Integrating realistic simulation engines within the MORSE framework

Arnaud Degroote, Pierrick Koch, Simon Lacroix

Integrating Realistic Simulation Engines within the MORSE Framework

Arnaud Degroote[1], Pierrick Koch[2,3] and Simon Lacroix[2,4]

***Abstract*— The complexity of robotics comes from the tight interactions between hardware, complex softwares, and environments. While real world experience is the only way to assess the efficiency and robustness of a robotics system, simulations help to pave the way to actual experiments. But an overall robotics system requires simulations at a level of realism which no holistic simulator can provide, given the wide spectrum of disciplines and physical processes involved. This paper presents a way to integrate various simulators, in a distributed, scalable and repeatable way, to benefit from their different advantages and get the best fitted and accurate simulation for a given robotics system. It depicts how the MORSE open-source robotics simulator is adapted to comply with the High Level Architecture standard, thus allowing the reuse of numerous dedicated realistic simulators. Two examples of the integration of simulators are provided.**

I. Introduction

As complex systems, robots integrate a variety of subsystems that rely on a large spectra of physical processes. Dynamics is of course the primal concern, and it varies a lot depending on the environment and kind of robots considered: rigid-body mechanics, fluid dynamics or wheel-soil interactions for instance. Perception implies optics, electromagnetism, acoustics, also depends on the kind of sensors used and the considered environments. Besides these robot-related physical processes, the environment itself is defined by a series of properties and dynamic processes, that either pertain to physics (*e.g.* atmospheric phenomena that may impact flight mechanics or perception) or are related to other actors governed by specific models (*e.g.* crowd dynamics, road traffic, humans interacting with the robots).

Robotics simulators developed within the robotics community are far from integrating this whole spectrum of processes and associated models. Their design is mostly driven by other concerns, such as real-time property, compliance with the software architecture within which the functions to evaluate are integrated, ease of deployment and use. They are mostly used to test, evaluate or validate the *integration* of a series of functions, and the realism of the simulated processes is often not an important concern.

While such simulators are very beneficial to robotics developments, their lack of realism does not properly fill the gap between simulations and actual tests: there is a growing interest in exploiting more realistic models within robotics simulators. The literature abounds with *specialized simulators*, that focus on a given physics phenomenon: the issue of integrating such simulators within a robotics simulator, in a composable and reusable manner, pertains to the *simulator architecture*.

This paper presents a way to tackle this integration issue, in a distributed (and also scalable) way. The current work is based on the existing open-source robotics simulator Morse [1] and the High Level Architecture standard (HLA, [2]). In the following section, we discuss the general software architecture of robotics simulators, and in particular, the architecture of Morse. Section III provides an overview of the HLA standard and the evolution of the Morse architecture to comply with this standard, and section IV presents two illustrations of distributed simulations deployement using Morse and HLA. A discussion concludes the paper.

II. Robotics Simulators

Robotics simulators such as Gazebo, V-REP, or Morse rely on a similar architecture: they are built upon one (or more) classic physics engine such as ODE or Bullet, and a graphical engine to edit the environment and simulate vision and depth sensors. Simulated sensors and controllers are embedded in a component / plug-in system. All these components run in the same process, using multiples threads for parallel computations [1]. While such a design allows to test in real time the integration of different robotics components and rather "high-level" algorithms, it fails to scale along two dimensions:

- The increase in the number of simulated robots;
- The augmentation of the realism of the simulated sensors, actuators and environments.

Besides performance limitations, integrating an existing specialized simulation may be difficult in practice, as existing simulators usually do not provide the interfaces defined by the component / plug-in system. Also, even if several physics engines are supported, only one can be used for a given simulation run: this is an issue for fine-grained simulation of teams of heterogeneous robots for instance, that may include aerial, marine and ground robots.

