NAVIDRO, a CARES architectural style for configuring drone co-simulation

Loic Salmon, Pierre-Yves Pillain, Goulven Guillou, Jean-Philippe Babau

NAVIDRO, a CARES architectural style for configuring drone co-simulation

LOIC SALMON, ISEA, University of New Caledonia, Nouméa, New-Caledonia
PIERRE-YVES PILLAIN, GOULVEN GUILLOU, and JEAN-PHILIPPE BABAU, Lab-STICC, UBO, Brest, France

One primary objective of drone simulation is to evaluate diverse drone configurations and contexts aligned with specific user objectives. The initial challenge for simulator designers involves managing the heterogeneity of drone components, encompassing both software and hardware systems, as well as the drone's behavior. To facilitate the integration of these diverse models, the Functional Mock-Up Interface (FMI) for Co-Simulation proposes a generic data-oriented interface. However, an additional challenge lies in simplifying the configuration of co-simulation, necessitating an approach to guide the modeling of parametric features and operational conditions such as failures or environment changes.

The paper addresses this challenge by introducing CARES, a Model-Driven Engineering (MDE) and component-based approach for designing drone simulators, integrating the Functional Mock-Up Interface (FMI) for Co-Simulation. The proposed models incorporate concepts from Component-Based Software Engineering (CBSE) and FMI. The NAVIDRO architectural style is presented for designing and configuring drone co-simulation. CARES utilizes a code generator to produce structural glue code (Java or C++), facilitating the integration of FMI-based domain-specific code. The approach is evaluated through the development of a simulator for navigation functions in an Autonomous Underwater Vehicle (AUV), demonstrating its effectiveness in assessing various AUV configurations and contexts.

Additional Key Words and Phrases: component-based design, co-simulation, Model-Driven Engineering, Cyber-Physical System, drone

1 INTRODUCTION

In recent years, drones have been increasingly used for environmental observation purposes due to their numerous benefits in terms of mission planning and coverage rate. For instance, Autonomous Underwater Vehicles (AUVs) are becoming increasingly popular for hydrographic applications. These vehicles are capable of conducting underwater surveys and collecting data in areas that are difficult or dangerous for human divers to access. However, these applications face challenges concerning the estimation of the underwater position of the AUV due to GPS signal loss [29]. Given the uncertainties and costs involved in testing such a system, simulation has become a key issue, for instance to evaluate the accuracy of the AUV position computed by a given navigation function considering a certain AUV configuration and context [17].

Simulation may be performed at different levels of abstraction, such as model-in-the-loop, software-in-the-loop, hardware-in-the-loop, and processor-in-the-loop simulations, for different stages of development process. In this paper, we focus on model-in-the-loop and software-in-the-loop simulations, where the simulation is based on running software, modeling the behavior of parts of the AUV and its environment, emulating hardware functions, or implementing pieces of embedded software.

Authors' addresses: Loic Salmon, loic.salmon@unc.nc, ISEA, University of New Caledonia, Nouméa, New-Caledonia; Pierre-Yves Pillain, pierre-yves.pillain@univ-brest.fr; Goulven Guillou, goulven.guillou@univ-brest.fr; Jean-Philippe Babau, babau@univ-brest.fr, Lab-STICC, UBO, Brest, France.

ACM 1539-9087/2024/3-ART
https://doi.org/10.1145/3651889

Drone simulation faces four challenges [13], as a drone may be considered as a Cyber-Physical System (CPS). The first challenge concerns simulation performance. This aspect is mainly addressed by optimizing simulation modules and by using parallel infrastructure [1]. The second challenge concerns accuracy of results which relies on the quality of simulations modules. The third challenge concerns CPS modeling [13]:

- *Preventing misconnected model components.* Indeed, not only because of the type of the components, one issue is the unit and the reference frame of data that can be different from one component to another one. Standardization of reference frames and units of data is a key element to prevent errors.
- *Modeling sensor behaviors and time disparities.* The accuracy and the frequency may vary from sensor to sensor. Each sensor is disturbed differently by the environment, and has its own sensor noise model.
- *System heterogeneity.* Integration of heterogeneous domain specific topics requires to implement principles of separation of concerns related to modeling system and simulation [14]. Due to different data sources, different models and algorithms, different simulation configurations are to be considered.

The fourth challenges pertains to the configuration of simulation runs and proposing agility in the simulation design.

Compared to a monolithic simulation, co-simulation allows simulation modules, which represent subsystems, to be coupled together, providing a more comprehensive simulation of the overall system. This approach brings many advantages such as reduced development time [43]. Co-simulation is a technique used in simulation to integrate and orchestrate heterogeneous models while considering modeling and time disparities (third challenge). In the literature, a lot of experiments show the effectiveness of co-simulation for AUV evaluation (trajectory tracking [27], safety and performance analysis [44], UAV swarms analysis [4] and communications [2]). One widely used co-simulation standard is the Functional Mock-Up Interface (FMI) [6]. FMI is a standardized interface for model exchange and co-simulation of dynamic models across different simulation tools. The FMI standard for Co-Simulation defines a standardized interface for coupling simulation units (called Functional Mock-Up Unit (FMU)) in a co-simulation environment. FMU are orchestrated by the Master module (orchestrating data exchange and scheduling).

While the FMI standard provides facilities to integrate and orchestrate different simulation modules, it does not provide any guidelines on how to design co-simulation (how to connect FMUs through a co-simulation graph) and to configure it (the fourth challenge). Indeed, the key aspects ”preventing misconnected model components” and "flexible co-simulation design" are software architectural challenges: what architectural style is required to avoid standardization errors and to ease configuring co-simulation description and runs. And, considering end-users, the architecture style has to be supported by an easy-to-use environment to design and run configurable simulators. In such a context, complexity of CPS encourages developers to implement co-simulation by using modeling approaches [13] and more specifically a Component-Based Software Engineering (CBSE) approach [12].

Recently, high-level modeling languages and standards have been proposed to ease the description of simulator structure and runs [3, 10, 11, 21, 25, 34, 42]. But to the best of our knowledge, there is no approach proposing a high-level modeling language integrating richness of CBSE and parametrization to build a flexible simulation environment for designing and configuring a dedicated drone simulator regarding different contexts.

In this paper, we propose first a model-based framework, called CARES [1], dedicated to the design of drone co-simulation. The framework provides three high-level modeling languages (based on concepts defined by CBSE and FMI 2.0 for co-simulation) to consider three modeling layers (definition of types of co-simulation unit, graph of co-simulation units and description of co-simulation run). From models, a code generator and a runtime library allow to build a simulator for a given scenario. Then, to ease design of co-simulation graph, we propose an architectural style called NAVIDRO. NAVIDRO provides patterns to ease configurable co-simulation development for the specific domain of AUV trajectory tracking. It proposes to implement generic API and adapters to ease

[1] https://framagit.org/jpbabau/cares

integration of different drone behaviors, sensor configurations and navigation functions. NAVIDRO is illustrated through a simulator dedicated to evaluation of AUV navigation functions. At last, the paper presents how to design and configure co-simulation runs for different configurations and contexts.

The paper addresses software engineering challenges to ease design of flexible co-simulation. CARES languages offer flexibility at different levels: a parameter of a co-simulation unit may be modified when defining the co-simulation graph or when describing a co-simulation run (at the beginning or at specific instants), a link between components may be setup when describing a co-simulation run, a co-simulation unit may be stopped or restarted at specific instants. And the NAVIDRO architectural style provides guidelines on how to connect different co-simulation units to ease design of flexible drone co-simulation for a specific family of applications. CARES and NAVIDRO experimental results have been presented in a first paper [38]. This paper details CARES and NAVIDRO principles. All concepts are presented and integration of FMU is detailed. In addition, the notion of scenario is described and illustrated using several configurations.

The remainder of the paper is structured as follows. Section 2 presents the application domain of AUV and more specifically the navigation function. Section 3 introduces the generic aspects of the model-driven and component-based framework, called CARES. Section 4 presents the NAVIDRO architectural style proposed to design a drone simulator. In Section 5, the co-simulation results of the case study are discussed. Section 6 presents related work before a conclusion in Section 7.

2 CASE STUDY : THE AUV NAVIGATION FUNCTION

We present here the case study used to illustrate CARES and NAVIDRO. An AUV is used to observe and collect data in a specific area. To fulfill its mission, it requires an evaluation of its position. This is the role of the navigation function. This section describes the different concepts and elements required to simulate the navigation function of an AUV.

The first element to consider is the trajectory of the AUV. This trajectory is the reference to evaluate the accuracy of the navigation function. For simulation, the trajectory can be extracted from a given log file (replay of a previous mission) or computed from a trajectory model. For the latter, the trajectory is usually specified by a list of way-points. Indeed, the trajectory cannot follow exactly the succession of way-points through segment lines due to physical constraints: the drone cannot turn directly and the motor control implies some uncertainties in the trajectory. To simulate a realistic trajectory, different models can be used as for instance splines [39]. For testing purpose, a simple trajectory may be computed as a list of consecutive segments and semi-circles (cf. Figure 1).

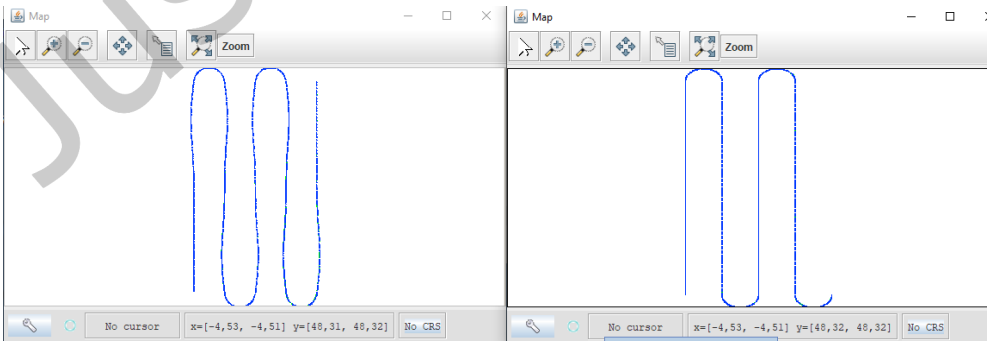

Fig. 1. Two different simulated trajectories built at the right with lines and semi-circles and at the left with splines.

The trajectory is impacted by the environment and meteorological conditions. In our case, the AUV evolves in an environment where current and depth play an important role, disturbing the intended trajectory of the AUV. To maintain its direction, reacting to the current, the AUV moves using crab steering mode. Moreover, the AUV may have to respect a given altitude (which is always calculated from the seabed). Then, to simulate a realistic trajectory, current and depth are a part of the model for drone navigation estimation. The environment behavior may be extracted from a log file, given by a forecast file or computed from an environment model.

