CARES, a framework for CPS simulation : application to autonomous underwater vehicle navigation function

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

CARES, a framework for CPS simulation : application to autonomous underwater vehicle navigation function

Loic Salmon
LabISEN
ISEN Yncrea Ouest
Brest, France
loic.salmon@isen-ouest.yncrea.fr

Pierre-Yves Pillain, Goulven Guillou, Jean-Philippe Babau
Lab-STICC, CNRS, UMR6285,
Univ. Bretagne Occidentale
Brest, France
pierre-yves.pillain, goulven.guillou, babau@univ-brest.fr

***Abstract*—One key objective of Cyber-Physical System (CPS) simulation is to evaluate different CPS configurations regarding a certain user objective. First, simulation of CPS necessitates frameworks to handle heterogeneity of CPS components (the software and hardware system control, the behavior of the CPS itself and its physical environment). Then, to build simulators, designers use paradigms like FMI (Functional Mock-Up Interface) that proposes a data-driven generic interface facilitating the integration of heterogeneous models. However, in order to facilitate simulation configuration, an approach is required to drive modeling of parametric features and operational conditions. In this paper, we present CARES, a component-based and model-driven approach to facilitate CPS simulation. CARES is applied to evaluate an Autonomous Underwater Vehicle (AUV) navigation function by simulation. The proposed models integrate both the principles of a generic simulation (integration of Component Based Software Engineering CBSE concepts and FMI paradigm) and domain specific aspects through a component-based architecture style. From a design model, a code generator builds the structural (Java or C++) code of the simulator. The generated code relies on a given run-time library for its execution and its structure facilitates integration of domain-specific code. The experiments show the effectiveness of the approach to build simulators for evaluation of different AUV configurations.**

***Index Terms*—component-based design, simulation, model-driven engineering, cyber-physical system**

I. Introduction

In recent years, AUV usage in the hydrographic field is a more and more recurring point. Indeed, AUVs offer many benefits in terms of bathymetry resolution, mission planning and coverage rate. However, there are some uncertainties concerning the estimation of the position of the AUV underwater due to GPS signal loss whereas engineers need to predict, before the mission, that their robot will not be in a forbidden area or lost [5]. Considering those uncertainties and the cost involved for such a system, simulation has become a key issue to predict the AUV position, computed by the navigation function of the AUV [1].

CPS simulation faces four challenges. The first challenge concerns simulation performance. This aspect is addressed by optimizing the domain specific code and by using parallel infrastructure [8]. The second challenge concerns precision of results which relies on the quality of the domain specific code. The third challenge concerns CPS modeling [3]:

- *Preventing misconnected model components*. Indeed, not only because of the type of the components, one issue is the unit and referential of data that can be different from one component to another one. Standardization of the referential and the unit of data is a key element to prevent errors.
- *Modeling sensor behaviors and time disparities*. The precision and the frequency of the sensors can be different. Different sensors are used to monitor the behavior of the CPS and each of them can be noised or disturbed by the environment.
- *System heterogeneity*. Integration of heterogeneous domain specific aspects requires the separation of concerns related to modeling system and simulation. Due to different data sources, different models and algorithms, different simulation configurations are to be considered.

For integration and orchestration of heterogeneous models, the FMI [4] standard is widely used. Even if some points (discrete events, timing semantic, distribution [24]–[26]) may be discussed, the basic concepts proposed by FMI 2.0 for co-simulation (communication between simulation modules and orchestration) are sufficient for our simulation problem.

Finally, the last challenge is the need of easy-to-use environment to design simulators. Classically, complexity of Cyber Physical System (CPS) encourages developers to change the development process by using modeling approaches [3] and more specifically a Component-Based Software Engineering (CBSE) approach [16]. Recently, high-level modeling languages and standards have been proposed to facilitate the description of simulator structures and simulator runs [27]–[29]. But from 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 dedicated CPS, here AUV navigation function.

