVE SYSTEMS EUROPE GMBH (PASEU) 6. EIGHT BELLS LTD (8BELLS) 7. UNIVERSITA DELLA SVIZZERA ITALIANA (USI), 8. TAMPEREEN KORKEAKOULUSAATIO SR (TAU) 9. UNIVERSITY OF PELOPONNESE (UoP) 10. CATALINK LIMITED (CATALINK) 11. ROBOTEC.AI SPOLKA Z OGRANICZONA ODPOWIEDZIALNOSCIA (RTC) 12. CENTRO RICERCHE FIAT SCPA (CRF) 13. PANEPISTIMIO PATRON (UPAT)
Website	www.cpsosaware.eu

Control Sheet

VERSION	DATE	SUMMARY OF CHANGES	AUTHOR
0.1		*Initial draft*	Evgeny Shindin
0.2		All chapter with simulator tools added	*All involved Author Partners*
0.3		Orchestrator and Data aggregator input provided	*IBM*
0.9		Final version	*IBM*
1.0		Reviewed version that addresses internal review comments	*IBM*

	NAME
Prepared by	IBM
Reviewed by	UoP, USI, ISI
Authorised by	ISI

DATE	RECIPIENT
30 March 2021	Project Consortium
8 April 2021	European Commission

Table Of Contents

1. Executive Summary 6
1.1 Structure of Document 6
1.2 Related Documents and Tasks 6
1.3 Definitions and Acronyms 7
2 Introduction and state of the art (IBM) 9
2.1 Co-simulation methods and tools 9
2.1.1 Methods 9
2.1.2 Functional Mock-up Interface (FMI) 9
2.1.3 Tools 10
2.1.4 Summary 10
2.2 Storage DB types 10
2.2.1 Relational databases 11
2.2.2 Key-value stores 11
2.2.3 Document databases 12
2.2.4 Graph databases 13
2.2.5 Object databases 13
2.2.6 Summary of Storage DB types 14
3 Simulation and Training Block Architecture 15
3.1 Architecture overview 15
3.2 Integration and Storage approach 16
3.3 Simulation workflow 18
4 Simulation and Training Block Components 20
4.1 Orchestration tool 20
4.2 Data storage and transformation services 24

4.3 Inter and Intra Communication Simulators 25
4.3.1 Intra – Communication Simulator 25
4.3.2 Inter – Communication Simulator 29
4.4 CPSoS HW-SW Simulators 31
4.5 CPSoS Use-case dedicated simulators 32
4.5.1 Simulator for ADAS/AV systems research 32
5 Simulation and Training Block Interfaces 39
5.1 Orchestrator interfaces 39
5.2 Data storage and transformation services interfaces 41
5.2.1 Low-Level Instance Base Interface 41
5.2.2 Class definition interface 41
5.2.3 Data definition interface 43
5.2.4 Data management interface 45
5.2.5 Ontology Alignment and Equivalence Rules 46
6 Conclusion 49

List of figures

Figure 1: SAT block architecture. 15

Figure 2: Two-Layered model....................17

Figure 3: Example of model in the intermediate format....................17

Figure 4: CPSoSaware CI/CD workflow....................21

Figure 5: Architecture of data storage and transformation service 24

Figure 6: NS-3 Basic Simulation Model....................26

Figure 7: NS-3 in CPSoSaware 29

Figure 8 SUMO Highway Scenario 29

Figure 9: OMNET++ propagation example....................30

Figure 10: Tool interface 30

Figure 11: Proxim's utilization visualization of a minimal co-processor architecture. The redness indicates the level of utilization (so far) by the program. 31

Figure 12: Basic structure of CARLA simulator for AV/ADAS research 32

Figure 13: GNSS attributes in CARLA simulator as specified in the GNSS sensor documentation. 35

Figure 14: LIDAR attributes in CARLA simulator as specified in the LIDAR sensor documentation. 36

Figure 15: Camera attributes in CARLA simulator as specified in the camera sensor documentation. Please note that camera has also camera lens distortions options as separate attributes group. 36

Figure 16: Inertial Measurement Unit (IMU) attributes in CARLA simulator as specified in the IMU sensor documentation....................37

Figure 17: CI/CD Orchestrator Control API....................39

Figure 18 NS3 Simulation Interface 40

Figure 19: API for namespace management....................41

Figure 20: API for class definitions management....................43

Figure 21: Schema management API....................45

Figure 22: Data management API....................46

Figure 23: Rules management API....................48

1. Executive Summary

This report constitutes the first output of task T4.4 "CPSoS Simulation Tools and integration" and describes the proposed architecture being considered by CPSoSaware for the simulation and training block. The SAT block functionality consists of two main functions: i) performing joint simulation, and ii) data storage for storing simulation data. We reviewed co-simulation as a method for performing joint simulation but come to the conclusion that this methodology is too complex for our needs and propose to use data transformation to the CERBERO Interoperability Framework format (CIF) instead. We compared and contrasted several storage DB types and suggest that a relational database would be the best choice for the simulation data. We presented the design of the architecture of the simulation and training (SAT) block which is driven by requirements to its functionality. We described this architecture and how it relates to an Integration and Storage approach for integrating different and diverse simulators that use different modelling paradigms and languages, and how it supports the Simulation Workflow. We then described each of the SAT and then described the SAT interfaces including the orchestrator and data storage interfaces.

The proposed architecture allows the project to achieve objective O4.2 "Implement CPSoSaware Simulation and Training block that constitutes the basic testing and training data extraction environment for the design and redesign procedures performed in the MRE System Layer component."

1.1 Structure of Document

This document is structured into six major sections:

- **Section 1** introduces the document, outlines its structure, and identifies terms and acronyms used across the document.
- **Section 2** provides the general introduction to the CPSoSaware simulation and training (SAT) block functionality and discuss state of the art co-simulation and data storage approaches that will be considered during design of SAT block.
- **Section** Error! Reference source not found.**3** describes SAT block high-level architecture, data integration and storage approach considered during SAT block design and generic simulation workflow that utilizes different architecture elements.
- **Section 4** discusses main SAT block components including orchestrator, data storage and transformation services and different simulators that used in the project.
- **Section 5** describes SAT block interfaces.
- **Section 6** concludes the document.

1.2 Related Documents and Tasks

This document is the first output of Task 4.4 "CPSoS Simulation Tools and Integration", that is scheduled to be performed during M6-M28 of the project. The final deliverable of Task 4.4 is D4.9 "Final Version of CPSoS Simulation Tools and Training Data Generation" will continue the current document. The choice of tools and methodologies for simulation and integration is partially based on the output of Task 1.1 "SoA analysis, technological selection and benchmarking of best practices" that is described in D1.1 "Supportive, Motivating and Persuasive Approaches, Tools and Metrics". Intra- and inter-communication simulation models developed as part of Task 2.2 "CPS Inter and Intra Communication Models" will be used by corresponding simulators discussed in section 4.3.1, 4.3.2 that in connection with use-case specific simulators discussed in section 4.5 will generate data required for Task 4.2 "CPSoSaware Networking for reliable communication and cooperation between CP(H)SoS". Simulation models of HW and SW components provided as output of Task 2.3 “CPS Models for HW & SW Components” will be used by HW and SW simulators discussed in section 4.4 in order to generate data required to perform HW-SW partitioning optimization in the Task 4.4. Moreover, properties of HW and SW components estimated during these simulations will be included to the models of these components and to the library of the HW & SW components will be provided as output of Task 3.6 “Development of HW-SW Library with reliable Components” in the deliverable D3.6 “Library of SW and HW components”. Furthermore, data generated by different simulators will be re-used in other tasks of WP3 “Model based CP(H)S Layer Design and Development supporting Distributed Assisted, Augmented and Autonomous Intelligence” in order to perform training and testing of different AI algorithms that will be developed as part of this WP.

1.3 Definitions and Acronyms

The following list includes the most relevant acronyms and recurring definitions used in the document:

Acronym / Term	Definition
ACID	Atomic, Consistent, Isolated, Durable
ADAS	Advanced driver-assistance systems
AV	Autonomous Vehicle
CAM	Cooperative Awareness Messaging
CERBERO	Cross-layer modEl-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid envirOnments. Horizon 2020 EU RIA project, grant agreement No. 732105
CIF	CERBERO Interoperability Framework (CIF)
CPHSoS	Cyber-Physical-Human System of Systems
CPS	Cyber-Physical System
CPSoS	Cyber-Physical System of Systems
CS	Co-Simulation
DB	Database
DBMS	Database Management System
ETSI	European Telecommunications Standards Institute
FMI	Functional Mock-Up Interface
FMU	Functional Mock-Up Unit
FMI	Functional Mock-Up Interface
FMU	Functional Mock-Up Unit
FPGA	Field Programmable Gate Array
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GPU	Graphics Processing Unit
IMU	Inertial Measurement Units
JSON	Javascript Object Notation
LiDAR	Laser Imaging, Detection and Ranging
ME	Model Exchange
OEM	Original Equipment Manufacturer

RADAR	Radio Detection and Ranging
RAM	Read-only Memory
RDBMS	Relational Database Management System
RGB	Red, Green, Blue
ROS	Robot Operating System
SAT	Simulation and Training
SQL	Structured Query Language
TCP	Transmission Control Protocol
V2X	Vehicle to *X* where *X* is one of several possible conversation partners
XML	eXtended Markup Language
YAML	Yet another Markup Language

2 Introduction and state of the art (IBM)

Driven by the project requirements one can identify two key functions that should be performed by SAT block: SAT block should be capable to perform joint simulation across different and diverse simulators and store simulation data in a way that will allow querying the data and obtaining a consistent dataset, that will be used in order to train ML algorithms. Thus, design of SAT block architecture starts from reviewing of the state-of-the-art approaches that are used to perform joint simulation and storage DB types that can be utilized to store simulation data.

2.1 Co-simulation methods and tools

It is challenging to develop CPSoSs, which are hybrid systems made up of loosely-coupled subsystems from different domains that operate together with a common purpose. The co-simulation methodology is to model each of the subsystems in a separate simulator. The main idea is that each subsystem has its own set of tools for which are specialized for the subsystem's domain. These may include programming languages, user interfaces, workflows, and more, which are well-established for modeling these subsystems. The modeling can then be done for each subsystem on its own, without concern for the coupled problem.

The simulators are then coupled together into a joint simulation, where each subsystem is run as a black box. The co-simulation is responsible for starting, stopping, and coordinating all of the simulators, as well as providing a mechanism for them to communicate and read each other's state.

2.1.1 Methods

Each subsystem can be represented digitally as a model. The model represents a dynamical system which relates to a set of physical laws, and a control system of some kind. There are domain specific tools to develop the models for each subsystem. The models can be executed and can communicate using standard interfaces. The standard supported by the most tools, is the Functional Mock-up Interface, described in the next section.

2.1.2 Functional Mock-up Interface (FMI)

The most common standard interface for computer simulations is known as Functional Mock-up Interface (FMI) (https://fmi-standard.org/). FMI is an open standard whose goal is to support the exchange of models (Model Exchange or ME) as well as the exchange of model data (Co-Simulation or CS) using a standard format. MEs and CSs are the two types of Functional Mock-up Units (FMU).

ME units represent the dynamic systems as sets of differential equations. These are imported into a tool in one batch, and connects the FMU to a numerical solver, which sets and computes the internal state and step size. Figure 1 shows the output flow of FMUs.

On the other hand, CS units each have their own solver. When they are imported, the tool sends requests to the FMU to step forward a given time, and then reads the output.

The tools discussed in section 3.1.3 below all support FMI, either version 1.0 or version 2.0.

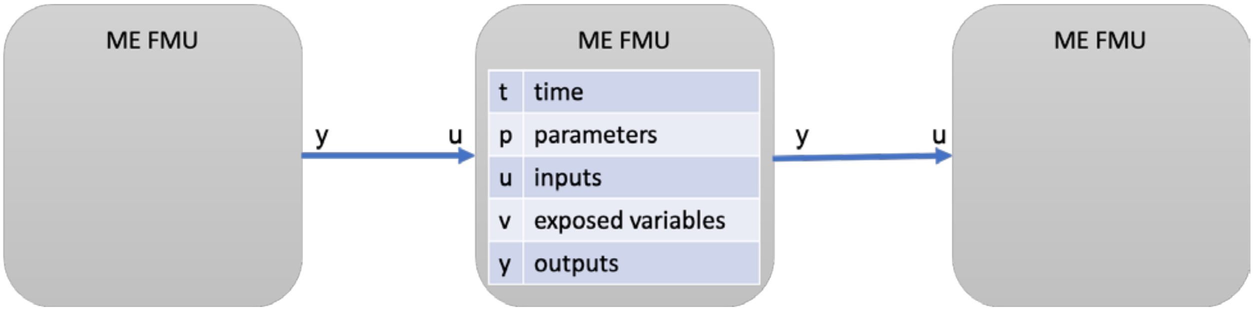


Figure 1 FMUs exchanging data. The output (y) of one simulator becomes the input (u) of another.

2.1.3 Tools

The FMI standards organization provides a list of tools that support the FMI standard for import and export of FMUs. This includes both version 1.0 and version 2.0 of the standard, as well as ME and CS FMUs. The list can be found here: https://fmi-standard.org/tools/.

2.1.3.1 IBM Engineering Systems Design Rational Rhapsody

IBM Engineering Systems Design Rational Rhapsody (https://www.ibm.com/products/systems-design-rhapsody) is a family of products for modelling and systems design that supports FMI version 1.0. It is a commercial system that includes collaborative design and testing using several modelling languages such as SysML.

2.1.3.2 INTO-CPS

The Integrated Toolchain for Cyber-Physical Systems (INTO-CPS) (https://into-cps-association.readthedocs.io/en/latest/tools.html) is an open-source collection of tools developed to aid in the development of CPSs. Included are desktop and cloud applications for configuring and orchestrating co-simulation scenarios, Modelio, "a combined UML/BPMN modeler supporting a wide range of models and diagrams." Maestro (https://github.com/INTO-CPS-Association/maestro), the co-simulation framework of INTO-CPS supports FMUs conforming to both version 1.0 and version 2.0 of the FMI standard.