The Morse simulator relies on the open-source modeller Blender and the physics engine Bullet. This tight integration within Blender allows to easily construct realistic robots and environments. Blender being fully scriptable in Python, one can import scenes from third party environment models, *e.g.* multi-layered terrain maps provided by geographic agencies or companies. As it is usual for robotics simulators, Morse provides a component system, allowing to easily integrate new sensors and actuators on-board the simulated robot.

[1]Institut Supérieur de l'Aéronautique et de l'Espace, 31055 Toulouse, France. arnaud.degroote at isae.fr

[2]CNRS, LAAS, 7 av du colonel Roche, F-31400 Toulouse, France; {firstname.lastname} at laas.fr

[4]Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France

[3]Univ de Toulouse, LAAS, F-31400 Toulouse, France

[1]In Gazebo, there are in fact two processes: a viewer allowing interactions with the simulation *gzclient*, and *gzserver*, the simulator itself

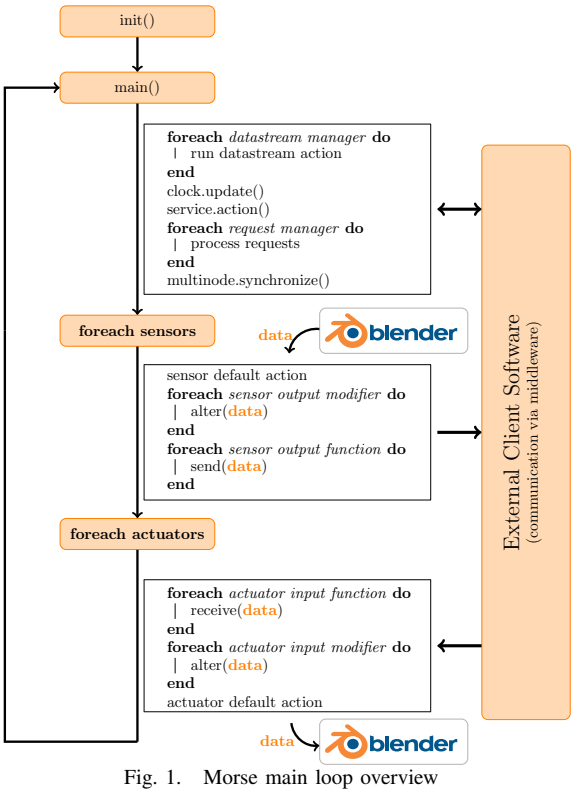

Fig. 1. Morse main loop overview

But Morse goes further in the component decomposition and the separation of concerns, by providing modifiers and datastream handlers:

- Modifiers are pieces of code allowing to change slightly the behaviour of a sensor, from simple data type conversion to arbitrarily complex noise models.
- Datastream handlers adapt the data to robotics middleware specific format, for both data-oriented interfaces and service-oriented interfaces. This feature allows to transparently use Morse with a variety of robotics architectures, such as standard ROS, GeNoM-based, Orocos-YARP, Orocos-ROS, and MOOS. Robots running different architectures can hence be jointly simulated – which proved to significantly ease multiple partners integration in collaborative projects [1].

Figure 1 illustrates how the different Morse components interact during one simulation loop. Simulation scenes, including the components to use, their configurations and interactions are described using the Builder API, an internal Domain-Specific Language (DSL) based on Python. This API hides the Blender complexity to users, but also allows to dynamically program a scene, thus easing robustness tests of algorithms in various situations, *e.g.* by triggering events in the environment according to specified scenarios.

III. Distributed simulations using HLA

To augment the level of realism of a robotics simulator using additional specialized simulators, or to augment the number of robots involved in a simulation, the only way is to distribute the simulation processes over a network of CPUs – which is largely used in more mature industries, such as space [3] or aeronautic [4] industries, because it allows to run precise and realistic simulations.

Distributed simulations raise the issue of *synchronization*, as the simulators deployed on the various CPUs have their own computation requirements and scheduler, or the CPUs may differ – not to mention the delays induced by the network. Besides, one needs tools to allow the exchange of information between the simulators. This is made possible thanks to the High Level Architecture standard, which defines solutions to the issues of information exchanges and time management between the simulators.