For sensors, the following Figure 2 illustrates the different elements that can be present in an AUV. Indeed, each type of AUV is equipped with different sensors. In our case these sensors are:

- the GPS provides an estimation of the position of the drone while it is on the surface.
- the accelerometer is a part of Inertial Measurement Unit (IMU) and the linear accelerations.
- the gyroscope is a part of IMU and provides an estimation of the angular speeds.
- the DVL (Doppler Velocity Log) provides an estimation of the linear speeds.
- the barometer provides an estimation of the depth.

Fig. 2. The Kongsberg Munin 1500 AUV and its components.
2

Each sensor is characterized by a level of quality (accuracy) and runs at a given frequency (in Hertz). The technical specification of a sensor characterizes its quality through a model of error (white noise, bias, scale factor, time drift, ...) [49]. Moreover, each data provided by the sensors are given in the frame and orientation related to itself.

Finally, different navigation functions can be used. Some of them use only a few sensors data (GPS on the surface, gyroscope and DVL integration, gyroscope and accelerometer double integration) [30], while others use hybrid process merging all data received by the different sources (i.e. Kalman Filter). A Kalman Filter needs parameters' tuning to provide a correct estimation of the AUV position [5].

In conclusion, the simulation of the navigation function for a drone is highly flexible depending on the chosen AUV trajectory, the environmental conditions, the quality of the sensors and the chosen algorithm. To test a simulation configuration, one can deal with real data logs or models of the different elements (environment, trajectory, sensors, ...), different values for parameters of models. Moreover during a mission, at any time, different events may occur, like a sensor failure or a quality loss.

3 CARES FRAMEWORK

This section presents the CARES[3] framework we propose for co-simulation. CARES provides model editors and tools to develop and run model-driven and component-based simulators.

The model-driven framework CARES proposes to prevent inherent problems of code-centric approaches by describing a simulator through a set of communicating components, each component simulating one part of the drone and its environment. At execution stage, CARES follows a time-driven orchestration paradigm and a co-simulation run is based on a high-level description of a parametric scenario.

3.1 Main concepts

In the following section, we present the main concepts of CARES.

Structure. As usual with component-based approaches, a component is either a leaf component or a composite component. A leaf component executes repeatedly a specific task, while a composite component can be composed of one or several leaf or composite components. The composite components are proposed for structural purpose. They ease navigation in the system description. From FMI perspective, a leaf component represents a Functional Mock-up Unit (FMU). It is a co-simulation unit in charge of simulating one part of the system. From implementation point of view, it is a FMU wrapper respecting the FMI interface.

Parameters. As usual with component-based approaches, a leaf component is characterized by a set of typed parameters (basic types, records and arrays). Considering the needs of FMI for co-simulation, parameters can be shared between components (the *boolean shared* attribute of the shared parameter is by default set to *true*).

Communication. As usual with co-simulation, communication between components is based on data links connected to input and output data ports. To ease exchange of complex information between components or management of components, we propose to add functional services to components. Following classical component-based models, services are declared using required and provided interfaces of components. Each interface defines a set of functions. Service links are represented by relying provided interfaces to required interfaces (many to many relationship). At design stage, a provided interface is not necessarily linked to a required interface, it can be used later to configure co-simulation runs (see later).

Data ports and interfaces may be declared for a leaf or a composite component. In case of a leaf component, an input data, resp. an output data, can be used, resp. produced, by the behavior of the component. And a provided, resp. a required, service is implemented, resp. can be called, by the behavior of the component. In case of a composite component, with no specific behavior, a port or an interface is simply linked to a port or an interface of an internal component.

Component and communication behaviors. An explicit behavior is described only for leaf components. First, a leaf component provides an implementation for each function defined by each provided interface. And, to be FMI compatible, each leaf component has to implement five functions : *initialize()*, *end()*, *doStep()*, *start()* and *stop()*. The functions have to be implemented in Java or C++ language, depending on the target language of the simulator.

The *initialize()* function is used to initialize the parameters of the components and internal services. The *end()* function is used to stop internal services. The *doStep()* function is in charge of computing parameters and outputs considering inputs. Its execution follows the Run-To-Completion paradigm: the data inputs are considered at the beginning and the data outputs are produced at the end [15].

To implement different modes, when describing co-simulation runs, each component may be activated (by default at initialization or after a call of *start()*) or deactivated (after a call of *stop()*) at a given time. After a leaf component becomes inactive, the corresponding *doStep()* function is not executed.

[3]https://framagit.org/jpbabau/cares

For data communication, two states of shared parameters and data are managed by the framework: the current value (timestamp t2) and the previous one (timestamp t1). If a parameter belongs to a component C1 or a data is produced as an output by a component C1, it is updated after each *doStep()* execution of C1 (t2 = t1 + period of C1). If a parameter or data is used by a component C2, when a *doStep()* execution is performed, simulating evolution of C2 from instant t3 to instant (t3 + period of C2), C2 may consider the last produced information (instant t2) or the previous one (instant t1). Thus, for the modeling of a continuous system, a component C2 only considers data produced before or at time t3.

It is the responsibility of the FMU designer to consider either data computed at the previous instant (a data dependency is not a timed dependency for continuous processes) or after a co-simulation step (a data dependency is also a timed dependency to implement data-flow semantic). Because the algorithm of data communication orchestration is generic, it is not described in the model. We just represent data links that rely output ports to input ports (one to many relationship).

Component types. A composite component does not relate to the definition of a component type. It has just a structuring role and then is defined when designing a given simulator. Rather, a leaf component is linked to the definition of a component type. The reason is that many leaf components may have the same structure (parameters, communication ports and interfaces) and behavior. Moreover, instantiating using types allows to share behavior whereas instantiating using copy/paste requires renaming components and may lead to a behavior redefinition using an error-prone copy/paste operation.

Master algorithm. Inspired by [22], CARES implements a centralized Master Algorithm (MA). The MA periodically launches the function (*doStep(timeInterval)*) for each active leaf component. The period is based on the frequency expressed in hertz for each leaf component. The value of *timeInterval* corresponds to the time used by the real system (simulated by the component) to produce data. The component outputs are updated after this delay. New data is then propagated to all components considering that data as input. The communication mechanism (see above) allows each component to consider the new computed value or the previous one.

For component scheduling, the execution order is based on the partial order defined by the I/O dependency graph. If a cycle exists, a priority associated to each leaf component is used to define the execution order between components. The priority is user-defined, with a value of 0 representing the highest priority level.

Scenario. The scenario modeling step corresponds to a timed sequence diagram modeling process. A scenario defines:

- start and end time for co-simulation;
- the time step size;
- at the beginning of co-simulation
  - a sequence of parameter initialization : each component may be configured for each co-simulation run.
  - binding of required and provided interfaces (if necessary): if at design stage a required interface is linked to many provided interfaces, when defining a co-simulation run, a required interface has to be linked to one and only one provided interface. This step is useful to configure co-simulation by selecting which component is used to provide a given service.
  - a list of components to stop: a unused component (for instance a component providing a unused interface) can be stopped to improve the performance of the simulator.
  - a list of provided function calls: it is possible to perform complex configuration of a component by calling dedicated provided functions.
- a sequence of timed events and for each timed event
  - a sequence of parameter initialization and a list of provided function calls: a component may change its configuration during a co-simulation run.

  - a list of components to stop or to restart: it is possible to simulate a transient component error by stopping it at a certain instant and restarting it some time after.
- at the end of co-simulation: a sequence of function calls to reinitialize components;
- when components contain random data, a scenario may be played many times.

A scenario definition provides a high-level description of a given co-simulation run. Using language facilities, it is then easy to adapt co-simulation to test a given configuration for a given context (parameter valuation, function call, component binding, component stop and start).

3.2 Modeling

CARES concepts are implemented through three metamodels:

- *ComponentType metamodel*: this metamodel contains the required elements to describe data types, functional interfaces and types of leaf components (list of required and provided interfaces, data inputs and outputs, parameters). Instances of types are described through a textual editor based on Xtext[4]. A textual description seems to be adequate to model types of data or components as in programming languages.
  An example of declaration of interfaces and a leaf component type is given in Figure 3. *CurrentSource* declares a generic interface for different sources of current (after setting time and position, if ok, current data is accessible), while *CurrentSource* declares a specific interface to configure a *CurrentModel* component. This leaf component type declares the interfaces as provided and parameters to implement a sinusoidal model of current.
- *ComponentSystem metamodel*: this metamodel contains the required elements to describe a graph of components (leaf and composite components, data and interface links). Figure 4 exhibits an excerpt of this metamodel. A graphical editor based on Sirius[5] allows to instantiate a model of components and links between them. A graphical description of a simulator is suitable to navigate in the structure of the simulator (see 4.5).
- *Scenario metamodel.* This metamodel contains the required elements to describe a co-simulation scenario. A scenario contains initialization steps and timed events (component initialization, parameter modification, function call). A scenario is described through a declarative language, based on a Xtext textual editor. A textual description seems to be suitable to describe a co-simulation run (see 4.3) as in programming languages.

The three metamodels are implemented using Ecore[6].

3.3 Code generation

From instantiated models, a code generator based on Acceleo[7] generates Java or C++ code by analyzing the whole model (checking typing constraints). The generated code contains all class declarations, the glue code for all the components and a main function is implemented for each modeled scenario. Data communication is directly handled by the components by analysing data links. Interface binding is dynamically defined in the main function considering both models of structure and scenario. A class is generated to declare all the timed events and is used by the main function to drive co-simulation. Co-simulation execution is based on the CARES runtime library. The runtime library implements execution facilities and an orchestration module (MA) to run scenarios. In particular, the CARES scheduler launches each active leaf component by executing the *doStep()* function considering its frequency. To run co-simulation, all that is missing is domain-specific code.