In this paper, we propose first a framework, called CARES, dedicated to CPS simulation. The framework is a tool-

978-1-5386-5541-2/18/$31.00 ©2018 European Union

independent high-level modeling language (based on concepts defined by CBSE and FMI 2.0 for co-simulation). From models, a code generator and a runtime library allow to build a simulator for a given scenario. Then, to integrate the domain specific aspects of the AUV navigation function, we propose an architectural style, called Navidro. Navidro provides model-driven design rules and templates to facilitate domain specific simulator development. It proposes generic API and adapters to facilitate integration of different AUV behaviors, sensor configurations and navigation functions.

The remainder of the paper is structured as follows. Section II presents the application domain of AUV and more specifically the navigation function. Section III introduces the generic aspects of the model-driven and component-based framework, called CARES. Section IV presents the Navidro architectural style proposed to design the model for an AUV navigation function. In Section V, the simulation results are discussed. Section VI presents related works before a conclusion in Section VII.

II. The AUV navigation function

An AUV is used mainly 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 precision 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 trajectory models can be used as for instance splines [12]. For testing purpose, a simple trajectory may be computed as set of consecutive segments and semi-circles (cf. Figure 1).

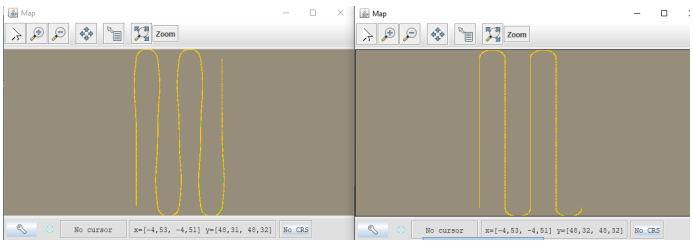

Fig. 1. Two different simulated trajectories built at the right with lines and semi-circles and at the left with splines.

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 adopts crab steering mode. Moreover, the AUV may have to respect a given altitude 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 (replay of a previous mission), given by a forecast file of computed from an environment model.

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

- 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): it provides an estimation of the linear accelerations.
- the gyroscope is part of IMU: it provides an estimation of the angular speeds.
- the DVL (Doppler Velocity Log): it 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

1

Each sensor is characterized by a level of quality (precision) 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, ...) [11]. 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) [10], while others use hybrid process merging all data received by the different sources (i.e. Kalman Filter). A Kalman Filter necessitates parameters' tuning to provide a correct estimation of the AUV position [13].

In conclusion, the simulation of the navigation function depends of the AUV trajectory, the environmental conditions, the quality of the sensors and the chosen algorithm. To evaluate one algorithm, one can deal with real data logs (replay of a given mission) or with models of the different components (evaluation for a specific AUV architecture).

III. CARES Framework

This section presents the CARES[2] (Component framework for Adaptative Real-Time Embedded Systems) framework we propose for simulation of CPS. CARES provides model editors and Java 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 CPS through a set of communicating components, each component simulating one part of the CPS. At execution stage, CARES follows a time-driven orchestration (such as FMI).

A. Main concepts

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

Structure. CARES derives from a component-based approach. Basically, a component is either a leaf component or a composite component. A leaf component executes a specific task, while a composite component can be composed of leaf or composite components. The composite components are only proposed for structural purpose. It facilitates navigation in the system description. From FMI perspective, a leaf component corresponds can be viewed as a Functional Mock-up Unit (FMU). It is a co-simulation unit in charge of simulating one part of the system.

Communication. Communication between components is based on data exchange through input and output ports. The communication protocol is implemented by following the Ravenscar profile [15]: a data producer overwrites data, so a module always considers the last produced data. The algorithm of communication orchestration is then generic and then not described in the model. We just represent datalinks that rely output ports to input ports (one to many relationship).

For our problem, the data exchange protocol is not sufficient. Classical functional services provide facilities to exchange complex data, to configure component behaviors and to program different simulation scenarios. For these needs, we add functional required and provided interfaces to components.

Components behavior. As with FMI, at execution, each leaf component executes a *doStep()* function following a Run-To-Completion paradigm. The *doStep()* function is executed considering data inputs at the beginning and producing data outputs at the end. Each component is initialized with an *initialize()* function and stopped with an *end()* function. To implement different modes during simulation, each component may be activated or not for a given scenario.