A. Overview of the High Level Architecture (HLA)

HLA (IEEE-1516) [5] is an open international standard, developed by the Simulation Interoperability Standards Organization (SISO) and published by IEEE. In the HLA terminology, a *federate* is an HLA compliant simulator, while a *federation* is the set of simulators connected for one distributed simulation. HLA defines an API covering the different needs for distributed simulations, allowing to:

- model the content of one simulator (the Simulation Object Model or SOM), and what is exchanged in the federation (the Federation Object Model or FOM) (*i.e.* the set of objects exchanged in a given simulation);
- manage the federation itself (simulator entering or going out the federation), and the objects that each federate manages;
- dynamically exchange objects update between federates and interactions (interactions as explicit actions between federates, at some discrete time);
- control time following different strategies.

Various implementations of this standard are available (*e.g.* [6]): these *RunTime Infrastructures* can be viewed as simulation middlewares (not to be confused with robotics middleware). One of the great benefits of using HLA is that a wide variety of specific simulators are compliant with the interfaces it defines. In robotics, there has been a few attempts to develop distributed simulation suites using HLA [7], [8], [9], but none became widely used yet.

B. Time management in HLA

Time management is particularly important, as it is the core of the correctness of the simulation. HLA defines two kinds of strategies for messages ordering:

- *Receive Order* strategy. This strategy basically sends messages to federates as soon as they are emitted. This strategy does not guarantee correctness or repeatability of the simulation, but allows the interactions with real systems. It is a "real" time mode.
- *Time Stamp Order* (TSO) strategy. In this strategy, senders assign a timestamp to each messages. The HLA implementation guarantees to deliver messages to each federate following the timestamp order. This ensures that the distributed simulation is repeatable and allows proper analyses.

To support this second strategy, a federate relies on the *Time Management* API described in the HLA standard. Each federate must explicitly request to the *HLA RunTime Infrastructure (RTI)* an advance of its *logical time (LT)* and

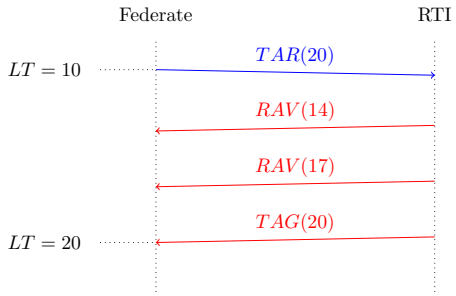

Fig. 2. Typical execution sequence.

waits from the RTI that this advance is granted (through a *Time Advance Grant (TAG)* message). Depending on the kind of federate, time advance can be requested through:

- *Time Advance Request (TAR)* for time-stepped federates;
- *Next Event Request (NER)* for event-stepped federates.

In one federation, the two methods can be used by different federates. The TAR mechanism, used for the Morse simulator, illustrated by a typical sequence for a time-stepped federate in Figure 2, consists of the following:

1) A federate sends a $TAR(T)$ message to the RTI
2) The RTI delivers, in order, the TSO messages which timestamp is $\leq T$
3) The RTI sends the $TAG(T)$ when it can guarantee that all TSO messages have been delivered
4) The federate can now advance its internal time, and compute a new simulation loop.

C. Adaption of Morse to HLA

We present here how Morse has been adapted for distributed heterogeneous simulations using the standard protocol HLA, allowing in particular a more formal and guaranteed way to deploy heavy simulations. The HLA Object Management API is analogous to publish-subscribe API's exposed by most middlewares for data exchanges, where:

- *Update Attribute Value (UAV)* allows to publish new data for an object (as port in YARP or topic in ROS);
- the *Reflect Attribute Value (RAV)* callback informs the interested federate about these new information;
- *Request Attribute Value Update* can be used to request the new or last value for an object.