[4]https://www.eclipse.org/Xtext/
[5]https://www.eclipse.org/sirius/
[6]https://www.eclipse.org/ecoretools/
[7]https://www.eclipse.org/acceleo/

navidro.scn | new Component System Diagram | env.dcl | Component System Diagram | Component System Dia

```
Declaration CurrentTypes {
    interfaces{
        Interface CurrentSource {
            // interface for generic source of current data
            // set relative time in ms and geo position (latitude, longitude)
            "core.bool" setTime("core.long" time, "core.Coordinate2D" position);
            // get U,V components in m s^-1
            "core.Coordinate2D" getCurrent();
            }
        Interface CurrentConfig {
            // configuration interface for a sinusoidal current model
            // set average speed in ms-1 and direction in degree
            "core.void" setSpeed("core.double" speed, "core.double" direction);
            // set amplitude, period and offset in m, s, s
            "core.void" setSignal ("core.double" amplitude, "core.double" period, "core.double" offset);
            }
    }

    leafComponentTypes{

        LeafComponentType CurrentModel {
            // sinusoidal model for current
            // provide interfaces as a configurable current source
            providedInterfaces ( CurrentConfig pConfigCurrentModel, CurrentSource pSinModelSource)
            parameters {
                "core.double" nominalSpeed;  // average speed in ms-1
                "core.double" direction; // direction in degree
                "core.double" amplitude;  // signal amplitude in m
                "core.double" period; // signal period in s
                "core.double" offset; // offset s
            }
        }
    }
}
```

Fig. 3. Interfaces and leaf component type for current.

Integration of domain specific code is required for provided interfaces and the three *initialize()*, *doStep()* and *end()* functions. For each leaf component, a corresponding Java or C++ class is generated declaring all provided functions and inheriting from a *LeafComponent* (generic classes defining *initialize()*, *doStep()* and *end()* functions). Then, the domain-specific code may be added directly or integrated by implementing the *delegator* pattern [18] in the generated class. Once the domain specific code is added, the simulator is ready to run.

3.4 Discussion

Design choices for CARES are proposed to provide facilities to the simulator designer:

- types are only defined for Leaf components;
- high-level textual and graphical languages ease simulator description;
- components may be configured at design and run time through modification of parameters, service calls, start and stop operations;
- execution semantic is directly implemented by the runtime library and then is transparent for the simulator designer.

The question is then "how to structure and configure a given simulator for a given drone with a given simulation objective?". For instance, considering the case study, we want to develop a simulator to evaluate accuracy of the navigation function for different configurations of an AUV, considering different contexts. Configurations are

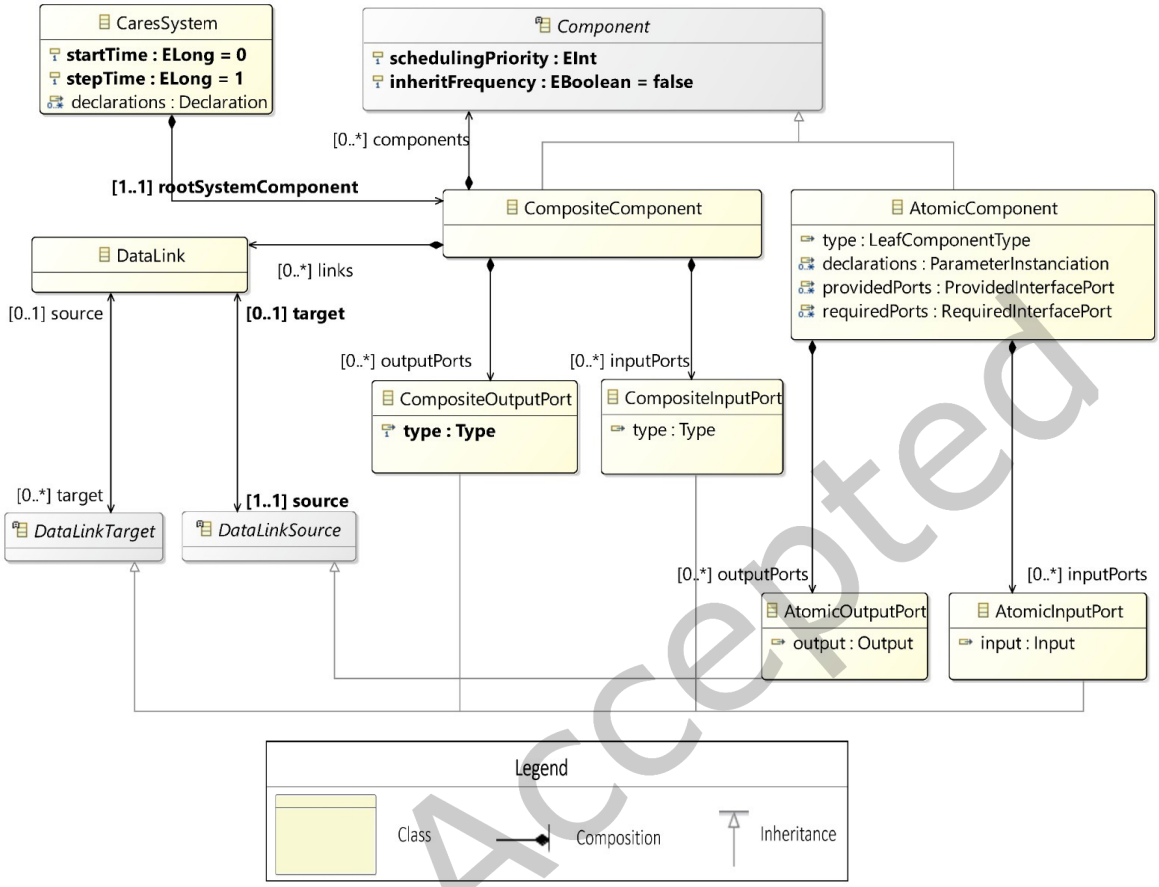

Fig. 4. CARES component System Metamodel.

here related to sensors quality and the choice of a given algorithm of navigation function. Context is related to environment conditions.

4 NAVIDRO ARCHITECTURAL STYLE

In the following part, model and architectural elements related to co-simulation of drone evolution are presented according to the CARES framework described previously. We here outline the main concepts involved in the NAVIDRO architectural style, illustrated by applying them to build a simulator of an AUV navigation function.

4.1 Application domain

While CARES is a generic model-driven framework for co-simulation, NAVIDRO is tailored to the specific domain of co-simulatiion to evaluate AUV architectures with respect to trajectory tracking applications. Within these applications, a drone navigates a given course while classical embedded sensors, such as the Global Positioning System (GPS), Inertial Measurement Unit (IMU), Doppler Velocity Log (DVL), compass, and barometer, generate

valuable data pertaining to drone behavior. This data is subsequently processed to assess the drone position and attitude.

NAVIDRO exhibits the capacity to evaluate the behavior of aerial, surface, or underwater drones. However, by using a Real-Time Kernel GPS, the position accuracy is not considered a critical concern for aerial and surface drones. As a result, the framework is more suitable to underwater drone applications where the sensors quality directly influences the analysis of drone trajectory.

Compared to conventional tools (see section 6), NAVIDRO emphasizes the adoption of high-level modeling techniques, thereby easing the incorporation of various requirements:

- First of all, with NAVIDRO, the specification of trajectories is performed through either a mission (defined by a sequence of successive waypoints to reach; each waypoint is described by its position (longitude, latitude, and altitude) and desired speed) or via an array of timed positions, such as those obtained from a log file. For the first method, the trajectory timed positions are computed using spline techniques, and for the second, the given positions and times are used directly to replay a given trajectory.
- Secondly, NAVIDRO provides facilities for integration of heterogeneous sensors considering both unit and referential and configuration of sensor errors to enable evaluations of specific technological design choices in drone architecture;
- Thirdly, NAVIDRO provides facilities for characterizing diverse physical conditions of the environment, such as current and seabed properties.
- Finally, NAVIDRO simplifies the process of defining scenarios for a specific usage of a drone within a particular environment.

4.2 Main principles of a CARES architecture style definition

For a specific application domain, co-simulation descriptions often share common concepts, and to ease the design of new co-simulation, we suggest developing an architectural style. An architectural style provides a set of guidelines on how to define and connect components. A CARES architecture style definition follows both top-down and bottom-up approaches.

4.2.1 Top-down. As usual with CPS control applications, co-simulations are based into a set of classical interacting subsystems. These subsystems include the *Controlled element* or plant (the system being controlled or supervised) , its *Environment* (external phenomena that affect the behavior of the controlled system), *Sensors* (elements that provide information on the controlled system or its environment), *Actuators* (elements that act on the controlled system) and a *Controller* (elements that implement a control or supervision policy). These elements play a structural role in a co-simulation configuration and are implemented through CARES composite components. Additionally, we introduce a *User* component (modeling user behavior) and an analysis component (which does not model an element of the system to simulate but is dedicated to computations used for simulation interpretation purposes). The definition of a CARES architectural style is achieved by selecting the required components and specializing them for the target domain:

- naming components
- standardizing inputs and outputs : name and unities

This step establishes the overall structure of the architecture style.

4.2.2 Bottom-up. In the context of the targeted application domain, each FMU or simulation module is transformed into a CARES leaf component. For example, a model of sensor behavior (simulating errors) or a sensor emulation (replaying a given mission) becomes a CARES leaf component. In this case, the inputs, outputs, and parameters of the FMU are transformed into the inputs, outputs, and parameters of the corresponding CARES

leaf component. Next, glue components are added to enable component integration and simulation configuration, by implementing:

- data type adaptation
- simulation component selection policy to add flexibility

The architecture style defines the way in which glue components and domain-specific leaf components are integrated through the use of patterns that can be employed across various applications.

4.3 NAVIDRO global architecture

This subsection describes the main components of the global architecture of NAVIDRO (see Figure 5).

