Using Integrative Modeling for Advanced Heterogeneous System Simulation

Tapasya Patki Hussain Al-Helal Jacob Gulotta Jason Hansen Jonathan Sprinkle

Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721-0104
{tpatki,alhelal,jgulotta,jchansen}@Email.Arizona.Edu/sprinkle@ECE.Arizona.Edu

Abstract

This paper is an academic experience report describing the use by researchers at the University of Arizona of a domain-specific language developed by the Institute for Software Integrated Systems (at Vanderbilt University). The domain in question is heterogeneous, distributed simulation of quad-rotor unmanned aerial vehicles (UAVs) as they respond to command and control requests from a human operator. We describe in detail how our individual designs of the controller and guidance laws for the UAV, as well as its rendering and position updates in a 3D environment, its onboard sensors detecting ground features, and the various commands to delegate mission-critical behaviors, all interact using the ISIS-developed modeling language. We then discuss the outlook for this domain (heterogeneous system simulation and integration) for domain-specific languages and models, specifically for unmanned vehicle control and interaction.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Integrated Environments

General Terms Domain-specific modeling languages, Heterogenous system simulation

Keywords HLA, RTI, GME, Command and Control (C2)

1. Introduction

Large projects with decentralized development face a critical issue in holistic system simulations. Maintaining a single simulation strategy, which may even include the use of proprietary tools and/or shared network drives, is quite difficult to achieve, and can lead to poor software engineering practices where elements are developed outside the simulation toolchain. These elements must be rewritten or adapted to fit inside the tools used by the project. Such practices are prone to problems that are subtle, such as mismatched models of computation, as well as problems that are widespread, such as software bugs while porting.

Many systems require development and design in proprietary tools (e.g., MATLAB/Simulink for the domain of control systems), and may take advantage of sophisticated models of computation available in such tools. Other portions of the system may depend on logic that is best expressed as a switch statement in C/Java, or may be run as an applet (e.g., human control through a command and control interface). How to integrate these portions of the system with various components written in other languages is best done through middleware, and many standard middlewares exist for such applications. However, for an expert in control, or discrete event simulation, middleware programming can be a treacherous and confusing addition to their own algorithms.

The simulation of these systems built with heterogenous tools, components, models of computation, and operating systems is a nontrivial task that is best tackled by a middleware expert. However, there exists the bootstrapping issue of confirming that all programmers for each component follow a styleguide, or include standard header files with standard object definitions. Enforcing such a styleguide early in the process can often lead to the 'chicken-and-egg' problem where experts cannot start working on their algorithms because they do not have a testing infrastructure, while infrastructure developers cannot develop the middleware because they do not have a set of algorithms to design around.

To address this issue for the specific domain of multi-vehicle command and control (C2), Balogh et al. [2008, Balogh] developed the HLA paradigm. Starting with a suite of tools that could utilize the infrastructure, and with a few examples, we began an experiment to continue advanced implementation of interactions between components, with the intention of integrating the components into an advanced demo that drew from many different simulation, design, and visualization tools. Importantly, we were able to do our component designs and simulations *independently* of the anticipated middleware, infrastructure, and global simulation strategy. Although it was known *a priori* that HLA was the likely candidate, this strategy enabled users to operate without that assumption[1]. Integration of these various components was somewhat trivial, which is a great result for the domain-specific modeling language, as it reduced the complexity of the expert developers significantly.

The scope of this paper does not include motivating the development of this HLA modeling language, nor a detailed description of the HLA middleware used. Readers interested in these details can refer to Balogh et al. [2008, Balogh][2]. In fact, there were many design choices and application domain choices made by the authors of the domain-specific language we use in this paper: we do not justify or motivate their work, but instead present this application example, which shows the tremendous amount of heterogeneous simulation, design, and rendering, which the use of this domain permitted in the period of just three months. For this paper's scope, we are most interested in the following qualities of a modeling language:

- the ability to specify tool-independent data structures;
- the ability to compose data structures with other data structures;
- the ability to synthesize "glue code" between various tools and software architectures;
- the ability to prototype component behaviors without running middleware as part of the test;

[1] In fact, early application domain choices utilized the ICE middleware by ZeroC.