To integrate HLA in Morse, it is natural to handle HLA as a standard datastream protocol and not as a specific custom mode. In particular, this allows to reuse all the existing infrastructure of Morse, both to build and run the simulations. This also leaves the possibility to adopt another technology or standard in the future, without breaking any other part of Morse.

1) Time management: As already mentioned, a key point for any distributed heterogeneous simulation is the time handling. Thanks to the reasonably simple design of Morse (single threaded loop as shown in 1), handling time according the HLA rules is relatively easy. The synchronization pattern can be implemented in the HLA specific datastream action, as shown in Algorithm 1.

As a consequence, the main loop is basically blocked until the RTI yields the authorization to advance time. Then, Morse executes normally the rest of the loop. Sensors will execute their code, and publish if needed the information using an *Update Attribute Value* with the right timestamp. The actuators will decode the last stored update of information and process them.

Algorithm 1 HLA datastream action handler

```
request time progression sending a Time Advance Request
granted ← False
while not granted do
    On Reflect Attribute Value reception : store received update
    On Time Advance Grant reception : granted ← True
end while
```

2) Complex interactions between simulators: Time management allows to export sensors computed by Morse to other simulators, and to get actuators driven by external simulators. But this does not enables to grab the sensor computation from another simulator or have an actuator driven by Morse with answers computed by an external simulator. For that purpose, we introduce the notion of *ExternalObject*: this is an object which can read streams for both input or output. Standard loop execution for such objects is described in Algorithm 2. No real new concept is integrated here, it is just a generalization of the existing *Sensor* or *Actuator* concepts in Morse. The only difficult part was to modify the builder API to be able to specify to which direction a modifier or a datastream handler is applied to an object.

Algorithm 2 Action of an *ExternalObject*

```
for all input datastream do
    new_data, data ← read(datastream)
end for
if new_data then
    for all input modifier do
        alter(data)
    end for
end if
Custom Action (can be void)
for all output modifier do
    alter(data)
end for
for all output datastream do
    write(data)
end for
```

These two mechanisms allows Morse to inter-operate with various other simulators through HLA. Thanks to the simple design of Morse, the process was relatively easy, with less than 500 lines of code modified to support a large part of the HLA possibilities and the new object kind *ExternalObject*.

IV. Illustrations

We now illustrate the presented architecture with three different instantiations. The two first examples exhibit interoperability between Morse and two very different simulators, one based on a graphical modeling tool, and the other based

on a plain C++ library. The third example shows how the HLA architecture allows to better handling big simulation with numerous robots. Lastly, we briefly discuss the possible bottlenecks of the architecture, and ways to overcome them.

A. Interaction between Morse and Ptolemy

Ptolemy II [10] is an open-source modeling and simulation tool for heterogeneous systems, developed at the UC Berkeley. It provides different models of computations such as continuous time or discrete events, allowing to model different aspects of a cyber-physical system. HLA actors to allow the interaction of the Ptolemy II model with other simulators have been recently proposed in [11].

Here, we want to design a command law for an Unmanned Aerial Vehicle to keep the drone at a constant height over the ground. Ptolemy is used here to model both the physical model of the quadrotor, and the cascading command law. Morse is used to simulate the environment and the drone's sensors, the drone being equipped of a radar altimeter (allowing to retrieve the distance to the ground) and a GPS. This model is showed in Figure 4: the model exports its position (x, y, z) through HLA, and receives a value $Height$ from $altimeter$, and the drone position from the GPS. Various parameters related to the federation are configured in the HLA manager named `quad1`. On the Morse side, this scenario is described by the builder script shown in figure 3. Morse receives the position (x, y, z) from Ptolemy, processes it through `read_pose`, and then passes it to the `teleport` actuator[2]. Whereas the radar computes the height to the ground, and processes it before sending it through the `write_height` method. The script looks like a nominal use of Morse datastream: the only HLA specificity is in the configuration line 16, to specify the file's location describing the Federation Object Model, and the specific federation name. The required Python conversion scripts are less than 20 lines each. This example is interesting for the following reasons:

- it validates the integration of several time management methodologies in the same federation (and the correctness of the Morse implementation), since Ptolemy uses here the NER approach while Morse use TAR,
- it shows how easy it is to define a HLA scenario for Morse,
- and it demonstrates the benefits of a HLA-distributed simulation, allowing to easily reuse a model defined in a high-level graphical language.

