REATSS: A Distributed Software Simulation Test Environment

Samuel Martin[1] and Robert Best[2]
ProLogic, Incorporated, Fairmont, WV, 26554

Daniel Solomon[3]
NASA Independent Verification & Validation Facility, Fairmont, WV, 26554

The Reconfigurable Environment for Analysis and Testing of Software Systems (REATSS) was developed to provide the NASA Independent Verification and Validation Facility an automatically distributed simulation environment supporting reusable object-oriented components and reliable real-time execution. Like many distributed simulation environments, capabilities are provided to communicate via network or shared memory, system executive control capabilities, and logging and monitoring capabilities. What differentiates REATSS from most distributed simulation systems are the technologies developed that provide the ability to rapidly reconfigure and modify the composition of a distributed application to create new target subsystems at runtime as well as providing an ideal integrated development environment and framework for the creation of composable real-time capable system of systems applications. The key technologies developed to accomplish these tasks are: the Data Distribution Service (DDS) based Generic Reconfigurable Interface Manager (GRIM), Reconfigurable System Executive (RSE), Eclipse based Service Development Environment (SDE), Application Development Environment (ADE), Simulation Control and Monitoring Service (SCM), and the OWL based Semantic Component Database.

This paper will discuss the architecture of REATSS and lessons learned in developing the architecture to meet a diverse selection of simulation models and services provided from many sources.

I. Introduction

The Reconfigurable Environment for Analysis and Testing of Software Systems (REATSS) is an advanced dynamic simulation environment tailored to the needs of software analysts at the NASA Independent Verification and Validation (IV&V) facility located in Fairmont, WV. The purpose of developing REATSS was to provide a flexible simulation and test environment to support dynamic analysis of diverse software systems through rapid integration of software simulation tools, models, and test articles. REATSS provides the system analyst the ability from their desktop to:

a) create stand alone simulation components
b) build larger component based simulations from these single simulation components
c) develop script based test scenarios for flight software analysis
d) execute the simulations on the REATSS server farm; two quad processor dual core AMD Opteron systems

[1] Program Manager, Advanced Technologies Division, 1000 Green River Drive, Suite 201, Fairmont, WV 26554, Member.
[2] Technical Lead, Advanced Technologies Division, 1000 Green River Drive, Suite 201, Fairmont, WV 26554, Member.
[3] Project Manager, Independent Verification and Validation, 100 University Drive, Fairmont, WV 26554.

e) perform batch mode testing with developed scripts
f) perform post processing analysis of results for additional testing or issue reporting

To provide this ability, it was necessary to develop unique new capabilities and integrate existing technologies where available. These technologies and capabilities include the development of a net-centric architecture for test component integration, the ability to dynamically manage hardware resource allocation, virtual user access, and tools for rapid component integration, test execution management and control. These capabilities leveraged new technologies in the areas of integrated development environments, ontologies, reliable high speed publish/subscribe protocols, and hardware architectures.

II. Capabilities and Technology

As mentioned previously, many technologies were developed and integrated in the creation of REATSS. This section will summarize those technologies, and provide a detailed description in follow-on sections.

At the heart of the flight software testing paradigm is the ability to accurately model the system and environment in which the software was designed to operate. In providing a simulation and modeling environment, REATSS has integrated many models to perform this tasking. First, a CPU and board emulation capability is needed to accurately execute the flight software being tested. REATSS is currently working with Virtutech's Simics emulator for the PPC based RAD 750. Second, in order to feed and stimulate sensor models, REATSS includes several physics based gravity, atmospheric, and propagator models. Finally, REATSS provides the flexibility to integrate simulation models from other vendors and user developed models into the REATSS framework. This capability greatly expands the library of models available to the users in developing their test environment.

To provide the users with an easy to operate environment for developing simulations, simulation components, and executing/monitoring simulation runs, a user interface based on the Eclipse framework was developed. Utilizing the modular architecture of the Eclipse framework, particularly the plug-in interface, it is possible to proficiently create a robust and feature rich user interface. This interface allows the user to develop simulation components, create large simulations by interconnecting single components, and the ability to execute and monitor simulations from their desktop.