2.1.4 Summary

The main benefit of co-simulation is the support of common, standard interfaces for simulating models of dynamical systems. The most represented standard for co-simulation is FMI. Unfortunately, several of the modelling tools used by the CPSoSaware project do not support this standard. Given the heterogeneity and diversity of the simulators, building additional components in order to support FMI would be cumbersome and not efficient. Therefore, the use of co-simulation methods would be too complex and inefficient as the simulators are too different and work in different scales. Thus, the use of a data transformation approach instead of co-simulation approach appears to be the best to adopt. To do this we utilize the concept of CERBERO Interoperability Framework (CIF) that allows to connect different tools using semantic data transformation.

2.2 Storage DB types

Data can be stored persistently in a database management system (DBMS). These systems allow for the creation, retrieval, update and deletion of data by different computers in the system. There are many types of DBMSs, with different advantages and disadvantages. This rest of this section is divided into subsections, of which the first five each describe a different class of DBMS, its strengths and weaknesses, with one or two examples. The final subsection summarizes the state of the art.

2.2.1 Relational databases

Relational database management systems (RDBMS) represent data in the form of tables consisting of rows and columns, which is called the relational model. Each table represents a collection of similar entities, with each row representing a single entity and each column representing an attribute of those entities. Each row in each table represents a data record and is represented by a unique key. Data from the tables can be combined, filtered, joined and otherwise manipulated using relational operators. This rigid structure is suitable for structured data that can be described by a separate database schema.

The strengths of RDBMSs include:

1. Support for complex queries using SQL which is standard across vendors and versions,
2. Support for high performance indexed queries, where the index can be on any column,
3. Support for normal forms for consistency,
4. Support for ACID transactions.

The weaknesses of RDBMSs include:

1. Heavy resource consumption,
2. The relational model can be restricting,
3. Inserts and updates can be slow,
4. Not all data fits the relational model.

While RDBMSs have been around since the 1970’s, they are still widely used and actively developed. There are many popular implementations of RDBMSs which are constantly seeing improved performance and new features. Some popular implementations:

1. Oracle,
2. IBM DB2,
3. MySQL,
4. Microsoft SQL Server.

2.2.2 Key-value stores

Key-value stores represent data as associative arrays. Each record consists of a key and a value. The key is usually an integer or something that can be easily mapped to an integer using a hashing function, while the value can be any type. In fact, the types of the values can be simple such as strings, compound collections, and need not be the same from record to record. Often the database is not divided into tables, although it can be. Associative arrays are also known as dictionaries, maps, or hashtables, and are suitable for unstructured data, with no set schema.

The strengths of key-value stores are:

1. Database records need not conform to strict schema,
2. Rapid retrieval of single records,

3. Very memory efficient.

The weaknesses of key-value stores are:

1. Poor support for complex queries,
2. Poor support for transactions,
3. Some implementations store in RAM only,
4. No standard query method.

The recent trend of NoSQL databases has caused a upsurge in usage as well as number of implementations of key-value stores. In particular, large, distributed cloud and edge applications often find a use for one or more key-value store implementations. Some of the more popular implementations include:

1. Redis,
2. Memcached,
3. Berkeley DB,
4. CouchDB.

2.2.3 Document databases

Document databases represent data in the form of documents. The documents stored in document databases are generally semi-structured. This means that they have a schema, but it is part of the document, in such forms as XML tags or JSON keys. This allows for more complicated queries than key-value stores, while allowing for some information to be represented that is not constrained by a predetermined schema. These documents are often in specific formats that support including schema information such as XML, JSON, or proprietary formats. Document databases have recently become popular as part of the NoSQL trend.

The strengths of document databases include:

1. Documents are not separated into tables, which fits better with object-oriented programming where one object may be stored in several tables,
2. Documents do not have to fit a standard schema, which allows for additional information to be managed without upfront planning and design,
3. Documents can be stored in a format that is easily generated and parsed such as XML, YAML, or JSON.

The weaknesses of document databases include:

1. Does not have as high performance as relational databases for highly structured data,
2. Has higher memory requirements than key-value stores.

Document databases have become much more popular as cloud and mobile computing have introduced a much more heterogeneous deployment architecture to many applications. Furthermore, they often take advantage of newer technologies, data formats, and other modern features that are difficult to implement in key-value stores and RDBMSs. Some popular implementations of document databases include:

1. MongoDB,
2. Elasticsearch,

3. CouchDB,
4. Cloudant.

2.2.4 Graph databases

Graph databases treat the relationship between different data to be of primary interest. Each datum is represented as a node in a graph structure, with arcs representing a relationship between nodes. The underlying storage mechanism is usually one of a key-value store, a document database, or even an RDBMS.

There are many types of data that can represented as graphs. For example, social network data, which connects individuals who have a relationship of interest, or consumption graphs, which connect consumers with products and payments.

The strengths of graph databases include:

1. Relations between data points are explicit, in the form of arcs,
2. Support for graph queries and analysis such as whether two nodes are connected or the shortest path,
3. Very good performance and scalability,
4. Some support for standardized query language such as GraphQL, Gremlin or SPARQL.

The weaknesses of graph databases include:

1. Not useful for data that is not network-like,
2. No single standard for query language,
3. Not good for bulk operations on many data points with one query.

For some specific applications noted above, graph databases are quite useful. Some popular graph databases include:

1. SAP Hana,
2. Oracle Property Graph,
3. Amazon Neptune,
4. Neo4j.

2.2.5 Object databases

Object databases are designed to support persistence of for applications built using object-oriented development. The main motivation is what is known as the object-relational impedance mismatch, where the application programming environment represents data as objects in source code, which is difficult to manage using the relational model. Object databases emerged in the 1980’s as this problem became apparent.

The strengths of object database include:

1. Objects can often be retrieved without explicit queries, by following pointers from other objects,
2. Object storage can be more efficient since the data model of the application matches the database. This is especially so for complex object structures,
3. They often support ACID transactions.

The weaknesses of object database include:

1. There is no standard query language,
2. There is no support for complex queries.

Recently, several open source object databases have emerged, renewing the popularity of this paradigm. Some popular object databases include:

1. Intersystems Caché
2. Actian
3. Db4o
4. ObjectStore.

2.2.6 Summary of Storage DB types

Each of the storage types discussed above has its advantages and disadvantages. Depending on the nature of the storage requirements and the applications using it, a different type may prove to be the most applicable. RDBMSs are suitable for most general-purpose data storage with heterogeneous applications and many complex queries, especially when the data is highly structured. Key-value stores are a low overhead and low memory usage option for simple store and retrieve applications. Document storage databases are a more flexible solution than RDBMSs when the data structure is less rigid, although they are not as strong on complex queries. Graph databases work especially well when the relations between data points are of primary interest. Object databases provide high performance with minimal code for object-oriented development. Therefore, the relational database type appears to be the best choice.

3 Simulation and Training Block Architecture

3.1 Architecture overview

The design of the architecture of the simulation and training (SAT) block is driven by requirements to its functionality. In particular the SAT block should:

- Provide integration and orchestration of different and diverse simulators in an extendable manner. That is, the architecture concept should allow connecting new simulators to the SAT in a plug-and-play manner, without any changes to the SAT itself.
- Provide data uniform storage of the simulation data that allows the generation of training/learning data sets for CPSoSaware AI components. The generation process should allow spanning data produced by different simulators and/or during different simulations into a single data record.

Following these requirements, CPSoSaware developed an extendable architecture of SAT block, presented in Figure 2.

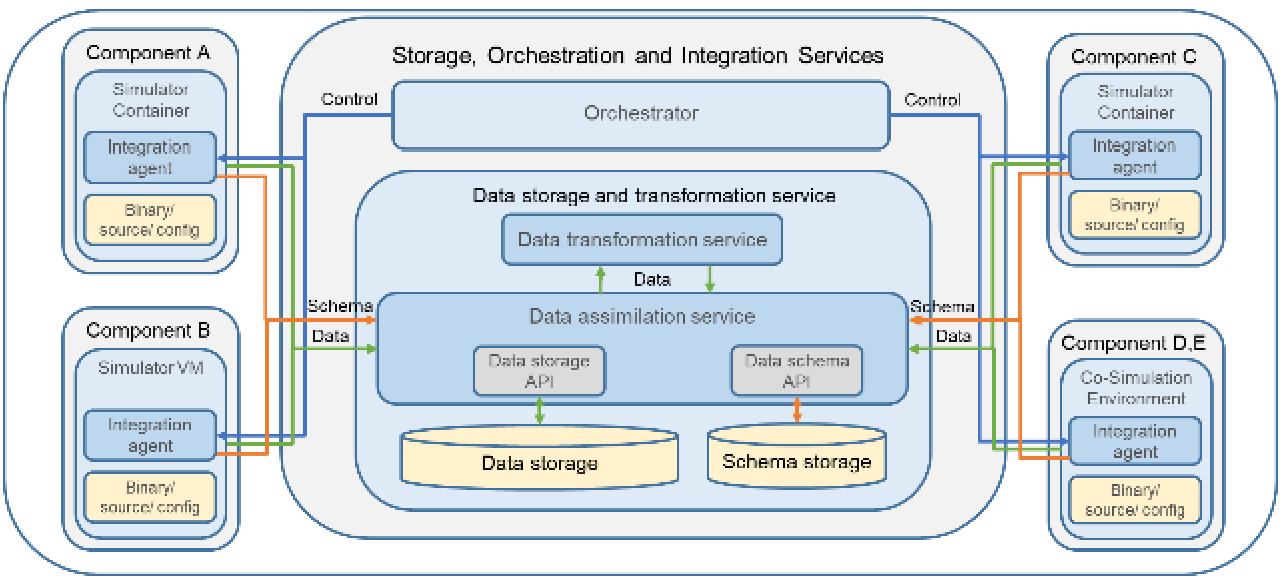


Figure 2: SAT block architecture.

According to this architecture, the SAT block consists of Storage, Orchestration, and Integration Services. Various simulators should implement an Integration agent that allows the establishment of a connection between simulator and aforementioned services. SAT block provides 3 different interfaces:

- Control interface allows to orchestrator to control simulation process. Different simulators have different abilities to control the simulation, so the implementation of this interface should be based on the following principles: first, the orchestration block should implement the most advanced version of the interface that includes all possible commands that could be supported by at least one of the connected simulators and be required for the joint simulation and data generation scenarios driven by CPSoSaware use-cases. Second, Individual integration agents should implement only a subset of the advanced interface supported by the respective simulator. Thus, the design of the advanced control interface will be performed in the iterative manner, driven by properties of the simulators that should be connected to SAT block. Once a simulator is connected to the SAT block its integration agent should expose the control features supported by the simulator. Such behavior allows the checking of the feasibility of orchestration scenarios and does not issue commands that are not supported by the particular simulator. Detailed design of the control interface is related to the CPSoSaware orchestration methodology that will be developed in Task 2.5.

- Schema interface allows the simulators to describe requirements of input and output data formats, required and optional data properties and correct types of their values. Once the simulator is connected to the SAT blocks its integration agent should send a schema for all data that the corresponding simulator should exchange with SAT block. Schema files are files in JSON formats that will be described in Section 5. Schema interface is required to enable uniform storage and data integration approach that will be discussed in Section 3.2.
- Data interface is a set of endpoints that allows sending the data from the simulator to the SAT block and from SAT block to the individual simulators. As previously described, all data formats should satisfy corresponding schemas. Initial description of data interface will be provided in Section 5.

SAT block consists of Orchestrator and Data Storage and Transformation services. Initial design of Orchestrator tool will be described in Section 4.1 and initial design of Data Storage and Transformation services will be described in Section 4.2. Different simulators that will be used in the CPSoSaware project and can be connected to SAT block discussed in Sections 4.3, 4.4, 4.5.

3.2 Integration and Storage approach

Integration of the different and diverse simulators is traditionally a complex engineering problem. It is characterized by several issues such as the usage of different modelling paradigms or languages and requires extreme effort to create and maintain the necessary integration infrastructure. This is particularly true in the CPS environment where you need to combine heterogeneous components suitable for the different aspects of the CPS. These motivations lead the designer to look for semantic integration of tools, and ontology-based integration is particularly suitable to the case.

The term “ontology” derives from ancient Greek “onto”, which means “being” and logos, which means, “discourse”. Ontology -- or roughly the "science of stuff" and how it is represented -- used to be a rather obscure branch of philosophy. It still is in some cases, but it is also an important and growing area of computer science and the web of things (WoT). Ontology has also assumed other relevant meanings, such as:

“A formal shared and explicit representation of a domain concept.”

or:

“A method for formally representing knowledge as a set of concepts within a domain, using a shared vocabulary to denote the types, properties and interrelationships of those concepts.”

or:

“A formal way to describe taxonomies and classification networks, essentially defining the structure of knowledge for various domains.”

Ontology-based data integration involves the use of ontology(s) to effectively combine data or information from multiple heterogeneous sources. Maintaining an ontology design facilitates keeping track of the terms and ensures integration efforts quickly get up to speed.

CPSoSaware consortium retains the idea that model-2-model transformation would not necessarily be the main mean of communication between tools (also, the feasibility of having fully automated model to model transformations from the system of system level down to the hardware is unlikely). Instead, each tool will manage its own model(s), and the intermediate representation will be used to exchange "cross-layers" and "cross-models" information between tools.

The intermediate format is, therefore, necessary to achieve the mediation between the application's class model conceptualization and the common domain ontology conceptualization since objects in the original format cannot be handled directly in the framework. Thus, CPSoSaware follows the Resource Description Framework (RDF)-like meta-model underlying common ontology. In particular CPSoSaware utilizes intermediate format that original developed in IBM for the semantic middleware (SEMI), and then re-used as intermediate format for the CERBERO Integration framework (CIF). This format based on the two-layered model structure that separates instances, properties and aggregations (lower level) from classes (upper level) (see Figure 3).

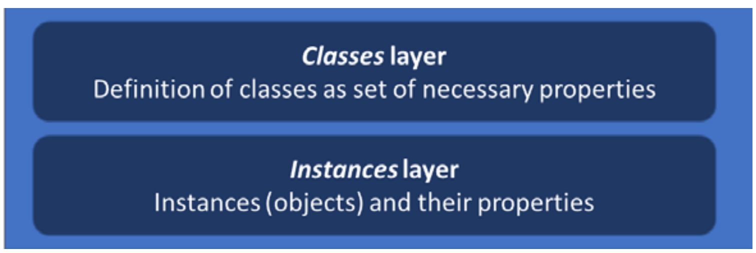


Figure 3: Two-Layered model

Each instance in this model represents a thing that possesses one or more properties within corresponding namespaces. The property itself possess a value that can be either simple (integer, float, string, etc.) or object (another instance). Aggregations are special instances that serve to represent one-to-many relations between instances, so each aggregation can "contain" several instances. An example of instance-level CIF model is presented on Figure 4.