Note that a very similar assembly would be possible using the closed-source Matlab/Simulink and its associated HLA toolbox [3].

B. JSBSim and Morse

Morse includes simplistic flight models for quadrotors or helicopters. These models are good enough for testing integration of high-level algorithms, but are far from allowing the testing and validation of flight control laws. We present here how Morse can be coupled to the realistic Flight Dynamic Model (FDM) JSBSim [12], so as to be able to

[2]The `teleport` actuator changes the position of a robot in one step.
[3]`http://www.forwardsim.com/products/hla-toolbox`

```
1  robot = Quadrotor()
2
3  teleport = Teleport()
4  robot.append(teleport)
5  teleport.add_stream('hla', 'read_pose')
6
7  radar = RadarAltimeter()
8  radar.add_stream('hla', 'write_height')
9  robot.append(radar)
10
11 gps = GPS()
12 radar.add_stream('hla', 'write_pos')
13 robot.append(gps)
14
15 env = Environment('...')
16 env.configure_stream_manager('hla', fom = '...',
17     name = 'Morse', federation = 'morse_ptII')
```

Fig. 3. Morse builder script to describe a HLA scenario

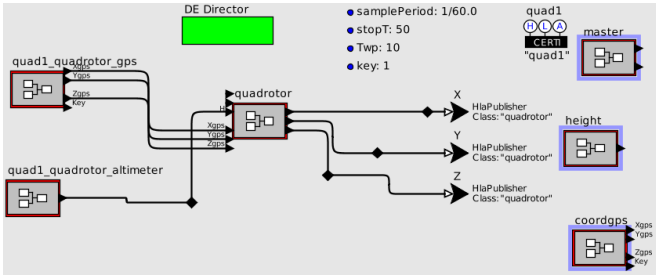

Fig. 4. A Ptolemy Actor modeling a control-law connected to the external world through HLA

test flight control laws. JSBSim includes fine aerodynamic model, several atmospheres and wind models, and numerous propulsion and engines models. It is already used in flight simulators such as FlightGear or OpenEagles, and is also exploited to test autopilot systems, such as PixHawk or the Paparazzi system. JSBSim provides a C++ interface under a dynamic library, so it is possible to use it directly into a Morse component. But this kind of integration is generally not possible, depending on the specialized simulator architecture, and it is not scalable. A better way to proceed is to develop in C++ a HLA wrapper for JSBSim, which makes it possible to couple JSBSim and Morse – and actually with any other HLA-compliant simulator. The development of such node is not very complex, the wrapper we developed is less than 1000 lines long.

Figure 6 depicts the interactions between an Orocos software architecture and a Morse/JSBSim integrated simulator. Orocos components communicates with the Morse simulator using the YARP middleware, while the Morse core interacts through HLA with the JSBSim node, sending commands (coming directly from Orocos, or after some computation of a Morse actuator), and receiving pose and velocity updates. These values can then be used to simulate on-board sensors output (*e.g.* IMU). The figure 5 provides a simplified configuration script for Morse of such a scenario. We can retrieve classic sensors (line 13 and 17), but also some ExternalSensor (line 21) and ExternalActuator (line 8) (hence we configure both input and output stream). The presence of the geodetic modifier (line 5) is worth noticing: JSBSim, as most of FDMs, is working in geodetic coordinates, while Morse is working internally in the Local Tangent Plane

```
1  robot = Quadrotor()
2
3  teleport = Teleport()
4  robot.append(teleport)
5  teleport.alter('geodetic')
6  teleport.add_stream('hla', 'AircraftPoseInput')
7
8  ctrl = DirectControl()
9  robot.append(ctrl)
10 ctrl.add_stream('yarp',  direction='IN')
11 ctrl.add_stream('hla', 'AircraftCtrl', direction='OUT')
12
13 pose = Pose()
14 robot.append(pose)
15 pose.add_stream('yarp')
16
17 imu = IMU()
18 robot.append(imu)
19 imu.add_stream('yarp')
20
21 jsbsim_mag = JsbsimMagnetometer()
22 robot.append(jsbsim_mag)
23 jsbsim_mag.add_stream('yarp', direction='OUT')
24 jsbsim_mag.add_stream('hla', 'MagnetometerInput', direction='IN')
25
26 env = Environment('sandbox')
27 env.properties(longitude=1.26, latitude=43.26, altitude=130.0)
28 env.configure_stream_manager('hla', name='Morse',
29     fom='aircraft.fed', federation='morse_fdm')
30 JSBSimExporter(env).dump()
```

