A Model-Based Integration of Network Emulation with HLA-based Heterogeneous Simulation Environments

Yuan Xue

ISIS,

Vanderbilt University

Timothy Busch Shelby Barrett

Michael Gacek

Air Force Research Lab

Himanshu Neema Gabor Karsai

 Janos Sztipanovits

ISIS, Vanderbilt University

**Abstract**—Evaluation of Command and Control (C2) concepts requires a sophisticated modeling, simulation and experiment infrastructure. This requires the integration of existing simulation tools, system prototypes and experiment platforms that can interact in a coordinated way. This paper presents our work on integrating the network experiment platform into the HLA-based simulation environment as a solution to this problem. Network emulation environments allow the use of real network devices, components and systems, thus providing greater realism in the network experiment. Integrating network emulation into the C2W system, however, is a challenging issue due to (1) the different time domains that the simulation and the real network platform operate on and (2) the need to controlling the communication overhead between the simulation and the emulation environment, while still ensuring the accuracy of the experiment. This paper presents our approaches to these two issues and shows our models, algorithms, and system protocols developed for such system integration.

Keywords- Command and Control System, Network Simulation, Network Emulation, Model-Based Integration

This research is sponsored by the Air Force Research Lab Visiting Faculty Research Program and the Air Force Research lab Summer Faculty Extension Grant.

# Introduction

Evaluation of command and control (C2) concepts requires a sophisticated modeling and simulation infrastructure that allows for the concurrent modeling, simulation and evaluation. This requires the integration of multiple existing simulation tools, system prototypes, and experiment platforms so that they can interact in a coordinated way.

To address this urgent need, the “Command and Control Wind Tunnel (C2WT)” tool suite [1] is developed to support rapid synthesis of heterogeneous simulation environments. The C2WT system uses the High-Level Architecture (HLA) [3]framework which provides supports for the integration at the API level and the interaction level via a Run-Time Infrastructure (RTI). Using C2WT system, a range of simulation tools such as MATLAB/Simulink, OMNeT++[5], Java or C++ HLA-federates can be rapidly synthesized.

Network components and policies are essential aspects of C2 systems. Their impact on the performance and the behavior of C2 systems need to be accurately characterized when evaluating C2 systems. However, the current tool suite uses the network simulators, which lack the implementation details of network protocols and algorithms, and is insufficient in providing the level of accuracy required by the evaluation of C2 systems. For example, C2WT provides support for integration of OMNeT++-based network simulations. Being a discrete-event simulator, however, the accuracy of its simulations is dependent on the level of abstraction of the network protocol models. The operating system details, such as buffer copy overhead, are not simulated in the OMNeT++.

To enable realistic and accurate evaluation of the network component in the Command and Control system, we propose to integrate network emulation environment into the heterogeneous simulation tool suite “Command and Control Wind Tunnel”. Different from simulation, network emulation environments, such as Emulab, allow the use of real network devices, components and systems, thus providing greater realism in the network experiment. It allows the study of part of the C2 system running on real distributed hosts and communicating over more realistic emulated networks. If integrated into the C2W infrastructure, the network emulation platform could bring great realism and much better accuracy to the evaluation of C2 systems with real network traffic injected into the network. The emulation environment also provides a platform where prototypes of C2 components can also be deployed and interact with other simulated components. This integration is also critical for assessing the impact of various network attacks, which (e.g., TCP SYN flooding) is highly dependent on the detailed operating system implementation.

Integrating network emulation into the C2W system, however, is a challenging issue. (1) Being a simulation environment, the C2W system runs under the discrete event-driven time model and uses the RTI time management; on the other hand, the network emulation platform runs in continuous real time. The time synchronization issue between the simulation and emulation environment is a non-trivial issue. (2) There is potentially large volume of data communicated between the simulation and emulation environment. Controlling the communication overhead while still ensuring the accuracy of the experiment is also a challenging issue. This paper presents our approaches to these two issues and shows our models and algorithms developed for such system integration.

