

HARVEST: A FRAMEWORK AND CO-SIMULATION ENVIRONMENT FOR ANALYZING UNMANNED AERIAL VEHICLE SWARMS

Christopher J. Augeri
Air Force Institute of Technology
Wright-Patterson AFB, OH
chris.augeri@afit.edu

Kevin M. Morris
Air Force Communications Agency
Scott AFB, IL
kevin.morris-03@scott.af.mil

Barry E. Mullins
Air Force Institute of Technology
Wright-Patterson AFB, OH
barry.mullins@afit.edu

ABSTRACT

Unmanned vehicles have the capability to transform military operations. One relatively unexplored application involves cooperative unmanned vehicle systems called sensor swarms. We propose a conceptual unmanned vehicle swarm: a Host of Armed Reconnaissance Vehicles Enabling Surveillance and Targeting (HARVEST). A HARVEST swarm is theoretically capable of autonomous refueling, cooperative search, information fusion, and munitions employment. To enable cooperative swarm capabilities, we identify a set of individual unmanned vehicle services, e.g., localization, querying, and routing.

The HARVEST concept, swarm capabilities, and unmanned vehicle services are embodied in our sensor swarm co-simulation environment. The goal is to improve simulation fidelity by integrating existing simulators used within the Department of Defense (DoD). The process of integrating multiple simulations is known as co-simulation; our design uses OPNET's External System Definition (ESD) to achieve co-simulation. This is the same interface OPNET provides to enable co-simulations based on the High-Level Architecture (HLA) defined by the Defense Modeling and Simulation Office (DMSO).

The simulators in the first sensor swarm co-simulation prototype are based on the technology behind NETWARS (OPNET) and the Java programming language. To integrate with OPNET, Java wraps the ESD C-based method calls. This co-simulation is known as a Java, OPNET, and C-Based Co-Simulation (JOCosim). The implementation details of and lessons learned from the first JOCosim prototype are described in this paper.

We also briefly discuss the second JOCosim prototype currently under development. The newer version places all simulation control within Java versus just receiving data from OPNET. This capability is crucial to achieving the goal of the second prototype — integrating additional simulation tools such as MATLAB, FalconView, and Digital Terrain Elevation Data (DTED).

This work was partially supported by the Air Force Communications Agency (AFCA). The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force (USAF), Department of Defense (DoD), or the U.S. Government.

INTRODUCTION

Our institution invests a considerable portion of its research activities in sensor networks and unmanned aerial vehicles (UAVs). Given that UAV swarms are not readily available, much of our research consists of theoretical analysis and extensive simulation scenarios. Although we are focused on UAV swarms, some of our results may be applicable to other unmanned vehicle swarms.

This paper presents three recent results of our research: a conceptual UAV swarm (HARVEST), a framework for designing a HARVEST, and a co-simulation development platform (JOCosim) for studying a HARVEST swarm. The HARVEST concept and design framework provide methods to describe the networks we are studying and provide guidance on their design. The co-simulation environment, JOCosim, integrates Java with OPNET, the simulation technology enabling NETWARS. We are only aware of one other research effort that even indirectly interfaces Java with OPNET [24].

UAV SWARM CONCEPTS

This section describes a specific type of sensor network, a conceptual UAV swarm. Key assumptions that separate UAV swarms from sensor networks are their mobility, power, and data rate. While a UAV has fuel (battery) remaining, it is flying, collecting significant data, and transmitting near its full power. However, in a typical sensor network, nodes often are significantly less mobile have low duty cycles, and collect limited data.

Given that UAV swarms are a relatively unexplored topic, we developed a conceptual model, a Host of Armed Reconnaissance Vehicles Enabling Surveillance and Targeting (HARVEST). To be economically viable, a swarm is expected to consist of small to medium-size vehicles, e.g., the Bat-Cam or Desert Hawk [19].

The HARVEST responds to both sensed data and user input. User input might include specifying an altitude or flight plan. Core cooperative functions would include localization, area search, and data routing. Potential applications of the HARVEST include target tracking, active decoys, tactical networking, terrain mapping, and environmental hazard plume detection.

The HARVEST proposed here is designed to stimulate research and may border on imaginative; this is intended. A HARVEST is expected to consist of heterogeneous UAVs. Heterogeneity can mean using different payloads on the same platform or distinct airframes. A swarm is expected to provide multi-spectral imagery, climate data, combat capability [27], and network routing. In the absence of GPS, the swarm should have some localization capability. Some vehicles may serve as supply UAVs and carry reserve UAVs to offset attrition. The swarm is also expected to have an autonomous refueling capability [1].

