A Distributed Real-Time Framework For Dynamic Management Of Heterogeneous Co-simulations

Jean-François Cécile, Loic Schoen, Vincent Lapointe, Alexandre Abreu, Jean Bélanger

Opal-RT Technologies Inc.
1751 Richardson, Suite 2525
Montréal, Québec, Canada, H3K 1G6
jean-francois.cecile@opal-rt.com

Keywords: Real-time, heterogenous, co-simulation, RT-LAB, UAV

ABSTRACT

Simulation of complex systems usually requires that heterogeneous models be integrated into a single simulation environment. Because these models are often developed by different teams, or depend on various commercial simulation tools (such as Simulink™, Dymola™ or SystemBuild™), considerable effort is expended in configuring the corresponding components into a cohesive co-simulation.

As part of its research and development efforts, Opal-RT has developed RT-LAB Orchestra, a software application that facilitates integration and interoperability between co-simulation components. RT-LAB Orchestra is an application-level data communication layer that sits on top of Opal-RT's RTLAB framework, a proven real-time architecture for distributed simulations.

RT-LAB Orchestra provides an application programming interface (API) that implements a data-centered communication mechanism between co-simulation components. Using the RT-LAB Orchestra API, components connect to the RTLAB framework and exchange data with other co-simulation components, synchronously or asynchronously. Prior to its connection, the data exchange for a co-simulation component is configured through a graphical user interface that controls XML connection elements.

A case-study is presented that models the formation flight of Unmanned Aerial Vehicles (UAVs). RT-LAB Orchestra's dynamic connection capabilities are used to model how the overall behavior of a flight is affected by the removal or addition of UAVs to an existing formation.

1. INTRODUCTION

Heterogeneous co-simulation consists of software components written in different programming languages, or generated by various simulation tools, and interacting together to form a cohesive simulation environment. Co-simulating is often required because simulation designers need to integrate components developed by different teams, or because individual components are best developed with a specific programming language or tool [1]. For example, Matlab/Simulink™, Dymola™ or MATRIXx™ are widely used commercial tools for model-based design, while embedded code is often written in C or C++. While co-simulations present advantages, such as encouraging code re-use, they are usually large and complex. As a result, high performance, scalable and real-time simulations are difficult to achieve. This paper presents a software framework for heterogeneous co-simulations, intended to work under real-time constraints.

The system, named RT-LAB Orchestra, provides a communication layer that sits on top of the RT-LAB framework, a real-time infrastructure for distributed simulations [2], which typically allows simulation time steps under 10 us. RT-LAB takes advantage of the considerable computing power available through clusters of

affordable PCs, connected by high-performance communication links such as InfiniBand [3]. Using the connectivity features provided by RT-LAB Orchestra, heterogeneous software components exchange simulation data with an RT-LAB framework and hence benefit from the array of facilities available through the RT-LAB infrastructure, such as multi-rate capabilities or extensive support for hardware in the loop [2].

The rest of this paper is structured as follows. In the next section, related work is presented and compared to RT-LAB Orchestra. Then Orchestra's software architecture is described, and its advantages discussed. As an application of RT-LAB Orchestra's connectivity features, a distributed real-time simuLation of Unmanned Aerial Vehicles (UAVs) has been carried out. Simulation results are presented that illustrate a data exchange between an RT-LAB framework and C-code software processes. These data demonstrate the dynamic connection management of Orchestra, and its usefulness in terms of rapid prototyping and scalability.

2. RELATED WORK

Technologies based on the Data Distribution Services for Real-Time Systems (DDS) are built on similar concepts to RT-LAB Orchestra. DDS is a specification written by the Object Management Group (OMG) that provides a standardized way of developing distributed real-time applications [4]. DDS provides an Application Programming Interface (API) that developers can use to build a distributed application, without defining how the communication between distributed nodes is implemented. DDS relies on the OMG Interface Definition Language (IDL) to define Quality of Service (QoS) parameters that correspond to various requirements attached to the data exchanged within a distributed system. IDL's syntax is similar to C++ [4].
The DDS API uses a publish/subscribe mechanism, data-centric communication system. DDS publishers and subscribers are distributed objects that notify readers and writers of the availability of data. The data is accessed through a data space, or domain, and readers/writers objects that are registered within a domain are referred to as domain participants. The publish/subscribe approach is preferred to the more classic client/server mechanism, such as the one used by CORBA, because it has no centralized server, therefore no performance bottleneck. Also this mechanism has no single point of failure, and it potentially displays better scalability [5]. ORTE (OCERA Real Time Ethernet) is an open-source implementation of a real-time communication protocol built on top of an UDP stack that is also based on a publish/subscribe mechanism [6].