## Time Synchronization

To reconcile the two different time models in the C2WT system and the network emulation system, we investigated the time management mechanisms of network emulation environment and analyzed their compatibility with the time management mechanism of RTI. Our work focuses on the real-time execution modes of execution, where the simulation system follows a time-stepped discrete time model, while the emulation system just uses the system time. Without communication delay between the simulation and the emulation environment, the two systems can synchronize to the physical real time separately. When communication delay exists, it will cause the time stamp discrepancy of the same packet at these two environments, which will result in measurement error and introduce causality violations in the simulation. To solve this problem, we proposed a novel time synchronization mechanism where the simulation runs under a virtual time that is separated from real time but still keeps close pace with real time.

## Data Communication Architecture

There is potentially a large volume of data that need to be communicated between the simulation and the emulation environment. Such communication overhead will significantly affect the performance of the integrated C2W system. Controlling the communication overhead while still ensuring the accuracy of the experiment is a challenging issue. In this paper, we first propose a new architecture for incorporating network traffic from real networking environment into the HLA-based simulation environment. In this architecture, a federate serves as the gateway of the communication between the simulation federates and the real network application processes. Compared with the decentralized architecture, the centralized approach simplifies the design and limits the communication overhead over RTI. It also implies more efficient network application process execution, since it does not need to call the RTI library for communication.

Observing that different C2 system experiment may have different communication needs and semantics, efficient communication can be only achieved by recognizing such differences and tailing the communication protocol based on each experiment unique semantics. To enable such capability in the integrated system, we enhance the meta model of the existing C2WT system with the emulation meta model which specifies the major attributes of the information communicated between the simulation and the emulation environment. Then the C2 system experiment configuration can specify the communication context through network modeling, which can in turn customize the communication protocol between the simulation and the emulation environment to achieve the best tradeoff between experiment realism and performance.

The remainder of this paper is organized as follows. Section II introduces the background information of our work and discusses the relevance and difference of our work to previous work. Section III, IV and V present our approach to the data communication architecture, meta-model, model and the time management design, respectively. Section VI describes the implementation details. Section VII presents the experiment results. Section summarizes the paper and discusses the possible future directions of this work.

# Related Work

There are a few existing works that support the integration of network simulation and emulation. For example, the ns2 emulation facility, so-called nse [8], supports introducing live traffic into the ns-2 simulator [9]. Special objects are developed within the ns-2 simulator, which inject traffic from the simulator into the live network and accept the traffic from the live network [7,19]. Based on this facility, the Emulab software also supports its integration with ns-2 simulation [10]. The network simulator OMNeT++ also has similar support [11] to incorporate live traffic. The work of [18] extends ns2 emulation facility with wireless multicast support. [15] improves the ns2 emulation facility with real-time capability.

The early work on network simulation and emulation integration usually requires simulation to be real-time capable as a time drift between the simulation and the real network may occur and corrupt the results. The work of [14] presents an approach on synchronizing network emulation with simulation by virtualizing real systems. The work of [13] further develops a new model of time called relativistic time that combines the controllability of virtual time with the naturally flowing wall-clock time to facilitate the time coordination between the simulation and emulation systems.

Our work is related to these previous works in that we both combine real network elements with simulated ones -- each modeling different portions of a networked distributed system. Our work is fundamentally different from the existing work in that (1) in the existing work, the network is simulated, the application is real, while in our work, the network is real, the application is simulated; (2) though both require time synchronization, in our case, the network communication (e.g., packets in fly) can not be controlled. In short, we need new designs for the simulation-emulation communication and time synchronization. Moreover, there has not been an effort, however, for integrating heterogeneous simulation environments (including not only networks, but also physical environment, organizations, etc) with network emulation environment.