Also, software had to be developed that would perform tasks associated with the distribution of the simulation components across all the systems that have been identified as hardware assets in the REATSS server farm. These software components represent the REATSS manager library. These modules are responsible for collecting hardware (CPU, GPU, PPU, amount of RAM, types of peripheral devices, etc) utilization information, configuring communications links between components, and maintaining a queue of simulations for batch execution.

In order to provide a reliable communications channel amongst simulation components running on multiple servers, it became necessary to develop network interface components based on a net-centric architecture that was both flexible to working with multiple protocols, and dependable to meet the real-time requirements of the system. This component is the Generic Reconfigurable Interface Manager (GRIM). GRIM's main communication conduit was developed based on the Network Data Distribution Service (NDDS) developed by Real-Time Innovations, Inc. NDDS was selected as the protocol of choice for its real-time performance guarantees as well as its ability to communicate locally via shared memory and remotely via Ethernet to both minimize latency and maximize throughput.

III. Network Interface Components

Today's high-performance simulations are being developed with the understanding that they require unique computational resources that may not be available through the use of a single computer, or even a single site. Additionally, the benefits of simulation interoperability are widely known and practiced to allow heterogeneous simulation components to be integrated and reused targeting varying problem domains. The scope of REATSS has always been to support large scale, complex simulations, and as such the architecture was created to support a server farm to provide the end user with an automatic interface to vast compute resources.

These were the driving forces behind the development of the Generic Reconfigurable Interface Manager (GRIM) which provides applications with a communications conduit to other applications outfitted with GRIM and configured for inter-process communications (IPC). GRIM is generic, in that its application programming interface (API) remains consistent while supporting several simulation interoperability standards including the Object Model Group's (OMG) Data Distribution Service (DDS) and the Institute of Electrical and Electronics Engineers' (IEEE) High Level Architecture (HLA) 1516 standard. GRIM is reconfigurable, in that it allows an application's publish-subscribe interface to be defined wholly in XML formatted metadata, allowing for a runtime configurable, data-centric distributed component.

A. GRIM DDS Interface

From its conception, REATSS was envisioned as a distributed simulation platform with a reliable real-time communication backplane capable of handling stringent timing and data delivery requirements with the goal of supporting distributed real-time applications as well as hardware-in-the-loop systems. This goal, coupled with the large scale scope of potential REATSS simulations, mirrored the purpose of the OMG's DDS for Real-Time Systems Specification, which targets real-time applications that have a requirement to model some of their communication patterns as a pure data-centric exchange, with the need to scale hundreds or thousands of publishers and subscribers in a robust manner. The Network Data Distribution Service (NDDS) is Real-Time Innovations, Inc's (RTI) DDS implementation. The NDDS architecture is designed to optimize performance by delivering minimal latency amongst distributed components by way of implementing the key features of the DDS specification including support for "DataReaders and DataWriters" for each data type, as well as the accompanying quality of service (QoS) specification. Expanding the DDS standard, NDDS provides each data type with its own DataReader/DataWriter buffers and dedicated threads to insure prompt delivery and handling of messages. NDDS supports several transports that are automatically selected to best fit the applications needs. For instance, if two applications are running on the same host machine, NDDS will select a shared memory transport to insure the fastest possible IPC mechanism. Alternatively, if applications are running on separate nodes, NDDS will use Ethernet, wireless, switched fabrics, or serial connections. The following diagram illustrates three NDDS enabled applications communicating through the appropriate transport layer based on their location.