[2] The maturity of the project and the short timeline for this workshop do not permit an in-print citation of the work.

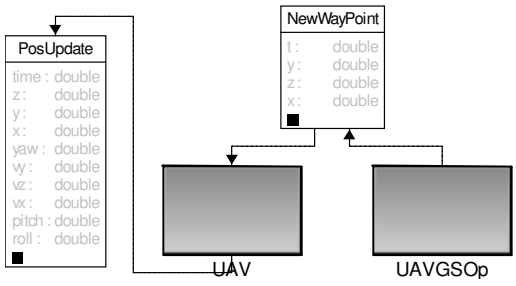


Figure 1. GME Model in the HLA paradigm, which describes an interaction between the `UAV` and the `UAVGSOp`. Note that `New Waypoint` information is passed to the `UAV` from the ground station, and the `UAV` publishes its position for interested parties. The object-oriented structure of those messages are present in the UML class diagram representation.

- the ability to use existing domain-specific tools and environments for design of models, and re-use those models in those tools at runtime; and
- the ability to have a single, unifying modeling language that permits all of the above.

In this paper, we describe our experience with using this domain-specific modeling language, specifically with its advancement of our design and simulation agenda from the perspective of "what would we have to do if we did not have the modeling language to help?" We first describe the tools which were at our disposal for design, simulation, and visualization. We next discuss the various implementations of component functionality. Finally, we describe the integrated demonstration, and how we envision our future work based on the capabilities of this modeling environment.

2. Modeling Language Description

The Generic Modeling Enviroment (GME) [Ledeczi et al. 2001] developed by the Institute for Software Integrated Systems (ISIS) at Vanderbilt University is a configurable modeling environment with a graphical interface. The basis of GME is Model Integrated Computing (MIC) [Ledeczi et al. 2005], which facilitates the creation and evolution of embedded software systems that are domain-specific. MIC gives us an end-to-end view of the entire system and lets us capture an entire class of applications. The Multi-Graph Architecture (MGA) supports *meta-level* processes and *meta-modeling*, and is an executable framework for MIC. A meta-level process is used to specify, define and validate domain-specific environments. These environments are then used to obtain metamodels, that let us specify the various properties of the *domain models*, like the concepts, relationships, compositions and integrity constraints. Metamodels are used to generate *paradigms*, that can be deployed to create applications.

The project required the integration of various heterogeneous tools. Specifically, we needed the ability to use

- MATLAB/Simulink
- OMNeT++ [Varga 2001]
- CPN Tools [Jensen et al. 2007]
- DEVSJAVA [Palaniappan et al. 2006]
- 3D-Viewers like Delta3D and Blender.

To facility communication between these tools, the High Level Architecture (HLA) [IEEE-HLA-1516 2000] was used to handle interoperability and synchronization issues. HLA is an object-based, DoD/IEEE standard that facilitates the building of simulation systems from heterogenous components, permits component reuse, and supports interoperability through publish/subscribe mechanisms. It also encourages distributed and multi-platform computing. An elementary HLA-compliant program is called a *federate*, and a set of interacting federates is referred to as a *federation*. The exchange of data among federates is supported via the Runtime Infrastructure (RTI). The key functions of the RTI involve Federation Management, Data Distribution Management and Time Management. To support these functionalities, various toolkits like the MATLAB HLA Toolbox, Open-HLA, pRTI and Portico are available. Authors used the Portico 8.0 environment to implement the RTI. Portico is a flexible, open-source, cross-platform implementation of the RTI (see `http://www.porticoproject.org/`).