Time aspects and scheduling. Following the FMI paradigm for co-simulation, time and scheduling are managed by a centralized scheduler. The scheduler launches periodically the (*doStep(timeInterval)*) operation for each leaf component. The period is based on a frequency expressed in hertz. The time interval is modelled through a delay associated to the component. It corresponds to the time used by the real system to produce data. The output of the component are then updated after this time interval. For scheduling, the execution order is defined by the partial order given by the dataflow. If a cycle exists, a priority associated to each leaf component is used to define the execution order between components.

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

- start and end time for the simulation;
- the time step size;
- at the beginning of a scenario: a sequence of parameter initializations; a list of components to stop (a not-used component may be deactivated); the binding of required and provided interfaces (if necessary); a list of operation calls on specific components for configuring them;
- a sequence of timed events and for each timed event a sequence of parameter initializations or operation calls;
- at the end of a scenario: a sequence of operation calls on specific components for configuring them;
- when a simulation contains random data, a scenario may be played many times. In this case, some operations may be performed at the beginning and the end of each scenario.

A scenario definition provides a high-level description of a given simulation run. Using language facilities, it is then easy to adapt simulation to test a given configuration (parameter valuation or component structure).

B. Modeling

The 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, attributes). Instances of types are described through a textual editor based on Xtext[3].
- *ComponentSystem metamodel*: this metamodel contains the required elements to describe a configuration of components (leaf and composite components, data and interface links). The Figure 3 exhibits an excerpt of this metamodel. A graphical editor based on Sirius[4] allows to instantiate a model of components and links between them.
- *Scenario metamodel*. This metamodel contains the required elements to describe a simulation scenario. A scenario contains initialization steps and timed events (component initialization, attribute modification, function call). A scenario is described through a declarative language, based on a Xtext textual editor.

The three metamodels are implemented using Ecore[5].

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

[3]https://www.eclipse.org/Xtext/

[4]https://www.eclipse.org/sirius/

[5]https://www.eclipse.org/ecoretools/

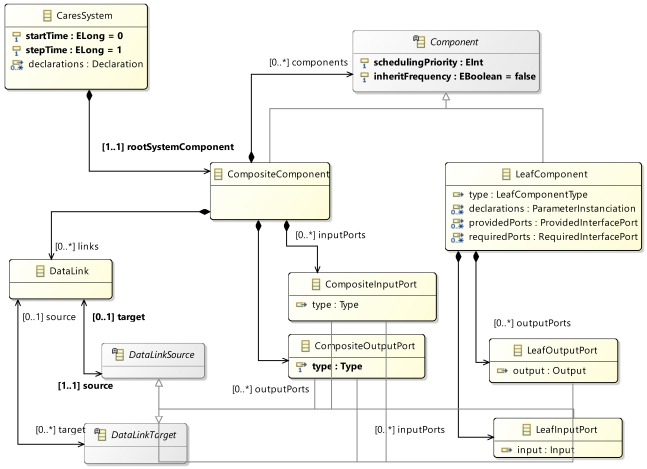

Fig. 3. Component System Metamodel

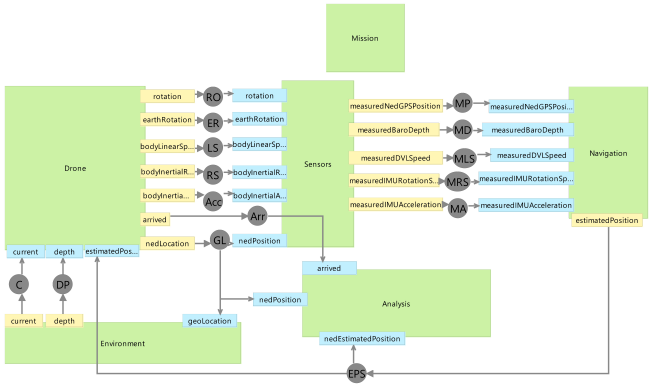

Fig. 4. Global architecture defined by Navidro

C. Code generation

From instantiated models, a code generator implemented using Acceleo[6] generates Java code by analyzing the whole model (checking typing constraints). The generated code contains all the declarations and the glue code for all the components.