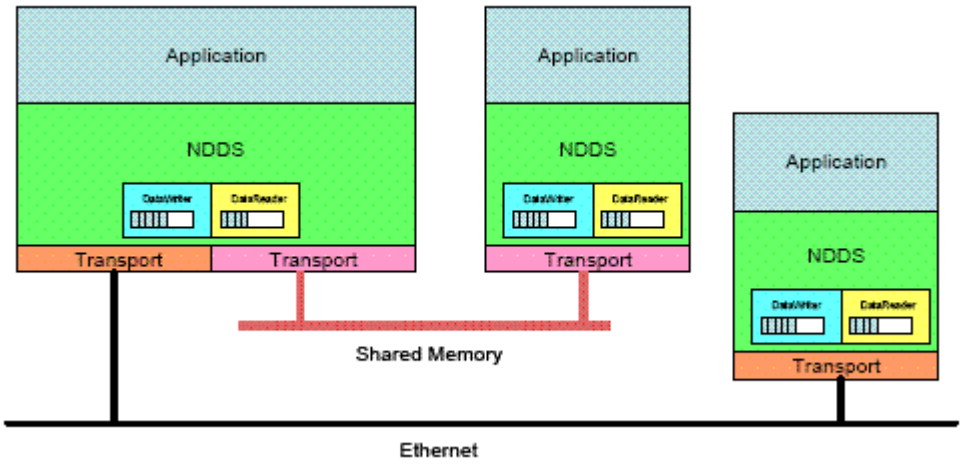

Figure 1. DDS Transport Mechanism

By dedicating DDS specified DataReaders and DataWriters for each data type, and selecting the quickest transport mechanism, NDDS provides reliable real-time IPC.

The GRIM library expands NDDS by allowing applications to define their publish-subscribe semantics through metadata which provides a dynamic, runtime configurable IPC interface. In this way, heterogeneous endpoints can be rapidly configured to communicate with one another with the granularity of variable to variable mappings. It is important to note however, that while the application developer is provided with an interface that provides this level of granularity through the underlying mechanisms of GRIM, packets are not sent at the variable level. Careful consideration was given to the NDDS architecture so as not to degenerate the middleware's performance.

B. GRIM IEEE HLA 1516 Interface

While NDDS was selected to be the main messaging system used for IPC amongst the core REATSS components, the need to support HLA 1516, the prescribed standard for military simulation interoperability within

all of NATO, was identified and implemented in GRIM's infancy. GRIM provides similar benefits to HLA 1516 federate development as it does to NDDS application development. It allows applications to define their data sharing intentions through metadata which can be dynamically changed and reloaded at runtime to allow federates to interact in varying federations supporting differing Federation Object Models (FOM). Additionally, by using the Runtime Infrastructure's (RTI) own FOM Document Data (FDD), GRIM based federates are guaranteed to be interoperable with other federates constituting a given federation. So as not to confuse the application developer, the developer is provided with the same API for HLA federate development as for NDDS application development, with only the arguments and return types of method calls differing. In this way, developers need only a high level understanding of the semantics behind publish-subscribe messaging architectures to create applications to partake in either a REATSS simulation or an HLA 1516 federation.

GRIM has always been thought of as a generic communications conduit, consideration has been given to its design to allow it to be extensible to other simulation interoperability protocols such as the Distributed Interactive Simulation (DIS) protocol, the Defense Modeling and Simulation Office's (DMSO) HLA v1.3, etc.

REATSS' HLA 1516 requirements were driven by the possibility that REATSS would be used to communicating with existing HLA 1516 models. In order to host HLA 1516 enabled models within a REATSS simulation an NDDS/HLA 1516 Gateway was created. The Gateway provides a runtime configurable, bidirectional communication channel that is both inherently FOM agile and REATSS architecture friendly due to GRIM's reconfigurable nature. By utilizing component and federate metadata, the gateway can be configured beyond direct "one-to-one" translations, and thoroughly check the veracity of the user's mappings. It is also capable of discovering and supporting new component mappings at runtime.

IV. User Perspectives

While GRIM provides simulation developers with the means to create a distributed simulation component that can be rapidly reconfigured to share data with several different applications running on the same machine or a remote node, and the REATSS server farm offers end users additional compute resources to execute their hardware intensive simulations, the need to provide a robust user interface (UI) remains. REATSS requires a user interface framework that is component based, distributable, extensible, and operator composable in order to fully realize the potential of its architecture. Through research and market analysis we identified the open source Eclipse platform as the candidate framework. Eclipse uses a modular plug-in based architecture to which REATSS developed user interface components adhere. Eclipse is an Integrated Development Environment (IDE) for IDE developers.