The HLA paradigm, developed by ISIS at Vanderbilt University, was used to create various models in GME, including a particular interaction model shown in Figure 1. The federates corresponding to the `UAV` and the `Ground Station Operator` are components developed in *any* tool that the paradigm supports (in our case, MATLAB/Simulink for `UAV`, and Java for `UAVGSOp`). In this case, the `UAV` publishes updates to its position, but the `UAVGSOp` can send new commands via the `NewWayPoint` message, which will affect the direction in which the `UAV` flies.

These domain models specify the fundamental structure of the interconnection of these components, and the messages they are allowed to send. Model interpreters specify a mapping between this structure and the RTI infrastructure, and various application components, in the following ways. Although their detailed description is out of the scope of this paper, the HLA paradigm has two model interpreters—the `C2WInterpreter` and the `CPNInterpreter`. The first of these interpreters produces the required federate files that are used by the RTI. These files include the names of various components, the data structures expected to be shared, etc. The second, (`CPNInterpreter`), imports information from a CPNTools model to obtain the discrete events expected from that tool, for use in integrating the overall demonstration. These discrete events are then used in the HLA model to specify which events are expected to be received from which components, and consequences of their receipt (i.e., actions). This permits the specification of complex decision-making processes in an existing tool (CPNTools), and the sending of various messages to computational components throughout the model.

The HLA paradigm provides the following abilities:

- layout of interaction models between various components;
- middleware-independent specification of data structures for messaging; and
- specification of runtime parameters for the overall simulation.

Recall from Figure 1 that a ground station interacts with a UAV. The data in that diagram, namely `NewWayPoint`, is fully specified in a larger diagram. The Object and the Interaction Diagram, as modeled in the HLA paradigm, are shown in Figure 2. The notations used are similar to those in standard UML, but the language definition has the additional semantics that their interconnection will denote specific structure in the federate synthesis.

3. Rendering

The aim of a simulation is to accurately replicate a real-life or hypothetical scenario, including its visualization. Tools that permit the high-fidelity design and simulation of a complex vehicle do not always provide an equivalent high-fidelity rendering of that vehicle, so there exists a need to enhance this visualization through external tools. One such tool is Delta3D (`http://www.delta3d.org/`), an open source gaming and simulation engine which provides a 3D