For integration of domain specific code, we impose that the code is "FMU-Compliant" by providing only the three *init()*, *doStep()* and *end()* functions. Then, for each leaf component, a corresponding Java class is generated inheriting from a *LeafComponent* generic classes defining *initialize()*, *doStep()* and *end()* functions. The domain-specific code is then integrated by implementing the *delegator* pattern [14] in the generated class.

Once the domain specific code is added, the simulator is ready to run by using the provided runtime library that implements the Master module. A scheduler launches each leaf component, viewed as a FMU, considering its frequency. Data communication is directly handled by the components without the need of the Master module to coordinate read/write operations.

At the end, from a given scenario, a class is generated to declare all the timed events considered by the main class to drive the simulation.

The code is now ready to perform a simulation.

IV. Navidro architectural style

In the following part, model and architectural elements related to the simulation of an AUV navigation function are presented according to the Cares framework described previously.

A. Global architecture

This subsection describes the global architecture style of Navidro (see Figure 4).

[6]https://www.eclipse.org/acceleo/

The *Environment* component is used to model and represent the environment in which the AUV evolves. The environment of the drone is composed of components generating current and level of the seabed at a given position. Both of those elements may be either simulated or extracted from log files.

The *Drone* component is responsible for the generation of the drone trajectory. This component is based either on a mission log reader (replay) or a trajectory model (a list of way-points defined by the *Mission* component). For the latter, the simulated trajectory can be computed simply with lines and arcs or more realistically with splines.

The *Sensors* component contains all the components simulating each sensor such as gyroscope, accelerometer, barometer or GPS. Data produced by a sensor can be computed considering a given error model or extracted from a log file. In the case of simulated data, data are produced by adding error to data obtained by analysing the simulated reference trajectory.

The *Navigation* component is responsible for the navigation function used to estimate the position of the drone. Different navigation functions can be plugged to build the trajectory from the initial point with information provided from the sensors. For instance, a basic navigation function integrating the speed (DVL output) considering the direction by integrating the rotation speed (gyroscope output) has been implemented. This component corresponds to the software part of the system.

The *Analysis* component compares the position given by the *Drone* component with the estimated position computed by the *Navigation* component. Its role is to compute the position incertitude.

As discussed in introduction, to prevent misconnected components, a standardization of units as well as referential frames is necessary. The global architecture style Navidro defines a list of referential frames and units fixed to avoid misconnected components. For instance, the speed, is given in m/s for the current and the measure of the DVL sensor. The position of the AUV is given according to the NED (North/East/Down) frame.

B. Heterogeneity integration

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

1) Units and referential adaptation: As previously noticed, data provided by each sensor is given in the frame and orientation related to itself. To adapt to the defined standard, we propose to provide a library of adapter components that make conversion and referential changes for classical data. Those components allow to standardize domain-specific processes.

The Figure 5 illustrates the utilization of such adapter component. 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 (*ErrorDVL* component) or real data (*readDVL* 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 of the *myBodyToDVL* component. Furthermore because the frame and units of log data are based on technical sensor choice, an adapter is also necessary (*readerDVL*).

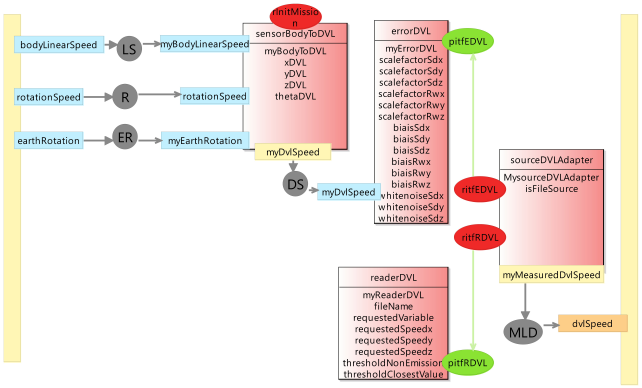

Fig. 5. DVL composite component

Reciprocally, DVL data necessitates to be converted in the body frame of the drone to be processed by the navigation function. This is the role of the *myDVL* component (see Figure 6 of the composite Navigation component) modeling the DVL driver of the system.