# Data Communication Architecture

Figure.1 illustrates the C2WT system framework extended with the network emulation capability under the centralized data communication architecture. In the figure, the *EmuGateway* federate serves as the gateway between the simulation federates and the network applications. At run time, it communicates with other simulation federates over RTI. The *EmuGateway* federate will publish and subscribe to all the network related interactions on RTI. The *EmuGateway* federate communicates with the network applications through the Simulation-Emulation tunnel. This tunnel is implemented using Interaction Delivery Protocol based on the TCP unicast socket.

Figure C2W system framework with network emulation capability

In this framework, the network models (including network interaction models, network application deployment models, and network topology models, as detailed in Section III) are integrated with other models such as controller models and environment models using GME at the model integration layer. Model transformations, implemented via custom interpreters, generate scripts for Emulab to configure the network experiment (topology, running commands) and glue code for the *EmuGateway* federate and the network applications that run on the emulated/real network.

# Meta Model and Domain Model for Network Emulation

We enhance the meta-model of the C2WT system with the network emulation meta-models which specify (1) the major attributes of the information communicated between the simulation and the emulation environment; (2) the network application deployment configuration. The C2 system experiment configuration can specify the communication semantics through network modeling, which can in turn customize the communication protocol between the simulation and the emulation environment to achieve the best tradeoff between experiment realism and performance.

The C2 Wind Tunnel project has developed DSMLs that are based on the semantics of each integrated simulation platform and an overarching DSML to model the interactions between simulation models. Custom translators (model interpreters) are developed to dynamically import and export platform-native models into the GME DSMLs and to generate HLA configuration files and glue code. These tools allow the composition and execution of the entire simulation to be configured using the central modeling environment. To support the integration of network emulation into the C2WT environment, we extend the overarching modeling environment in GME to capture the operational semantics of the network emulation and the interaction between the network application processes in the emulation environment and the simulation models. In what follows, we will present our extension from three aspects: (1) Application Process Deployment Model, and (2) Network Emulation Interaction Model.

## Application Process Deployment Model



Figure 2 Application Deployment Meta-model

This model specifies the deployment of application processes on the hosts and the transport-layer communication model that will be used. This model includes the following objects.

(1) *NetworkApp*, which models the network application process that is running on the emulation host. It has at least a port. The *TCPBasicApp* and *UDPBasicApp* can be prebuilt as two basic forms of network application processes to facilitate network experiments (e.g., for background/attack traffic generation).

(2) *OnHost*, which models the deployment of a network application on an end host (i.e., *Host* in the topology model). Note the host defined in the network model is a “virtual host”. When Emulab is used, the “virtual host” will be mapped to a real host where the IP address is assigned. In this case, the emulation gateway needs to keep track of the IP address and performs host mapping.

(3) *NetworkAppInteraction*, which models the communication connection between network applications. There are two types of communication models between application processes: Connection-oriented and connectionless communication. TCP protocol is connection-oriented. UDP protocol can work in both connection-oriented and connectionless communication manners. For connection-oriented communication, two application processes need to establish a connection. The destination application process information is predefined (prewired) in the Application Deployment Model so that it does not need to be carried in the Network Emulation Model (detailed in the next subsection). In the connectionless model, the destination process information is unknown during the deployment. Thus it requires to be carried in the Application Deployment Model.



Figure 3 shows an example of application deployment model over the network as shown in Figure 2.



Figure 4 Network Interaction Meta-Model

Figure 3 Example Application Deployment Model

## Network Emulation Interaction Model.

This specifies how the network application processes in the C2 system communicate through the network, particularly, what information needs to be exchanged between the simulated C2 elements and the real network applications through *EmuGateway* in the experiment. In our design, this function is enabled through *NetworkInteraction*, a special type of HLA *Interaction*.

As shown in Figure 4, each *NetworkInteraction* needs to have three attributes in order to be processed by the *EmuGateway*:

(1) *Timestamp*, the time this interaction should be processed by the network application process. Note that this is not the timestamp of the *NetworkInteraction* itself. The *NetworkInteraction* needs to be sent at an earlier time though RTI before this *Timestamp* for the sake of time.