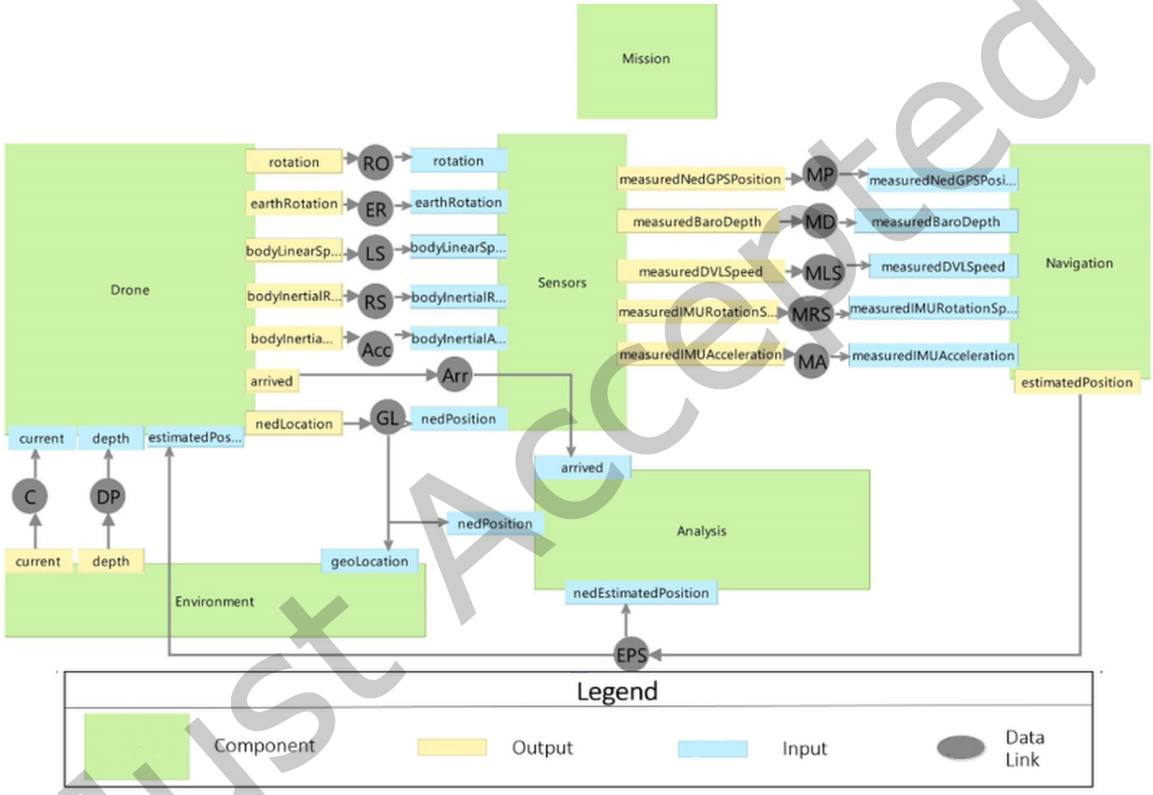

Fig. 5. Global architecture of the AUV simulator following the NAVIDRO architectural style.

The *Environment* component is used to model and represent the environment in which the drone moves. The environment of the drone is a composite built with components modeling each aspect of the environment. For the AUV, these components provide sea current and level of the seabed at a given position at a certain time. In most of approaches, environment description is implicit or directly implemented in the system description. We prefer here to model a distinct component to ease environment configuration (context of the drone) and source selection. Source selection consists in selecting a certain component emulating the environment behavior or a component replaying conditions encountered in a mission already executed. The latter is generally based on extracting data from a log file.

A *User* component represents the user requirements for a given mission. It may also be used to configure co-simulation, but we prefer to use scenario modeling for such activity. For the AUV, this component is represented by the *Mission* component. It corresponds to a list of way-points that the trajectory of the drone has to follow.

The *Controlled element* is here a *Drone* component responsible for generating the drone behavior, a trajectory of an AUV for the case study. This component is based either on a behavioral model or a replay of a mission (based on a log reader). Behavioral model may be a very complex process and depends on the targeted system and simulation objectives. For the AUV, we simulate the trajectory either by lines and arcs or more realistically by splines.

The *Sensors* component contains all the components simulating each sensor providing information on the drone or its environment behavior. Data produced by a sensor are computed considering a given error model or extracted from a log file. Moreover, with the facilities provided by CARES, it is easy to add a delay on the sensor production to simulate timing behavior. For AUV, we have a gyroscope, an accelerometer, a barometer and a GPS. Depending on the sensor, classical errors like bias, derive, scale factors and white noise are considered. Implementation of errors is based on a generic error library.

The *Actuators* component contains all the components simulating each actuator acting on the drone behavior. Like for sensors, a delay may be added to simulate timing behavior. For AUV trajectory tracking, because behavior is a pre-computed trajectory and due to targeted objectives (position computation based on sensor production), actuators are not considered for NAVIDRO.

A domain-specific *Controller* component represents the embedded control part. For AUV, the *Navigation* component is responsible for simulating the navigation function used to estimate the position of the drone. Different navigation functions can be plugged to assess the trajectory of the drone from the initial point with information provided by different sensors. For instance, a basic navigation function integrating the speed (*measuredDVLSpeed* provided by the DV component) and considering the direction by integrating the rotation speed (*measuredIMURotationSpeed* provided by the gyroscope component) has been implemented. A Kalman Filter has been also written in C++.

The last component is not an existing component of the system to simulate. It implements a tool to evaluate results of co-simulation. For AUV, the *Analysis* component computes the position uncertainty by comparing position given by the *Drone* component with the estimated position computed by the *Navigation* component.

Data or service communication links between components follows some connection rules:

- the environment component provides data to the drone (the behavior of the drone depends on environment conditions) and to sensor components (some environment behaviors may be observed);
- the drone component provides data to the sensor components (some drone behaviors may be observed);
- the user component provides information to the drone component (the drone behavior depends on the user requirements).
- the sensor components provide data to the control part (the programmable part depends on sensors data); in case of replay (simulation is based on sensor log files), the drone and environment components, that provide data to sensor components, are unused and can be stopped;
- the actuator components receive data from the control component (the control part acts on the drone by using actuators) and provide data to the drone component;
- all components may provide data to the analysis component.

The underlying principles of NAVIDRO global architecture is the classical control loop : (process and its environment) -> sensors -> process control -> actuators -> process.

As mentioned earlier in the introduction, to ensure that all components are connected correctly, it is important to standardize both units and referential frames. Therefore, when defining the overall architecture of the system, a list of referential frames and units must be established. For an AUV, the speed of both the AUV and the current is

measured in meters per second (ms^{-1}), and the AUV's position is given according to the NED (North/East/Down) frame.

4.4 Configuration and heterogeneity integration

When designing simulators, different issues have been identified such as adaptation of heterogeneous frames and units, specific component selection for a given co-simulation and integration of domain specific code in any language.

4.4.1 Data Source Selection. For a given co-simulation, the user has to select sources of data in a transparent way. The data source selection from logs, models or forecasts is necessary for the following components of the global architecture: *Environment*, *Sensors*, *Drone*. To configure data source selection, several approaches exist:

- The first idea is to link at design stage the output of the chosen data source component to the corresponding input.
- The second idea is to plug all possible sources to a *switch* component. The number of inputs is then the number of possible sources. Source selection is done when modeling scenario by setting a *select* parameter (one value by source).
- The third approach (such as in the Model System Logic (MSL) [51]) proposes to switch the components during the execution according to a mode defined by the user.

CARES offers facilities to implement the three possibilities. The first solution is simple and efficient. This solution implies that the user has to modify the design model for each source configuration. However, this solution may not be very user-friendly for individuals who simply want to configure a simulator run without having extensive knowledge of the underlying structure of the simulator. The second solution requires only to change the value of the *select* parameter of the *switch* component for each new input to consider. The third solution is simple at design stage but requires source selection during scenario modeling. The first and third solutions require that all sources respect the same frame and unit standard. They are simpler to implement and to manage by the CARES designer. And the second one has the advantage of considering heterogeneous data inputs (adaptation performed by the *switch* component). All the three solutions imply that all data source components are embedded in the simulator. But with CARES, in case of heavy simulation, it is possible, when modeling the scenario, to deactivate the unused components.

We consider now patterns for each component of the global architecture.

4.4.2 Sensor component. For sensor behavior description, we propose the following pattern:

- because data provided by a model of sensor is given in the frame and orientation related to itself, to adapt to the defined standard by the global architecture, we propose to provide a library of adapter components that make conversion and referential changes;
- a component is then in charge of error modeling by defining a set of generic error parameters (bias, scale factor and white noise for any dimension) to provide realistic sensor data;
- a log reader component is in charge of considering log files for replaying missions; frame and unit adaptations are also necessary according to the log file format and is specific to each reader component;
- a component is used to select a component source to consider during co-simulation.

Figure 6 illustrates the use of such pattern. The picture corresponds to the composite component DVL, a part of the *Sensor* composite component. This component is responsible to produce DVL data. Data may be either simulated by a model error (*myErrorDVL* component) or real data (*myReaderDVL* component). The latter produces data by reading a log file, whereas the first one simulates classical measurement errors. For this component, adaptations are necessary for inputs and outputs. Firstly, it is necessary to adapt data provided by the *Drone* component to the body frame of the sensor since data is computed in the body frame of the drone. This is the role

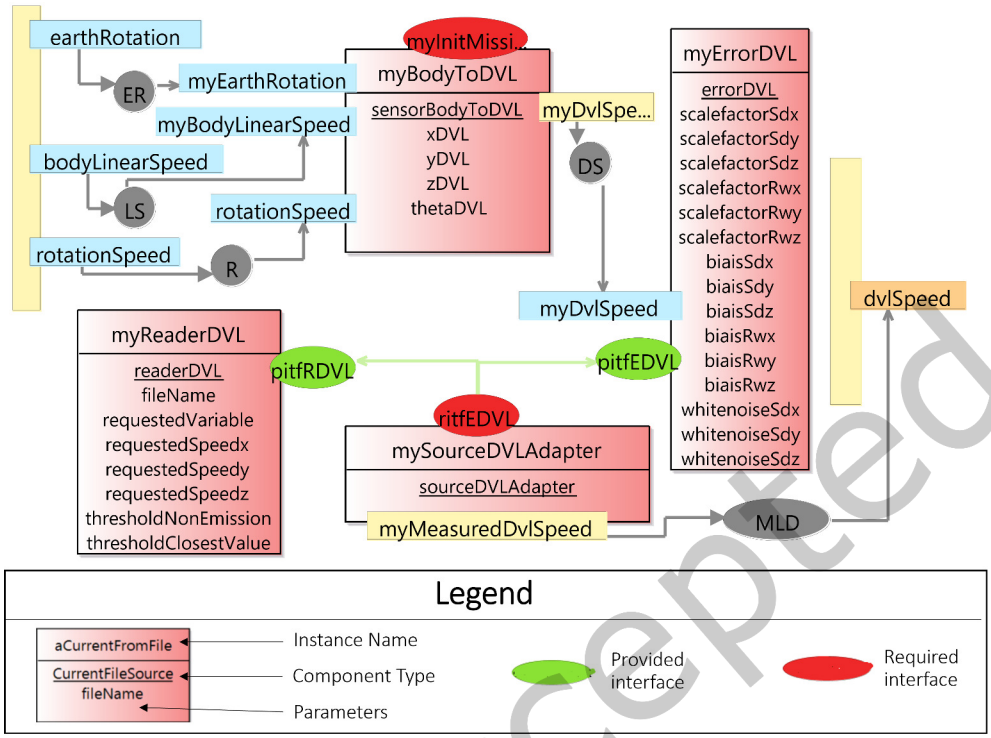

Fig. 6. DVL composite component.

of the *myBodyToDVL* component. Furthermore because the frame and units of log data are based on technical sensor choice, required adaptation is performed by *myReaderDVL*.