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

2) Data Source Selection: For a given simulation, the user has to select specific sources of data in a transparent

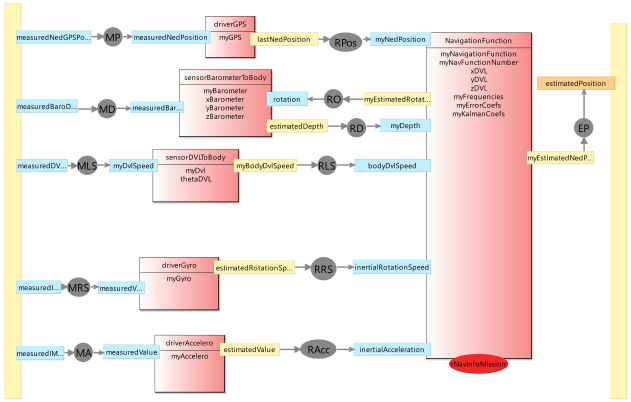

Fig. 6. Navigation composite component

way. The data source selection from logs, models or forecasts is necessary for the following components of the global architecture: *Environment*, *Sensors*, *Drone*. For the *Navigation* component, the user has also to choose a specific algorithm for the estimation of the position. To configure simulation (data source selection), several approaches exist:

- the first idea is to link at design stage the chosen data source component to the corresponding input: this solution requires that the source respects the input standard (respect of the architectural style);
- the second idea is to plug all possible sources to a *switch* component: the *switch* component acts as a selector configured by the scenario;
- the third approach has been addressed in [2] where the authors propose a Model System Logic (MSL) to switch the components during the execution according to a mode defined by the user.

The first solution is simple and efficient at execution stage. But this solution implies that the user has to modify the design view. This point is not easy to users who only want to configure a simulator, without having a knowledge of how it is built. The third solution requires a more complex runtime library for dynamic adaptations. This is not necessary in our case: the configuration of the data sources is fixed for the whole simulation. We propose then to follow the second approach as illustrated in Figure 5 with the component *mySourceDVLSwitch*. This solution implies that all data source components are embedded in the simulator. To optimize execution, this solution requires to do not execute un-selected data source components. So the not used components are deactivated for performance purposes.

Another illustration of the *switch* component is the selection of the simulated trajectory as depicted by the Figure 7. 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.

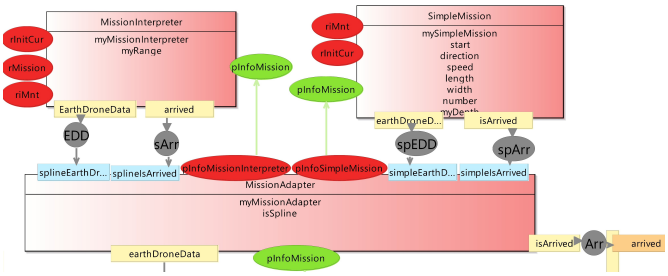

Fig. 7. Excerpt of the Drone composite component

The two examples show the facility of the approach to integrate different sources of data. The *switch* component has the advantage to impose an API, standardizing frame and units to prevent computation errors.

3) Integration of legacy code: To integrate domain-specific code, we impose that the code respects a given Java interface composed of three functions (*init()*, *doStep()* and *end()*). To integrate legacy codes developed in several languages, a first solution is to use the FMI facilities. FMI proposes a Java API to consider heterogeneity of languages for FMU execution. This solution requires to use the corresponding FMI tool licences. In our experiments, we develop a simple adaptor to integrate C++ code. The original C++ code (a Kalman filter for the navigation function) has been first extended to implement the *init()*, *doStep()* and *end()* functions. Then a language adaptor has been implemented by using JNI facilities.

C. Scenario definition

An example of scenario *AuvSimulation* for Navidro is presented in Figure 8. It implements a simulation for 100 seconds. The *MissionInterpreter* component is used to provide the trajectory, so the *SimpleMission* component is stopped. To evaluate the impact of random errors on sensors, the scenario is played 10 times. For each scenario execution, the initial speed and direction is set to 5.0 and 1.0 respectively. After 20 seconds, resp. 50 seconds, the speed, resp. the direction, is modified. During simulation, the estimated positions are stored each ms in a csv log file.