We assume that the HARVEST produces *significantly* more information than it can store or externally transmit. Thus, the HARVEST’s information-producing capability greatly exceeds its transmission and storage capacity. At least some data must be aggregated or discarded over time. To facilitate energy-efficient operation, the UAVs can deactivate various components to conserve energy.

A user can only extract information from a subset of the nodes within the swarm at any one time [23, 30]. In other words, much of the swarm communication is internal to the HARVEST and may be at a lower transmission power to nearby UAVs. An additional novel concept is that one or more UAVs actively rove the HARVEST, hopefully providing graceful service degradation in the event there is a spatial gap or internal routing problem. Information exchanges with the HARVEST theoretically are formatted in XML format and nodes are addressed via IPv6.

One possible HARVEST application involves being used to monitor suspected enemy activity in a nearby area. Historical intelligence knowledge of this area is minimal and a HARVEST is launched to provide more current intelligence. To conserve energy, only passive sensors are initially energized. Upon detection of suspected activity correlated between several UAVs, a ground-based bi-static radar transmitter is activated. Some reserve UAVs detach from support UAVs and activate their radar antennas. High-bandwidth imaging UAVs also activate their cameras and transmitters.

Based on the radar returns, the HARVEST conducts in-network image processing based on the area of interest and image quality. The chosen frames of this multi-source feed are then available via the HARVEST edge nodes. After receiving this video stream, imagery analysts confirm the presence of enemy activity, as does intercepted electronic traffic. The enemy downs several UAVs, but supply UAVs release reserve UAVs to compensate. After operator authorization, some unmanned combat aerial vehicles (UCAVs) deploy munitions and destroy the fleeing enemy — mission accomplished!

HARVEST DESIGN

While many of the individual capabilities envisioned for a HARVEST are available today, there is no known UAV swarm configuration that can function to the integrated level or on the dense scale envisioned for the HARVEST. Due to the complexity of both sensor networks and UAV swarms, previous research suggests that UAVs require an integrated design paradigm [26, 28]. This paradigm is commonly referred to as *cross-layer* design.

This cross-layer paradigm is counter to the traditional engineering model that uses functional decomposition. One example can be found in the traditional OSI network stack, where it has been advocated that network functions should be integrated to achieve optimal performance [26].

Although optimal performance may be achievable using this cross-layer paradigm, it is risky [25]. In particular, this approach fosters proprietary solutions versus standards-based inter-operable components. Second, statistically determining the performance and interaction effects of components would be difficult. Third, correcting faults is greatly complicated due to internal component complexity.

We advocate a paradigm that adheres to the successful traditional network stack (with *minor* modifications) and has been previously suggested in [20]. To implement this new paradigm, the traditional OSI model is expanded via a set of vertical *management policies*. We propose two basic policies, *preservation* and *employment* (Table I).

Table I – HARVEST Design Paradigm

Layers	Services	Policies
Application	User Interaction Swarm Services Vehicle Services	Preserve fuel defense Employ recon Target
Transport	IPv6	
Network	IPv6	
Link	+Timing	

A *preservation* policy encompasses activities related to swarm survival. Preservation examples include updating locations (collision avoidance), minimizing power usage (longevity), and maintaining connectivity (C2). An *employment* policy focuses on doing useful tasks. Examples include area monitoring (data collection) and cooperative tasks (swarming). These policies influence the network layer decisions to achieve HARVEST objectives.

The paradigm of “horizontal execution (*layers*), vertical control (*policies*)” is similar to “centralized command, decentralized execution”. To enable these policies, each layer passes information to the layer above and below it via policy interfaces, an approach suggested in [31]. One example is passing the physical layer packet transmission times to assist application layer localization services.

Although the preservation and employment policies appear simple, they encompass the majority of use cases envisioned. A key issue is that these management policies may have competing goals. For instance, a user may task a HARVEST to minimize energy usage while maximizing area exploration. The user should provide guidance on the relative importance of competing policies. This would enable the system to make autonomous choices about the specific algorithm or protocol to use at a given time.

Our final contribution in this section outlines a set of UAV functions to promote swarm application research (Table II). Our goal is to provide a framework enabling concurrent work by several independent researchers versus a specific design. The Joint Architecture for Unmanned Systems (JAUS) is better suited to engineering specific solutions on a particular set of components [8].

Table II – HARVEST Application Layer

Swarm Services		Vehicle Services	
Aerial Docking	Data Indexing	Collision Avoidance	Payload Tasking
Swarm Services	Cooperative Flight	Telemetry Reporting	Information Sensing
User Interaction			
Swarm Management	Flight Planning	Query Generation	Munitions Deployment

The functions shown in Table II are expanded from the application layer of the HARVEST application layer and the proposed management policies, both shown in Table I. The HARVEST application layer provides for user interaction, swarm services and (unmanned aerial) vehicle services. Each of these categories contain similar, but distinct functional components.