(2) *ProcName*, the name of the network application process that handles this *NetworkInteraction*. This name should be exactly the same as the process’s name used at the end host, as it will be passed directly by the *EmuGateway* to the *TapClient* (details in Section IV.).

(3) *NodeName*, the name of the node that hosts the network application process which handles this *NetworkInteraction*. Note that the deployment of the application process onto real hosts may not be known at the modeling time, the *EmuGateway* will convert this name to the real hostname/IP address at runtime.

Inheriting from Interaction, *NetworkInteraction* has a list of Parameters, which are used to model the content/information carried in *NetworkInteraction*.

*NetworkInteractionWithPeerInfo* extends *NetworkInteraction* with two more attributes -- *PeerProcPort* and *PeerNodeName*, which are the communication peer’s application process name and node name. The *PeerNodeName* will be converted to the node’s hostname/IP address at the *EmuGateway* federate at run time. The *PeerProcPort* will be directly passed. This type of interaction is used where peer network application information is needed, typically in two scenarios: (1) the network application is a connectionless UDP sender. The *PeerNodeName* and *PeerProcPort* information is needed to tell the UDP sender the destination application information. This type of *NetworkInteractionWithPeerInfo* is published by other federates and subscribed by *EmuGateway* federate; (2) the network application is a server (either TCP or UDP) which accepts requests from multiple clients. The client information cannot be known at the modeling time. This type of *NetworkInteractionWithPeerInfo* is published by *EmuGateway* federate and subscribed by other federates.

Figure 5 shows an example of the Interaction Model for the network scenario outlined in Figure 3. Note that the *NetworkInteraction* has one Parameter *ImageURL*, which is the information carried in this interaction. This implies that in this simulation experiment, only *ImageURL* is passed from the simulation environment to the emulation environment.

Figure 5 Example Network Interaction Model

This modeling framework is flexible to handle a variety of communication semantics. In the example shown in Figure 5, the UDP traffic from the application process *SendImage* is triggered by the *SendImageToNetwork* interaction, which specifies the location of the image. Real image data will be communicated within the emulated/real network. This experiment setup could allow us to evaluate the impact of data corruption (e.g., by malicious attacks) on the C2 performance. If the experiment is only interested in the transmission delay of the data (the traffic load it incurs), instead of its content, we can setup another interaction model with lower communication overhead between the simulation and the emulation environment. In this case, no real data is sent between for image transfer application. Instead, dummy data is sent with the specified sending rate and packet size to create the appropriate traffic load. Using the *SendImageToNetwork* *NetworkInteraction*, the simulated UAVs (UAVfederate) notify the network application process *SendImage* two parameters: *FrameInterval* and *FrameSize*.

Together with other domain models, this network interaction model will be used to generate HLA configuration files and glue code for the C2WT system.

# Time management

To reconcile the two different time models in the C2WT system and the network emulation system, we investigated the time management mechanisms of network emulation environment and analyzed their compatibility with the time management mechanism of RTI. Our work focuses on the real-time execution mode, where the simulation system follows a time-stepped discrete time model, while the emulation system just uses the system time.

Under the real-time execution mode, the simulation system follows a time-stepped discrete time model, while the emulation system just uses the system time. The simulation (ts) and operation system (to) time are synchronized with real time (tr) separately. ts=to=tr. Without communication delay between the simulation and the emulation environment, such time synchronization will not violate local causality constraint.

When communication delay exists, it will cause the time stamp discrepancy of the same packet at these two environments, which will result in measurement error and introduce causality violations in the simulation. The following figure illustrates this problem.

### Since the simulation environment will receive events from the emulation environment, simulation time should lag behind or equal to emulation time so that the events from emulation will not arrive at simulation in its past time. Similarly, the emulation environment will receive events from the simulation environment, thus simulation time should lead or equal to emulation, so that the events from simulation will not arrive at emulation in its past time. Without delay, simulation and emulation should be synchronized to the same time. With delay in both directions, this issue is non-trivial.