Fig. 5. A simplified Morse-JSBSim scenario builder script

coordinate system. The geodetic modifier allows to do the conversion between the two coordinates systems, based on the properties identified line 27. Of course, in this case, it is also possible to do the conversion in the JSBSim C++ wrapper, but it would break the common usage in FDMs. Lastly, the line 31 "analyses" the rest of the builder script, and generates a parameter file for the JSBSim node, to share common information, such as initial position, model to use, and timestep. This capacity to generate automatically these parameters, due to the fact that builder DSL is based on the Python programming language, eases a lot the deployment of distributed simulations.

This coupling between Morse and JSBSim allows to simulate, test and validate in the same environment a complete aerial robot that comprises an autopilot to handle flight control and higher level processes, *e.g.* exteroceptive sensors data processing and mission re-planning.

C. Multiple robot simulations

Previous examples demonstrate the use of HLA to couple heterogeneous simulators to allow more realistic simulation. We now describe how HLA can be used to scale the simulation for scenarios with a large number of robots.

The idea here is to distribute the simulation between several Morse nodes, each node handling only a subset of robots (and their associated sensors and actuators). Nodes are synchronised through HLA, by exchanging the robot positions [4]. This guarantees that all nodes, at each logical instant, have the same view of the global simulation, and so, typically, that exteroceptives sensors perceive a correct and up-to-date state of the world (i.e. the other robots present in the environment). The overall configuration is depicted in Figure 7. The same mechanism can also be used for hybrid

[4]Hence the call to `multinode.synchronize()` in the `main()` of the Morse loop shown Figure 1.

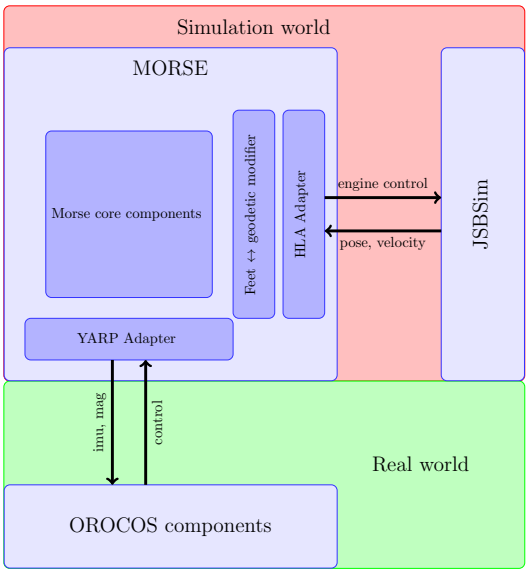

Fig. 6. An Orocos-based robot simulation with Morse and JSBSim. The red area denotes the simulation world, where data are exchanged through HLA. The green area denotes the interaction between the simulation and real software to be tested, using a robotic middleware (here YARP).