For instance, a *user* may generate a query, but the *swarm* is responsible for indexing the information that is sensed by each *vehicle*. Similarly, the *user* plans a specific mission that the *swarm* cooperatively flies while each *vehicle* performs collision avoidance. The HARVEST policies, preservation and employment, are defined in the swarm management component for user interaction.

Our research framework provides the motivation for our co-simulation environment. Specifically, we desire to leverage existing simulators in use by the USAF and DoD, increase simulation fidelity, and enable several researchers to work independently. We began our co-simulation research with OPNET, the primary network simulator used by DoD. Since OPNET is our core simulator, much of the material in this section is OPNET-specific, although we try to be as general as possible.

Though OPNET is a robust network simulator, those who use it may experience a significant learning curve. Many students at our institution are only available for a period of less than two years and often have little or no previous experience with OPNET. In addition, they often develop new models for their research and rarely extend existing models. Finally, OPNET provides limited support (in our opinion) for mobility and application-layer protocol simulations.

For instance, although nodes can be mobile, node movement does not mirror a flight simulator’s realism. Another issue is that node trajectory cannot be changed during simulation. If a speed/bearing trajectory (vector) method is used, as in JOCosim I, the *XY* coordinates are not able to be modified. For this reason, JOCosim II and later versions use *XY* coordinate updates to simulate UAV movement [21]. The only disadvantage is the update frequency required for network fidelity, as OPNET requests a position update *every* packet transmission.

Given our long-term goal to integrate with a myriad of additional simulation engines [2], a key design decision was selecting the core co-simulation environment. OPNET is written in C and provides a custom version, Proto-C. OPNET’s co-simulation library is also in C. Since we desired an object-oriented language, we could have selected C++. However, we chose Java for its robust network, threading, and graphics capabilities — thus was born our Java, OPNET, and C co-simulation, JOCosim.

JOCOSIM I

This section describes our experience with implementing JOCosim I using OPNET 10.5 and Java 5.0 [3]. Our source material is a set of co-simulation tutorials presented at past OPNETWORK conferences [15–18]. The basic tutorial provided a C-based co-simulation with co-simulation execution control provided via standard input (STDIN) [15]. Our goal was to extend this tutorial to control and display the position of a single OPNET node from Java. In particular, we sought to demonstrate that data could be exchanged between Java and OPNET, given the goal of coordinating co-simulation control in Java.

The first core technology that merits attention is OPNET’s external system (ES) implementation [11–12]. First, *every* node model can contain *one or more* external system (ESYS) process models. Once added, every ESYS process model must have an external system definition (ESD). If the ESD file is not specified, the simulation can be run within OPNET, but no co-simulation activity can occur. It is worth noting that having multiple ESYS modules can *significantly* increase both the amount of source code and the end complexity of the co-simulation.

The ESD file contains a binary version of each external system interface (ESI) that is defined. The user also specifies a system definition (SD) text file that defines whether OPNET or an external executable will control the co-simulation. The SD file *optionally* specifies any external link libraries OPNET will require. When OPNET interacts with the ESIs given in the ESD, it does so via `op_esys*` method calls. When an external simulator interacts with the ESIs given in the ESD, it does so via external system access (ESA) `Esa_*` method calls.

Our external C code then interacts with Java via Sun’s Java Native Interface (JNI)¹ [4]. JNI provides the ability for Java to link with pre-compiled libraries written in other programming languages, e.g., C. Typical reasons for using JNI are if significant investment has been made in an existing library or if execution speed is a concern. In our case, we needed to access OPNET libraries.

Building a JNI application is reasonably complex [20]. First, the source files must be constructed in both Java and C. The external method stubs are first specified in Java and then the Java source is compiled. Method signatures that are called from C are extracted via `javap` for use in `GetStaticMethodID` calls (Figure 1) [5]. Then, the `javah` utility is used to extract an obfuscated C header (Figure 2) for method calls from Java. The C source code is compiled (with the `javah` header) to a dynamic link library (DLL). The JNI obfuscation specification is difficult to implement manually; we strongly recommend developers use the Java `javap` and `javah` utilities to determine the Java and C method signatures.

We constructed a second C library that served as our pseudo-random number generator (PRNG) *and* allowed us to test calling C *from* Java. To periodically provide a speed and bearing to OPNET, the controller queried the Java Virtual Machine (JVM) for a random value. The JVM then calls the PRNG library for a random value and returns it to the controller (Figure 1). We also used Java’s `sleep` method by passing a sleep duration to Java, effectively pausing the C thread of execution (Figure 2).