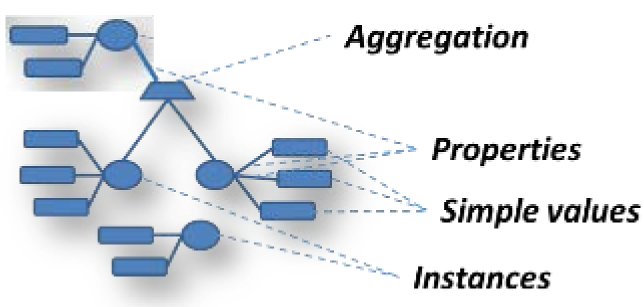


Figure 4: Example of model in the intermediate format

Classes are implemented using the classification-by-property paradigm [20]. That is, any instance that possesses some predefined set of properties becomes an instance of the corresponding class. This predefined set of properties denoted as a class definition. The set of class definitions related to the specific namespace form ontology.

Ontology helps with revealing meaning and relations of each property from the whole graph by referring to a property by its name. All properties relevant to the model are present in the ontology. Ontologies can be either simplified (i.e. system model features only a subset of all properties of the real system), or full ontology where all properties in the system model are presented in the ontology. To enable interoperability between different tools and preserve the integrity of holistic model mappings between ontologies are provided. These mappings expressed through equivalence rules between classes and define relations between instances, classes and properties coming from different namespaces. As a result, data storage contains a single model combined from different viewpoints provided by different tools.

3.3 Simulation workflow

In this section we discuss generic simulation workflows where different components of simulation and training block are involved.

A simulation workflow starts from the preparation of the simulation script and configuration data. The simulation script is passed to the orchestrator in order to define the orchestrator workflow which defines the type, number and order of simulation components/nodes that will be executed during simulation process. Configuration data stored by data storage and transformation service according to corresponding schemas and should be provided before the simulation. In order to invoke the simulation process, the user provides the name of the simulation script and identification of the configuration data.

The simulation process starts from preparation of all simulation components. When a component is ready, the integration agent of the component checks if all input and output data schemas are already registered by data storage and transformation service and performs the registration as necessary. In order to distinguish data produced by a single simulation across several different simulators / simulation nodes integration agents, the orchestrator and data storage services support the simulation id property. The simulation id allows distinguishing the data produced by one simulation script run from other runs. When the orchestrator launches a new simulation, it requests a new simulation id from the data storage service. When the orchestrator invokes simulation in the one of available simulation nodes it sends the current simulation id to the integration agent of this node and the integration agent sends simulation id together with all data produced during the simulation run. Data storage and transformation service also stores a reference to the simulation script that invoked run of the simulation with specific id.

If during the simulation process output data produced by one simulation node should be transformed to the input data of another simulation node, then corresponding transformation rules should be prepared before the simulation. These transformation rules should be written using equivalence rules syntax discussed in section 5.2.5. Written rules submitted to the ontology alignment block of the data storage and transformation service allow to perform data transformation in background. Once the output simulation data are passed and stored in the data storage service this data could be retrieved in a different format as input to other simulation nodes. All work required to transform the data from one representation format to another is performed by the data transformation service and is transparent to the end user.

All data that produced during the simulation workflow is stored by data storage and transformation service and can be queried/modified or deleted by the end user using API discussed un the section 5.2.4.

4 Simulation and Training Block Components

4.1 Orchestration tool

As presented in Section 3.1, CPSoSAware incorporates simulators from various domain introducing significant heterogeneity in various aspects. Therefore, the integration and orchestration of these diverse simulating environments that will form the end – to – end CPSoSaware platform introduces significant challenges.

These challenges are to be tackled by the orchestrator, one of the core components of the SAT architecture that is responsible to apply the continuous integration / continuous deployment (CI/CD) principles of automatically building and integrating changes as they are committed. In a nutshell, the orchestrator will be responsible to pick up the latest requirements' definitions and simulators' configurations and trigger the execution of the simulations. The simulations could consist by several simulators integrated through well-defined interfaces. Each simulator may produce outcomes to be consumed from another simulator executed sequentially.

Orchestration in the context of CPSoSaware, is based on Jenkins[1], an open source & free software that implements an automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat and it supports several version control tools (e.g. CVS[2], Subversion[3], Git[4], Mercurial[5], etc.) and can execute various build tools commands as well as arbitrary shell scripts and Windows batch commands.

---

[1] https://www.jenkins.io/
[2] https://www.nongnu.org/cvs/
[3] https://subversion.apache.org/
[4] https://git-scm.com/
[5] https://www.mercurial-scm.org/

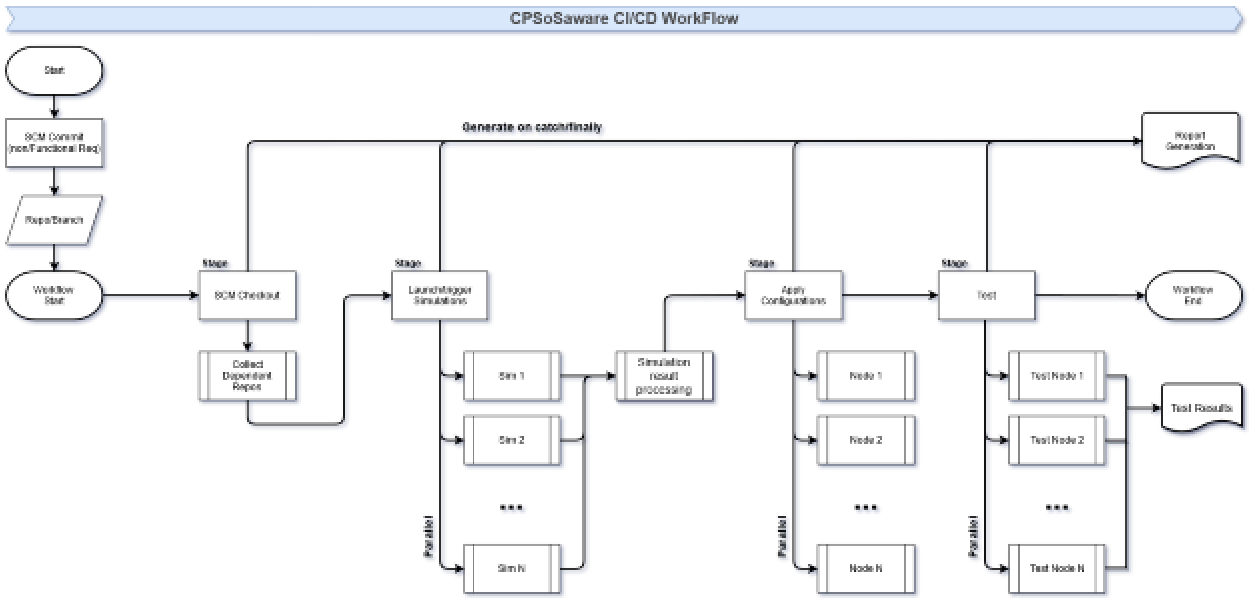


Figure 5: CPSoSaware CI/CD workflow

The workflow that will be adopted by the CPSoSaware project is presented in Figure 5. This workflow is designed based on Jenkins Pipelines[6] and there will be configured with a source code management (SCM) polling trigger.

The SCM system adopted by the CPSoSaware is Git. Git is a distributed version-control system for tracking changes in any set of files, originally designed for coordinating work among programmers cooperating on source code during software development. Its design goals include speed, data integrity, and support for distributed, non-linear workflows (thousands of parallel branches running on different systems).

Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins. A continuous delivery (CD) pipeline is an automated expression of your process for getting software from version control right through to the users. Every change to the software (committed in source control) goes through a complex process on its way to being released. This process involves building the software in a reliable and repeatable manner, as well as progressing the built software (called a "build") through multiple stages of testing and deployment. Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via the Pipeline domain-specific language (DSL) syntax. The definition of a Jenkins Pipeline is written into a text file (called a Jenkinsfile) which in turn can be committed to a project’s source control repository. This is the foundation of "Pipeline-as-code"; treating the CD pipeline a part of the application to be versioned and reviewed like any other code.

As already imposed, all the involved components in the CPSoSaware platform will be version controlled and stored in Git Repositories. These components will be:

[6] https://www.jenkins.io/doc/book/pipeline/

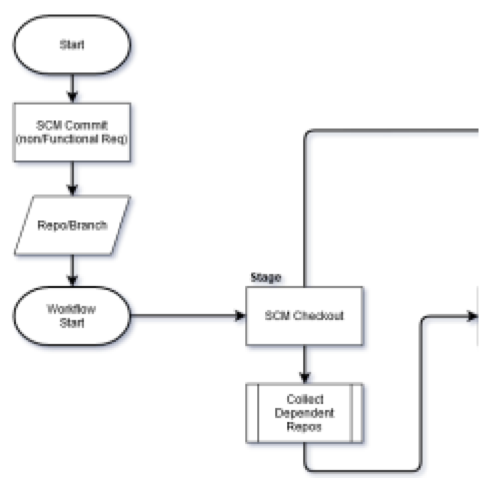
- Functional/non-Functional requirements
- Simulation suite code
- Components configurations (raspberry, FPGA, etc.)
- Components codes:
    - Bitstreams codes
    - Service codes
    - Scripts
- Test automation scripts: The testing scripts will verify that the configurations are applied/deployed successfully in the components and there is communication between them.

Also, a binary repository manager (also known as artifactory) will be configured to store 3rd party libraries and/or the outcome of the build process. This repository will store binaries such as:

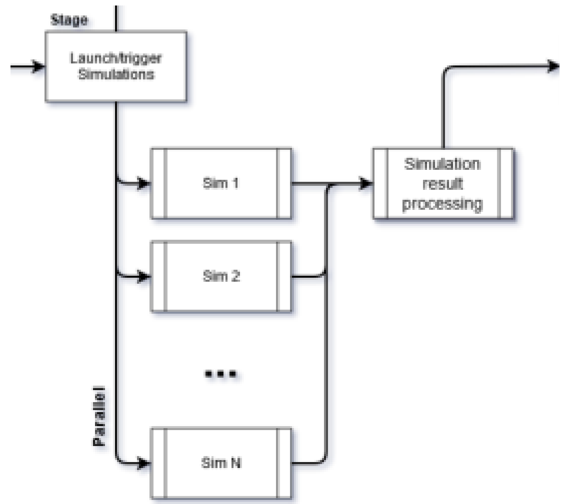
- Customized OS images
- FPGA bitstreams
- Simulation suite binaries

Workflow

The workflow setup as described will be applied in both the integration/simulation and the deployment phases of the project as well. The workflow execution is distinguished in 4 discrete stages as presented in Table 1.

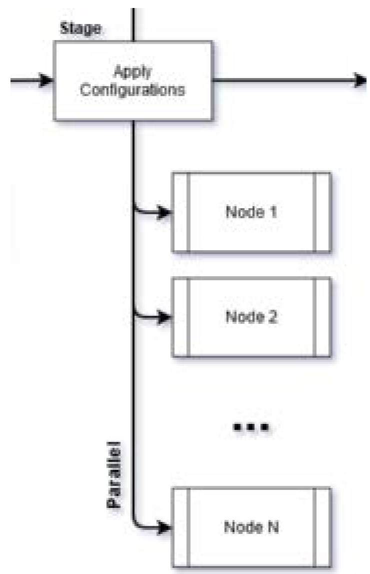
1. Main Workflow As already mentioned, the workflow will be triggered by SCM polling. When a commit is performed on Functional/non-Functional requirements repository then the workflow will be triggered. The first stage will be executed, which responsible for collecting the latest requirements.	

<table>
<tr>
<td>
2. Simulation Stage

Since the execution of the first stage completes the flow continuous to the 2nd stage. A trigger (e.g. http request) will be sent to launch the simulations. At this phase, a process will wait for all parallel simulations to finish and sequentially process the results. Processing the results regards the analysis on whether the applied configuration of the simulation met the functional/non – functional requirements.
</td>
<td>


</td>
</tr>
<tr>
<td>
3. Configuration Stage

By the completion of 2nd stage, the 3rd stage is triggered. This stage is a subject of the Continuous Deployment part of the workflow. At this phase the configuration for the nodes is available and the deployment is started.

Stage 3 is a subject of the Continuous Deployment part of the workflow. The insights of this stage is not sub
</td>
<td>


</td>
</tr>
</table>

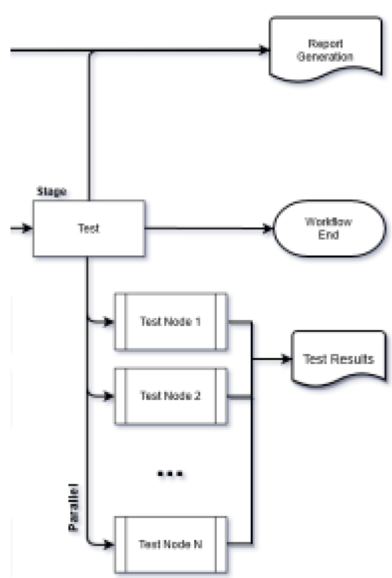
4. Test Stage The last stage execution is triggered by the end of third stage where the test automation scripts will be executed to verify that environment is working, and the nodes operate as expected. Stage 4 is a subject of the Continuous Deployment part of the workflow. The insights of this stage is not sub	

Table 1 CI/CD workflow stages

4.2 Data storage and transformation services

Initial architecture of data storage and transformation service shown on Figure 6. This architecture is based on CERBERO interoperability framework architecture and introduces several external and internal APIs that will be discussed in Section 5.2 and also several functional and storage blocks.