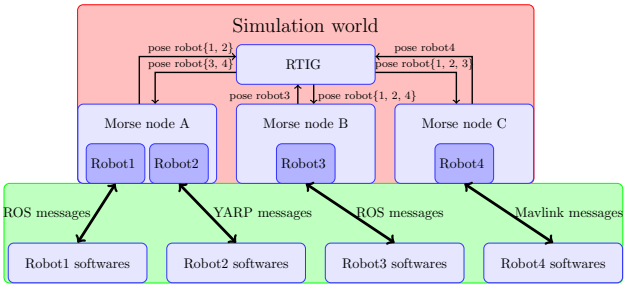

Fig. 7. A Morse multinode instantiation with four robots on three different nodes. Each robot software layer interacts with Morse using its specific middleware. The red area denotes the simulation world, where data are exchanged through HLA. The green area denotes the interaction between the simulation and real software to be tested.

simulation, *i.e.* in which real robots interact with simulated robots – this specific case imposing that the simulation runs in real-time.

The HLA multi-node mode supersedes the historical in-house multi-node socket protocol developped for Morse. It can handle more complex scenarios (such as 'as fast as possible' simulation) and provides guarantee on the correctness of the distributed simulation. Moreover, through the reuse of a standard architecture, it eases the deployment and debug of large multi-robots simulations.

D. On performance

One possible performance bottleneck in such distributed scenario is the volume of data exchanged on the network. HLA designers took this in consideration, and defined the DDM (Data Distribution Management). The aim of DDM API is to limit and control the volume of data exchanged between federates. It allows to define multi-dimensional coordinates system, and so to register and grab information only from specific geometric zones. [13] reviews and analyzes (from a performance point of view) most of the approaches proposed to implement this API.

Another possible issue is the centralized aspect of the RTIG. Several proposals has been made, such as a hierarchical federations approach [14] or a bridge system [15] or surrogates [16] between heterogeneous federations. Unfortunately, none of these solutions has yet reached the HLA standard.

V. Discussion

We have presented an approach allowing the existing Morse simulator to inter-operate with various other specialized simulators / tools, thus fostering the deployment of more realistic simulations in robotics. It demonstrates the versatility architecture of Morse, which clearly separates components, modifiers, and middleware interfaces. The use of a DSL to describe simulation scenarios also eases the deployment of such simulations. The use of the HLA standard is very beneficial, as it is a heavily tested standard for distributed simulation (in space, aeronautic, defense, ... communities). It allows in particular the reuse of mature tools, in a distributed, scalable and repeatable way. Moreover, the use of HLA also allows the deployment of hybrid simulations.

Examples presented in Section IV illustrate the different possibilities of the new architecture, mainly interoperability with heterogeneous simulators, and scalability over large simulations. No performance analysis has been conducted. There is no equivalent simulation model in a homogeneous centralized framework for the two first ones, and not efficient enough hardware to execute large scale multi-robot simulations. The essential point is that HLA guarantees a sound state of the simulation, independently of the underlying simulators performance. The implementation of the presented architecture is integrated in the current implementation of Morse [5], the JSBSim part is in a specialized repository [6].

Future work will consist in consolidating the current implementation and extending the realism of simulations by integrating additional specialized simulators (*e.g.* an atmosphere simulator for autonomous soaring developments, or a traffic simulation for smart vehicle development). Another important point is the user-interface, which should ease the definition and deployment of distributed simulations. Among the longer term issues that remain to be tackled, the two following are worth to consider:

- The definition of simulation descriptions in order to provide generic interfaces and interoperability between simulators. Indeed, while HLA defines a grammar, it does not define the vocabulary exchanged between federates. To provide real interoperability between simulators, this vocabulary (the Federation Object Management) must be generic and descriptive enough. This model problem is still open, and should be discussed within the community.
- The maintenance of the consistency of the environment models that are distributed over an HLA federation: *e.g.* when ruts are caused by robot traversing a terramechanics terrain model, the terrain appearance must be updated accordingly for the vision / Lidar simulators. Solving this mainly pertains to the ability to maintain a consistent soil model and propagate its variations to the visual layer: HLA could easily handle the propagation of such changes.