¹ JNI can be pronounced as “{JAIN, JIN, JEN}-ee”. It is *not* Sun’s Jini™ (Java network technology). Jini™ is a pun on genie (“JEEN-ee”).

```

Java: Define Method
    public static void xSimSleep(int ms)...

Java: Compile & Extract Signatures (javac & javap -s callbacks >
callbacks.s)
    public static void xSimSleep(int);
    Signature: (I)V

C: Find Method (from callbacks.s)
    jMIDs[JVM_SIM_SLEEP] =
    (*jEnv)->GetStaticMethodID
    (jEnv, jCls, "xSimSleep", "(I)V");

C: Call Method
    (*jEnv)->CallStaticVoidMethod
    (jEnv, jCls, jMIDs[JVM_SIM_SLEEP], 5000);

```

Figure 1 – Calling Java from C

```

Java: Define Method Stub
    private static native void cSetSeed(int seed);

Java: Compile & Extract C Header (javac & javah callbacks)
    JNIEXPORT void JNICALL
    Java_Callbacks_cSetSeed(JNIEnv *, jclass, jint);

C: Define Method (from callbacks.h)
    JNIEXPORT void JNICALL Java_Callbacks_cSetSeed
    (JNIEnv *, jclass, jint);

C: Compile Dynamic Link Library (DLL)

Java: Call Method
    private static native void cSetSeed(int s);

```

Figure 2 – Calling C from Java

Our ESD specification for our single ESYS module included with our modified `manet_station_adv` node model is shown in Figure 3. `DURATION` and `SEED` were the execution time and global seed used when initiating a simulation run. The `xPsn/yPsn` ESIs were written once per second by the ESYS module. The `speed` and `bearing` ESIs were updated by Java via C to emulate an external flight simulator providing mobility updates. The simulator description associates an SD file (Figure 3).

Name	Type	Direction	Dimension	Comment
<code>esd_DURATION</code>	double	OPNET to Cosim	0	1: Simulation duration time to set the GUI progress bar
<code>esd_SEED</code>	integer	OPNET to Cosim	0	1: Use the same seed in the C/Java co-sim
<code>esd_xPsn</code>	double	OPNET to Cosim	0	1/s: Update GUI tracker
<code>esd_yPsn</code>	double	OPNET to Cosim	0	1/s: Update GUI tracker
<code>esd_speed</code>	double	Cosim to OPNET	0	IRQ: From co-sim
<code>esd_bearing</code>	double	Cosim to OPNET	0	IRQ: From co-sim

Figure 3 – External System Definition (ESD)

After becoming familiar with OPNET’s co-simulation environment and JNI, we had to make two key design decisions: where and how should simulation control occur. Given our time frame and the project’s complexity, we opted to use an external C controller that instantiated both OPNET and a JVM [6–7]. Thus, data is passed in both directions between Java and OPNET via C.

Our JOCosim system design is shown in Figure 4. The C-based controller first initializes OPNET. This task is accomplished via calls to `Esa_Main`, `Esa_Init`, and `Esa_Load`. The C controller loads the ESI specification with `Esa_Interface_Group/Name_Get`. Then, the C controller loads the JVM via `JNI_CreateJavaVM` and locates the desired class with `FindClass`. After this, the controller parses the command-line script that lists the simulation time steps — in our case, one-second intervals. The C controller then calls the JVM with periodic updates.

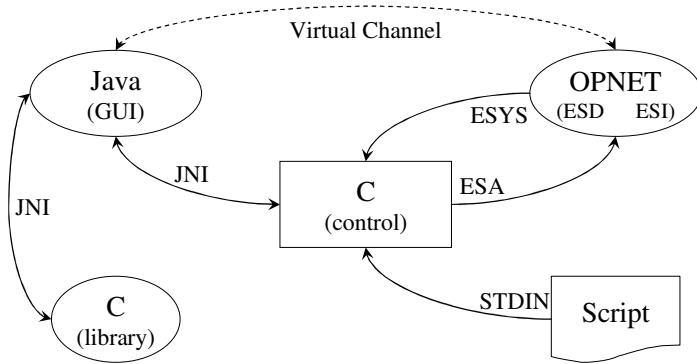


Figure 4 – JOCosim I Design

A functional decomposition of our design is provided in Figure 5. From OPNET’s perspective, although our external code is controlling simulation execution, it exists inside of an ESYS process model that was inserted in the OPNET-provided `manet_station_adv` node model. Our ESYS model is an independent entity that does not modify any of the existing source code, but rather modifies node attributes and state variables. The sequence of the C controller calls is particularly important and not easily determined. The `JOCosim.obj` file (SD block) contains the compiled object code of our C controller.