Orchestra is based on a data-centric simulation data exchange comparable to the mechanism defined by the DDS. However, in developing RT-LAB Orchestra our intent was not to implement an abstract specification such as DDS, but rather to enhance an existing and proven real-time framework, RT-LAB, by adding connectivity features that open up the framework to heterogeneous software components.

3. ORCHESTRA'S ARCHITECTURE

The core of Orchestra's architecture is a configurable communication layer that sits on top of the RTLAB framework, whose role is to provide a transport layer between distributed simulation nodes, and a real-time scheduler for the co-simulation, see Figure 1. The communication layer consists of a set of shared-memory segments, one per domain. As a result, domain participants are co-located within the same RT-LAB simulation node.
Domain participants exchange simulation data via the communication layer by calling functions of the Orchestra RTAPI, described below. Orchestra distinguishes two types of domain participants, namely the RT-LAB framework itself, and external components. Logically, an external component is a software process that is not part of the Simulink-based model that the RTLAB framework instantiates for real-time execution. Physically, an external component is a cohesive software entity that embeds calls to the RTAPI, and that is compiled and linked to form a stand-alone process. A single domain participant acts as a reader or a writer by using the RTAPI to send or receive data respectively. Orchestra is designed to be extensible; this is accomplished by clearly decoupling the RTAPI from its implementation on the RT-LAB framework side. As a result, different domains could possibly rely on separate implementations of the communication layer. For example, we could envision that the communication layer for a given domain be implemented as an High

Level Architecture (HLA) federate that exchanges data through an HLA Runtime Time Infrastructure (RTI) [7].

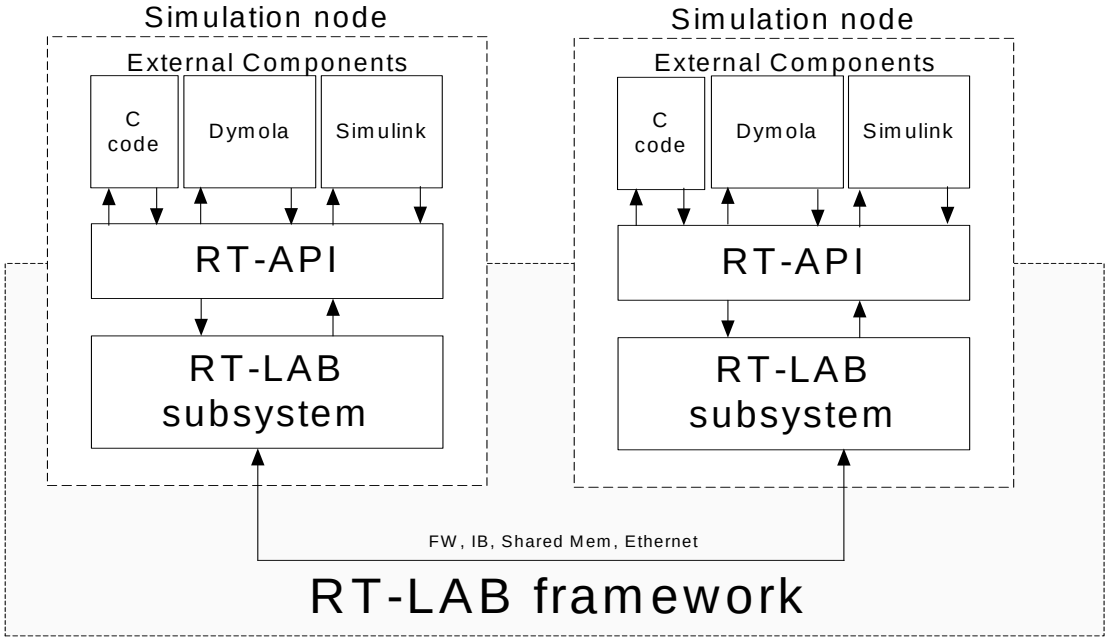

Figure 1. RT-LAB Orchestra's architecture.