Recently, a consortium of industrials and academics developed *Functional Mockup Interface* (FMI), a new standard to ease the interoperability between simulators [7]. This standard is relatively orthogonal to HLA and focuses more on interfaces between simulators, by giving standardized access to simulation model equations. It is even possible to combine them [17], using HLA technologies as a master ("scheduler") for the different simulators exporting FMI interfaces. Morse interoperability with the FMI standard will be an interesting point to investigate.

[5] https://github.com/morse-simulator/morse

[6] https://github.com/adegroote/morse-jsbsim

[7] https://www.fmi-standard.org

References

[1] G. Echeverria, S. Lemaignan, A. Degroote, S. Lacroix, M. Karg, P. Koch, C. Lesire, and S. Stinckwich, "Simulating Complex Robotic Scenarios with MORSE," in *SIMPAR*, 2012.

[2] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR, 1999.

[3] M. R. Reid and E. I. Powers, "An evaluation of the high level architecture (HLA) as a framework for NASA modeling and simulation," in *25th NASA Software Engineering Workshop, Goddard Space Flight Center, Greenbelt (USA)*, 2000.

[4] J.-B. Chaudron, D. Saussié, P. Siron, and M. Adelantado, "Real-time aircraft simulation using HLA standard," in *IEEE AESS Simulation in Aerospace 2011, Toulouse (France)*, June 2011.

[5] "IEEE standard for modeling and simulation (M&S) high level architecture (HLA)-IEEE std 1516-2000, 1516.1-2000, 1516.2-2000. Institute of Electrical and Electronics Engineers, New York," 2000.

[6] E. Noulard, J.-Y. Rousselot, and P. Siron, "CERTI, an Open Source RTI, why and how," in *Spring Simulation Interoperability Workshop*, 2009, pp. 23–27.

[7] L. Winkler and H. Wörn, "Symbricator3D – a distributed simulation environment for modular robots," in *Intelligent Robotics and Applications*. Springer, 2009.

[8] L. Xiang, L. Xunbo, and C. Liang, "Multi-disciplinary modeling and collaborative simulation of multi-robot systems based on HLA," in *International Conference on Robotics and Biomimetics*, 2007.

[9] P. Nebot, J. Torres-Sospedra, and R. J. Martínez, "A New HLA-Based Distributed Control Architecture for Agricultural Teams of Robots in Hybrid Applications with Real and Simulated Devices or Environments," *Sensors*, vol. 11, no. 4, pp. 4395–4400, April 2001.

[10] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, " Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[11] G. Lasnier, J. Cardoso, P. Siron, C. Pagetti, and P. Derler, "Distributed Simulation of Heterogeneous and Real-Time Systems," in *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, Washington, DC, USA, 2013.

[12] J. S. Berndt, "JSBSim: An open source flight dynamics model in C++," in *AIAA Modeling and Simulation Technologies Conference and Exhibit, Providence, Rhode Island*. Citeseer, Aug. 2004.

[13] C. Dzermajko, "Performance comparison of data distribution management strategies in large-scale distributed simulation," Master's thesis, University of North Texas, Denton, Texas, 2004. [Online]. Available: http://digital.library.unt.edu/ark:/67531/metadc4524

[14] W. Cai, S. J. Turner, and B. P. Gan, "Hierarchical federations: an architecture for information hiding," in *Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 2001.

[15] B. Bréholée and P. Siron, "Design and Implementation of a HLA Interfederation Bridge," in *European Simulation Interoperability Workshop, Stockholm (Sweden)*, 2003.

[16] M. W. Yoo and T. G. Kim, "Design and implementations of surrogates for interoperation of hla federations," in *Proceedings of the 2009 SISO European Simulation Interoperability Workshop*. Citeseer, 2009.

[17] M. Awais, P. Palensky, A. Elsheikh, E. Widl, and S. Matthias, "The high level architecture RTI as a master to the functional mock-up interface components," in *Computing, Networking and Communications (ICNC), 2013 International Conference on*, Jan 2013, pp. 315–320.