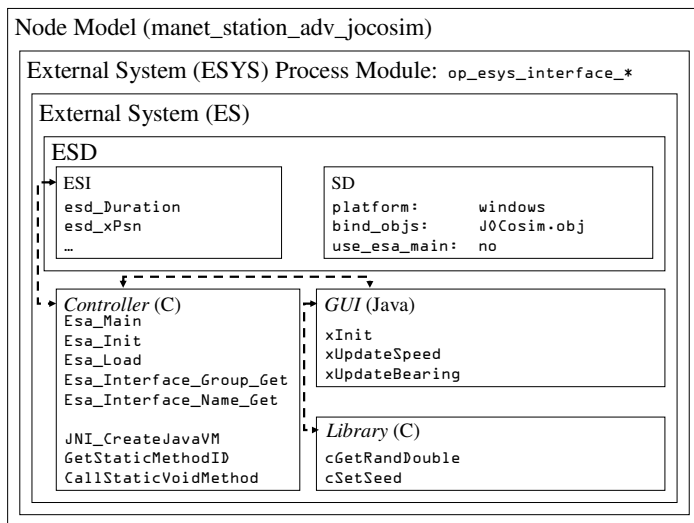


Figure 5 – JOCosim I Functional Decomposition

The JOCosim GUI is shown in Figure 6. It should be noted that the node’s position is being displayed during simulation execution, as confirmed by the progress bar near the top. The speed and bearing that Java generates for OPNET’s use are also displayed. This animation view was compared with the concurrent “live” animation provided by OPNET’s animator, `op_vuanim`. The image in the upper left of Figure 7 is a screen capture of OPNET’s animator. The emphasis in this first model was to verify we could successfully interact with the simulation via Java and emulate a flight simulator. Notably, network activity was left as a task to be completed for JOCosim II [21].

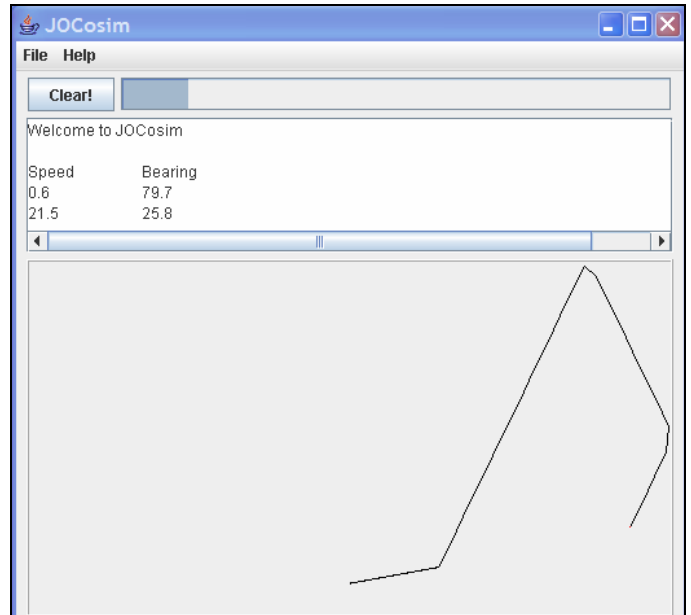


Figure 6 – JOCosim I Screen Capture

The data flow of JOCosim I is described by Figure 7. Although JOCosim I consisted of only one node, the method calls, timing synchronization, and concurrent development of the multi-language source code demanded it. The items in C and Java, e.g., `xUpdateTimer` of the Java block, represent the requisite method call to use. The ESD items and the bottom of the C controller, e.g., `yPsn` and `DURATION`, identify the data being exchanged. The lined items specify when the information is exchanged, e.g., at startup or every second.

The data flow of Figure 7 also captures that the C controller queries Java every 100 seconds for speed and bearing (Figure 4 and Figure 5). Java then queries the second C library for a random number. This value is relayed via the C controller to OPNET on the appropriate ESI. When OPNET receives the ESI interrupt, our interrupt handler computes a new XY coordinate and sets the corresponding attribute on the mobile wireless node, as reflected in the Java and OPNET animators (Figure 6).