To solve this problem, we proposed a novel time synchronization mechanism where the simulation runs under a virtual time that is separated from real time but still keeps close pace with real time. In particular, the operation system time is synchronized with real time (to=tr), while the simulation time is separated from real time. The simulation environment should have at least a lag of (L ≥ 2) from real time to accommodate the communication delay emulation to simulation environment, if any incoming traffic is expected (ts = tr- L). Simulation clock advances at the same pace as real physical clock. All outgoing traffic event with time stamp t will be actually scheduled/tunneled to emulation environment at simulation time t - 1- Lto compensate the delay from simulation to emulation and the lag between simulation and emulation so that it could arrive at emulation at real time t. For incoming traffic with time stamp (t+), it will arrive at the simulation at simulation time ts = t-L+ +2. Since L ≥ 2, the event can be scheduled at ts = t+. The figure below illustrates this scheme.

The above scheme works well when the outgoing traffic event with time stamp t is always generated at a simulation time t’ smaller than t - 1- L. If t’ is larger than t - 1- L, the system will issue a run-time exception.

# Implementation

## Figure 6 outlines the components in the implementation.

Figure 6 Implementation Architecture

## Interaction Delivery Protocols

We present how the *NetworkInteractions* are delivered and triggers appropriate network operations at the emulated/real hosts. First, the *InteractionHandler* in *EmuGateway* will publish and subscribe to all *NetworkInteractions*. For incoming *NetworkInteractions* from RTI, the handler will wrap them into tasks and send them directly to the *TapServer*. A task object has as its attributes (1) *ProcName*, which handles the task, (2) *Timestamp*, when the task should be executed, (3) list of *Parameter* objects, which specify the content of the task. For each *Parameter* object, it has (1) *ParameterName*, (2) *ParameterLength*, and (3) *ParameterValue*. The tasks will be serialized and transferred from the *TapServer* to the *TapClient* according to the *Interaction Delivery Protocol*. Figure 11 shows the message format in the *Interaction Delivery Protocol* and an example message.

Figure 7 Interaction Delivery Protocol Message Format

The *TapServer* is a multi-thread TCP server in the *EmuGateway* federate. It communicates with *TapClient* (a TCP client) that resides on each emulation host. Once receiving the message from the *TapServer*, the *TapClient* will convert it back to a task object. During the conversion, the *TapClient* also need to convert its *Timestamp* from the simulation time to the real time on the emulation host (details discussed in subsection B.) If the *Timestamp* is larger than the current time, which means the task should be executed in the future, the task will be sent to a buffer. When the *Timestamp* becomes current, the task will be scheduled for execution, which means the parameter will be set to the application process indicated by *ProcName*. In our example here, the *SendImage* process will receive the value of *ImageURL*, retrieves the image data and send them out via UDP. (Further performance optimization can be done here to prefetech the image before schedule.)

When the image is received by the *TapClient*, the *TapClient* will create a new task based on the *NetworkInteraction* Model. As in the example shown in Figure 8, the task will have (1) *RecvImage* as its *ProcName*, (2) the time the packet (here we put a frame in one UDP packet) is received as *Timestamp* (note that the time is converted from the real time to the simulation time here) (3) calculated packet delay as the *ParameterValue* for *Parameter* *PacketDelay*. The tasks will be serialized and transferred from the *TapClient* to the *TapServer* according to the *Interaction Delivery Protocol*. When it arrives at the *TapServer*, its *Timestamp* will be compared against the current simulation time, and it will be sent to the *LocalTask* buffer waiting to be scheduled if the *Timestamp* is larger than the current time.