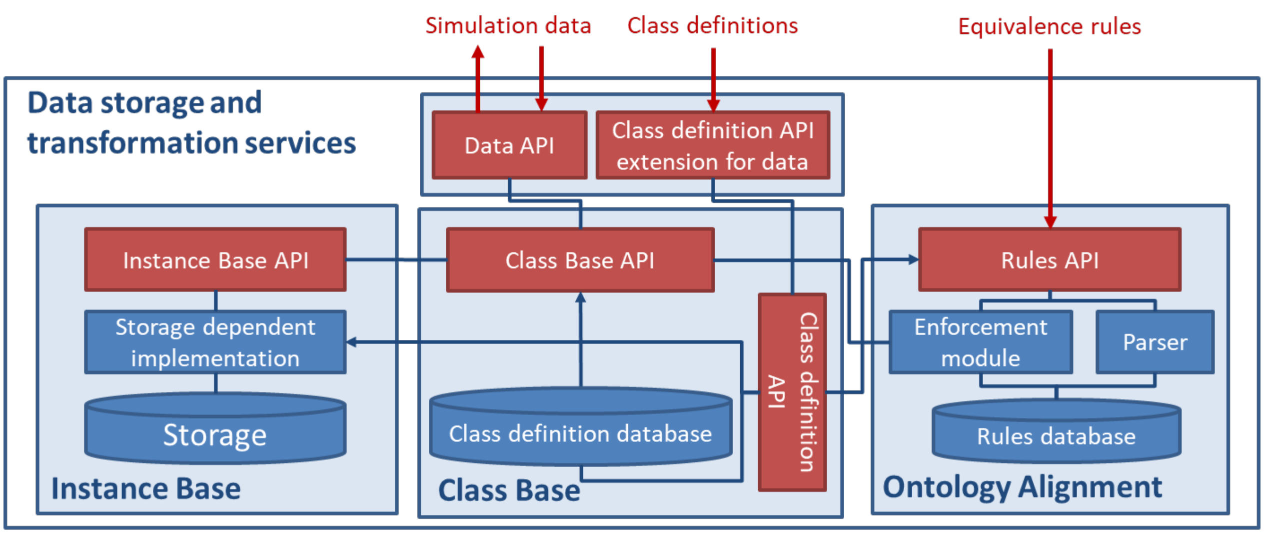


Figure 6: Architecture of data storage and transformation service

The architecture includes 4 logical blocks:

- Instance base block corresponds to the lower layer of two-layered model discussed in the Section 3.2. This block consists of data storage, storage-dependent instance base representation and instance base API that will be discussed in Section 5.2.1. Based on discussion provided in Section 2.2 CPSoSaware is considering MySQL database as data storage. Storage dependent implementation translates queries to the instance base API to SQL requests to the database. Another function of the storage-dependent implementation is to maintain MySQL database schema including tables, indexes, and keys. In particular this allows to create new tables each time when new class definition received by upper-level class definition API.

- Class base block corresponds to the upper layer of the two-layered model. discussed in the Section 3.2. This block consists of class definition storage and two internal APIs: class definition API that will be discussed in Section 5.2.2 and class base API that serve for the internal purposes. Class definition storage does not have special requirements for the large data storage and operational performance and then class definitions stored as JSON files in the service block local file system.
- Data interfaces block builds on top of class base. This block consists of two APIs. Data definition API extends class definition API with data specific featured and will be discussed in section 5.2.3. Data API allows to store and query simulation data and will be discussed in section 5.2.4.
- Ontology alignment block is required to perform data transformation from one representation to another. This transformation is based on equivalence rules that define relations between different classes. Equivalence rules syntax and API will be discussed in the section 5.2.5. User defined equivalence rules parsed and checked by parser block and applied to the class base layer by the enforcement module. Parsed rules are stored in the rules database that similarly to the class definitions database do not have special requirements for the large data storage and operational performance and then stored in the local file system in the machine-readable representation.

Data storage and transformation services plays important role in the SAT block, supporting control interface and providing schema and data interfaces.

4.3 Inter and Intra Communication Simulators

4.3.1 Intra – Communication Simulator

The NS-3 Simulator

A very common practice for network designers to evaluate network performance before deploying in a real-life deployment, is to use network simulators. A well-accepted network simulation for the research community, capable of carrying out large scale network simulation, with excellent performance is NS-3[7][8][9][10] . NS-3 is a Discrete-Event simulator (DE) which is an open, extensible network simulation platform, dominant

[7] https://www.nsnam.org/

[8] Jha, Rakesh Kumar, and Pooja Kharga. "A comparative performance analysis of routing protocols in MANET using NS3 simulator." International Journal of Computer Network and Information Security 7.4 (2015): 62-68.

[9] Mai, Yefa, Yuxia Bai, and Nan Wang. "Performance comparison and evaluation of the routing protocols for MANETs using NS3." (2017).

[10] Amina, Bengag, and Elboukhari Mohamed. "Performance evaluation of VANETs routing protocols using SUMO and NS3." 2018 IEEE 5th International Congress on Information Science and Technology (CiSt). IEEE, 2018.

in the research community. NS-3 provides models for simulate network-related use case scenarios, in respect to different wireless communication technologies.

NS -3 is built using C++ and Python with scripting capability. The ns library is wrapped by Python based on the pybindgen library which delegates the parsing of the ns C++ headers to castxml and pygccxml to automatically generate the corresponding C++ binding glue. These automatically generated C++ files are finally compiled into the ns Python module to allow users to interact with the C++ ns models and core through Python scripts. The ns simulator features an integrated attribute-based system to manage default and per-instance values for simulation parameters.

The general process of creating an NS-3 based simulation can be divided into several steps:

1. Topology definition: To ease the creation of basic facilities and define their interrelationships, ns-3 has a system of containers and helpers that facilitates this process.
2. Model development: Models are added to simulation (for example, UDP, IPv4, point-to-point devices and links, applications); most of the time this is done using helpers.
3. Node and link configuration: models set their default values (for example, the size of packets sent by an application or MTU of a point-to-point link); most of the time this is done using the attribute system.
4. Execution: Simulation facilities generate events, data requested by the user is logged.
5. Performance analysis: After the simulation is finished and data is available as a time-stamped event trace. This data can then be statistically analyzed with statistical tools (e.g. R, Matlab, python libs, etc.) to draw conclusions.
6. Graphical Visualization: Raw or processed data collected in a simulation can be graphed using tools like Gnuplot, matplotlib or XGRAPH.

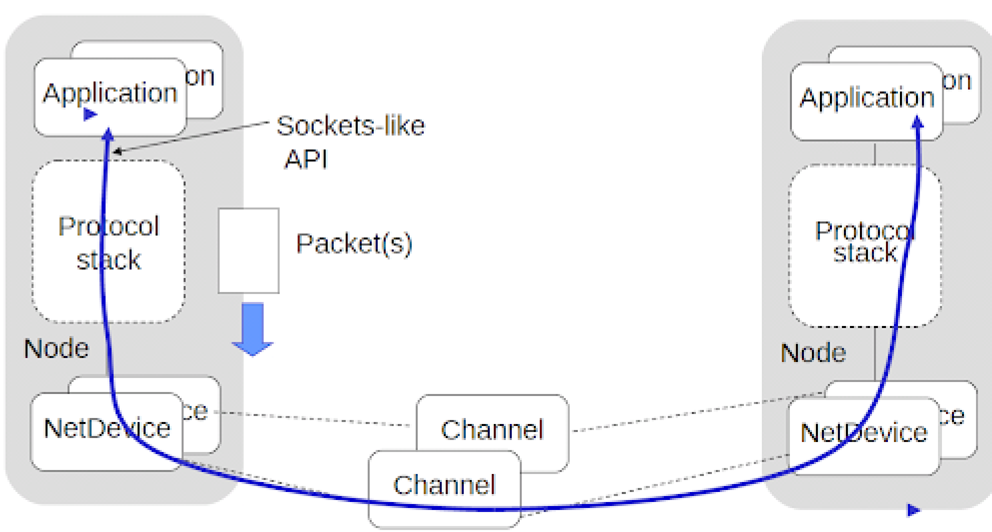


Figure 7: NS-3 Basic Simulation Model

In NS-3 the basic computing device abstraction is called, Node. Node representing a class that provides methods for managing the representations of computing devices in simulations, with added functionality.

As shown, in Figure 7, each node contains one or more sub-modules, to represent different applications, protocol stacks and communication technologies. Finally, in order to accomplish inter-node communication inside the simulation environment, simulation designer must define Channels.

Following a bottom-up approach, at the bottom level of a NS-3 Node, the actual network interfaces are setup (which can be more than one), namely NetDevice. NetDevices can be realized as the unix “eth0” for example. Just as in a real computer, a Node may be connected to more than one Channel via multiple NetDevices. In NS-3 the net device abstraction covers both the software driver and the simulated hardware. A net device is “installed” in a Node in order to enable the Node to communicate with other Nodes in the simulation via Channels. NetDevices allow simulation designed to setup models for simulating different PHY-related models as well as MAC-related Models. Specialized versions of the NetDevice are PointToPointNetDevice, WifiNetDevice.

The basic abstraction for the communication between nodes inside the simulation is the Channel. NS-3 provides multiple specialized versions of Channel, that is PointToPointChannel and WifiChannel. Channels in NS-3 represents a basic communication interface, which provides methods for managing communication subnetwork objects and connecting nodes each other. Channels allow simulation designers to setup models for simulating different propagation models that affecting end-to-end delays and packet loss.

Finally, NS-3 provides helpers for simplify the process of the creation of a simulation using the Topology Helpers. A simple process for setup, configure and run a simulation in NS-3, contain the following the process:

1. Firstly, the nodes that will participate in the simulation must be created. For that purpose, the NodeContainer helper is needed, used to install and configure simulation Nodes. The NodeContainer topology helper provides a convenient way to create, manage and access any Node objects that is created in order to run a simulation.
2. Then a channel is created that will be used for the communication between the simulation nodes. A simple Helper that can be used is PointToPointHelper, which creates a point-to-point channel between simulation nodes.
3. The next thing, that must be configured is the net devices that will be installed in the node of the simulation. The NetDeviceContainer, with help from the PointToPointHelper will install the NIC on the nodes and establish the connection between the nodes with a PointToPointChannel.
4. Finally, the internet stack is installed on the net devices of the nodes, using the InternetStackHelper, using giving IP/TCP capabilities to the nodes.

On top of the simulation nodes that are created, it’s time to install the application that will generate the traffic for the simulation.

Finally, in order to run the simulation, the start and the end of the simulation in seconds must be defined.

The NS-3 Benefits

One of the fundamental goals in the ns–3 design was to improve the realism of the models, i.e., to make the models closer in implementation to the actual software implementations that they represent. Different simulation tools have taken different approaches to modelling, including the use of modelling-specific languages and code generation tools, and the use of component-based programming paradigms. While high-level modelling languages and simulation-specific programming paradigms have certain advantages, modelling actual implementations is not typically one of their strengths. In the authors' experience, the higher level of abstraction can cause simulation results to diverge too much from experimental results, and therefore an emphasis was placed on realism. The ns–3 Network Simulator programming language in part because it better facilitated the inclusion of C-based implementation code. ns–3 also is architected similar to Linux computers, with internal interfaces (network to device driver) and application interfaces (sockets) that map well to how computers are built today. NS–3 also emphasizes emulation capabilities that allow NS–3 to be used on testbeds and with real devices and applications, again with the goal of reducing the possible discontinuities when moving from simulation to experiment.

Another benefit of realism is reuse. ns–3 is not purely a new simulator but a synthesis of several predecessor tools, including ns–2 itself (random number generators, selected wireless and error models, routing protocols), the Georgia Tech Network Simulator (GTNetS)[393], and the YANS simulator[271]. The software that automates the construction of network routing tables for static topologies was ported from the quagga routing suite. ns–3 also prioritizes the use of standard input and output file formats so that external tools (such as packet trace analyzers) can be used. Users are also able to link external libraries such as the GNU Scientific Library or IT++.

A third emphasis has been on ease of debugging and better alignment with current languages. Architecturally, the chosen design was to emphasize purely C++-based models for performance and ease of debugging, and to provide a Python-based scripting API that allows ns–3 to be integrated with other Python-based environments or programming models. Users of ns–3 are free to write their simulations as either C++ main() programs or Python programs. ns–3's low-level API is oriented towards the power-user, but more accessible "helper" APIs are overlaid on top of the low-level API.

THE NS-3 in CPSoSaware

In the context of the CPSoSaware, NS-3 will be adapted to run as a Software as a Service in order to facilitate the integration with the CPSoSaware platform. External software interfaces will be responsible to feed NS-3 with new configurations and trigger simulation experiments. Additionally, a mechanism for extracting and processing the simulation traces will be also designed. This mechanism aims to support the post – analysis of the simulation results and the decisions on whether the network configuration meets the application functional and non – functional requirements. The internals of this design and implementation are not subject of this deliverable and are described in more detail in D4.2. An overview of this architecture is depicted on Figure 8.

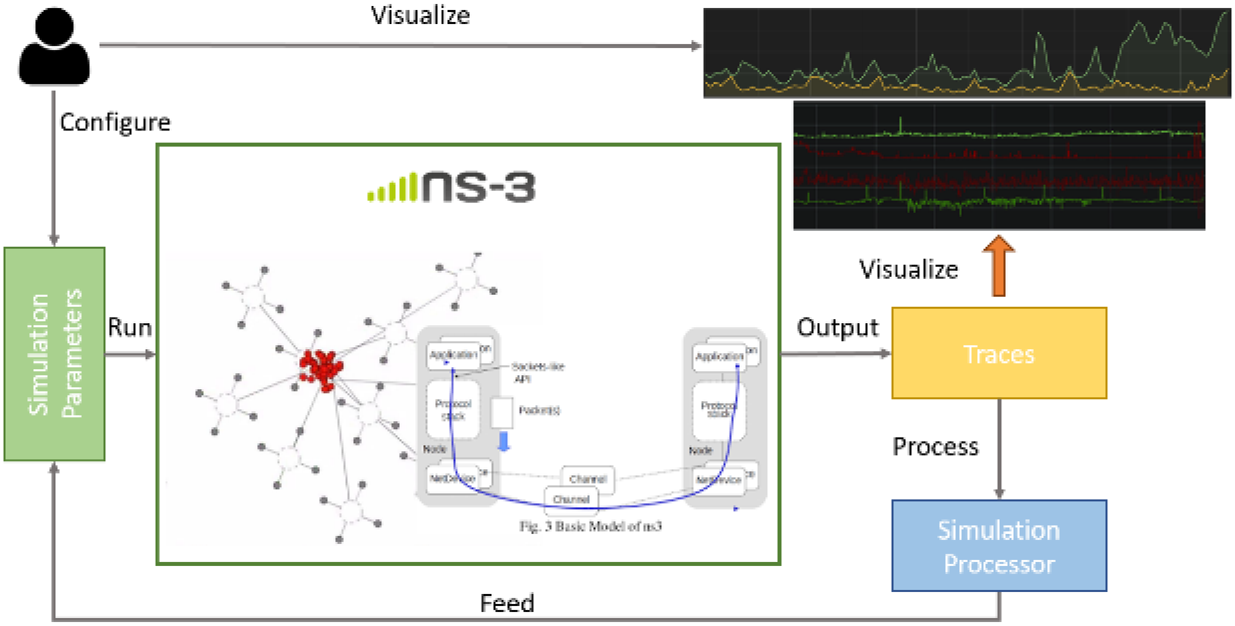


Figure 8: NS-3 in CPSoSaware

4.3.2 Inter – Communication Simulator

In order to simulate communication between the devices, it is necessary to simulate both the movement of these vehicles in a scenario, and the radio propagation of the different messages sent by each one of the devices. This propagation is affected both by distance and obstacles, which can attenuate and produce signal reflections.