In the example, as illustrated in Figure 6, the component *mySourceDVLAdapter* implements the third approach, the data source selection is not set at design stage but when defining the scenario (see later).

DVL is an example of the pattern used for each sensor: local adaptation of the frame, error modeling, standardized log reader and source selection. Indeed, several reference frame Earth-Centered, Earth-Fixed), NED, body frame of the drone or of the sensors) adaptations are proposed by the framework. The proposed pattern allows then to easily integrate heterogeneous sources of data.

4.4.3 Drone and environment components. As with sensors, a drone or environment component is a choice of a simulation model, except that errors are not considered here.

For the AUV, we use a switch component *myMissionAdapter* as depicted by the Figure 7. We consider here the second approach where the switch considers both data sources. The boolean parameter *isSpline* is set to configure the data source. In this figure, the *MissionInterpreter* component considers splines while the *SimpleMission* component considers lines and semi circles (see results on Figure 1). The latter is useful for testing purposes while the spline allows to model a more realistic trajectory.

4.4.4 Control component. First, sensor data must be converted in the body frame of the drone to be processed. For the AUV, this is the role of the *myDVL* component (see Figure 8 of the composite *Navigation* component) modeling the DVL driver of the system.

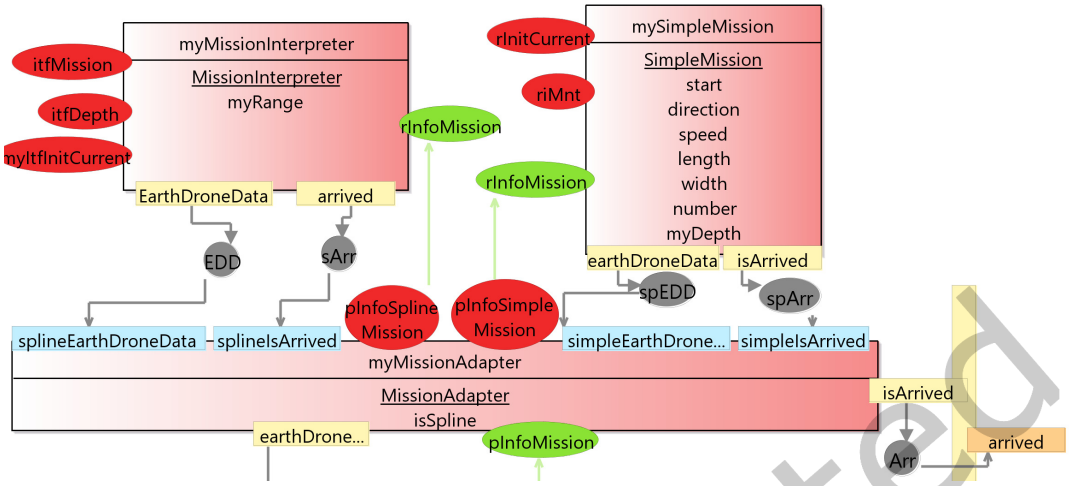

Fig. 7. Excerpt of the Drone composite component.

The last required configuration for the simulator is the choice of a specific algorithm for the control component. The idea is here to declare a *select* parameter, *myNavFunctionNumber* for the AUV, set when defining the scenario.

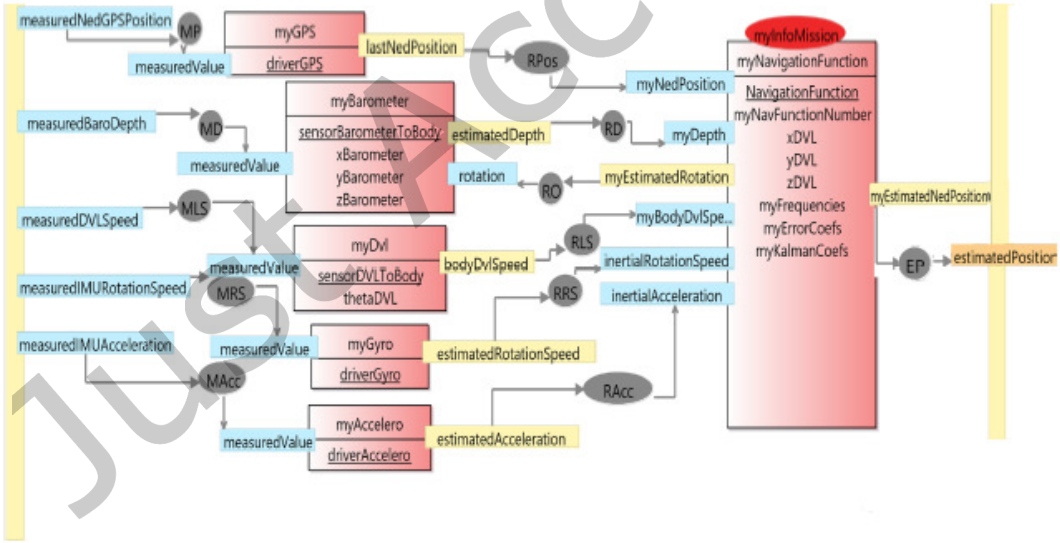

Fig. 8. Navigation composite component.

4.4.5 Integration of legacy code. To integrate domain-specific code, we require that the code respects a given interface composed of three functions (*initialize()*, *doStep()* and *end()*) derived from FMI.

If legacy code is developed with the same language as the generated language, we propose to use the delegator design pattern. We add a wrapper to the legacy code to implement the same interface as the generated component. Then, the generated component is instrumented to call the corresponding services provided by the wrapper.

If legacy code is developed with another language, we propose to implement the delegator design pattern by using either FMI facilities or language adaptors. Using FMI, legacy code is first encapsulated in a FMU. A FMU is a co-simulation unit and can be developed from a few programming languages. To deal with FMU, libraries propose Java (javaFMi[8]) and C++ (FMI4cpp [9]) wrappers. An FMI wrapper respects the FMI interface and can be called directly by the CARES generated component. We provide in the Figure 9 a pseudo-code based on javaFMI to drive FMU simulation by a leaf component *FmuLeafComponent*. FMU is declared through its corresponding FMU file *fmuFileName*. In the constructor, the *Simulation element* is created. The *element* is then initialized by calling *init()* function. In the *doStep()* function, the inputs are sent to the FMU (*write* call), the FMU is executed for a delay of a period (*doStep()*) and the result is recovered to set the outputs (*read* call). The wrapper calls are inserted in specific portions of code (between Acceleo comments *Start of user code* / *End of user code*). These portions of code are protected when regenerating code from the model (Acceleo facilities). For the case study, we also develop a language adaptor to integrate C++ code in a Java simulator. The original C++ code (a Kalman filter for the navigation function) has been first extended to implement the *initialize()*, *doStep()* and *end()* functions. Then a language adaptor has been implemented by using JNI facilities.

Integration of legacy code to implement provided functions is based on the same delegator design pattern. Except that FMI wrapper cannot be used in this case because the FMI wrapper is limited to a generic interface (*initialize()*, *doStep()* and *end()*).

4.5 Scenario definition

Defining a CARES scenario allows to configure a specific co-simulation sequence. In particular, the scenario serves as a space to describe dynamic behaviors and specify failures.

The NAVIDRO architectural style imposes just that all unused components are stopped in the first configuration part of the scenario (first *begin* section). A unused component is a component that produces unused data and does not provide required interfaces.

For AUV co-simulation, an example of scenario *AuvSimulation* is presented in Figure 10. It simulates half an hour considering steps of 1 ms. In the initialization part, first, each data source is selected and configured depending on the pattern used for data source selection. For *Environment*, models of seabed and current are selected. The model of current is then initialized by calling the *setSpeed(speed=1.0,direction=45.0)* function. For DVL, the *dvlSpeed* data is computed considering the error model. So the required interface of the adapter is connected to the provided interface of *myErrorDVL* and the unused component *myREaderDVL* is stopped. A DVL misalignment is defined (15 °) for this co-simulation by setting the *theThetaDVL* parameter. By setting *isSpline* to *false*, the *SimpleMission* component is selected to provide the trajectory, so the *MissionInterpreter* component is stopped. Position is estimated by using data provided by DVL and gyroscope components. The DVL misalignment is not considered by the navigation function. The scenario is the place to specify failures. Fisrt, to evaluate the impact of random errors performed by the sensors (simulated by the *myErrorDVL* component), the scenario is played 10 times. For each scenario execution, the initial drone speed is set to 2.5 ms^{-1}. After 45 minutes, the drone speed is reduced to 2.0 ms^{-1} due to engine limitations. At last, a transient DVL error (no sent data during 0.5 second) is simulated by stopping the DVL component after 10 seconds and restarting it after 10.5 seconds. During co-simulation, the estimated positions are stored each ms in a *DronePosition.csv* log file.

[8]https://bitbucket.org/siani/javafmi/wiki/Home
[9]https://github.com/NTNU-IHB/FMI4cpp

```java
package cares.application;

import cares.framework.Clock;
import cares.components.compFmuLeafComponent;

//Start of user code  : Additional imports for FmuLeafComponent
import org.javafmi.wrapper.Simulation;
//End of user code

public class FmuLeafComponent {

    protected compFmuLeafComponent myContainer;

    // Inputs
    protected Double input;
    // Outputs
    protected Double output;
    // Parameters
    protected String fmuFileName;

    // Start of user code  : Properties of FmuLeafComponent
    protected Simulation element;
    // End of user code


    public FmuLeafComponent(compFmuLeafComponent container) {
        myContainer = container;
    // Start of user code  : Implementation of constructor method
        element = new Simulation(fmuFileName);
    // End of user code
    }
    public void initialize() {
    // Start of user code  : Implementation of initialize method
        element.init(0.0);
    // End of user code
    }
    public void doStep(int nStep) {
    // Start of user code  : Implementation of doStep method
        element.write("inputName").with(input);
        element.doStep(Clock.getStepTime()*nStep);
        output = element.read("outputName").asDouble();
    // End of user code
    }
    public void end() {
    // Start of user code  : Implementation of end method
        element.terminate();
    // End of user code
    }
    // Start of user code  : Additional methods
    // End of user code
}
```


Fig. 9. Java pseudo-code based on the javaFMI 2.26 API to drive a FMU simulation from a leaf component.