V. Evaluation

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

First, simulations are performed 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, the simulation takes less than 47 seconds and the position error is under 5 mm. Considering this ideal configuration, the simulator performance and precision 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, altitude and desired speed, a drone trajectory is simulated with splines. After applying a model error, to qualify the quality of the navigation function, we draw the two trajectories produced by the *Drone* component and by the *Navigation* component. The following Figure 9 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 trajectory corresponds to the estimated one while the green one represents the trajectory of the drone.

The following example (Figure 10) 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. The simulation shrinks in 20 seconds a duration of thirty minutes. For this scenario, the maximal error is about 13.5 m.

VI. Related work

The literature proposes a lot of works on usage of CBSE paradigm for architecting CPS. [16] 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, model-based approaches, architectural and component languages, and design. Our approach concerns model-based ad-hoc architecture for integration purpose. The main difference from the literature is that we are interested in simulation while most focus on design and verification as some works about CPS and robot control ([20], [21], [18], [19], [7]).

Concerning architectures for simulation, some generic solutions have been proposed like FMI [4]. FMI is an independent standard for modeling and co-simulation . The aim of FMI is to associate several simulation tools into one co-simulation environment orchestrated by the Master module (data exchange and scheduling). FMI constitutes a generic proposal while our approach addresses domain specific concerns. In another way, DirectSim [23] 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 a guide on how to build a simulator of an AUV navigation function.

CoSim20 proposes a modeling language to define a correct coordination of different executable models for co-simulation, which can be distributed at execution stage [25]. In our approach, due to the simulation domain, distribution is not necessary and coordination is simplified because of the time-driven simulation.

FIDE is an Integrated Design Environment (IDE) for FMI [27]. It allows the modeling and design of co-simulation by integrating FMUs and considering discrete events. It proposes to implement a deterministic FMI Master Algorithm.

```
Simulation AuvSimulation (ms)  //name of the scenario and time unit
   system Navidro;   //name of the system model under study
   simulationTime [0,100000]:1; //simulation from 0 to 100000 ms, step of 1 ms
   begin{  // initialization of parameters and stop unused components
      "Navidro.Drone.SimpleMission.isSpline"=false;
      "Navidro.Drone.MissionInterpreter".Stop();}
   scenarios {Scenario scenarLog [10] // the scenario scenarLog is played 10 times
         begin { // initialization of drone parameters
            "Navidro.Drone.SimpleMission.speed"=5.0;
            "Navidro.Drone.SimpleMission.direction"=1.0;
         events {
            // at instant 20s, the speed of the drone is set to 2.5
            instant 20000 { "Navidro.Drone.SimpleMission.speed"=2.5; }
            // at instant 50s, the direction of the swimmer is set to -1.0
            instant 50000 { "Navidro.Drone.SimpleMission.direction"=-1.0;} }
         end {// operations performed at the end of each simulation execution}
         logs {// each ms, time and the estimated position of the drone is stored
            DronePosition.csv timed (1.0) {
               "Navidro.Drone.NavigationFunction.myEstimatedNedPosition";}};}
   end { // operations performed at the end of the simulation execution}
```