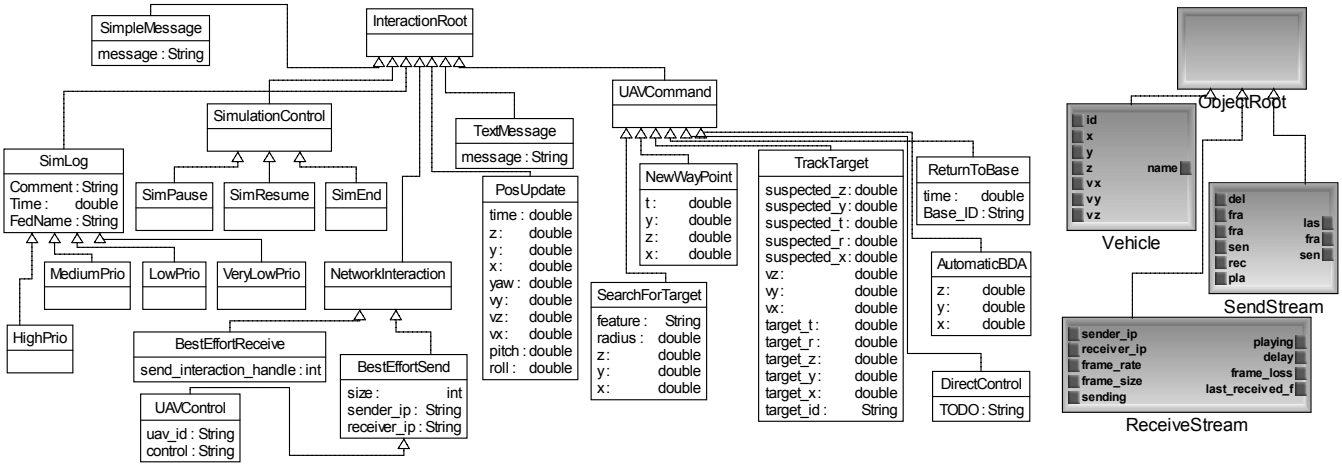


Figure 2. The model of interactions, in the HLA paradigm. Models describing the runtime parameters for simulation, logging, and network interactions are in the left-hand side of the figure. The right hand side concentrates on messages sent back and forth between various components (`UAV`, `Ground Station`, etc.), including the commands used by various components. Finally, the rightmost portion of the figure shows the various objects that are passed by the RTI, including the sending/receiving streams of the network simulator.

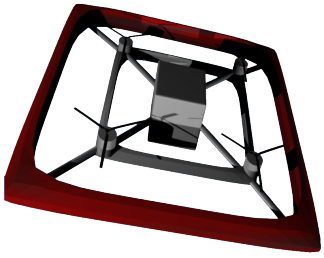
visualization of the virtual world as the simulation executes (refer to the manuscript by Balogh et al. [2008, Balogh]).

Delta3D is packed with numerous sprites which can be used to represent a UAV, such as that in Figure 3(a). This is not an accurate representation of the STARMAC [Hoffmann et al. 2007] that is to be simulated, so a more suitable rendered model was created using Blender (`http://www.blender.org/`), an open source 3D content creation suite used to create and render the 3D model seen in Figure 3(b). This is a much more accurate representation of the STARMAC than the included sprite.

In order to correctly render the UAV, Delta3D requires position (x,y,z), velocity ($\dot{x}$, $\dot{y}$, $\dot{z}$), attitude (ϕ, θ, ψ) and a time stamp ($time$). Since Delta3D can natively interface with HLA, no special GME interpreter is required to generate the glue code for HLA integration. Implementing the HLA Run-Time Infrastructure allows published `PosUpdate` information to be subscribed to by a federate. The Delta3D federate can then be configured to subscribe to `PosUpdate` interactions. HLA allows any source to publish the `PosUpdate` information making development flexible.

Actual image data is not feasible to transfer over the network; network bandwidth causes a large bottleneck. Fortunately, it is possible to send simulated data from Delta3D to look like new data from a sensor. The UAV state data and GPS location corresponding to a target can be published to the RTI. If the UAV is in the `Search Target` State, Delta3D will publish confirmation that the UAV has started its search. The Delta3D federate will constantly publish target position information ($target\_x$, $target\_y$, $target\_z$, $target\_r$, $target\_t$), target velocity ($\dot{x}$, $\dot{y}$, $\dot{z}$), suspected target location ($suspected\_x$, $suspected\_y$, $suspected\_z$, $suspected\_r$, $suspected\_t$) and target id ($target\_id$). These comprise the parameters available for the `TrackTarget` interaction. Section 5 discusses the algorithm for deducing and publishing the target location from a simulated camera.

(a) An example sprite from the catalog, which does not represent our vehicle.



(b) 3D rendering of the STARMAC helicopter, used in Delta3D.

Figure 3. Rendered 3D models that can be used in Delta3D. Note that we are now able to use a sprite that represents our actual vehicle, based on a model developed in the Blender tool, and the actual dynamical model in Simulink.

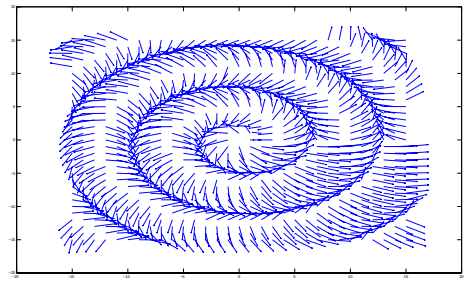
4. Controller

The UAV chosen for this project was the STARMAC, a quadrotor UAV being developed by a group at Stanford University. Both a description of its dynamics and a demonstration of its abilities can be found in [Hoffmann et al. 2007]. For easy visualization and to take advantage of Mathworks design tools (such as the SISO Design Tool), Simulink was preserved as the modeling language for designing the controller.