4. COMMUNICATION LAYER CONFIGURATION

The Orchestra communication layer is configurable offline through a Data Description File (DDF). A DDF is an eXtended Markup Language (XML) decomposed into a four-level hierarchy. Part of a sample DDF is shown in Figure 2.

```
<?xml version="1.0" standalone="no" ?>
<orchestra>
<domain name="multiuav_uav1" >
      <synchronous>yes</synchronous>
      <multiplePublishAllowed>no</multiplePublishAllowed>
      <states>no</states>
      <block name ="multiuav_uav1/Publish">
            <item name="Autopilot">
            <type>double</type>
            </item>
            <item name="X">
            <type>double</type>
            </item>
...
      </block>
      <block name ="multiuav_uav1/Subscribe">
            <item name="Latitude">
            <type>double</type>
            </item>
            <item name="Longitude">
            <type>double</type>
            </item>
            <item name="Elevation">
            <type>double</type>
            </item>
...
      </block>
</domain>
```

Figure 2. Section of a sample Data Description File.

The top element is the <orchestra> tag; this element serves as a place holder that includes all the simulation data for a given co-simulation. The next level down from the top element contains a list of <domain> elements. An Orchestra domain is a named set of uniquely named data items to be exchanged between participants, which is similar to the DDS concept of data space. The <domain> element contains various QoS elements that define the connection policies applied by domain participants. The QoS parameters corresponding to these elements are:

- Synchronicity

This parameter can be either synchronous or asynchronous. The asynchronous mode is similar to the bucket mode described in [1].

- Writer Access Exclusivity

This parameter determines whether several publishers can read to the same domain. Any number of domain participants can subscribe to a given domain at the same time.

- Writer Access Priority

This parameter determines whether an external component needs to seed the data exchange within a given domain, by publishing to the domain first.

Other QoS parameters, such as deadline policies, are configured through the Simulink-based RT-LAB framework. For example, a deadline policy is provided by the RT-LAB framework that specifies the rate of execution for a given simulation.

Each <domain> also contains a reader and a writer block, which in turn contain a list of named data items. The publisher/subscriber role is defined vis-à-vis the RTLAB framework, namely: data items in a publisher block are written to a domain by the RT-LAB framework, and therefore available for reading by an external participant to the domain. Conversely, data items in a subscriber block are written by an external participant, and read by the RTLAB framework. Data items are named identifiers that map to numerical variables exchanged within a domain.
Users not familiar with editing XML have the option of using a graphical user interface provided as part of RT-LAB Orchestra to create a DDF. Figure 3 shows one of the panels of this graphical user interface.

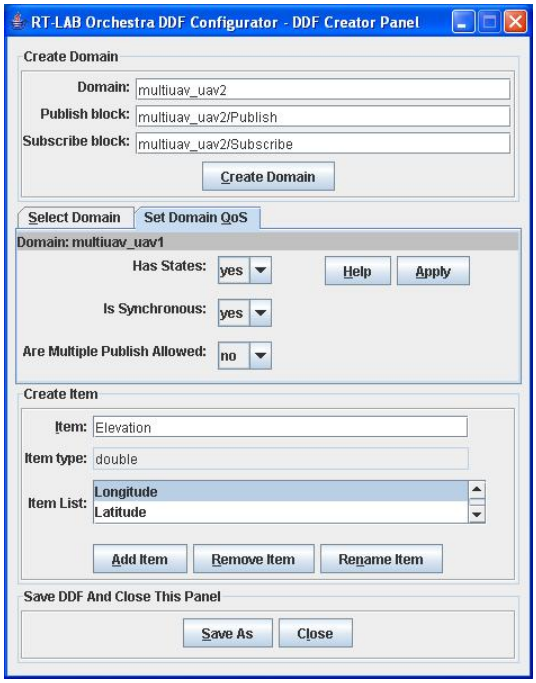

Figure 3. Graphical user interface used to create an Orchestra DDF.