Fig. 8. A simulation scenario for Navidro

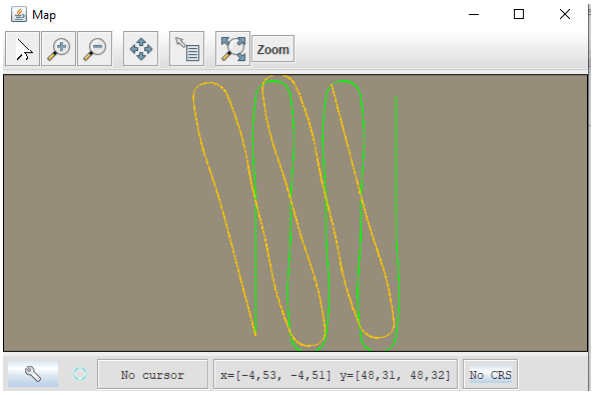

Fig. 9. Example of simulation with a DVL misalignment

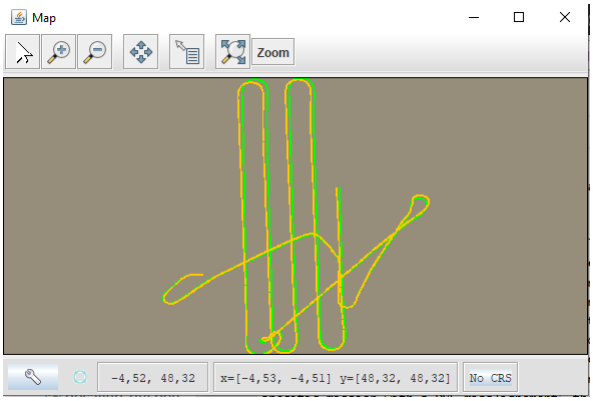

Fig. 10. Example of navigation function evaluation with real data

Daccosim NG proposes a Graphic User Interface (for co-simulation graph description) and a Command-Line Interface (to drive co-simulation run) to run FMI 2.0 co-simulations based on both centralized and distributed architectures [28]. 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. CARES provides facilities to consider parametrization and Client/Server interfaces for component description. AIn addition CARES proposes a high-level declarative language to facilitate the description of different co-simulation runs (the CARES scenarios).

The SSP standard proposes a tool-independent XML-based format for the description, packaging and exchange of system structures and their parameterization [29]. The standard integrates notions of units, valuation and mapping between parameters and components. SSP may be used to describe a system parametrization precisely. In the same idea, CARES proposes parameter definition, but directly into the co-simulation graph. Moreover, the parameter valuation is used to drive co-simulation runs through a high-level declarative language.

Some simulators exist for drones like the open source OpenAUV testbed [17]. The simulator is proposed because 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 [22] 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 this work the idea of defining a component-based 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 [6] 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 [1] proposes a software with the same objective as in the paper: the simulation of navigation function of AUV. It has been developed with MATLAB. Like in our

approach, it 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, it is developed as a black-box with no information on its architecture.

VII. Conclusion

In this paper, the CARES framework for the design of simulators of CPS has been proposed. This model-based approach combines classical CBSE concepts and FMI paradigm. To use it in the specific domain of AUV navigation function, the Navidro architectural style has been proposed. It helps to standardize units and deals with heterogeneity of data source. The approach has been tested for different AUV configurations considering simulated or real trajectory and measurement errors.

Future works concern performance improvements by adding parallelism at execution stage. For instance, each sensor component may be executed in parallel. In addition, because of the iterative aspect of simulation, each iteration may be also parallelized. Another idea is to split the simulator in two parts, one for the trajectory creation and another one for sensors and navigation function analysis.

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, internal and external events [30]. CARES has to be extended to facilitate the driving of simulations integrating FMU following the FMI 3.0 Co-Simulation standard.

Finally, a simulation could launch several simulations with different parameter values in order to compare the impacts of some parameters on the performance of the navigation function. The idea is to develop an architecture exploration tool for the navigation function.

References

[1] K. Gade, NavLab, a generic simulation and post-processing tool for navigation, Modeling, Identification and Control, 135-150, vol(26), 2005

[2] H. Yin and H. Hansson, Mode switch timing analysis for component-based multi-mode systems, Journal of Systems Architecture - Embedded Systems Design, 1299-1318, vol(59), 2013

[3] P. Derler, E. Lee and A. Sangiovanni-Vincentelli, Modeling cyberphysical systems, Proceedings of the IEEE, 13-28, vol(100), 2012

[4] T. Blochwitz et al., The functional mockup interface for tool independent exchange of simulation models, Proceedings of the 8th International Modelica, Dresden, 2011

[5] J. C. Kinsey, R. M. Eustice, and L. L. Whitcomb, A survey of underwater vehicle navigation: recent advances and new challenges, Proceedings of IFAC Conference of Manoeuvering and Control of Marine Craft, Lisbon, 2006

[6] M. Prats, J. Perez, J.J. Fernandez, P.J. Sanz, An open source tool for simulation and supervision of underwater intervention missions, 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2577-2582, Algarve, 2012

[7] S. Becker et al., The MECHATRONICUML design method - process and language for platform-independent modeling, HNI, Paderborn University, Tech. Rep. tr-ri-14-337, 2014

[8] IEEE Standard for Modeling and Simulation (M-S) High Level Architecture (HLA)– Framework and Rules, 2010