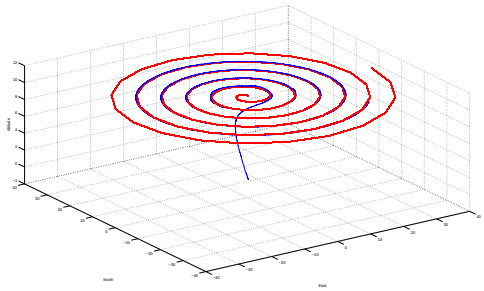
In order to respond to the varying command and control commands of the `Ground Station`, the Simulink controller can switch between various control laws. A top level view of the modified Simulink block diagram can be found in Figure 5. Both the waypoint and spiral search controllers can be seen on the left, with an input flag that specifies which one should be active.

The waypoint controller for the STARMAC was achieved using a 3rd order controller across the motor voltage command and setting up feedback loops around translational acceleration, velocity, and position.

The search algorithm for the UAV was chosen to be in the form of a spiral, originating and spiraling out from whatever location the UAV was at when the flag was switched over. The velocity vector field for the spiral/search guidance algorithm can be found in Figure 4(a). Ideally, the UAV should fly along the spiral. In order to achieve this, the algorithm takes the UAV position as an input, and returns the velocity it would like the UAV to achieve. Given a UAV with a position not on the spiral, the velocity issued is dependent on the distance from the nearest point. The farther the UAV is from the point, the more its velocity will be commanded directly towards that location on the spiral. As the craft nears the spiral, more and more of it's velocity vector is directed tangential to the spiral, until eventually the craft merges with the spiral and tracks it. A typical response can be found in Figure 4(b).



(a) Vector field showing spiral path following.



(b) Dynamic simulation of STARMAC following a spiral path in Simulink.

Figure 4. The controller design, done in Simulink, allows for component-scale simulation and analysis. In (a) the vector field shows the controller's response based on position (this represents the analysis and design phase.). In (b) a simulation of the controller from an initial condition below the spiral's origin is shown (this is the component-scale simulation).

The importance of these plots to this work is that they were the results of design done *inside* the design and simulation toolchain of Mathworks, including the ability to generate plots, utilize summing and feedback functions, and a fundamentally model-based approach. If we had decided to prototype the designs in MATLAB/Simulink and later port them to C/C++, errors could have been introduced either by changing to those languages (subtle errors of models of computation) or typographical errors (explicit errors not easy to find). With the integration of MATLAB/Simulink into the HLA infrastructure, we were able to use the same models developed in design and analysis as executable models.

5. Ground Feature Detection

For unmanned vehicles outfitted with a camera sensor, one potential application is unmanned and autonomous surveillance. The UAV could be given an instruction as, for example, "find the blue trucks in some area." Analyzing the output image from the camera is a crucial operation for the UAV to complete this task. It would have to determine first whether there was a blue truck in its field of view and report the rough location of that truck in some meaningful way, such as GPS coordinates, based on the image and its present state.

In order to closely emulate an actual implementation it is necessary to have a picture. However, simulation in Delta3D has a major drawback with respect to the sensor: the image cannot be directly analyzed. Therefore a software workaround is required to determine whether an object of interest is contained within the image. Taking advantage of the fact that knowledge of the simulation world is absolute, it is possible to bypass the image analysis phase. Instead of locating an object via the camera's image, an algorithm (given the known position of an object) can report where that object would appear in the image, if at all. From there the final task of meaningfully reporting that location is identical as if the pixel location came from an actual picture. Such an approach allows easy transition from canned simulation data to data obtained from analysis.