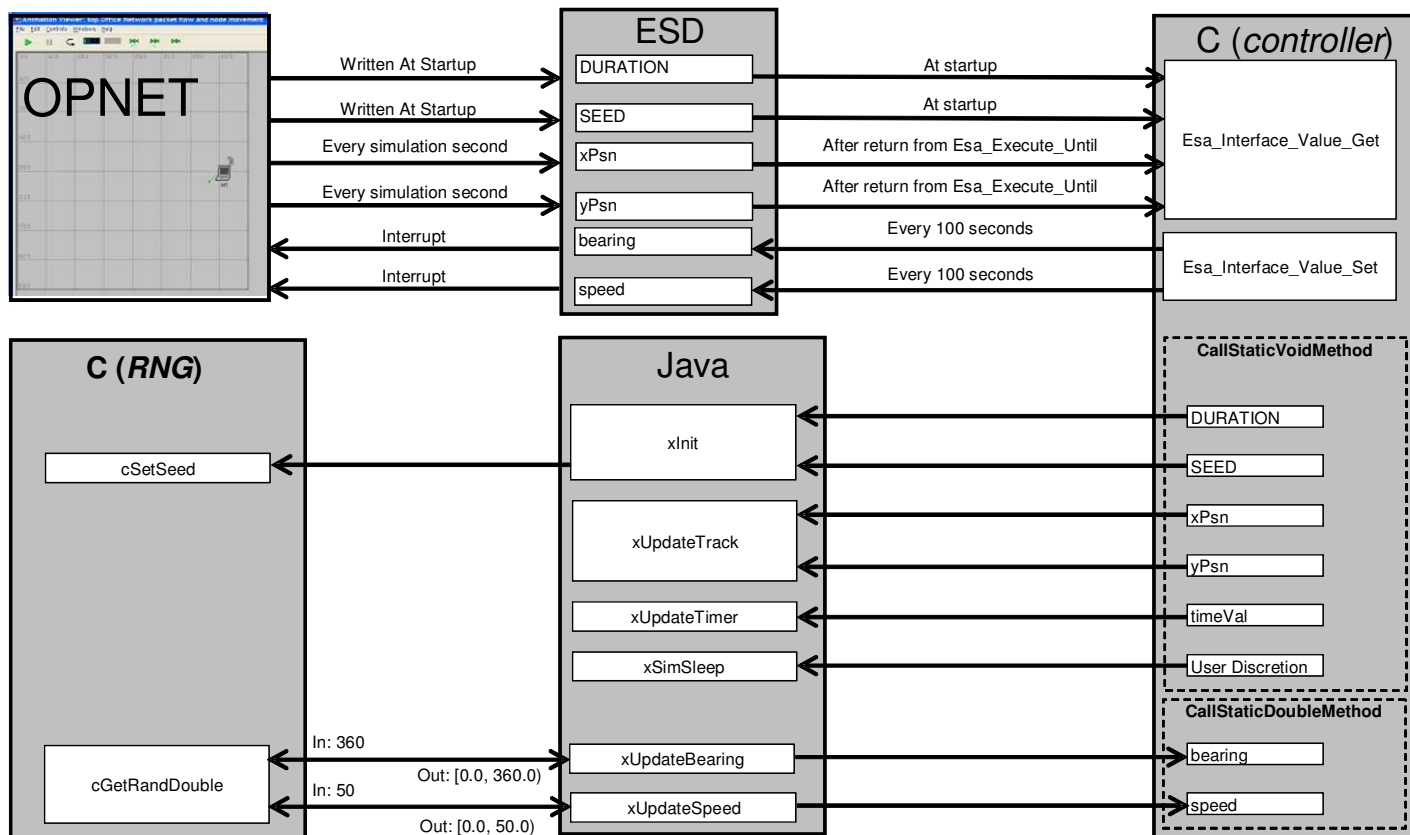


Figure 7 – JOCosim I Data Flow

The ESYS process model’s state transition diagram is shown in Figure 8. At the start of the co-simulation, the `init` state is invoked and the process then advances to the `idle` state. The process remains in or is transitioning back to the `idle` state for the rest of the co-simulation.

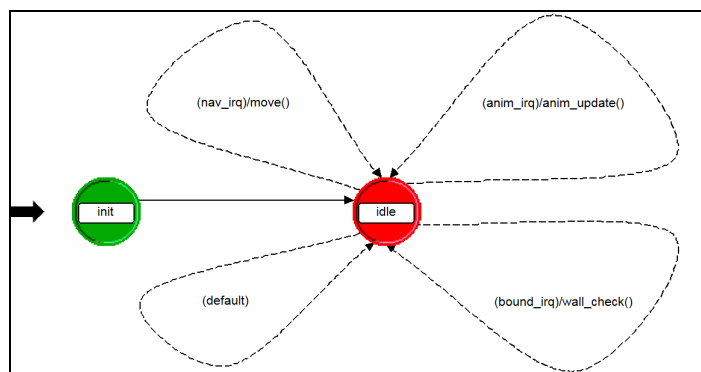


Figure 8 – ESYS Process State Transition Diagram

Each transition from the `idle` state is invoked by an interrupt. The `nav_irq` transition is triggered by the external co-simulation setting an ESYS model’s ESI, e.g., `Esa_Interface_Value_Set`. After the `nav_irq` is triggered, the process calls the `move` method (Figure 9). The `move` method reads the co-simulation ESIs and sets the corresponding node attributes, `speed` and `bearing`.