[9] D. Cook, A. Vardy, R. Lewis, A survey of AUV and robot simulators for multi-vehicle operations, IEEE/OES Autonomous Underwater Vehicles (AUV), 1-8, Oxford (USA), 2014

[10] M. Kok, J. D. Hol, T. B. Schn, Using inertial sensors for position and orientation estimation, Foundations and Trends in Signal Processing, 1-153, vol(11), No. 1-2, 2017

[11] O. J. Woodman, An introduction to inertial navigation, Technical report of University of Cambridge, 2007

[12] J.H.B. Sampaio Jr, Planning 3D well trajectories using spline-in-tension functions, Journal of Energy Resources Technology-transactions of The Asme, vol(129), 2007

[13] P. Benet, F. Novella, M. Ponchart, P. Bosser and B. Clement, State-of-the-art of standalone accurate AUV positioning - Application to high resolution bathymetric surveys, IEEE OCEANS, Marseille, 2019

[14] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: abstraction and reuse of object-oriented design, ECOOP'93 Object-Oriented Programming, 406-431, Berlin, 1993

[15] A. Burns, B. Dobbing, B. and G. Romanski, The Ravenscar tasking profile for high integrity real-time programs, Reliable Software Technologies - Ada-Europe, 263-275, Berlin, 1998

[16] I. Crnkovic, I. Malavolta, H. Muccini and M. Sharaf, On the use of component-based principles and practices for architecting cyber-physical systems, 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE), 23-32, Venice, 2016

[17] M. Schmittle et al., OpenUAV: A UAV Testbed for the CPS and Robotics Community, ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), 130-139, Porto, 2018

[18] N. Gobillot, C. Lesire and D. Doose, A modeling framework for software architecture specification and validation, International Conference on Simulation, Modeling, and Programming for Autonomous Robots, 303-314, Bergamo, 2014

[19] Arne Haber, Jan Oliver Ringert and Bernhard Rumpe, Montiarc-architectural modeling of interactive distributed and cyber-physical systems, arXiv:1409.6578, 2014

[20] R. Volpe, I. Nesnas, T. Estlin, H. Das, the CLARAty architecture for robotic autonomy, IEEE Aerospace Conference, 121-132, vol(1), Big Sky, 2001

[21] G. Guillou and J.-P. Babau, ImocaGen : A model-based code generator for embedded systems tuning, 4th International Conference on Model-Driven Engineering and Software Development, 390-396, Roma, 2016

[22] J. L. Sanchez-Lopez, M. Molina, H. Bavle, C. Sampedro, R. A Surez Fernndez and P. Campoy, A multi-layered component-based approach for the development of aerial robotic systems : the aerostack framework, Journal of Intelligent and Robotic Systems. 683-709, vol(88), 2017

[23] https://www.directsim.fr/projects/directsim-framework/wiki

[24] S. Tripakis, Bridging the semantic gap between heterogeneous modeling formalisms and FMI, International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation. 60-69, SAMOS, 2015

[25] G. Liboni and J. Deantoni, CoSim20: An Integrated Development Environment for Accurate and Efficient Distributed Co-Simulations, International Conference on Big Data in Management. 76-83, Manchester, 2020

[26] S. Centomo, M. Lora and F. Fummi, Generation of Functional Mockup Units for Transactional Cyber-Physical Virtual Platforms, chapter of Languages, Design Methods, and Tools for Electronic System Design. 27-46,2020

[27] F. Cremona, M. Lohstroh, S. Tripakis, C. Brooks and E. A. Lee, FIDE: An FMI Integrated Development Environment, the 31st Annual ACM Symposium on Applied Computing. 1759-1766, Pisa, 2016

[28] J.E. Gomez, J.J. Hernandez Cabrera, J-P Tavella, S. Vialle, E. Kremers and L. Frayssinet, Daccosim NG: co-simulation made simpler and faster, 13th International Modelica Conference. 157:82, Regensburg, Germany, 2019

[29] The SSP Standard: Project Status and Roadmap, Modelica Projects User Meeting. Cambridge, MA, 2018

[30] Functional Mock-up Interface Specification, Version 3.0, The Modelica Association Project FMI. 2021