SUMO (https://networksimulationtools.com/sumo-simulator/) is a simulator extensively used which can model traffic systems including road vehicles, from a microscopic level to a highly complex macroscopic, It can manage a fleet of vehicles which follow traffic rules, including traffic lights, zebra crossings, intersection precedence, etc. By itself, it is possible to extract from it relevant metrics related to traffic throughput in different scenarios by adding new lanes or improving traffic light plans. Also, other indicators such as vehicle emissions can be assessed from the simulations.

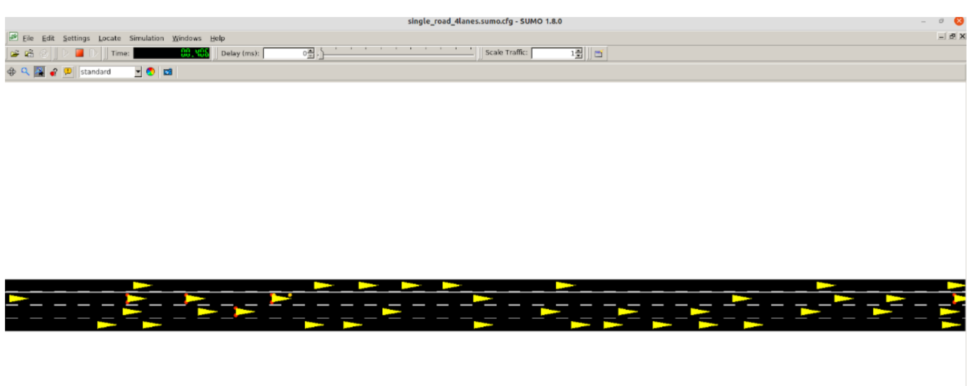


Figure 9 SUMO Highway Scenario

As commented, modelling vehicle movement is half of the for the simulation environment. A second layer of modellization has been included by adding OMNET++ to simulate the V2X message radio propagation for each vehicle. It is a simulation software which allows to visualize the signal propagation of the messages.

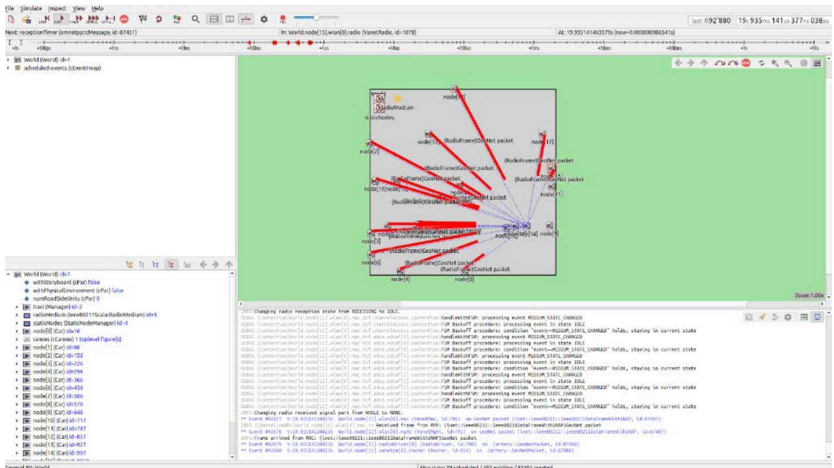


Figure 10: OMNET++ propagation example

OMNET++ can use various simulation protocols in the simulations. A package for V2X simulation has been used, compliant with ETSI ITS-G5 protocol including GeoNetworking and BTP. The V2X Cooperative Awareness Message (CAM) , broadcasted by each unit at 10 Hz is used as main message. It includes information about the position, heading and speed of each V2X actor, among others.

The connectivity between booth tools is done with the TraCI protocol. Once started, SUMO accepts clients connected using a TCP connection. Once this connection is established, the client, in our case OMNET++, can trigger each simulation step. Radio propagation simulation for many vehicles takes a high amount of time and cannot run in real time. Some effort in reducing the number of vehicles has been done, trying to alleviate this situation and obtain better performance.

Initially the connection between both simulators was created directly. However, to gain more flexibility a custom proxy has been developed in the middle, which is transparent in the communication between SUMO and OMNET++, but can resend the information either to other tools, a log file, etc.

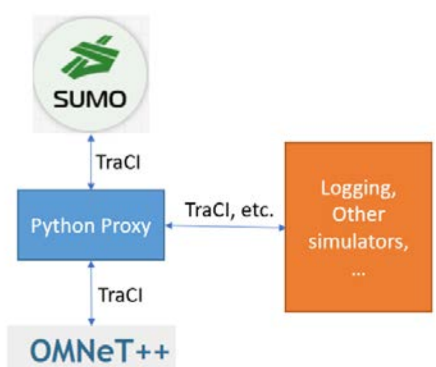


Figure 11: Tool interface

In the context of the CPSoSaware, these tools model the movements and interaction of the vehicles and other possible actors such as pedestrians. The latter are involved in the V2X ecosystem either by sending VAM (Vulnerable User Awareness Messages, similar to CAM) from their mobile phones , or by proxy, detected with a camera and them the VAM or CAM message generated from the network on their behalf.

4.4 CPSoS HW-SW Simulators

The OpenASIP-based soft core processors that are studied in the project for FPGA programming as well as reliable co-processing can be simulated with a retargetable instruction-set simulator. The simulator is called *ttasim* and it provides instruction cycle accurate results for the Transport-Triggered Architecture based co-processors developed using the OpenASIP tools. The simulator is driven with an architecture description format called Architecture Description File (ADF) which contains all the necessary information required for cycle accurate modelling, but it does not provide dynamic latency information e.g. from unideal memory hierarchies. For this level of accuracy, OpenASIP provides System C hooks that can be used to connect ttasim co-processor models to larger system level simulations with desired accuracy, along with more accurate memory models. The simulator can provide also utilization statistics for how many times operations were executed and in which function unit, the bus utilization and so on. It provides also basic software debugging features such as breakpoints and single stepping.

There is also a graphical user interface for the OpenASIP simulator engine called *Proxim* (from *processor simulator).* This simulator also has visualizations for utilization, as exemplified in Figure 12.

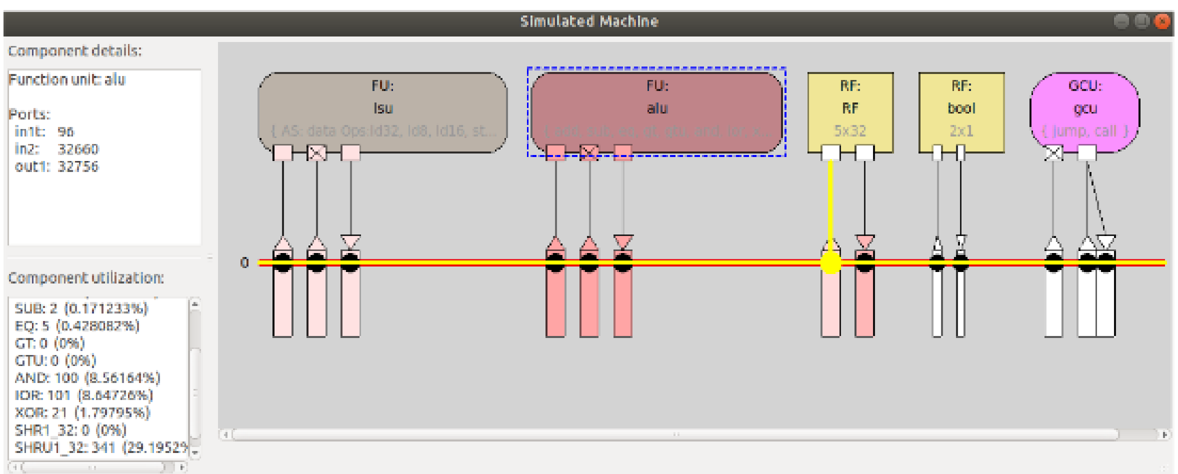


Figure 12: Proxim's utilization visualization of a minimal co-processor architecture. The redness indicates the level of utilization (so far) by the program.

4.5 CPSoS Use-case dedicated simulators

4.5.1 Simulator for ADAS/AV systems research

4.5.1.1 CARLA simulator

CARLA is a tool for AV/ADAS systems research that is also an open-source simulator based on MIT license allowing commercial and research use. CARLA assets are distributed using CC-BY License.
CARLA Simulator was developed in a flexible and modular way with dedicated API that allows easy integrations with autonomous driving applications. These applications can include AV stack related to perception and control algorithms including these based on deep learning and rule-based frameworks. CARLA is based on following set of technologies:

- Core engine: Unreal Engine, popular engine with powerful 3D rendering capabilities allowing the development of photorealistic simulations.
- Road network logic system: OpenDRIVE (as of February 2021 version 1.4 is used) which contains information specific for simulation applications like road geometry, surface properties, signs, lane types, directions and markings. Several road editors use OpenDRIVE standard for creating AV/ADAS testing grounds (e.g. parts of specific cities) that can be later on imported to CARLA simulator.
- Basic architecture principle: scalable client-server architecture. Server manages the simulation including physics computations, rendering of sensors, real world data updates (actors and other relevant objects). Server can be run using GPU processing capabilities.
- Simulation control: API that works with both C++ and Python.

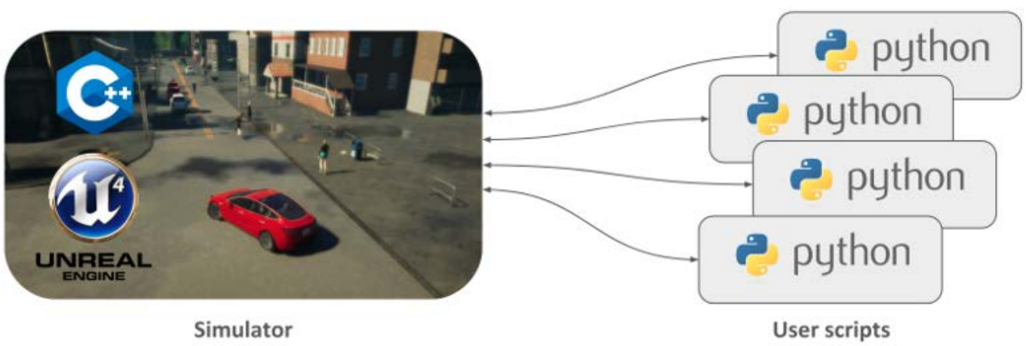


Figure 13: Basic structure of CARLA simulator for AV/ADAS research

CARLA simulator contains of following key modules:

- Sensors. Sensors acquire information from the surrounding world by being mounted on the vehicle. CARLA supports multiple sensor types e.g.
    - Camera – RGB camera.
    - GNSS – geolocation of the vehicle and sensor.
    - IMU – inertial measurement unit that contains gyroscope, compass and accelerometers.
    - LIDAR – rotating LIDAR with 4D point cloud coordinates and intensity.
    - Radar – 2D point map.
    - Semantic LIDAR – rotating LIDAR with 3D point cloud including semantic segmentation.
- Traffic manager – vehicles/agents controller (non-ego vehicle controlled by user scripts).
- Data recorder – it is recording the whole scenario (step-by-step approach) allowing to replay every time step of the recorded scenario.

- ROS bridge – Robot Operating System integration – allowing two-way communication between CARLA and ROS, however with performance limitations.
- Additional assets – multiple urban settings with basic weather conditions control.

4.5.1.2 Robotec.ai Real World Simulator

Robotec.ai has developed Real World Simulator which is a proprietary tool for developing, testing and validating autonomous vehicles. Real World Simulator is currently used by OEMs and Tiers to develop mainly commercial autonomous vehicles (e.g. cargo delivery ODD). Still simulator can be used in passenger vehicle and public roads use case.

Simulator has been developed in modular way to allow engineers integrate multiple AV/ADAS stack elements (e.g. perception stack or control system of the vehicle). Dedicated ros2cs module enables fast integrations with software developed on top of ROS and ROS2 middleware (https://design.ros2.org/articles/ros_middleware_interface.html).

Robotec.ai offers a modular, extendable, ROS(2)-based simulation platform to configure, develop and integrate AV/ADAS components. Architecture of the simulation platform is shown in Figure 1 below.