The REATSS UI is composed of three major IDE components; the Service Development Environment (SDE), the Application Development Environment (ADE), and the Simulation Control and Monitoring (SCM) perspectives. Each perspective focuses on specific groupings of activities that are to be performed by the end user from inception through execution to post processing and analysis. We will examine the REATSS UI in more depth in the following sections, focusing on each individual perspective.

A. SDE – Service Development Environment

The SDE focuses on the simulation development not in terms of a full up simulation but in terms of the components, or services, that compose a simulation. These components can be models, services, or a collection of each. REATSS provides a framework for developing and interacting with component based simulations, with the first step being the design and development of REATSS enabled components. The SDE is the first stop whether the user is creating a new simulation component from scratch, or integrating an existing Simulink, MATRIXx, or other model. In order for a component to interact with other models in a REATSS simulation in an intelligently distributed manner, a component's publish-subscribe interface and resource requirements must be made known to the runtime. The SDE provides wizards to walk the user through defining the data types that comprise the component's publish-subscribe interface allowing them to both create new data types and choose from existing types to increase reusability and reduce work. These data types can be assigned units to allow the development environment to check for logical inconsistencies, and the data can be grouped together into "topics" for better readability and error checking. These wizards also allow the users to specify hardware and software requirements, such as a component's target operating system, CPU, network I/O, memory, GPU, and PPU utilization. All of these parameters are saved in the Web Ontology Language (OWL) format to allow the runtime to intelligently distribute the component and configure the communication channels used by GRIM. In addition to defining a component's metadata, the SDE

uses this data to generate skeleton code, Microsoft Visual Studio 2003/2005 (VC++ 7.1/8.0) project files, and GNU make files, making it easy for the developer to integrate their component into the REATSS component catalog. The SDE, like the other perspectives within the REATSS UI, provides an interface to a Subversion repository so that all work products are automatically version controlled and backed up. Lastly, the SDE provides a schematic editor that can be used to develop a more complex component from several smaller components. Think "System of Systems" architecture.

B. ADE – Application Development Environment

After the user has defined the needed REATSS runtime metadata through the SDE and written the code for the model, the component is ready for use in the ADE. The ADE provides a repository browser which reads the OWL based metadata for a REATSS component within the repository to provide a graphical toolbox of components that can be dragged onto a schematic editor. Incorporated into the metadata are robust access control features ensuring that the end user is only privy to those components that are permitted for their use from the browser. When a component is selected from the browser and dragged onto the schematic it is represented as a block with publications and subscriptions represented as input and output pins defined completely by the component's OWL metadata. As the user drags components onto the schematic editor to build up their simulation, they are provided with the means to wire component input and outputs to one another either at the topic-to-topic level or zoom-in to interconnect at the variable-to-variable level, which allows components to interact with one another in a very generic way, where the only restriction is the primitive data type and units associated with the variable. When the user makes a connection in the schematic editor, the ADE's reasoners are initiated to check the user's wiring against the OWL metadata to ensure logical consistency. The reasoner's rules themselves are written in OWL providing an ontological rule set that is in itself an extension of the component metadata. The drag-and-drop semantics associated with the ADE's schematic editor are extended to provide toolbars and context menus that are also populated by the component's metadata to provide consistent user activities, while working with very different types of data depending on the component. In this way the REATSS application development process is truly object oriented. These tools allow the user to set initial conditions for any published or subscribed variable, select different versions of a component from the repository, set breakpoints, etc. The save process in the ADE works as the simulation build process, where the user defined initial conditions, breakpoints, IPC wiring, resource requirements, etc are translated and written into the format required by the runtime. Lastly, the ADE provides a script editor and debugger with syntax highlighting, auto-completion, and other tools for use with the REATSS Script Engine which will be described in more detail in the following sections.