## System Initialization

When the integrated C2WT system starts, the *TapClient* will initialize the network application processes and register itself to the *TapServer* (the *TapClient* needs to know the address of the *TapServer* a priori) by sending its address (real hostname or IP address), its virtual host name and its current OS time. Based on this information, the *TapServer* will build the *HostMap*, which keeps the mapping information from the *NodeName* to the host IP addresses. The *TapServer* then provides this *HostMap* back to the *TapClient*. The *TapServer* also returns the current simulation time to *TapClient* as a response after the simulation starts to run. Due to the delay variation, multiple rounds of handshake are necessary to have a good estimation of the worst-case delay, as we will show in the experiment study. The TapClient will use this information to synchronize the simulation time with the emulation time, which is discussed below.

## Time management

To implement the time management scheme, we have a time converter at each emulation host which converts the *Timestamp* on each task. The time converter keeps the difference between the simulation time and the emulation host OS time, which is initialized when the system starts. We make the following notations to illustrate the timestamp calculation:

* *Tos\_client\_start*: the OS time (real time) when the *TapClient* register itself to the *TapServer*.
* *Tos\_server\_start*: the OS time at the *TapServer* when the client registration message is received.
* *Ts\_start*: the value of the simulation time when the client registration message is received.

Then the difference between the simulation time and the OS time at the *TapServer* is Tos\_server\_start - Ts\_start. We assume that the system clocks of the emulation hosts and the simulation host are synchronized a priori (e.g., via Network Time Protocol). Thus this difference (Tos\_server\_start - Ts\_start) is also the difference between the simulation time and the real time at the *TapClient*. At the *TapClient*, each incoming task with *Timestamp* will be converted from the simulation time to its emulation time with respect to the *TapClient* OS time for scheduling according to the following relation.

*Timestamp\_emu = Timestamp + (Tos\_server\_start – Ts\_start) – L*.

For outgoing tasks, the converter will convert its *Timestamp\_emu* with respect to the emulation time to the simulation Timestamp as follows.

*Timestamp* = *Timestamp\_emu - (Tos\_server\_start – Ts\_start) + L*

Here L is the lag from the simulation to the emulation time. As we have discussed earlier, L needs to be larger than 2, the delay from the *TapClient* to the *TapServer*, which can be calculated as 2 = Tos\_server\_start - Tos\_client\_start. Here we take L to be twice the maximum value of 2 measured multiple times during system initialization, taking into account the delay variation.

# Experiment Results

## Experiment Platform

To validate the design of our integrated C2 simulation and emulation environment, we conduct experiment over a wireless network testbed. The experiment scenario is the same as the one in Figure 3 and 5.

In our wireless experiment testbed, the wireless nodes which emulate the UAVs are laptop. Each laptop is equipped with two adaptors: one IEEE 802.11b wireless adaptor is used for data communication between the experiment network applications (e.g., *SendImage* and *RecvImage*), one Ethernet adaptor is used for the interaction communication between the *TapClient* and the *TapServer*. In this way, the experiment data will not interfere with the control data for the integrated simulation and emulation environment. For experiment, the wireless adaptors communicate with a wireless Access Point which is connected to a wired network. The control station node is emulated using a desktop that is connected to the same wired network. The simulation environment runs on a high performance workstation that is connected to the same wired network (on the same switch) with the desktop (ControlStation), Ethernet adaptors of laptops (UAVs) for extremely low delay communication.

## Experiment Results

We first study the delay between the simulation and the emulation environment. This value is an important parameter in our system as it decides the time synchronization between the two environments. Figure 7 shows the measurement from one laptop that emulates the UAV to the simulation host. We perform multiple rounds of measurements. The x-axis shows the measurement number. The y-axis shows the corresponding delay in nanosecond. The packet size used in the experiment is 20K byte. We observe that the delay varies a lot across different experiments, with the maximum value as 6ms and average around 1ms.

Figure 7 delay between simulation and emulation environment

Since tasks in the Interaction Delivery Protocol may have different sizes due to the different types of parameter lists, we would like to study whether the size of packets will affect the delay. Figure 8 shows the delay measured for packet sizes with 10K byte, 5K byte and 20K byte, sorted individually according the delay value in each experiment.