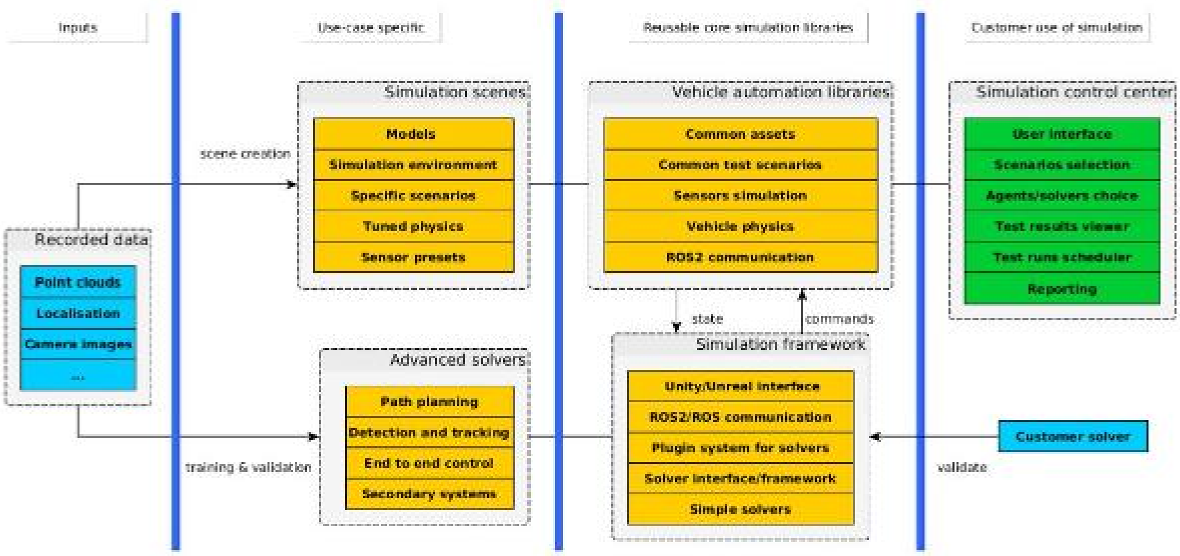


Figure 14: High level architecture of Robotec.ai Real World Simulator.

This modular architecture allows customers to use custom environments developed with detailed real world input data such us point clouds or geodata. It is also possible to use existing compatible outdoor or indoor scenes and additional external models that can be positioned in the scene.

Robotec.ai simulation platform also allows to test and validate AV solutions through the system of advanced solvers. Solvers are AV/ADAS algorithms such as localization, path planning or perception that the customer would like to test. Existing customer solvers can be plugged into the simulation.

Robotec.ai Real World Simulator development is driven by the ROS2 middleware. We have developed unique ROS2 module for Unity3D engine.

ROS and Unity3D don't scale easily the programming languages, key concepts and coordinate frames are all different. Robotec.ai has developed a high-performance communication module for Unity3D. This unique module delivers following results: the performance of communication is increased by approximately 800 times comparing to third-party bridge solutions and it supports all standard as well as custom messages with message generation.

Robotec.ai simulator contains following key modules:

- Real World Simulator architecture including handling of agents, support for multiple ego vehicles, launch files configuration.
- Management system of states and logs from integrated autonomous driving software.
- Sensors simulation library:
    - Sensor template interface supporting ROS2 messages.
    - GPS (Global Positioning System) - sensor simulation providing location in global coordinate system. Conversion point with rotation has to be defined in the simulation scene to properly convert local coordinate system of simulation to global coordinate system.
    - IMU (Inertial measurement unit) – sensor simulation combined of accelerometers and gyroscopes, providing measurements related to angular and linear movement (translation, velocity, acceleration).
    - LIDAR (LIght Detection and Ranging) – sensor simulation illuminating the target with laser and creates 3D point cloud by measuring the Time of Flight of reflected rays to the detector. Simulated LIDAR has multiple parameters enabling simulation of variety of sensor models (2D and 3D).
    - Camera – RGB camera simulation with management mechanism for handling multiple field of views.
- Ros2cs module enabling high performance ROS2 communication for Unity3d.
- Vehicle physics adapted for vehicles.
    - Engine, transmission and suspension simulation.
    - Support for multiple controllers (including steering wheel controllers).
- Support tool for creating realistic simulation scenes based on real world data:
    - 3D mesh models generation from real LIDAR measurements.

4.5.1.3 CARLA and Robotec Real World Simulator in CPSoSaware

One of the key aspects of the developed CPSoSaware components for the automotive AV/ADAS use case are following:

- Deep multimodal scene understanding that provides multimodal sensor data (RGB/Lidar, RGB/depth) to be analysed by Computer Vision/deep learning mechanisms and produce high-level observations/detections.
- Multimodal localization - Requests the production of localization data from the combination of measurements (e.g. GPS, LiDAR) for metrics like arrival/departure and trajectories.
- Path-planning path generation between nodes.

Evaluated use cases in automotive pillar of CPSoSaware project will be following:

- Human in the loop control use case in single vehicle scenario

- Cybersecurity issues in connected cars scenario
- Cooperative awareness scenario

Cybersecurity and cooperative awareness use cases needs to be validated in the environment with multiple controlable ego-vehicles, what makes Robotec Real World Simulator the best fit for those scenarios.

For the efficient implementation (development, testing and validation) of the listed modules following key assets are required from the simulation:

- Realistic representation of the environment. Achievable in both simulation environments (CARLA and Robotec Real World Simulator).
- Available sensors (LIDAR, GNSS, IMU, RADAR, Camera, vehicle data). All sensors are available in both simulation environments (CARLA and Robotec Real World Simulator). However, Carla provides more photorealistic environment for simulations focused on camera applications.
- Machine learning support. Support provided in both simulation environments (CARLA and Robotec Real World Simulator).
- Communication interfaces (ROS with preference for new version ROS2). Communication provided in both simulation environments (CARLA and Robotec Real World Simulator).
- Possible simulation of multiple agents. Achievable in both simulation environments (CARLA and Robotec Real World Simulator).
- Possible extension with additional modules: driver behavior, V2X communication, cooperative collision warning system. Achievable in both simulation environments (CARLA and Robotec Real World Simulator). However, none of both simulation environments provides V2X communication for now (this is also planned as one of the CPSoSaware components).

Below Figures show all relevant AV/ADAS sensors attributes for CPSoSaware use cases:

Blueprint attribute	Type	Default	Description
`noise_alt_bias`	float	0.0	Mean parameter in the noise model for altitude.
`noise_alt_stddev`	float	0.0	Standard deviation parameter in the noise model for altitude.
`noise_lat_bias`	float	0.0	Mean parameter in the noise model for latitude.
`noise_lat_stddev`	float	0.0	Standard deviation parameter in the noise model for latitude.
`noise_lon_bias`	float	0.0	Mean parameter in the noise model for longitude.
`noise_lon_stddev`	float	0.0	Standard deviation parameter in the noise model for longitude.
`noise_seed`	int	0	Initializer for a pseudorandom number generator.
`sensor_tick`	float	0.0	Simulation seconds between sensor captures (ticks).

Figure 15: GNSS attributes in CARLA simulator as specified in the GNSS sensor documentation.

Blueprint attribute	Type	Default	Description
`channels`	int	32	Number of lasers.
`range`	float	10.0	Maximum distance to measure/raycast in meters (centimeters for CARLA 0.9.6 or previous).
`points_per_second`	int	56000	Points generated by all lasers per second.
`rotation_frequency`	float	10.0	LIDAR rotation frequency.
`upper_fov`	float	10.0	Angle in degrees of the highest laser.
`lower_fov`	float	-30.0	Angle in degrees of the lowest laser.
`atmosphere_attenuation_rate`	float	0.004	Coefficient that measures the LIDAR instensity loss per meter. Check the intensity computation above.
`dropoff_general_rate`	float	0.45	General proportion of points that are randomy dropped.
`dropoff_intensity_limit`	float	0.8	For the intensity based drop-off, the threshold intensity value above which no points are dropped.
`dropoff_zero_intensity`	float	0.4	For the intensity based drop-off, the probability of each point with zero intensity being dropped.
`sensor_tick`	float	0.0	Simulation seconds between sensor captures (ticks).
`noise_stddev`	float	0.0	Standard deviation of the noise model to disturb each point along the vector of its raycast.

Figure 16: LIDAR attributes in CARLA simulator as specified in the LIDAR sensor documentation.

Blueprint attribute	Type	Default	Description
`bloom_intensity`	float	0.675	Intensity for the bloom post-process effect, `0.0` for disabling it.
`fov`	float	90.0	Horizontal field of view in degrees.
`fstop`	float	1.4	Opening of the camera lens. Aperture is `1/fstop` with typical lens going down to f/1.2 (larger opening). Larger numbers will reduce the Depth of Field effect.
`image_size_x`	int	800	Image width in pixels.
`image_size_y`	int	600	Image height in pixels.
`iso`	float	100.0	The camera sensor sensitivity.
`gamma`	float	2.2	Target gamma value of the camera.
`lens_flare_intensity`	float	0.1	Intensity for the lens flare post-process effect, `0.0` for disabling it.
`sensor_tick`	float	0.0	Simulation seconds between sensor captures (ticks).
`shutter_speed`	float	200.0	The camera shutter speed in seconds (1.0/s).

Figure 17: Camera attributes in CARLA simulator as specified in the camera sensor documentation. Please note that camera has also camera lens distortions options as separate attributes group.

Blueprint attribute	Type	Default	Description
noise_accel_stddev_x	float	0.0	Standard deviation parameter in the noise model for acceleration (X axis).
noise_accel_stddev_y	float	0.0	Standard deviation parameter in the noise model for acceleration (Y axis).
noise_accel_stddev_z	float	0.0	Standard deviation parameter in the noise model for acceleration (Z axis).
noise_gyro_bias_x	float	0.0	Mean parameter in the noise model for the gyroscope (X axis).
noise_gyro_bias_y	float	0.0	Mean parameter in the noise model for the gyroscope (Y axis).
noise_gyro_bias_z	float	0.0	Mean parameter in the noise model for the gyroscope (Z axis).
noise_gyro_stddev_x	float	0.0	Standard deviation parameter in the noise model for the gyroscope (X axis).
noise_gyro_stddev_y	float	0.0	Standard deviation parameter in the noise model for the gyroscope (Y axis).
noise_gyro_stddev_z	float	0.0	Standard deviation parameter in the noise model for the gyroscope (Z axis).
noise_seed	int	0	Initializer for a pseudorandom number generator.
sensor_tick	float	0.0	Simulation seconds between sensor captures (ticks).

Figure 18: Inertial Measurement Unit (IMU) attributes in CARLA simulator as specified in the IMU sensor documentation.

4.5.1.4 Robotec V2X Simulator

Robotec.ai is developing V2X Simulator as ROS2 module, that can be integrated with any AV simulator having support for ROS communication. The simulator works as external module with replicated both static environment (scene) and all dynamic objects.

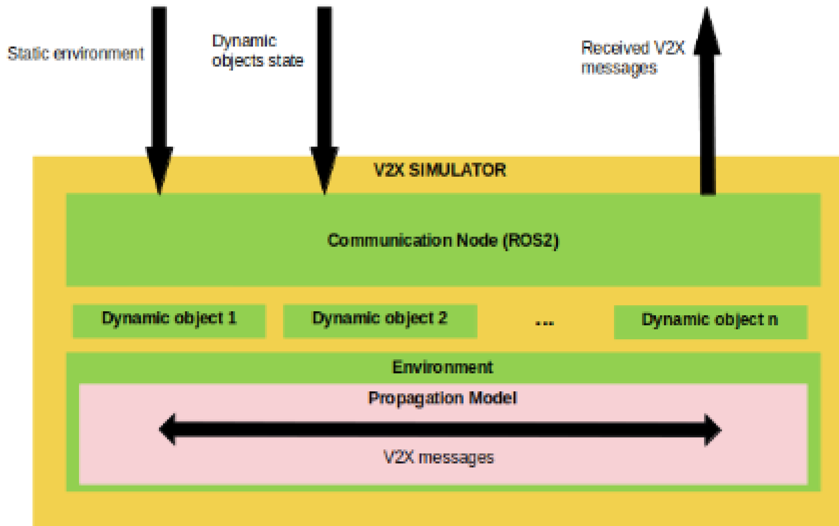


Figure 19: V2X simulator

Integration with AV simulator consists of 3 types of interfaces:

- Static environment communication – ROS message containing information about the scene. Environment is created in V2X simulation only once, on initialization of simulated use case.
- Dynamic objects state – ROS message send periodically from each traffic agent. This message is responsible for sharing locations of agents and all the data transmitted in V2X message from AV simulator to V2X Simulator.
- Received V2X messages – ROS message sending all received V2X messages back to AV Simulator

V2X communication is performed in V2X simulator, based on locations of agents, environment and Propagation Model. Currently only Free Space Path Loss model is implemented, in future more advanced models will be developed.

5 Simulation and Training Block Interfaces

In this section we present several APIs used for communication within the SAT. Good practice demands that publicly accessible APIs be protected against unauthorized access. The requirements for authentication and authorization will be addressed in the next deliverable.

5.1 Orchestrator interfaces

The orchestration tool described in Section 4.1 is responsible for triggering, handling and monitoring all the intermediate steps of the simulation workflow. The execution of a workflow is handled through a REST API and can be triggered either on demand or event – driven. This API is exposed via a REST API Gateway that publishes the two aforementioned approaches via the services presented on Figure 20.

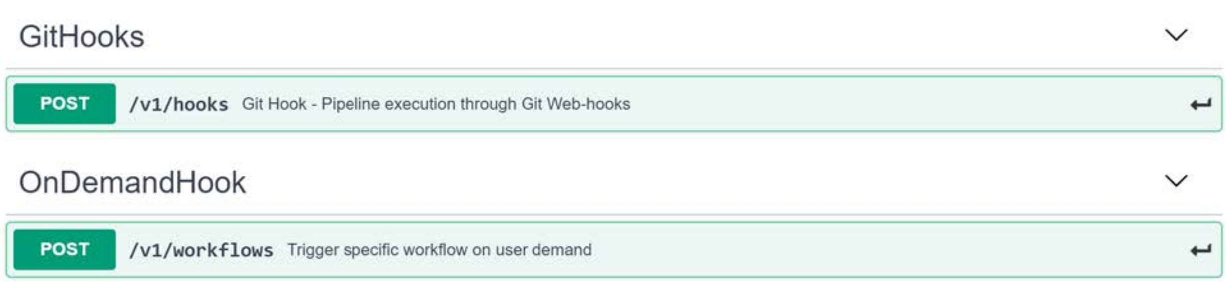


Figure 20: CI/CD Orchestrator Control API

The pipeline in the context of the CPSoSAware consists of one or more steps that each reflect on specific simulation execution. These workflows are registered in independent isolated branches along with their configurations. Every new commit on each of the branches will trigger the execution of the workflow by the orchestrator. This is performed by a software agent that monitors the git repository of CPSoSAware.