5. COMMUNICATION LAYER ACCESS

External components participate to a domain data exchange by embedding calls to the RTAPI, a set of C-code functions distributed as part of Orchestra. The data exchange is a data-centered publish/subscribe mechanism, which means that the data are referred to using named items, without the need to specify another domain participant explicitly. This mechanism allows external components to send or receive data that are part of a domain, regardless of which component is actually connected to the RTLAB framework. Simulation data are described in the DDF processed by an RTLAB component at simulation startup.
Prior to exchanging data, a connection to a domain must be established by calling RTConnect(). Disconnection from a domain is achieved by calling the function RTDisconnect(). Data items are published (sent) and subscribed to (received) by calling RTPublish() and RTSubscribe() respectively, referring to data items by name. The data exchange within a domain can be synchronous or asynchronous, as specified by the QoS defined in the DDF. Synchronization is implemented through a producer/consumer locking mechanism.

6. INTEGRATION WITH THE RT-LAB TRANSPORT LAYER

RT-LAB users partition a Simulink [8] model to form subsystems, then assign subsystems to simulation nodes using a graphical user interface. The Orchestra communication layer is integrated to the RT-LAB framework through a library of Simulink blocks that are dragged and dropped into the Simulink model. These blocks are referred to as proxy blocks because the RT-LAB framework uses them as substitutes for external components. For example, to permit the connection of a C-code process to the RT-LAB framework, a C-code Simulink block is added to the model. Dymola blocks are currently part of this library. Extending Orchestra's connectivity to other simulation tools simply amounts to adding the corresponding blocks to the Orchestra library. Once the path to a DDF is defined, and a domain chosen, the data items inside the domain are mapped to the inputs and outputs of the proxy block.
A model is compiled and loaded onto a real-time operating system such as QNX [9] or Linux RedHawk [10]. During model initialization, Orchestra validates the syntax and the semantics of the DDF, then shared-memory segments are allocated for every domain defined in the DDF.

7. UAV CO-SIMULATION

We have developed a multi-UAV co-simulation as a sample application that takes advantage of Orchestra's connectivity features. Many applications are possible, but this is a good example of a research area that benefits from a system such as Orchestra because autonomous

UAVs are often flown in formation, so distributed architectures are well suited to studying their interactions [11]. Also, fleets of UAVs require real-time operating mode because of the constraints resulting from the wide range of priorities in the tasks they should perform [12, 13].

This multi-UAV co-simulation consists of 4 autonomous RT-LAB UAV models that send position data to an RT-LAB framework. In addition, C-code software processes, henceforth "external code", embeds calls to the RTAPI and connect to the RT-LAB framework to obtain samples of the UAV positions. Each UAV block implementation is based on the Pioneer UAV model developed using AeroLib, a library of Simulink blocks developed by Opal-RT. The Pioneer UAV model uses nine subsystems from the aircraft dynamics, the main subsystems are listed below with a brief description for each:

Aerodynamic Subsystem. This subsystem calculates the aerodynamic forces and moments acting on the aircraft, expressed in the body-fixed reference frame, based on the flight condition, aerodynamic coefficients and control inputs.

Autopilot Subsystem. The Autopilot Subsystem consists of a flight path tracking system and a damper system. The flight path tracking system tracks a series of waypoints defined by latitude, longitude and altitude at predefined airspeeds. A guidance system converts the waypoints to a desired bearing, altitude and cruise speed, and PID controllers ensure that the appropriate control inputs are generated to ensure tracking.

Equations of Motion Subsystem. This subsystem implements the 6-DOF flight dynamic equations of motion for the aircraft, based on the forces and moments calculated by the Aerodynamic Subsystem, the thrust calculated by the Piston Engine Propeller Subsystem and the mass properties of the aircraft.

Simple Controls Subsystem. This subsystem models the actuator dynamics for the control surfaces of the aircraft.

The four UAV models are identical in terms of aircraft characteristics, but are offset in space. For the demo, they follow a flight path defined by a set of waypoints at predefined airspeeds using an autopilot, and always maintain their relative positions. The models are initialized in a trimmed state. A UAV Simulink model is shown in Figure 4.