The process starts by feeding the algorithm the position of an object from Delta3D in x,y,z coordinates. Additional required data are the x,y,z coordinates of the UAV, its roll-pitch-yaw orientation, and the intrinsic parameters of the camera. The most important properties of the camera are its focal length, the size of the CCD, and the resolution. Given these, the algorithm produces the i,j pixel coordinates that represent where in the picture the specified object lies. These coordinates are then adjusted by some noise factor to simulate receiving actual data. Finally the adjusted pixel coordinates are reverse transformed and reported as an approximate location for the object. In practice only the latter portion is necessary because the image will actually be available for analysis.

To accomplish this, Delta3D and MATLAB/Simulink must communicate with one another. Delta3D provides MATLAB the x,y,z coordinates of the object and UAV while Simulink provides the state and orientation data directly to the camera (i.e., not published through the RTI). The camera parameters are fixed and so are simple constants. All the calculations are carried out in MATLAB. An overview of the process can be seen in Figure 6.

6. Integration

Section 2 describes how the GME Environment was used to develop an application model with the help of the HLA paradigm. The `UAV` and the `Ground Station` federates and the associated interactions were discussed, including how the UAV is controlled through Simulink. Target detection was discussed in Section 5, and requires state information of the `UAV` as well as information regarding the target's location.

Now with each of these pieces developed, simulated, and tested individually, we should integrate them into a demonstration. To do this, we follow the overall structure as described in Figure 7. Using Portico as the RTI infrastructure, we use a Java implementation of a `Ground Station` which provides human command input into the simulation. These commands include direction for the `UAV` to search for a target, fly to a waypoint, track a target, etc. The other Java implementation (a work in progress) publishes information about the location of a target, in order for the MATLAB component discussed in Section 5 to publish information about a target being acquired.

These Java-based components express basic control-flow, and also have their own GUI, utilizing Java's user-interface libraries.

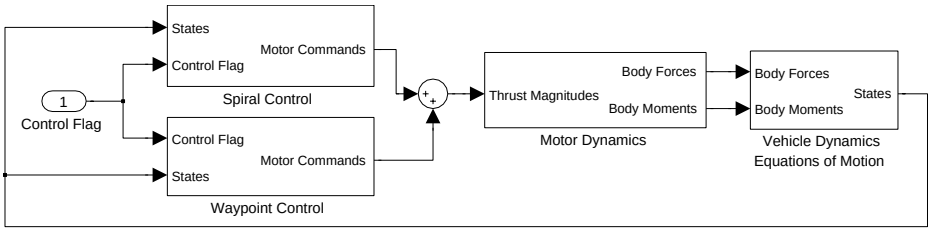


Figure 5. Top-level block diagram of Simulink STARMAC model. Note that the switching controller on the left side of the figure represents the hybrid behavior of the controller, whether the vehicle is flying the spiral search pattern, or the waypoint following pattern.