C. SCM – Simulation Control and Monitoring

Once a component based simulation has been configured, and the needed runtime metadata has been built by the ADE, the simulation is ready to run. This is when the end user will begin to utilize the SCM perspective. The SCM provides several mode controls to start, stop, pause, restart, and single step an individual component, or an entire simulation. Users are provided with the means to queue simulations, as well as disconnect/reconnect to executing simulations including those instantiated by other end users. In addition, the SCM provides facilities to set breakpoints at runtime, override variables, save and load checkpoints, monitor variables (both textually and graphically), and dynamically add or remove components from the simulation. An OpenSceneGraph (OSG) based viewer has also been developed which allows simulated objects with associated geometry data to dynamically register and be rendered in a 3D scene at runtime. Lastly, any variables can be graphed in real-time for analysis. All of these functions define the control and monitoring aspects of the REATSS UI, however it is not always desirable or feasible to expect an end user to systematically interact with an executing simulation. Moreover, there are times when an analyst knows precisely what must be done before running a simulation to illicit the desired results, but the tedium involved with predicating these conditions isn't time effective. These considerations drove the development of the REATSS Script Engine, which is a Python based tool for scripting simulations that provides the end user with all the functionality associated with the SCM perspective. Being Python based, the Script Engine also provides easy to use mechanisms for logging data, emailing results, and interacting with other open-source software products that have a Python extension module. One of these packages is PyROOT, a scientific analysis and visualization framework, which adds an extra layer of domain specific tools to the end user. Provided with the SCM is a simulation queue which allows users to schedule numerous simulations or simulation scripts if the required

resources are not currently available as deemed by the REATSS Managers, which will be explained in more detail later. In this way, users can script Monte Carlo simulations and queue them to run overnight or over a weekend. They can set break-conditions and save the appropriate checkpoints. They can specify that the viewer capture the simulation as a movie, and instantiate loggers to analyze the simulation results from several different vantage points later.

V. Modeling

A. Simics RAD 750 Emulator

In providing a platform to perform flight software testing, it became necessary to make a decision on running the flight software on actual hardware or to acquire an emulator. After analyzing the upcoming IV&V mission support programs, it was decided that the target platform should initially be a RAD 750 based system. Next, a trade study was conducted to determine the most cost effective solution based on the needs of the customer. The results of this trade study determined that an emulator based solution should be pursued.

The requirements necessary for an emulation based solution were very aggressive. They include:

a) the ability to accurately run flight software binaries as provided by NASA programs, including real-time operating systems, device-drivers, and protocol stacks
b) run object code without access to source code
c) the ability to run faster than real-time to support Monte Carlo based testing
d) the ability to scale to platforms running multiple processors, in the case of multiple flight computers communicating within a single subsystem of the flight vehicle being tested

In researching existing software based full system simulation, only one platform currently supported the RAD 750. Virtutech's Simics meets all of these requirements and provides a full API for the integration into REATSS as a stand alone simulation component, as well as full simulation mode control from an external source. For these reasons, the Simics based RAD 750 platform was selected.

B. Physics Models

In order to stimulate the sensors that feed data to the flight software, it became necessary to provide a base set of models to perform this task. The current library of models is very simple, and is focused towards REATSS' current customer. The architecture implemented in REATSS makes integrating new models into the REATSS library a seamless effort. As more programs are supported by REATSS, this model library will rapidly increase allowing for the reusability of models between programs and decreasing the cost incurred by future programs.

Currently, REATSS has the following models integrated in the library:

a) *NORAD SGP4/SDP4 Propagator Models* to predict the location of any number of satellites in orbit at any time based on a Two Line Element set
b) *GRAM99 Earth Atmospheric Model* to provide the user with the means to obtain atmospheric temperatures, density, and concentrations
c) *EGM96 Earth Gravitational Model* to provide a detailed model depicting the variations in Earth's gravitational field
d) *SHGJ180U Venus Gravitational Model* to provide a detailed model depicting the variations in Venus's gravitational field
e) *JGL165P1 Moon Gravitational Model* to provide a model depicting the variations in the moon's gravitational field
f) *GMM2B Mars Gravitational Model* to provide a detailed model depicting the variations in Mars's gravitational field
g) *JPL Planetary and Lunar Ephemerides Model* to provide the end user the ability to incorporate DE405 ephemeride data.