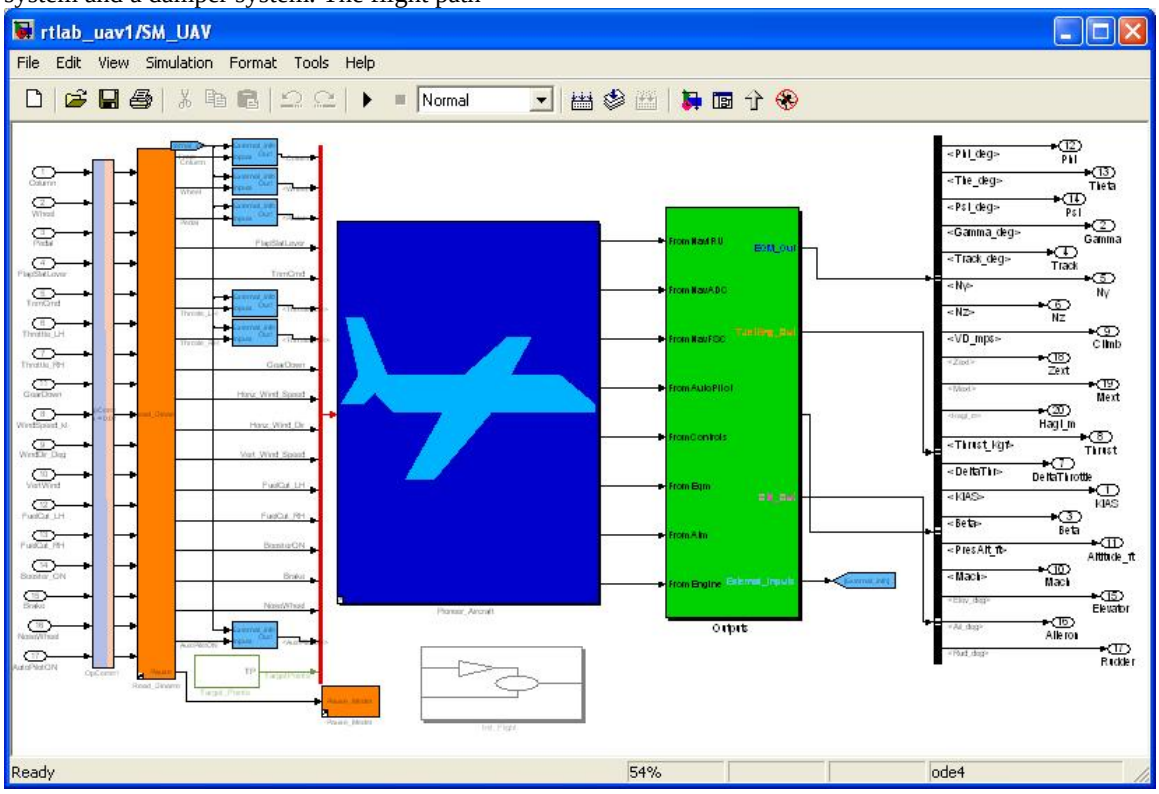

Figure 4. UAV model used in the multi-UAV co-simulation.

The host machine on which the Simulink co-simulation was developed is a PC running Windows XP. Real-time code was generated by the Simulink RTW code generator. The code is then loaded onto a cluster of target PCs that run the real-time Linux RedHawk operating system [10]. Nodes in the clusters communicate through an InfiniBand link. Data are retrieved from the target nodes through a TCP/IP connection, and displayed on the host machine using the X-Plane visualization tool [14]. Dynamic connection/re-connection of components is achieved through a sequence of events automatically sent by running a Python script.

Figure 5 shows the altitude variations of simulated UAVs as a function of time, as well as the effect of connection/disconnection requests sent by the monitoring script to the UAV models. The UAV dotted lines correspond to altitudes computed by the UAV models. The solid lines are the same altitudes published by the UAV models, and read by the RT-LAB framework. The broken and dotted lines diverge when a UAV component sends a disconnection request to the Orchestra communication layer by calling the RTDisconnect() function of the Orchestra RTAPI. As a result, the altitude read by the framework remains level. When a connection event is sent by the monitoring script to the UAV model, the UAV component calls the RTConnect() function of the RTAPI and, as the published altitude varies again, the dotted and solid lines merge. As the connection and disconnection requests are received by the RT-LAB framework, the number of connections is retrieved from Orchestra and is displayed as a real-time parameter of the co-simulation.