On the other hand, the *"OnDemandHook"* gives control to the user to trigger a workflow with specific configuration. This configuration defines the steps that compose the user defined workflow and the application requirements (use case dependent). This is performed through a POST HTTP call with payload the yaml file presented in Table 2. This file describes an array of steps to be executed sequentially. Each step is defined by the following properties:

- Unique identifier (id),
- A trigger_uri that actually caries the http call that will be used by the orchestrator to trigger the particular simulation. This trigger_uri is simulator specific and the respective http services (where applicable) will be implemented in the context of the CPSoSAware,
- The file path of the configuration file for the particular simulation step. This configuration file is simulator specific as well and must be accessible by the simulation environment that will consume it.

Table 2 OnDemandHook service payload


```
workflow:
  - id: "step1"
    trigger_uri: "http://esdalab.ece.uop.gr /ns3/step/run1"
```

```
    input: "http:// esdalab.ece.uop.gr /ns3/step/input.json" #
Optional
  - id: "step2"
    trigger_uri: "http:// esdalab.ece.uop.gr /ns3/step/run2"
    input: "http://esdalab.ece.uop.gr /ns3/input.json" #
Optional
  - id: "step3"
    trigger_uri: "http:// esdalab.ece.uop.gr /ns3/step/run3"
    input: "http:// esdalab.ece.uop.gr /ns3/step/input.json" #
Optional
```

Apart from the interfaces defined under the scope of the orchestration tool, additional interfaces are to be defined and implemented for the control of the simulators by the orchestrator. The first simulator to be integrated with the orchestrator is the Network Simulator 3 (NS3) used in the intra – communication layer. This simulator is a command line executable written in C++ and python. In the context of CPSOSoSAware, a wrapper is implemented that delegates the execution of a simulation to a REST API. This API exposes a POST http call as presented in Figure 21. The payload of this request is a json file with two main fields:

- Callback: The callback property supports the execution of actions after the completion of a simulation step
- Filename: The filename of the configuration that the simulator must execute. In the NS3 case, this file is a .cpp file that describes the simulation in terms of network topology, device attributes, channel attributes, Error Rate Models, Channel models, etc.

Figure 21 NS3 Simulation Interface

Further API interfaces are to be designed and developed while the simulators used in the CPSoSAware project are integrating to the CI/CD workflow and the orchestrator. The enhanced version of API will be further presented in the next version of this deliverable.

5.2 Data storage and transformation services interfaces

5.2.1 Low-Level Instance Base Interface

As mentioned in section 4.2, we describe the low-level interface intended for direct manipulation with basic structures that includes instances, properties and aggregations.

Instance – represents a simple object that has unique identifier (UID) that allow to distinguish one instance from another. Instances can possess properties.

Property - represents a relation between instance (property owner) and property value, that can be either simple value, another instance or aggregation. Each property has name and namespace. Any instance can possess (be owner of) only one property with specific name within specific namespace.

Aggregation – generalization of the instance that served to represent one-to-many relations. May contain any number of members that can be either instances or simple values. In the current implementation cannot possess properties. The question on whether aggregation can or cannot possess properties is under investigation.

The low-level interface includes methods to get/create/delete instances, properties, aggregations and namespaces. It also includes additional methods that are created to fulfill functionality required by high-level APIs such as class base or equivalence rules. This interface in general is not supposed to be used by connected simulators or other tools, unless connected tool do not introduce extension to data storage and transformation itself.

5.2.2 Class definition interface

Class definition interface introduces two APIs. The first API is developed to manage namespaces, where namespace is an object that includes two fields: name and version. Once namespaces are created, the interface receives unique id and further operations which are performed using this id. The API to manage namespaces shown on Figure 22

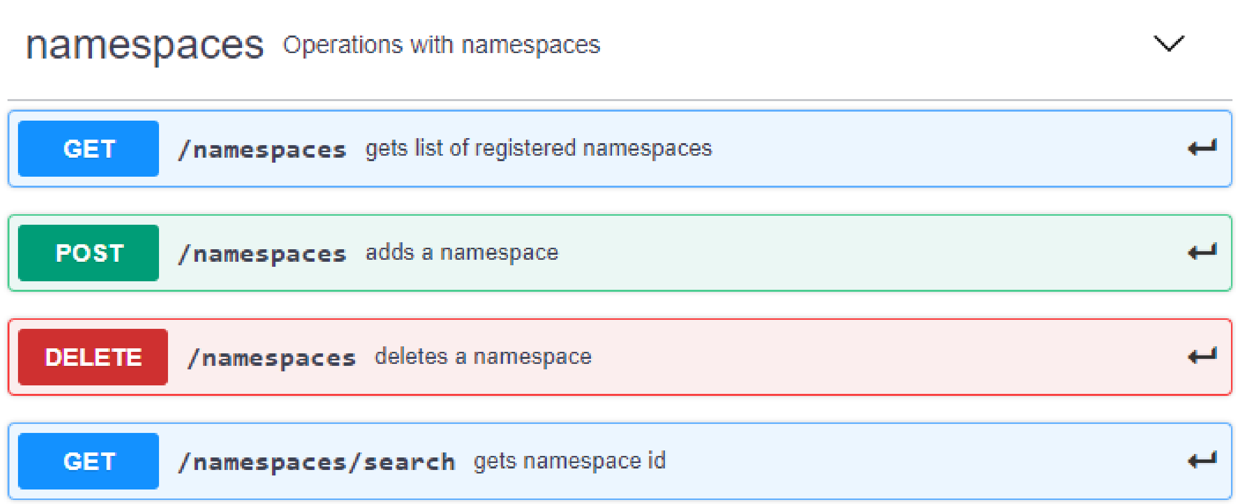


Figure 22: API for namespace management

The API includes methods to get a list of all registered namespaces, add new namespace, delete namespace by id, and get the namespace id by name and version if namespace is already registered. When new simulation component connects to the data storage and integration service, its integration object checks if the corresponding namespace is already registered by sending a GET namespace/search request. If the namespace is not registered, then the integration agent registers the namespace by sending a POST /namespaces request, and then registers the corresponding classes and data schemas.

The second API is developed to manage classes. The CPSoSaware simplified ontology introduces a class definition object that includes the following fields:

- "namespace" – namespace object for which belongs class definition.
- "class" – name of the class.
- "schema" – schema of the class.

Schema consists of properties definitions enlisted under “properties” key, where each property includes The following fields:

- "name" - name of the property.
- "namespace" – optional. Should be provided if property has different namespace (not same as defined in the schema namespace).
- "optional" – true/false indicates if property is optional or not. Instance of respective class may not possess optional properties.
- "value" – describes value of the instance.

Value field includes following properties:

- "type" – describes type of property value may be either “str”, “int”, “float” or “object”, indicating that related property should have value of corresponding type.
- "optional"– true/false indicates that value is optional (i.e. can have “null”) value or not.
- "collection" – can be “set”, “list” or null; indicates that the property points to collection of objects of the corresponding type. Null value in this field indicates that property possesses a single value. Properties possessing collection values (having collection “set” or “list”) will contain an aggregation value in the instance.
- "constrains" – set of constraints on property value (functionality of this field does not implement yet).
- "default" – default value of the property. Optional. If default is provided and the property’s value is null, then this default is used.
- "object" – either null (for simple types) or object specification (for object type).

Object specification consist of two fields:

- "namespace" – namespace to which the object value belongs.
- "class" – name of the class to which the object value belongs.

API to manage classes is shown in Figure 23.

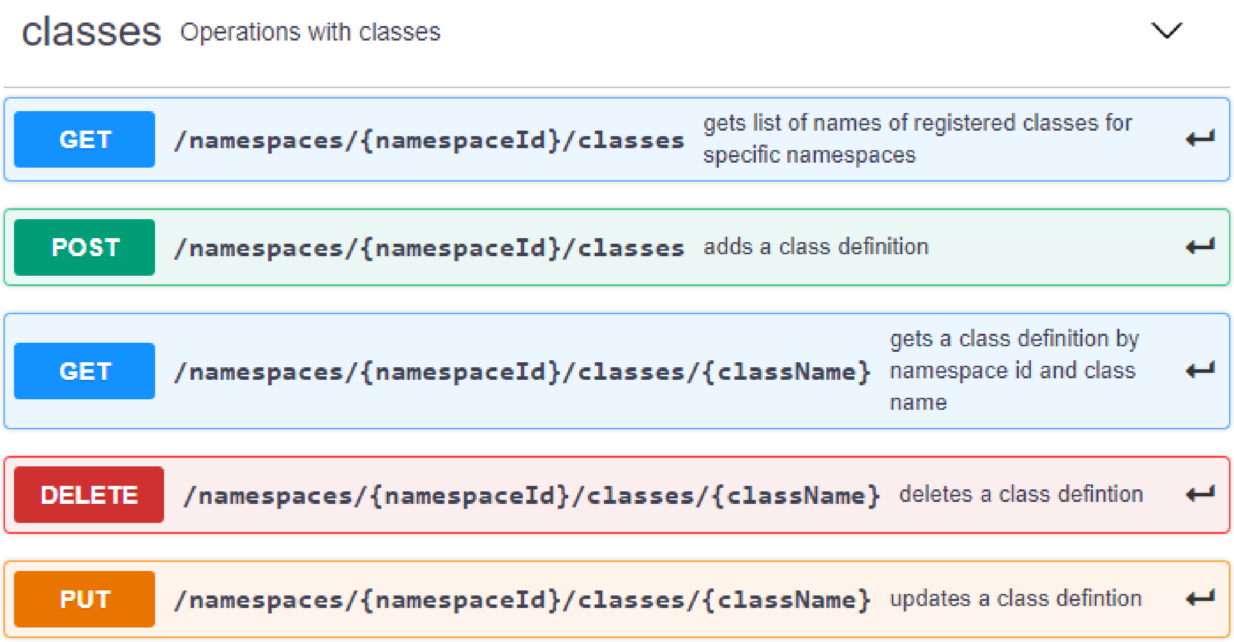


Figure 23: API for class definitions management

The API includes methods to add/update/delete class definition, to get class definition by namespace id and class name and to get list of all registered classes for specific namespace. After registering new namespace, the integration agent registers all classes sending POST /namespaces/{namespaceId}/classes request.

5.2.3 Data definition interface

Successful tool integration necessitates that system model data to be serialized or rendered into a preferably standard, format or syntax that can be parsed later and transformed into another format as per need of subsequent layers.

JSON representation of a set of *RDF* triples as a series of nested data structures has become increasingly popular as a data serialization format thanks to its more lightweight structure compared to XML, making it a useful format for data exchange in a way that requires less bandwidth than a bulky XML document. Thus, CPSoSaware choose JSON format as a primary format for data serialization. To enlarge compatibility with simulators that using XML as primary serialization format automatic XML-to-JSON and JSON-to-XML translators are provided.

However, JSON data format (especially after XML-to-JSON translation) are lacking several important features that are natural parts of underlining object models. For example, JSON format lacking possibility of referencing object that are defined in other parts of JSON document. To overcome difficulties introduced by the missing features we provide class definition extension for data. This extension provides following properties:

On the schema level:

- "name" - name of the schema. Since objects of the same class can be serialized with different data formats several different schemas can be provided to deserialize these objects. These schemas are distinguished by their names.
- "representation" – defines properties representation into JSON serialized object. Includes the following sub-properties:

  - "type" – can be one of "key_value_base" / "property_base" / "mixed". Describing data representation type: "key_value_base" is a native JSON representation, where JSON key describes property name and JSON value describes property value; "property_base" is a special representation often produced by XML-to-JSON converters, where each property defined by two JSON key-value pairs one having property name as its value and another having property value as its value; "mixed" is a representation where both types of representation are used.
  - "base_key" – parameter that defines JSON key which value defines the class of JSON serialized objects. This value will be parsed according to corresponding schema. All other JSON key-value pairs will be ignored.
  - "key_prefix" – parameter that defines prefix that should be stripped from JSON key on JSON-to-CIF conversion or added to JSON key on CIF-to-JSON conversion.
  - "key_value_base" – describes parameters specific for "key_value_base" representation of properties. Includes "base_key" and "key_prefix".
  - "property_base" – describes parameters specific for "property_base" representation of properties. Includes "base_key", "key_prefix" and two additional parameters:
    - "property_name_key" – describes which JSON key is used by parser to identify key-value pair that defines property name as its value.
    - "property_value_key" – describes which JSON key is used by parser to identify key-value pair that defines property value as its value.
- "keys" – define list of unique keys that allow to distinguish one object of corresponding class from another. Includes the following sub-properties:
  - "name" – name of the key.
  - "properties" – defines list of properties which form unique identifier of the object of corresponding class.

On the property level:

- "representation" – defines representation of the specific property if schema has "mixed" type of representation.
- "base_key" – same meaning as "base_key" in representation description but applied only to a specific property.
- "key_prefix" – same meaning as "key_prefix" in representation description but applied only to a specific property.

On the object level:

- "schema" – defines name of the schema of the nested JSON object. Object will be parsed according to corresponding schema.
- "extensible" – true/false. Defines if nested object can represent a new object of corresponding class (true), or only reference to existing object of corresponding class (false).
- "id_type" – defines how nested object are identified and can be either:
  - "object" – nested object itself provided according to corresponding schema,
  - "uid" – reference to the existing instance in the CIF database provided as UID of the CIF instance,
  - "key"–indicates that only several properties are provided and set of provided properties. It includes at least properties enlisted in the unique key provided in the "id_key" property value,
  - "key_property" – indicates that provided value of a property that uniquely identify the object. In this case "id_key" property refers to key that based on one property only.

- "id_key" – required if "id_type" property has value "key" or "key_property" defines a name of the unique key in the corresponding schema.