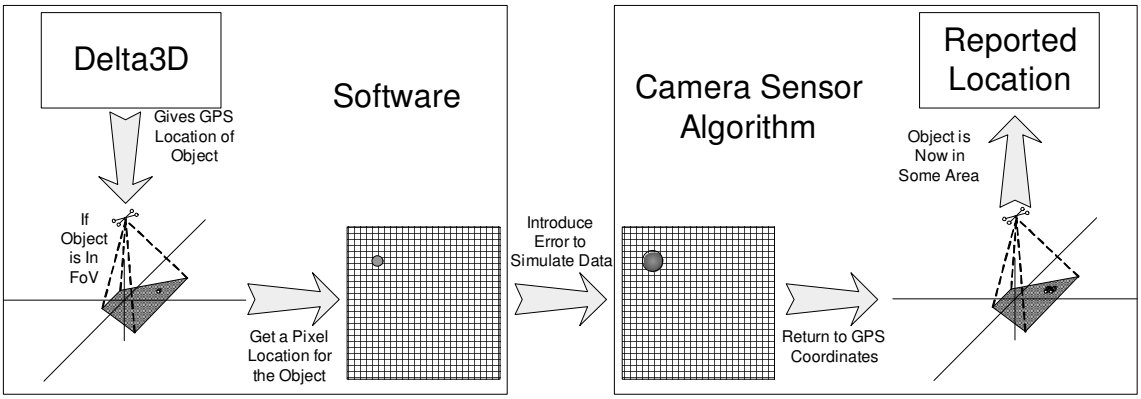


Figure 6. Overview of GPS target identification scheme, given that no image processing is available. If the object is in the field of view (FoV), it is passed along to the targeting component to report location.

For the physical-system simulation, the Simulink models discussed in Section 4 are called, which publish updated position information. This positional information, as well as location of the target, are read by the Delta3D component, and visualized for the benefit of the `Ground Station` human operator. The final result is an

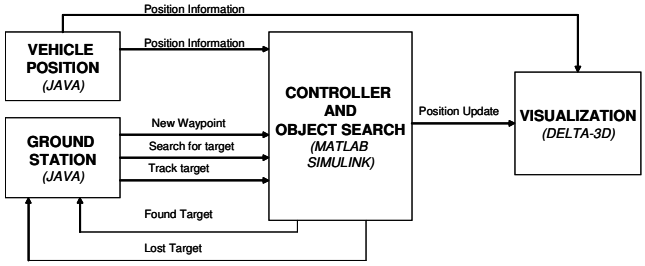


Figure 7. Integrated Schema of the Federation Execution

integrated demo that can be run from a Windows `.bat` file, reducing the possibility of human error in starting up components in the wrong order, or forgetting to pass in parameters. Such an automation for running the demonstration also reduces the effort required to run tests to confirm that certain tools (e.g., MATLAB) are properly integrated into the demonstrator's machine.

7. Results and Analysis

We successfully integrated several demonstrations that showed our various technical contributions. Depending on the number of interactions that we utilized in each demonstration model, about 5000 lines of software were generated for the entire set of federates available. This included the standard "getter" and "setter" methods for various objects, but more importantly the "publish" and "subscribe" methods were provided, reducing the complexity of programming for domain experts. For the Simulink interaction, some hand-editing of the model is required to integrate, i.e., replacing the state reading and writing blocks with HLA reading and writing blocks. This is important not just for information exchange, but also to prevent Simulink from advancing more rapidly than other portions of the simulation, and thus not synchronizing data with other components.

Based on the amount of generated code discussed in Section 7, it would require a *significant* amount of human effort to code the various integration points for each tool. The HLA modeling language provided an integration point for *each* software tool we needed, as well as many others that we did not need. This not only provides a late-stage integration freedom, but also gives a design freedom, where alternative tools can be explored in parallel tested upon integration for selection of the optimal behavior. In addition to the raw effort of programming the interaction points, there is significant effort required to understand *how* the tools could interact with the middleware. Thankfully, this task has already been done by the HLA modeling language designers.

There are, however, several areas in which the tool can be improved. As of now, the integration of components running on different machines is performed through shared drives. This could be improved to use TCP/IP across a network. To mitigate this shortcoming, such integration is currently performed through code generation, so a better integration solution will also be transparent to the users.

Another area for improvement is the integration with MATLAB/Simulink, which currently requires some user editing the MATLAB/Simulink model to include the generated interfaces to HLA. We leave this solution up to the language designers, though

one possible approach is to generate a library of blocks that can be used, and then updates to models in these blocks will automatically update any simulation models.