Figure 6 illustrates how heterogeneous domain participants exchange simulation data within Orchestra. The solid and dotted lines for the UAV2 are similar to the ones shown in Figure 5.

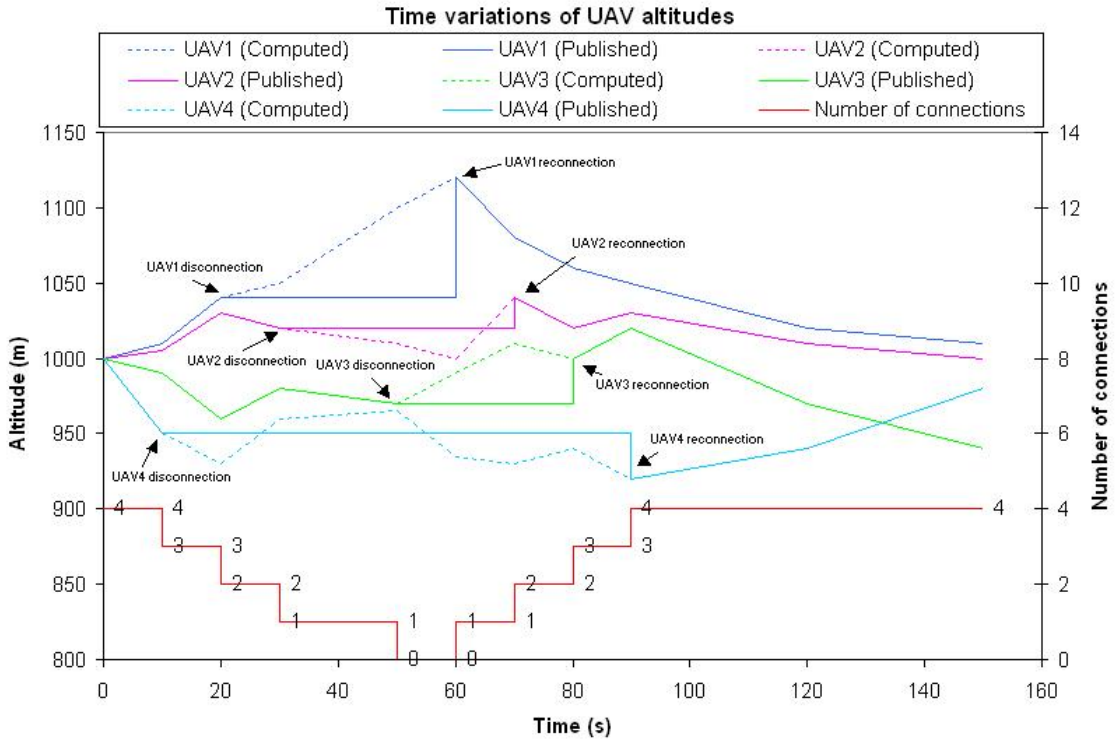

Figure 5. Altitude variations computes by the UAV models as a function of time.

At time 60 s, a C-code external component sends a connection request to Orchestra, and subscribes (reads) the UAV2 altitude from the corresponding domain, “multiuav_uav2”. At time 80 s, another instance of the C-code external component repeats the same sequence. The merged lines indicate that altitudes published by the UAV model are read by both the external components. Figure 6 also shows the number of connected domain participants as a function of time.

8. CONCLUSION

Orchestra is a real-time framework for heterogeneous software co-simulations. Orchestra provides an API based on a publish/subscribe mechanism that heterogeneous components use to send data to and receive data from an RT-LAB framework. An example application demonstrating the connectivity features of RT-LAB Orchestra has been presented, along with its dynamic connection management capabilities. These capabilities are useful for rapid prototyping as faulty components can be removed from a co-simulation, fixed, re-compiled and re-connected without affecting other components.

Future work includes the implementation of QoS policies to handle event-driven synchronization, and support for exchange of data with composite types.