VI. Manager Software

A. REATSS Managers

As mentioned earlier, when an end user launches a simulation, REATSS intelligently distributes the simulation components across all the systems that have been identified as REATSS assets. This functionality is provided by the four REATSS services; the REATSS Manager, the Asset Manager, the Sim Manager, and the Site Manager, which are best explained by the manager interactions that take place when an end user or a script launches a simulation. These interactions are detailed in Figure 2.

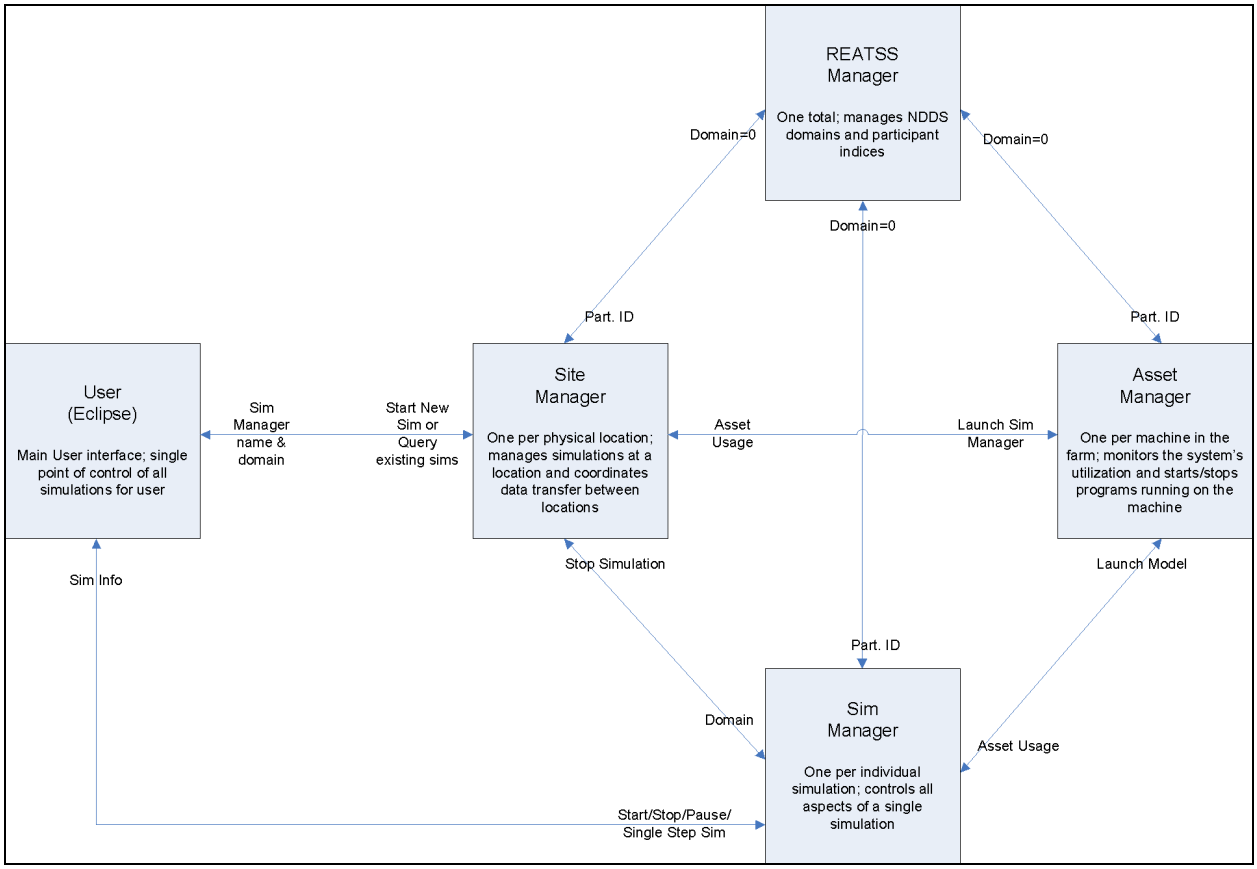

Figure 2. The REATSS Managers provide the means to intelligently and automatically distribute simulations