Another OPNET interrupt type is a self-interrupt and are internally scheduled by a process to trigger future events. The `anim_irq` fires once per second to schedule two tasks: an animation update and an ESI update of the node’s `XY` position for co-simulation processing in Java.

```

1 void move()
2 {
3     // Initialize local variables
4     double speed;
5     double bearing;
6     char speed_str[256];
7
8     // Get speed and bearing from ESIS
9     op_esys_interface_value_get (interface_speed, &speed, 0);
10    op_esys_interface_value_get (interface_bearing, &bearing, 0);
11
12    // Create string format used by node's speed attribute
13    sprintf (speed_str, "%-.1f meter/sec", speed);
14    op_sim_message (speed_str, "");
15
16    // Set node's speed and bearing attributes
17    op_ima_obj_attr_set (my_parent, "ground speed", speed_str);
18    op_ima_obj_attr_set (my_parent, "bearing", bearing);
19 }
20

```

Figure 9 – ESYS Process “move” Method

The `bound_irq` and `anim_irq` transition conditions are examples of self-interrupts. The `bound_irq` is used to keep the node within the network’s physical boundaries. If the node reaches a boundary it will “bounce”, following the rule that the angle of reflection equals the angle of incidence. The default interrupt enables the node to correctly process any other interrupts it may receive.

After achieving success with JOCosim I, we determined that we had created a viable co-simulation environment for further research. There are both design and technical modifications that are necessary to fully simulate a HARVEST swarm. These steps are listed in Figure 10.

1. Active statistic collections via probes
2. Create models via external model access (EMA)
3. Develop management cloud for ESYs access
4. Enable broadcast & routed transmissions
5. Extend Java GUI for simulation control
6. Integrate a distributed indexing application
7. Migrate from OPNET 10.5 to 11.5
8. Move co-simulation control to Java from C
9. Port a node mobility algorithm to Java
10. Support node failure and recovery

Figure 10 – JOCosim II Modification

A major technical issue has been item (7) — migrating our OPNET models to version 11.5. This is mostly attributable to changes in the wireless model. We could use FAILURE/RECOVERY interrupts for node failures, but that can be challenging. We would then have to create FAILURE/RECOVERY interrupt handlers for each of eight process models within the node model. To avoid this complexity, we adjust the transmit/receive power limits, and disable the packet generator(s) on a destroyed node. This approach effectively disables a node and ensures the collected statistics are valid system performance measures.

The most significant change is item (8) — moving the co-simulation execution control from Java to C. This involves wrapping the ESA method calls and handling discrete event timing. The new design, motivated by the changes in Figure 10, is given in Figure 11. The goal for JOCosim II is to study data indexing and querying distributed across nodes whose mobility is controlled by an external algorithm, e.g., a cooperative search [29]. The details of JOCosim II are discussed in [21].