This scenario illustrates the multiple possibilities provided by CARES to configure a co-simulation run by modifying parameters, binding interfaces, calling provided function and activating or deactivating components.

navidro.scn | new Component System Diagram | env.dcl | Component System Diagram

```
Simulation AuvSimulation (ms) // name of the scenario and time unit
    system auv; //name of the system model under study
    simulationTime [0,1800000]:1; // simulation from 0 to 1 800 s, step of 1ms
    begin{ // simulation initialization
        // simulation with a model of sea bed
        bind "auv.comp.Environment.MNT.myMntModel.pMntModel"
            to "auv.comp.Environment.MNT.myMntAdapter.rMntSource";
        "auv.comp.Environment.MNT.myGeoTiff".Stop();
        // simulation with a model of current, with an amplitude of 1.0 meter and a direction of 45°
        bind "auv.comp.Environment.Current.myCurrent.itfCurrentModel"
            to "auv.comp.Environment.Current.myCurrentAdapter.rCurrentSource";
        "auv.comp.Environment.Current.myCurrent"."environment.CurrentConfig.setSpeed"
        (p1::"environment.CurrentConfig.setSpeed.speed"="1.0";
        ,p2::"environment.CurrentConfig.setSpeed.direction"="45.0";);
        "auv.comp.Environment.Current.myS111CurrentFile".Stop();
        // simulation with a model of noised DVL, with a DVL misalignment of 15.0 °
        bind "auv.comp.Sensors.DVL.myErrorDVL.pitfEDVL"
            to "auv.comp.Sensors.DVL.mySourceDVLAdapter.ritfEDVL";
        "auv.comp.Sensors.DVL.myReaderDVL".Stop();
        "auv.comp.Sensors.DVL.myBodyToDVL.thetaDVL" = 15.0;
        // simulation with a  simple trajectory
        "auv.comp.Drone.myMissionAdapter.isSpline" = false;
        "auv.comp.Drone.myMissionInterpreter".Stop();
        // simulation with a navigation function based on DVL and gyroscope analysis
        "auv.comp.Navigation.myNavigationFunction.myNavFunctionNumber" = 2;
        // the position computation is based on no Dvl misalignment
        "auv.comp.Navigation.myDvl.thetaDVL" = 0.0;
    }
    scenarios {
        Scenario scenar [10]
            begin{
                "auv.comp.Drone.mySimpleMission.speed"=2.5; // initial speed(ms-1) of the auv
            } events {
                // after 10 seconds, the DVL stops
                instant 10000 {
                    "auv.comp.Navigation.myDvl".Stop();
                }
                // after 10.5 seconds, the DVL restarts
                instant 10500 {
                    "auv.comp.Navigation.myDvl".Start();
                }
                instant 2700000 {
                    "auv.comp.Drone.mySimpleMission.speed"=2.0; // reduced speed(ms-1) of the auv
                }}
            end {}
            logs {// each ms, time and the estimated position of the drone is stored
                DronePosition.csv timed(1.0)
                    { "auv.comp.Navigation.myNavigationFunction.myEstimatedNedPosition";}
            };
    }
    end {}
```

Fig. 10. A co-simulation scenario for the AUV navigation function.

5 EVALUATION

In order to evaluate the ability of our approach to simulate an AUV navigation function, co-simulations are performed on realistic configurations considering different contexts.

We perform first co-simulations to qualify the performance and the quality of the simulator itself. We consider here a drone following a straight line during 20 km with a constant speed of 5 m/s (duration of 4000 seconds, a little bit more than 1 hour). We simulate ideal sensors (no error) acting at a frequency of 1000 hz. The drone and the navigation function are also simulated at the same frequency. With a PC i7 of 3.6 GHz CPU speed

with 8GB RAM, co-simulation takes less than 47 seconds and the position error is under 5 mm. The error is computed between the positions of the emulated trajectory (the straight line) and the computed trajectory based on simulated sensor data. Considering this ideal configuration, the simulator performance and accuracy is quite acceptable. Then we consider some classical scenarios.

Considering the mission contained in a csv file specifying a list of way-points characterized by their position and desired speed, a drone trajectory is computed using spline technique. After applying a model error on sensors, to qualify the quality of the navigation function, we draw the two trajectories produced by the *Drone* component and by the *Navigation* component. Figure 11 depicts the results obtained for a specific mission with a DVL misalignement (the DVL has a different orientation from the drone and the navigation function does not considered it), the yellow (bright color) trajectory corresponds to the estimated one while the green (dark color) one represents the trajectory of the drone.

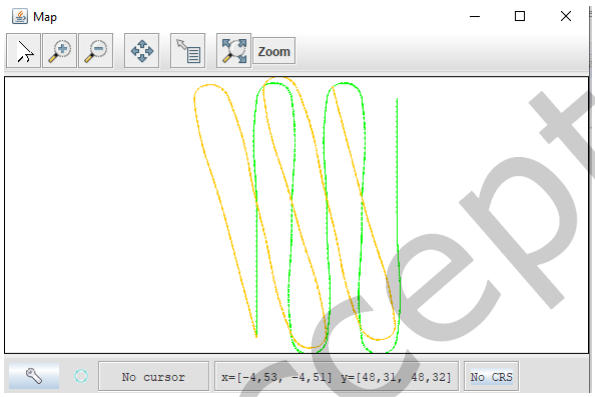

Fig. 11. Example of co-simulation with a DVL misalignment.

Different cases have been studied to test and validate the capacity of the simulator to consider different configurations and contexts. The simulator allows co-simulation of realistic sensors considering different configurations of error models. For another aspect, the simulator allows the re-execution of a real mission performed by an AUV by reading sensor data logs. In this scenario, sensor data is not simulated but rather replayed. For those two different cases, several navigation functions have been tested. For instance, one integrates data from the DVL and the gyroscope. Another one is based on the application of a Kalman filter. In this case, we had to test a lot of parameter values to tune the performance of the Kalman filter. The last presented example (Figure 12) is obtained with real data logs and a navigation function based on the integration of the DVL speed, taking into account orientation through the gyroscope data. Co-simulation shrinks in 20 seconds a duration of thirty minutes for a 2.5km trajectory length (average speed of 1.39 ms^{-1}). For this scenario, the obtained maximal error between real trajectory position and estimated position using a Kalman filter for navigation function is about 13.5 m (drift of 0.5 % after thirty minutes of mission). Depending on the environmental condition (underwater visibility ranges between 10 and 30 meters depending on the depth and water turbidity) or on the mission goal (observation allows for a positioning error, unlike the docking process), this distance may be acceptable or not .

6 RELATED WORKS

The literature proposes a lot of works promoting usage of CBSE paradigm for architecting CPS. [12] shows that the main concerns handled by CPS component models are those of integration, performance, and maintainability. The instruments to satisfy those concerns, while architecting CPS, are ad-hoc software/system architecture,

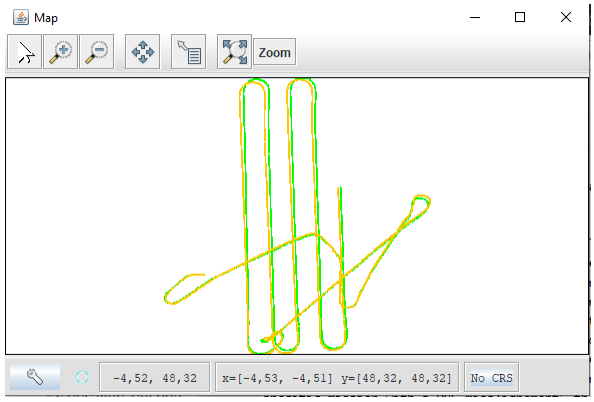

Fig. 12. Example of navigation function evaluation with real data.

model-based approaches, architectural and component languages. Our approach concerns model-based ad-hoc architecture for integration purpose. The main difference with the literature is that we target co-simulation while most focus on design and verification of CPS [16, 19, 23, 24, 48].

As an example, Ulgen [50] proposes a framework for autonomous cyber-physical systems (CPS). Although Ulgen focuses primarily on the design and programming of the control aspect of autonomous CPSs, we share several key concepts with Ulgen. These include the use of high-level models and domain-specific languages, decomposition into blocks with dynamic links controlled by a switching mechanism, and an architectural style guiding the combination of blocks to implement the modules.

The literature proposes also a lot of works on how to use a General Purpose Modelling Language (GPML like UML, AADL and SysML) for FMI co-simulation [22, 28, 35]. The main idea of these papers is to integrate FMI concepts in a dedicated profile of the considered GPML, to target classical co-simulation environments. The goal of CARES is to define high-level concepts to ease design, configuration and execution of FMI co-simulations. These concepts are proposed through a Domain Specific Modeling Language (DSML) and a framework to hide GPML complexity but, the manipulated concepts could have been defined with a GPML and bridges may be developed between CARES and GPML. These aspects would be investigated later but are out of scope of this paper.

In concurrent way, DirectSim [10] proposes a framework to develop agent-based simulators. It provides a run-time library, configuration and observation facilities, a set of bricks to model different kinds of vehicle, to model errors and 2D and 3D monitors. It is an open-source project but it does not propose an architectural style or guidelines on how to build a simulator of a drone or integrate FMUs.

CoSim20 proposes a modeling language to define a correct coordination of different executable models for co-simulation, which can be distributed at execution stage [33]. [26] proposes a language and platform independent framework for running distributed FMI co-simulations, based on RPC (Remote Procedure Call) technology. [46] proposes a tool to execute distibuted FMI co-simulation using a fixed or variable step algorithm, based on the REST API. If distribution is not necessary for the targeted application domain, one could imagine distributed implementation of CARES models, which would require extending CARES, code generator and runtime library with component allocation features and integration of distributed communication and orchestration mechanisms.

FIDE is an Integrated Design Environment (IDE) for FMI [11]. It allows the modeling and design of co-simulation by integrating FMUs and considering discrete events. It proposes to implement a deterministic Master Algorithm. DACCOSIM NG proposes a Graphic User Interface (by describing a co-simulation graph) and a Command-Line

Interface (to drive co-simulation run) to run FMI 2.0 co-simulations based on both centralized and distributed architectures [21]. Experiments show correct performances for DACCOSIM.

As DACCOSIM and FIDE, CARES proposes high-level graphical interface to describe co-simulation graphs. CARES implements a similar Master Algorithm as FIDE without considering roll-back. Moreover, CARES provides facilities to consider parametrization and Client/Server interfaces for component description. In addition CARES proposes an architecture style and a high-level declarative language to ease the description of different co-simulation runs (the CARES scenarios).