8. Conclusions and Future Work

In under three months, the authors were able to integrate a new demonstration of C2 behaviors, including new controllers for the quad-rotor vehicle, new commands sent to the vehicles, new models of the demonstration, and summary simulations that verify behavior on a new installation of the infrastructure. These summary simulations are important for a distributed team, as they confirm to other team members that various functional components are behaving correctly, and also confirm to those teams that they can run the simulation tools required.

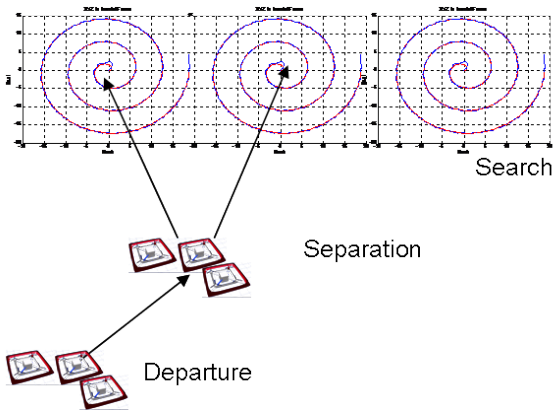


Figure 8. Co-operative Search Operations

Our future work includes development of high-level control algorithms for managing a group of vehicles that co-operatively search for target(s) at a specified location(s). This would be an implementation of mixed-initiative control. The key issues would involve ensuring a stable formation and generating optimal search algorithms. The UAVs would depart as a group in response to a command, and would separate mid-way to perform individual search operations spanning the entire search area, as in Figure 8. Dividing the search space optimally, avoiding collisions and reporting back appropriate information would require the inclusion of intelligent real-time algorithms in the controller. Mesh stability is a good model to obtain a stable formation, as it attenuates disturbances acting on one vehicle as they propagate to other vehicles. Thus the UAVs travel in a mesh. This calls for decentralized control laws and intelligent search strategies.

DSMs present a significant advantage in the high-level specification of system interaction, especially when the generation of the software that produces their interaction (i.e., the "glue-code" that holds an interaction together) is computational, and not a case-by-case design. We believe that future uses of domain-specific modeling environments in this domain will further enable experts in control, visualization, computer vision, etc., to put experiments of system-level simulations together more easily than a brute-force integration strategy.

Acknowledgments

This work would not have been possible without the efforts of Gyorgy Balogh, Himanshu Neema, Harmon Nine, Gabor Karsai, and Janos Sztipanovits of the Institute for Software Integrated Systems. Special thanks are due to Gabe Hoffman, Claire Tomlin, and Hal Tharp for their generous advice in the development of the quadrotor controllers. This work is supported by the Air Force Office of Scientific Research, under award #FA9550-06-1-0267, titled "Human Centric Design Environments for Command and Control Systems: The C2 Wind Tunnel".

References

Gyorgy Balogh, Himanshu Neema, Graham Hemingway, Jeff Green, Brian W. Williams, Janos Sztipanovits, and Gabor Karsai. Rapid Synthesis of HLA-Based Heterogeneous Simulation: A Model-Based Integration Approach. *Unpublished manuscript*, 2008.

Gabriel M. Hoffmann, Haomiao Huang, Steven L. Wasl, and Claire J. Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. In *Proc. AIAA Guidance, Navigation, and Control*, 2007.

IEEE-HLA-1516. IEEE standard for modeling and simulation high level architecture (HLA)-framework and rules. *IEEE Std 1516-2000*, pages i–22, Sep 2000.

K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3–4):213–254, 2007.

A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.

A. Ledeczi, G. Balogh, Z. Molnar, P. Volgyesi, and M. Maroti. Model integrated computing in the large. *Aerospace Conference, 2005 IEEE*, pages 1–8, March 2005.

Sivakumar Palaniappan, Anil Sawhney, and Hessam S. Sarjoughian. Application of the DEVS framework in construction simulation. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 2077–2086. Winter Simulation Conference, 2006. ISBN 1-4244-0501-7.

András Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference*, June 2001.