Figure 8 delay under different packet sizes

From the figure, we observe that the packet size is not a significant factor in the delay. The delay is mostly due to the OS scheduling overhead instead of the network transmission time. We use L = 12ms in the system experiment.

Now we show the experiment result under the simple scenario as shown in Figure 3 and 5. In the experiment, UAV1 starts to send images at 2 frames/sec at time 0. Then the control station instructs UAV2 to send images after 100 frames are delivered.

Figure 9 below shows the packet delay for UAV1 in millisecond for each packet. As we could observe from the figure, the packet delay increases after UAV2 joins the wireless network, due to contention.

Figure 9 packet delay from UAV1

# Summary and Future Directions

This paper presents our work on integrating the network emulation into the C2 Wind Tunnel system – a model-driven HLA-based heterogeneous C2 system simulation tool suite. We present the architecture and the detailed implementation of the integration under the real-time mode time synchronization model. A system prototype is developed to validate the proposed design. In the future, we will expand our results to larger experiment platforms and experiment on more diverse network application scenarios.

##### References

1. Command and Control Wind Tunnel, <https://wiki.isis.vanderbilt.edu/OpenC2WT/index.php/Main_Page>
2. GME, <http://www.isis.vanderbilt.edu/projects/gme/>
3. HLA standard – IEEE standard for modeling and simulation (M&S) high-level architecture (HLA) – framework and rules, <http://ieeexplore.ieee.org/servlet/opac?punumber=7179>
4. Portico RTI, <http://www.porticoproject.org>
5. OMNeT++, <http://www.omnetpp.org/>
6. Emulab, <http://www.emulab.net/>
7. DETERlab -- cyber-DEfense Technology Experimental Research laboratory Testbed, <http://www.isi.edu/deter/>
8. Nse -- the ns emulation facility, <http://www.isi.edu/nsnam/ns/ns-emulation.html>
9. ns-2 simulator, <http://www.isi.edu/nsnam/ns/ns-emulation.html>
10. Integration of ns2 and emulab, <https://users.emulab.net/trac/emulab/wiki/nse>
11. OMNeT++ support for emulation, <http://www.omnetpp.org/pmwiki/index.php?n=Main.OmnetppInNutshell>
12. Sztipanovits, J., and Karsai, G. 1997. “Model-Integrated Computing”, IEEE Computer, vol.30, pp. 110-112.
13. C. Bergstrom S. Varadarajan G. Back, “The Distributed Open Network Emulator: Using Relativistic Time for Distributed Scalable Simulation”, Workshop on Principles of Advanced and Distributed Simulation, pp19-28, 2006.
14. Weingartner, Elias and Schmidt, Florian and Heer, Tobias and Wehrle, Klaus, “Synchronized network emulation: matching prototypes with complex simulations”, SIGMETRICS Perform. Eval. Rev., vol. 36, no. 2, pp58-63, 2008.
15. Mahrenholz, D.; Ivanov, S., “Real-Time Network Emulation with ns-2”, IEEE International Symposium on Distributed Simulation and Real-Time Applications, pp 29-36, 2004.
16. Richard M. Fujimoto, “Time Management in the High Level Architecture”, Simulation, vol. 71, pp388-400, 1998.
17. Guruprasad, S., Ricci, R., and Lepreau, J., “Integrated Network Experimentation using Simulation and Emulation”, In Proceedings of the First international Conference on Testbeds and Research infrastructures For the Development of Networks and Communities, 2006.
18. S. Penz, "Wireless Multicast Support for the NS-2 Emulation Environment," 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp.267-273, 2007.
19. Shashikiran B. Guruprasad, “[Issues in Integrated Network Experimentation using Simulation and Emulation](http://www.cs.utah.edu/flux/papers/guruprasad-thesis-base.html),” Master's Thesis, University of Utah, August 2005.