The SSP standard provides a tool-independent XML format for the description, packaging and exchange of system structures and their parameterization [34]. The standard integrates notions of units, valuation, adapters and mapping between parameters and components. SSP can be used to precisely describe co-simulation system setting. In the same idea, CARES offers the possibility of parameterizing co-simulation, but by providing high-level modeling languages. Moreover, with CARES, modification of parameters may be programmed at specific instants by using the scenario language, bringing dynamics in the description of co-simulation runs.

Based on FMI, MOKA [3] proposes an Object-Oriented co-simulation framework for development, integration and co-simulation of FMU. The main contribution of MOKA is to provide a generic FMU API to ease integration. The idea of a built-in orchestrator is shared with CARES. But CARES proposes a more high level modeling component-based language integrating facilities to configure co-simulation (parametrization, types of co-simulation units, dynamic binding of components). It also proposes an architecture style and a high-level language to program co-simulation runs.

Vico [25] is an entity-component-system (ecs) based co-simulation framework. The ecs architecture coming from gaming world is well suited to achieve performance. The approach is suitable to launch a 3D simulation but, as the authors said, FMU are difficult to fit into an ecs architecture. Using ecs, behaviors and data have to be separated. The solution is here to propose a generic FMU interface as Moka. At execution stage, [25] proposes to program scenarios using the Kotlin programming language. [25] is ecs-based to promote performances while CARES is component-based to promote application of software engineering principles (architectural style, high-level languages, configuration facilities).

Because the manual configuration of complex large-scale co-simulation scenarios can be error-prone, MOSAIK [45] proposes an approach for assisting the user in the development of co-simulation scenarios. MOSAIK provides a simplified FMI-based API to build (component selection and connection), to parameter and to execute co-simulations. MOSAIK is code-oriented whereas CARES with the same objective is based on a high-level modeling language. From orchestration point of view, we propose a similar Master Algorithm to execute a co-simulation unit step (get inputs/ doStep/write outputs). Based on MOSAIK, to prepare complex co-simulations, [42] proposes an ontology that is used to get assistance in the process by getting recommendation of suitable co-simulation units. CARES covers the co-simulation part of the simulation process proposed by MOSAIK.

There exists different ways to implement a Master Algorithm to orchestrate FMU execution [8, 22, 32]. For this specific aspect, [47] proposes a DSL to design Master Algorithms. The proposed DSL eases integration of FMU considering different orchestration policies. It also eases parallelization and optimization. This work is complementary to ours, CARES proposes a DSML for all aspects of co-simulation except for the orchestration part. The modeling and configuration of Master Algorithm is a perspective for CARES.

The orchestration of multi-periodic simulations raises some semantic issues. Base on the work of Caspi et al. [9], CoCoSim [7] introduces a framework that tackles the issue of data transfer semantics with multi-periodic Simulink blocks. The idea is to transfer the most recently produced data between blocks, even when the production period varies. CARES implements the same semantic but for simulation purposes.

FMI provides a FMU interface specification, but it does not ensure that FMUs interact with in a semantically correct manner (different references, units, models of computation or accuracy). [20] proposes a language that allows for the descriptions of common semantic adaptations that can be used in FMI co-simulation. This paper

shows the importance of mastering co-simulation unit integration through a high-level language. Our paper addresses this adaptation challenge through three aspects: services defined by components may implement adapters inside generated FMUs wrappers, the architectural style defines common references and units and data communication protocol allows different models of computation. CARES should be extended to ease description and integration of more adaptation policies.

Regarding programming frameworks, some solutions exist for drone simulation like the open source OpenAUV testbed [41]. The simulator is proposed as extensive physical testing can be expensive and time consuming because of short flight times due to battery constraints and safety precautions. This approach remains code-centric and is dedicated to Multirotor Unmanned Aerial Vehicles. In the same domain, Aerostack [40] proposes a layered architectural style to design and simulate controllers of Unmanned Aerial Systems. Even if this work addresses a different domain, we share with it the idea of defining an architecture style, the development of domain-specific library and the idea of designing a system by configuring existing blocks.

The literature proposes also specific simulators for AUV. UWSIM [36] integrates a mechanical model of a submarine drone for simulation of its movements. Different sensors can be simulated as well as the sea state. However, this work focuses on visualization rather than simulating the sensors and the environment. Navlab [17] proposes a software with the same objective as the case study of the paper: the simulation of navigation function of AUV. The solution is code-centric, developed with MATLAB. As with NAVIDRO, the architecture style is based on three layers (trajectory simulation, sensors and navigation function). In addition simulated sensors (integrating noise) or real data may be considered. However, this project is not open-source and developed as a black-box.

AVS [31] is an architectural style that has been proposed to develop simulators for autopilots of racing sailboats. Some concepts of AVS inspire the definition of CARES and NAVIDRO like the ability to integrate heterogeneous models. But CARES and NAVIDRO offer a more mature and powerful framework (interface adaptation, scenario definition, FMI integration).

In conclusion, compared to the literature, the main originality of CARES and NAVIDRO is to propose high-level modeling languages, independent of technologies and platforms, easing the description and configuration of co-simulation and co-simulation runs. The proposal is based on two levels of abstraction. At the first level, CARES components and scenarios define the generic concepts for easy-to-configure co-simulations. The co-simulation implementation code is then generated from the models, which eases co-simulation development. At the second level, the NAVIDRO architecture style shows how to use and combine CARES concepts for a specific application domain, here trajectory tracking applications.

7 CONCLUSION

In this paper, the CARES framework and the NAVIDRO architectural style have been presented. The CARES model-based approach combines classical CBSE concepts and FMI 2.0 for co-simulation paradigm for the flexible design of drone co-simulation. NAVIDRO helps to standardize units and deals with heterogeneity of data source. It eases integration of legacy code like FMI co-simulation units. The approach has been tested by developing a simulator to evaluate different AUV navigation functions, considering different AUV configurations (trajectories and sensor behaviors) and contexts (sea current and level of seabed).

The paper addresses software engineering challenges while future works concern improving performance by adding parallelism to the execution stage. Each component can be executed in parallel and according to the dependency graph, each iteration can also be parallelized. This perspective requires developing new models, languages and runtime library to enable the distribution of components, ensure their interactions and coordinate their execution.

Another classic optimization consists in integrating multivariate frequencies for co-simulation units. In this case, to avoid simulation errors, a rollback mechanism should be implemented, which is not the case in the current version of CARES where a small co-simulation step is the only way to provide realistic, but not efficient, simulations.

CARES has been defined to integrate FMUs following the FMI 2.0 Co-Simulation standard. The FMI 3.0 Co-Simulation standard offers new facilities to consider initialization, configuration and events [37]. CARES has to be extended to ease the driving of co-simulations integrating FMU following the FMI 3.0 Co-Simulation standard. We also work on developing bridges between CARES models and GPML languages (to reuse existing modeling techniques and tooling) or other simulation languages (to ease reuse of legacy code).

Finally, we are working on developing a design space exploration tool that drives co-simulations with different configurations to compare the impacts of some parameters on the simulation objective. The exploration tool has the objective to define, for instance, what is the impact of a certain level of a sensor error on the performance of the navigation function. This work may be related to research on hyperparameter optimization.

REFERENCES

[1] 2010. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Object Model Template (OMT) Specification - Redline. *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000) - Redline* (2010), 1–112. https://doi.org/10.1109/IEEESTD.2010.5953408

[2] Srikrishna Acharya, Amrutur Bharadwaj, Yogesh Simmhan, Aditya Gopalan, Parimal Parag, and Himanshu Tyagi. 2020. CORNET: A Co-Simulation Middleware for Robot Networks. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. 245–251.

[3] Memduha Aslan, Halit Oğuztüzün, Umut Durak, and Koray Taylan. 2015. MOKA: An Object-Oriented Framework for FMI Co-Simulation. In *SummerSim '15: 2015 Summer Simulation Multiconference*. Chicago, 1–8. https://doi.org/10.5555/2874916.2874920

[4] Christopher J. Augeri, Kevin M. Morris, and Barry E. Mullins. 2006. Harvest: A Framework and Co-Simulation Environment for Analyzing Unmanned Aerial Vehicle Swarms. In *MILCOM 2006 - 2006 IEEE Military Communications conference*. 1–7.

[5] Pierre Benet, Fabien Novella, Marie Ponchart, Pierre Bosser, and Benoit Clement. 2019. State-of-the-Art of Standalone Accurate AUV Positioning - Application to High Resolution Bathymetric Surveys. In *OCEANS 2019 - Marseille*. 1–10. https://doi.org/10.1109/OCEANSE.2019.8867041

[6] Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, Christoph Claub, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, Manuel Monteiro, Thomas Neihold, Dietmar Neumerkel, Hans Olsson, Jorg-Volker Peetz, and Susann Wolf. 2011. The functional mockup interface for tool independent exchange of simulation models. *Proceedings of the 8th International Modelica* (2011). https://doi.org/10.3384/ECP11063105

[7] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Éric Noulard, and Claire Pagetti. 2020. CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models. *Embedded Real Time Systems (ERTS) 2020* ARC-E-DAA-TN74591 (2020).

[8] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. 2013. Determinate composition of FMUs for co-simulation. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*. 1–12. https://doi.org/10.1109/EMSOFT.2013.6658580

[9] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. 2003. Translating discrete-time Simulink to Lustre. In *International Workshop on Embedded Software*. Springer, 84–99.

[10] Florent Cornu, Alexandre Garnier, and Christian Audoly. 2008. Simulation of the efficiency of a system of drones in a mine warfare scenario. *Undersea Defence Technology* (2008), 1–8.

[11] Fabio Cremona, Marten Lohstroh, Stavros Tripakis, Christopher X. Brooks, and Edward A. Lee. 2016. FIDE: an FMI integrated development environment. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, Sascha Ossowski (Ed.). ACM, 1759–1766. https://doi.org/10.1145/2851613.2851677

[12] Ivica Crnkovic, Ivano Malavolta, Henry Muccini, and Mohammad Sharaf. 2016. On the Use of Component-Based Principles and Practices for Architecting Cyber-Physical Systems. In *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*. 23–32. https://doi.org/10.1109/CBSE.2016.9

[13] Patricia Derler, Edward Lee, and Alberto Sangiovanni-Vincentelli. 2012. Modeling cyber-physical systems. *Proc. IEEE* 100 (2012), 13–28. https://doi.org/10.1109/JPROC.2011.2160929

[14] Edsger W. Dijkstra. 1982. *On the Role of Scientific Thought*. Springer New York, New York, NY, 60–66. https://doi.org/10.1007/978-1-4612-5695-3_12

[15] Brian Dobbing and Alan Burns. 1998. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. *Ada Lett.* XVIII, 6 (nov 1998), 1–6. https://doi.org/10.1145/301687.289525

[16] Stefan Dziwok, Uwe Pohlmann, Goran Piskachev, David Schubert, Sebastian Thiele, and Christopher Gerking. 2016. *The MechatronicUML Design Method: Process and Language for Platform-Independent Modeling.* Technical Report tr-ri-16-352. Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Zukunftsmeile 1, 33102 Paderborn, Germany. https://www.hni.uni-paderborn.de/pub/9478 Version 1.0.