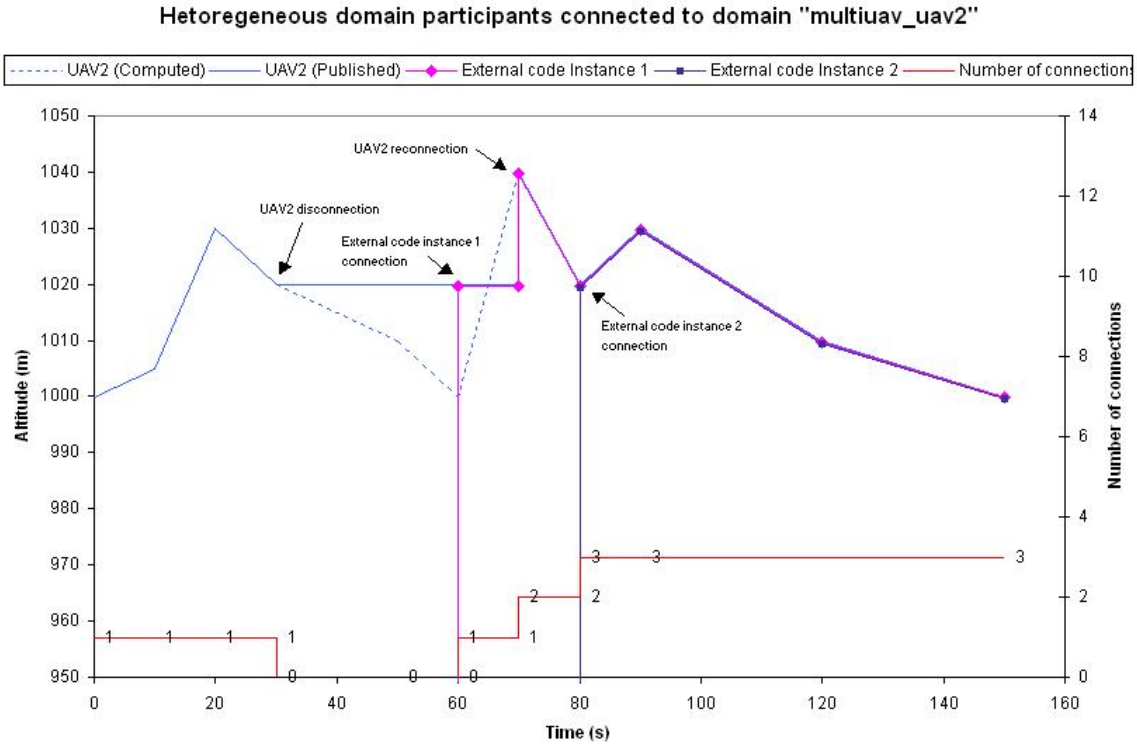

Figure 6. Connection of heterogeneous domain participants to a UAV domain.

ACKNOWLEDGEMENTS

The authors would like to thank Dr. Ravi Venugopal for enhancing the Pionner UAV model so as to permit simulations of in formation flights.

REFERENCES

[1] P. Bjureus and A. Jantsch, "Heterogeneous System-Level Cosimulation with SDL and Matlab, Electronic Chips & System Design Languages", Kluwer Academic Publisher, 2001.
[2] www.opal-rt.com
[3] http://www.infinibandta.org
[4] www.omg.org
[5] P. Eugster, P. Felber, R. Guerraoui, A.-M. Kermarrec, "The Many Faces of Publish/Subscribe", ACM Computing Surveys, Vol.35, No 2, 2003.
[6] http://www.ocera.org
[7] http://www.dmso.org
[8] http://www.mathworks.com
[9] http://www.qnx.com
[10] http://www.ccur.com
[11] P. Doherty, P. Haslum, F. Heintz, T. Mertz, P. Nyblom, T. Person and B. Wingman, "A Distributed Architecture for Autonomous Unmanned Aerial Vehicle Experimentation", Proceedings of the 7th international Symposium on Distributed Autonomous Systems, 2004.
[12] W. Eui Hong, J. Shin Lee, L. Rai and S. Ju Kang, "RT-Linux based Hard Real-Time Software Architecture for Unmanned

Autonomous Helicopters", 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2005.
[13] N. Lechevin, C.A. Rabbath and P. Sicard, "Stable Morphing of Unicycle Formations in Translational Motion", To be published in Proceedings of the American Control Conference, 2006.
[14] http://www.xplane.com