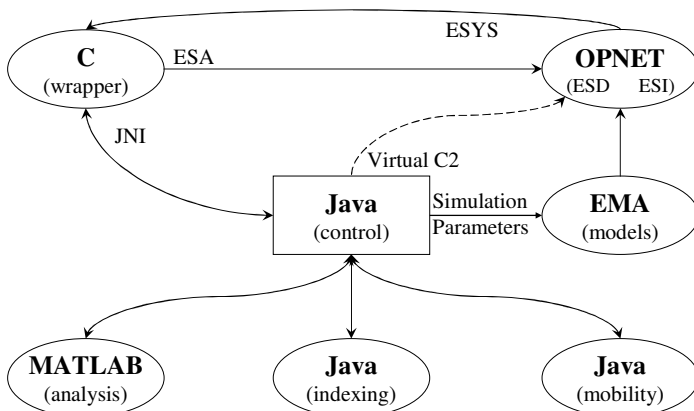


Figure 11 – JOCosim II Design

C. J. Augeri and K. M. Morris thank Scott Graham for significant latitude on the assigned “individual” project.

REFERENCES

- [1] *Automated Air Refueling*. [Online]. Press Release. Air Force Institute of Technology (AFIT). Available: <http://www.afit.edu/pa/news.cfm?vwk=65>.
- [2] *High-Level Architecture (HLA)*. [Online]. Available: <https://www.dmsomil/public/transition/hla/>.
- [3] *Java 5.0*. Sun Microsystems. [Online]. Available: <http://java.sun.com/>.
- [4] *Java Native Interface (JNI)*. Sun Microsystems. [Online]. Available: <http://java.sun.com/docs/books/jni/>.
- [5] *JNI Calling Java Methods*. Sun Microsystems. [Online]. Available: <http://java.sun.com/docs/books/tutorial/native1.1/implementing/method.html>.
- [6] *JNI Invocation Interface*. Sun Microsystems. [Online]. Available: <http://java.sun.com/docs/books/jni/html/invoke.html>.
- [7] *JNI Loading the JVM*. Sun Microsystems. [Online]. Available: <http://java.sun.com/docs/books/tutorial/native1.1/invoking/example-1dot1/invoker.c>.
- [8] *Joint Architecture for Unmanned Systems (JAUS)*. Office of the Under Secretary of Defense (OUSD) for Acquisition, Technology and Logistics. [Online]. Available: <http://www.jauswg.org/>.
- [9] *OPNET Discrete Event Simulation*. Chaps. 2,4,8,9,15,24. Ver. 10.5+.
- [10] *OPNET Editors Reference*. Chap. 6. Ver. 10.5+.
- [11] *OPNET Model File Access API Reference*. Ver. 10.5+.
- [12] *OPNET Modeling Concepts Reference*. Chaps. 1,5,8. Ver. 10.5+.
- [13] *OPNET Simulation Control API Reference*. Ver. 10.5+.
- [14] *OPNET Wireless Module User Guide*. Chap. 2. Ver. 10.5+.
- [15] Sess. 1532. “Interfacing Multiple Simulators Using the OPNET Co-Simulation API,” *OPNETWORK*. 2005.
- [16] Sess. 1901. “Co-Simulation w/OPNET Using HLA,” *OPNETWORK* 2004.
- [17] Sess. 1933. “HL Co-simulation with OPNET,” *OPNETWORK*. 2005.
- [18] Sess. 1938. “Effectively Planning and Implementing HLA Federations using OPNET,” *OPNETWORK*. 2005.
- [19] *Unmanned Aircraft Systems Roadmap, 2005–2030*. Office of the Secretary of Defense (OSD), August 2005. [Online]. Available: <http://www.acq.osd.mil/usd/Roadmap%20Final2.pdf>.
- [20] I. F. Akyildiz, W. Su, Y. Sankarasubramanian, and E. Cayirci. “A survey on sensor networks,” *IEEE Comm.*, vol. 40 (8), 2002, pp. 102–114.
- [21] C. Augeri, K. Morris, and B. Mullins. “JOCosim: Using Java, OPNET, and C to Enable Mobile Co-Simulation”, *OPNETWORK*. 2006. Wash., D.C.
- [22] V. Dham. *Link Establishment in Ad Hoc Networks Using Smart Antennas*. M.S. Thesis, Virginia Polytechnic Institute and State University, 15 Jan 03.
- [23] A.G. Dimakis, V. Prabhakaran, and K. Ramchandran. “Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes,” *Proc. of the Sym. on Info. Proc. in Sensor Networks*. 2005.
- [24] K. Fujita. *Extending OPNET Modeler with Client Profiles for Selecting Data Sources in WAN*. University of Pittsburgh. [Online]. Available: <http://db.sis.pitt.edu/projects/Nebula/public/opnet/report2.html>.
- [25] V. Kawadia and P.R. Kumar. “A cautionary perspective on cross-layer design,” *IEEE Wireless Communications*, vol.12 (1), pp. 3–11, Feb. 2005.
- [26] U.C. Kozat, I. Koutsopoulos and L. Tassiulas, “A framework for cross-layer design of energy-efficient communication with QoS provisioning in multi-hop wireless networks,” In *Proc. Joint Conf. of the IEEE Comp. and Comm. Societies (INFOCOM)*, vol. 2, pp. 1446–1456, 7–11 March 2004.
- [27] C.A. Lua, K. Altenburg, and K. Nygard, “Synchronized Multi-Point Attack by Autonomous Reactive Vehicles with Simple Local Communication,” In *Proc. of the IEEE Swarm Intelligence Sym.*, April 2003, pp. 95–102.
- [28] R. Mahajan. *Cross Layer Optimization: System Design and Simulation Methodologies*. M.S. Thesis, Virginia Polytechnic Institute. Nov 2003.
- [29] K.M. Morris, B. Mullins, D. Pack, G. York, and R. Baldwin, “Impact of Limited Communications on a Cooperative Search Algorithm for Multiple UAVs,” In *Proc. of the IEEE Intl. Conference on Networking, Sensing and Control*, Fort Lauderdale, FL, April 23–25, 2006.
- [30] H. Parunak, S. B. Brueckner, and J. J. Odell. “Swarming coordination of multiple UAVs for collaborative sensing,” In *Proc. of the AIAA Unmanned Unlimited Systems Tech. and Operations Air/Land/Sea Conf.* 2003.
- [31] S. Shakkottai, T. S. Rappaport, and P. C. Karlsson, “Cross-layer design for wireless networks,” *IEEE Comm. Magazine*, vol. 41(10), pp. 74–80, 2003.