Schema management API shown on Figure 24

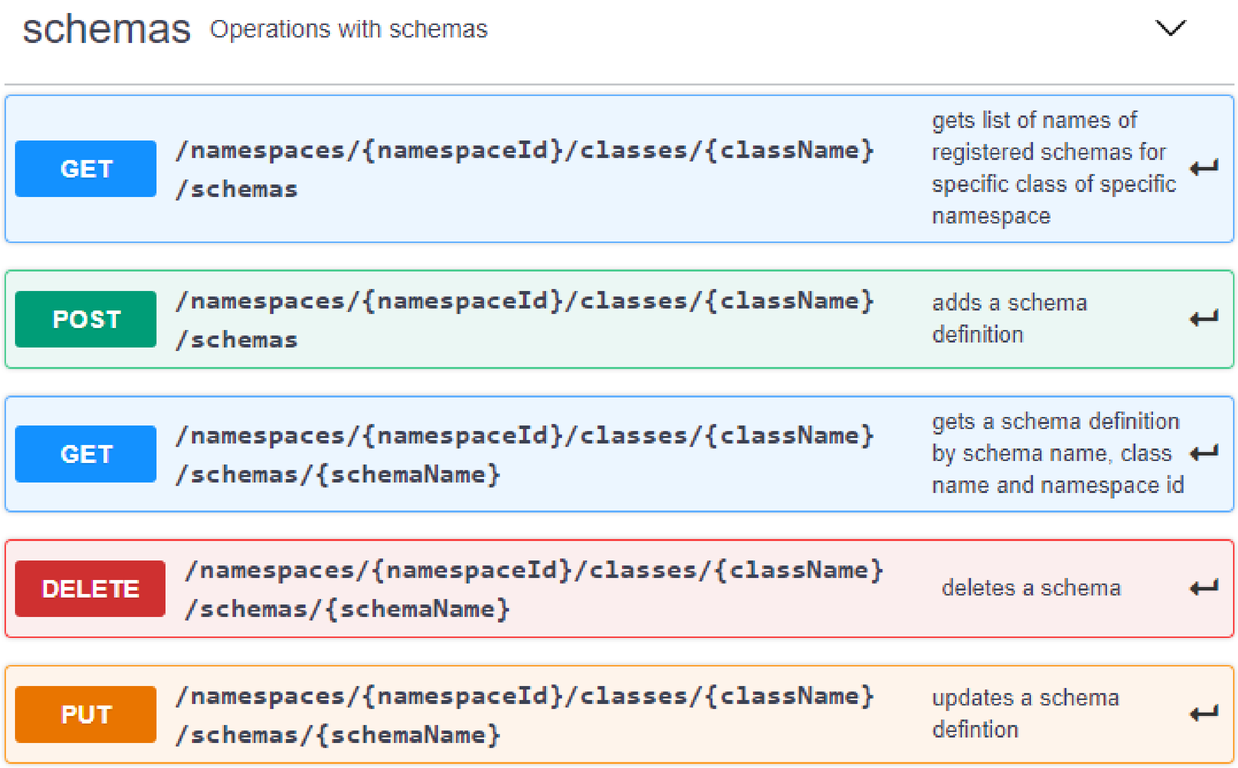


Figure 24: Schema management API

Schema management API introduces methods that are similar to the class management API methods. After class registration the integration agent should register data schemas for this class.

5.2.4 Data management interface

Data management API is shown in Figure 25.

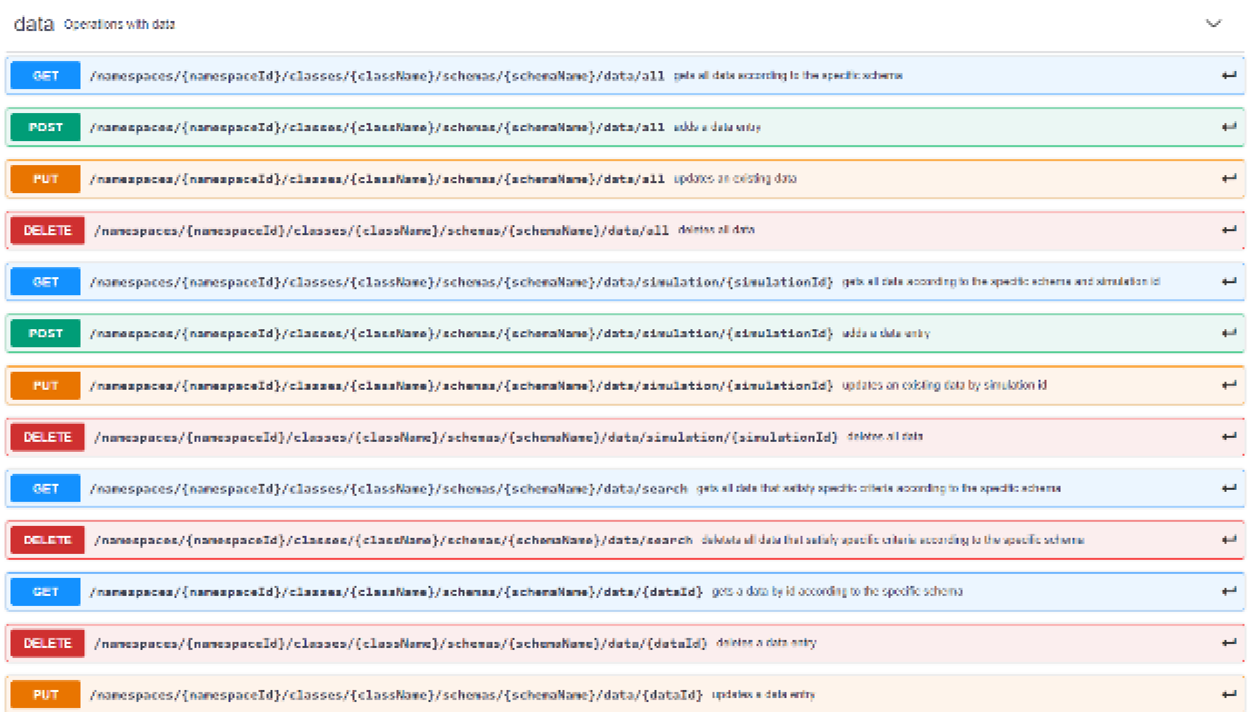


Figure 25: Data management API

The data management API provides several groups of methods to manage the data:

- Global methods allow to add/delete/update or get all data according to the specific namespace, class, and schema. These methods are applied by sending corresponding request to /namespaces/{namespaceId}/classes/{className}/schemas/{schemaName}/data/all URL.
- Integration agents use another group of methods that allows to add/delete/update or get all data according to the specific namespace, class, schema, and simulation id. These methods are applied by sending corresponding request to /namespaces/{namespaceId}/classes/{className}/schemas /{schemaName}/data/simulation/{simulationId} URL.
- Another two methods allow to get or delete all data according to the specific namespace, class, and schema that satisfy search criteria. These methods are applied by sending corresponding request to /namespaces/{namespaceId}/classes/{className}/schemas/{schemaName}/data/ search URL.
- Finally, there are methods that allows get, update or delete the data entry according to the specific namespace, class, and schema and having specific id.

5.2.5 Ontology Alignment and Equivalence Rules

Ontology alignment, or ontology matching, is the process of determining correspondences between concepts in ontologies. In the tool-integration context involving many tools providing their own ontologies, ontology matching has taken a critical place for helping heterogeneous tools to interoperate. Ontology alignment is provided by the data transformation service as set of the equivalence rules between objects of two or more classes. Equivalence rules allow automatic transformation of objects between different simulators working on different levels of abstraction. The set of the rules describing all equivalence relations between objects of all classes of two different namespaces represents a mapping between corresponding ontologies. Notwithstanding the different existing tools and languages for ontology alignment, CPSoSaware found that most of these tools and languages are not suitable for simplified ontologies for various semantic and syntactic reasons. The most promising language for the alignment of simplified ontologies was proposed in the CERBERO project, and CPSoSaware will continue develop and utilize this language. This choice supported by common principles that lies under CIF developed by CERBERO data storage and transformation services developing by CPSoSaware.

Data transformation service provide following syntax of equivalence rules.

- Main rule syntax:

ns1:class1 operator ns2:class2 [*…] [ON …] [IMPLYING …];

ns1, ns2 - names of namespaces

class1, class2 - names of classes

operator - one of: ===, <==, ==>, <==>

[*…] - optional multiplication part

[ON …] - optional "on" part

[IMPLYING …] - optional "implying" part

; - termination symbol

- Optional multiplication part syntax:

* ns3:class3 | int_expression [*…]

ns3 - name of namespace

class3 - name of class

|int_expression – any integer expression that can be provided instead of ns3:class3

[*…] - optional multiplication part

- Optional on part syntax:

ON (bool_expression [, bool_expression])

bool_expression – any bool expression

[, bool_expression] – optional additional comma-separated bool expressions

- Optional implication part syntax:

IMPLYING (implication [, implication])

- Implication syntax 1:

ns1:class1.property_expr1 operator ns2:class2.property_expr2 [*…] [ON …] [IMPLYING …]

ns1, ns2 - names of namespaces

class1, class2 - names of classes

operator - one of: ===, <==, ==>, <==>

property_expr1, property_expr2 - expressions defining (sub)properties names

[*…] - optional multiplication part

[ON …] - optional "on" part

[IMPLYING …] - optional "implying" part

- Implication syntax 2:

ns1:class1.property_expr1 = gen_expression

ns1 - name of the namespace

class1 - name of the class

property_expr1 - expression defining (sub)property name

gen_expression - general mathematical expression

Equivalence rule semantics.

Semantics of main rule operators (void multiplication part).

=== - means that corresponding classes are equivalent, i.e. each instance of class 1 is also instance of class 2 and vice versa.

==> - means that class 2 equivalent to class 1, i.e. each instance of class 2 is also instance of class 1, but instance of class 1 is equivalent to instance of class 2 only if both met matching criteria provided in "on" part.

<== - means that class 1 equivalent to class 2, i.e. each instance of class 1 is also instance of class 2, but instance of class 2 is equivalent to instance of class 1 only if both met matching criteria provided in "on" part.

<==> - means that instance of class 1 is equivalent to instance of class 2 only if both met matching criteria provided in "on" part.

Semantics of multiplication part.

Multiplication part change equivalence rules operator semantics in the following sense:

- when multiplication part contains class reference this means that class 1 equivalent to cartesian product of instances of class 2 and class 3, each instance of class 1 has two different instances (one of class 2 and one of class 3) as his counterpart with respect to corresponding relation operator.
- when multiplication part contains integer expression this means that each instance of class 1 corresponds to number of instances of class 2, and this number defined by integer expression that may depend on properties of corresponding instances.

Semantics of "On" part.

"On" part can include several logical expressions that treated as matching criteria between instances that are equivalent according to corresponding rule. These expressions can be treated as one single expression with "and" operator between corresponding parts.

Semantics of implications.

Implication of kind 1 (syntax 1) can be treated as nested equivalence rule and define equivalence relations between property values of instances of corresponding classes. Implication of kind 2 (syntax 2) define property value that should be assigned during rule execution process. This value is a result of calculation of general mathematical expression.

Data transformation service provides rule management API that allows add, delete, or get equivalence rules. API description represented on Figure 26

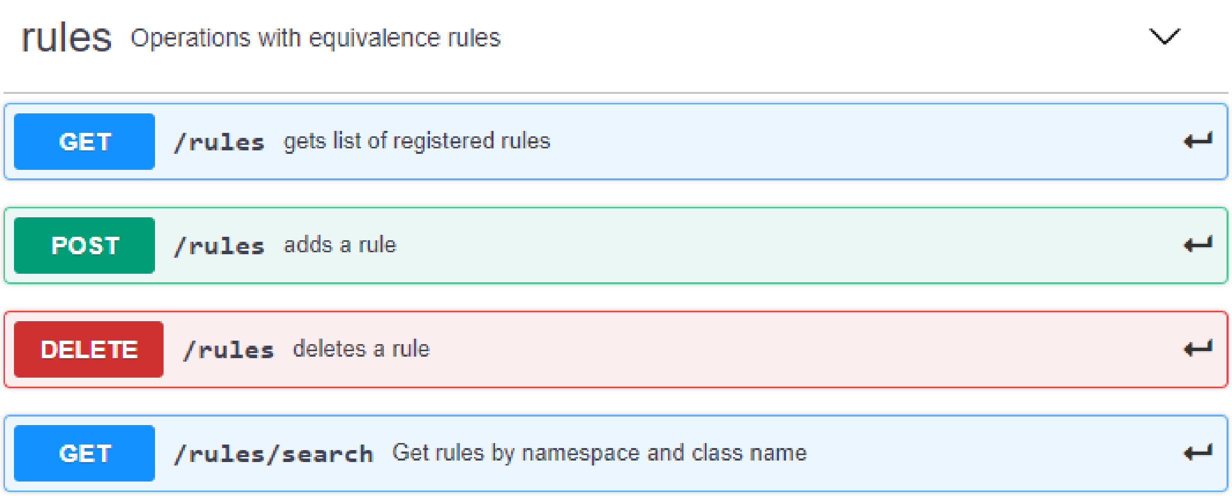


Figure 26: Rules management API

6 Conclusion

This document is the first version of the architecture for simulation and training (SAT) block. We have identified two key functions that should be performed by the SAT block: SAT block should be capable of: i) performing joint simulations across different and diverse simulators and ii) storing simulation data in a way that will allow queries to obtain consistent datasets that are further to be used to train ML algorithms.

The design of SAT block architecture starts with a review of the SAT block functionality and state-of-the-art approaches used to perform joint simulation, and storage DB types that can be utilized to store simulation data. We review co-simulation as a method for performing joint simulation based on the Functional Mock-up Interface (FMI), and several prominent tools that support it. However, we come to the conclusion that this methodology is too complex for our needs and propose to use data transformation to the CERBERO Interoperability Framework format (CIF) instead. Our comparison of persistent storage DB types suggests that a relational database would be the best choice for the simulation data. The design of the architecture of the simulation and training (SAT) block is driven by requirements to its functionality. We describe this architecture and how it relates to an Integration and Storage approach for integrating different and diverse simulators that use different modelling paradigms and languages, and how it supports the Simulation Workflow. We then lay out the SAT components, including how it will be implemented in Jenkins, the Orchestration tool; how the data will be stored, and how inter and intra-communication will be implemented. We also discuss the HW-SW simulators based on the OpenASIP simulator engine Proxim, and the use-case dedicated simulators based on CARLA and Robotec. We then describe the SAT interfaces including the orchestrator and data storage interfaces.

In summary, the proposed architecture allows the project to achieve objective O4.2 “Implement CPSoSaware Simulation and Training block that constitutes the basic testing and training data extraction environment for the design and redesign procedures performed in the MRE System Layer component.” The next version of the deliverable will be focused on the implementation and evaluation of the proposed architecture.