[17] Kenneth Gade. 2005. NavLab, a Generic Simulation and Post-processing Tool for Navigation. *Modeling, Identification and Control* 26, 3 (2005), 135–150. https://doi.org/10.4173/mic.2005.3.2

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 2001. *Design Patterns: Abstraction and Reuse of Object-Oriented Design.* Springer Berlin Heidelberg, Berlin, Heidelberg, 361–388. https://doi.org/10.1007/978-3-642-48354-7_15

[19] Nicolas Gobillot, Charles Lesire, and David Doose. 2014. A Modeling Framework for Software Architecture Specification and Validation. In *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots - Volume 8810* (Bergamo, Italy) *(SIMPAR 2014)*. Springer-Verlag, Berlin, Heidelberg, 303–314. https://doi.org/10.1007/978-3-319-11900-7_26

[20] Claudio Gomes, Bart Meyers, Joachim Denil, Casper Thule, Kenneth Lausdahl, Hans Vangheluwe, and Paul De Meulenaere. 2019. Semantic adaptation for FMI co-simulation with hierarchical simulators. *SIMULATION* 95, 3 (2019), 241–269. https://doi.org/10.1177/0037549718759775

[21] José Évora Gómez, José Juan Hernández Cabrera, Jean-Philippe Tavella, Stéphane Vialle, Enrique Kremers, and Loïc Frayssinet. 2019. Daccosim NG: co-simulation made simpler and faster. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4-6, 2019 (Linköping Electronic Conference Proceedings, Vol. 157)*, Anton Haumer (Ed.). Linköping University Electronic Press, 157:082. https://doi.org/10.3384/ecp19157785

[22] S. Guermazi, S. Dhouib, A. Cuccuru, C. Letavernier, and S. Gérard. 2016. Integration of UML models in FMI-based co-simulation. In *2016 TMS/DEVS Symposium on Theory of Modeling and Simulation, TMS/DEVS 2016, Part of the 2016 Spring Simulation Multiconference.* Pasadena, United States. https://hal-cea.archives-ouvertes.fr/cea-01843174

[23] Goulven Guillou and Jean-Philippe Babau. 2016. ImocaGen: A model-based code generator for embedded systems tuning. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD).* 390–396. https://doi.org/10.5220/0005804103900396

[24] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. 2014. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. (2014). https://doi.org/10.48550/arXiv.1409.6578 arXiv:1409.6578 [cs.SE]

[25] Lars I. Hatledal, Yingguang Chu, Arne Styve, and Houxiang Zhang. 2021. Vico: An entity-component-system based co-simulation framework. *Simulation Modelling Practice and Theory* 108 (2021), 102243. https://doi.org/10.1016/j.simpat.2020.102243

[26] Lars Ivar Hatledal, Arne Styve, Geir Hovland, and Houxiang Zhang. 2019. A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface. *IEEE Access* 7 (2019), 109328–109339.

[27] Nadjim Horri and Mikolaj Pietraszko. 2022. A Tutorial and Review on Flight Control Co-Simulation Using Matlab/Simulink and Flight Simulators. *Automation* 3, 3 (2022), 486–510. https://doi.org/10.3390/automation3030025

[28] Jérôme Hugues, Jean-Marie Gauthier, and Raphaël Faudou. 2018. Integrating AADL and FMI to Extend Virtual Integration Capability. Toulouse, France. https://doi.org/10.48550/arXiv.1802.05620

[29] James C. Kinsey, Ryan Michael Eustice, and Louis L. Whitcomb. 2006. Underwater vehicle navigation: Recent advances and new challenges. (September 2006).

[30] Manon Kok, Jeroen D. Hol, and Thomas B. Schön. 2017. Using Inertial Sensors for Position and Orientation Estimation. *CoRR* abs/1704.06053 (2017). arXiv:1704.06053 http://arxiv.org/abs/1704.06053

[31] Emilien Lavigne, Goulven Guillou, and Jean-Philippe Babau. 2018. AVS, a model-based racing sailboat simulator: application to wind integration. *3rd IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control CESCIT 2018* 51 (2018), 88–94. https://doi.org/10.1016/j.ifacol.2018.06.242

[32] Renan Leroux-Beaudout, Ileana Ober, Marc Pantel, and Jean-Michel Bruel. 2017. Modeling co-simulation : a first experiment. In *5th International Workshop on the Globalization of Modeling Languages (GEMOC 2017) co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017)*, Vol. 2019. Austin, TX, United States, 292–297. https://hal.archives-ouvertes.fr/hal-02603696

[33] Giovanni Liboni and Julien Deantoni. 2020. CoSim20: An Integrated Development Environment for Accurate and Efficient Distributed Co-Simulations. In *ICISE 2020 - 5th International Conference on Information Systems Engineering.* Manchester / Virtual, United Kingdom. https://hal.inria.fr/hal-03038547

[34] Modelica Projects User Meeting. 2018. The SSP Standard: Project Status and Roadmap. (2018).

[35] Christian Nigischer, Sébastien Bougain, Rainer Riegler, Heinz Peter Stanek, and Manfred Grafinger. 2021. Multi-domain simulation utilizing SysML: state of the art and future perspectives. *Procedia CIRP* 100 (2021), 319–324. https://doi.org/10.1016/j.procir.2021.05.073

[36] Mario Prats, Javier Perez, Fernandez J. Javier, and Pedro J. Sanz. 2012. An open source tool for simulation and supervision of underwater intervention missions. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2012), 2577–2582.

https://doi.org/10.1109/IROS.2012.6385788
[37] The Modelica Association Project. 2021. Functional Mock-up Interface Specification, Version 3.0. (2021).
[38] Loic Salmon, Pierre-Yves Pillain, Goulven Guillou, and Jean-Philippe Babau. 2021. CARES, a framework for CPS simulation : application to autonomous underwater vehicle navigation function. In *2021 Forum on specification & Design Languages (FDL)*. 01–08. https://doi.org/10.1109/FDL53530.2021.9568380
[39] Jorge Jr Sampaio. 2007. Planning 3D Well Trajectories Using Spline-in-Tension Functions. *Journal of Energy Resources Technology-transactions of The Asme - J ENERG RESOUR TECHNOL* 129 (12 2007). https://doi.org/10.1115/1.2790980
[40] José Luis Sánchez-López, Martin Molina, Hriday Bavle, Carlos Sampedro, Ramón A. Suárez Fernández, and Pascual Campoy Cervera. 2017. A Multi-Layered Component-Based Approach for the Development of Aerial Robotic Systems: The Aerostack Framework. *J. Intell. Robotic Syst.* 88, 2-4 (2017), 683–709. https://doi.org/10.1007/s10846-017-0551-4
[41] Matt Schmittle, Anna Lukina, Lukas Vacek, Jnaneshwar Das, Christopher P. Buskirk, Stephen Rees, Janos Sztipanovits, Radu Grosu, and Vijay Kumar. 2018. OpenUAV: A UAV Testbed for the CPS and Robotics Community. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. 130–139. https://doi.org/10.1109/ICCPS.2018.00021
[42] Jan Sören Schwarz, Rami Elshinawy, Rebeca P. Ramírez Acosta, and Sebastian Lehnhoff. 2020. Ontological Integration of Semantics and Domain Knowledge in Hardware and Software Co-simulation of the Smart Grid. In *Knowledge Discovery, Knowledge Engineering and Knowledge Management*, Ana Fred, Ana Salgado, David Aveiro, Jan Dietz, Jorge Bernardino, and Joaquim Filipe (Eds.). Springer International Publishing, Cham, 283–301. https://doi.org/10.1007/978-3-030-66196-0_13
[43] Gerald Schweiger, Cláudio Gomes, Georg P. Engel, Josef-Peter Schoeggl, Alfred Posch, Irene Hafner, and Thierry Nouidu. 2019. An Empirical Survey on Co-simulation: Promising Standards, Challenges and Research Needs. *Simulation Modelling Practice and Theory* 95 (05 2019).
[44] Leydson Silva, Ewerton Salvador, Alisson Brito, Jose Sousa Barros, and Vivek Nigam. 2019. A Multi-UAV Co-Simulation Environment for Safety and Performance Analysis. In *2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC)*. 1–8.
[45] Cornelius Steinbrink, Marita Blank-Babazadeh, André El-Ama, Stefanie Holly, Bengt Lüers, Marvin Nebel-Wenner, Rebeca P. Ramírez Acosta, Thomas Raub, Jan Sören Schwarz, Sanja Stark, Astrid Nieße, and Sebastian Lehnhoff. 2019. CPES Testing with mosaik: Co-Simulation Planning, Execution and Analysis. *Applied Sciences* 9, 5 (2019). https://doi.org/10.3390/app9050923
[46] Casper Thule, Kenneth Lausdahl, Cláudio Gomes, Gerd Meisl, and Peter Gorm Larsen. 2019. Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory* 92 (2019), 45–61.
[47] Casper Thule, Maurizio Palmieri, Cláudio Gomes, Kenneth Lausdahl, Hugo Daniel Macedo, Nick Battle, and Peter Gorm Larsen. 2020. Towards Reuse of Synchronization Algorithms in Co-simulation Frameworks. In *Software Engineering and Formal Methods*. 50–66. https://doi.org/10.1007/978-3-030-57506-9_5
[48] Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. 2001. The CLARAty architecture for robotic autonomy. In *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, Vol. 1. 1/121–1/132 vol.1. https://doi.org/10.1109/AERO.2001.931701
[49] Oliver J. Woodman. 2007. An introduction to inertial navigation. *Technical report of University of Cambridge* UCAM-CL-TR, 696 (august 2007). https://doi.org/10.48456/tr-696
[50] Beyazit Yalcinkaya, Hazem Torfah, Ankush Desai, and Sanjit A. Seshia. 2023. Ulgen: A Runtime Assurance Framework for Programming Safe Cyber–Physical Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 11 (2023), 3679–3692.
[51] Hang Yin and Hans Hansson. 2013. Mode switch timing analysis for component-based multi-mode systems. *Journal of Systems Architecture - Embedded Systems Design* (2013), 1299–1318. https://doi.org/10.1016/j.sysarc.2013.09.004