Every piece of hardware that makes itself available to REATSS to offer its compute resources to a simulation does so by running a Windows Service or Linux daemon called the Asset Manager. This manager collects all the hardware (CPU, GPU, PPU, amount of RAM, types of peripheral devices, etc) and software (operating system (OS), commercial off the shelf (COTS) packages installed, etc) information and monitors their utilization in real-time. When a simulation is launched, the REATSS Manager is consulted to retrieve the needed NDDS parameters to launch an independent simulation as well as the available resources and their current utilization from any executing Asset Managers. Next, the Site Manager is queried for all simulations executing at a given site, and the new simulation is added to the list. This provides the capability to both execute different simulations concurrently and also for users to monitor executing simulations from a different geographical location (sites). If the needed requirements outlined in the simulation metadata are not available as reported by the Asset Managers, the user is prompted to place the simulation in the queue, which is managed by the Site Manager. Finally, if the needed resources are available to the user, an instance of the Sim Manager is launched to interact with the Asset Managers to launch RSE instances on the various nodes. The RSE, which will be further explained later, loads the component and issues REATSS UI commands to the loaded component library.

B. RSE

While GRIM, as a library, provides the IPC mechanism for REATSS components, which are also libraries, the Reconfigurable System Executive (RSE) is the actual executable that the SCM interacts with while a simulation is executing. The RSE interface requires that REATSS enabled components derive from an abstract base class that the executive retrieves a pointer to after loading the specified component library. It uses the object-oriented C++ (OOC++) concepts of dynamic binding and polymorphism to issue the REATSS UI commands to the component library based on their implementation of the base classes' pure virtual functions. Additionally, components can request to run in real-time mode, where they specify their requested tick rate to the RSE and act as a slave to the RSE's interrupt driven clock as opposed to being event driven based on receiving inputs from other components. This clock can be either a hardware or software based interrupt generator that all RSE instances can register with for synchronization. Mechanisms are provided for the model to flag real-time violations based on the data, or lack there of, that it requires from other components to process a frame.

VII. Conducting Flight Software Testing

The tools and services that comprise the REATSS system as described thus far provide the framework for the REATSS systems main purpose which is conducting flight software testing. While developing a simulation environment to conduct flight software testing, the user will start with the flight software to be tested utilizing the RAD 750 emulator. Additionally, the engineer is responsible for developing the simulation components that feed the sensors associated with the system under test. By running the flight software in an emulated environment, the user has complete access to the entire memory space available to the flight software. An example of a sensor-model combination would be the implementation of a star tracker sensor. The user would start with a vendor provided star tracker model. The model would be integrated and turned into a REATSS compatible component. This includes the mapping of data I/O, creation of semantic metadata, and the definition of performance criteria for the model. Once this is complete, a model must be developed that can feed the location of celestial bodies to the star tracker model. These models are then combined in the ADE to form the simulation test environment with the celestial model feeding the start tracker model, which in turn feeds data to the emulator for reading by the flight software. This gives the flight software the illusion that it is communicating with actual flight system sensors instead of simulated models. Simulation and modeling is only a subset of the activities associated with performing flight software testing, the remainder of the effort is encompassed by the generation of test scenarios to assist in the IV&V of the target system. REATSS users can develop an infinite amount of test scenarios by way of the REATSS Script Engine. The capabilities outlined above provide the user an end-to-end simulation, testing, and post-processing analysis environment for the batch execution of simulations.

The reconfigurable nature of REATSS is showcased by the varying types of flight software testing that this architecture can support, which can be broken down into four major categories.

A. Avionics Test Bench

In this category of testing, real system hardware is utilized in the testing phase. The system demands real-time cycle accurate performance levels due to the rigid constraints imposed by the use of real flight hardware. In this test scenario, actual flight software binary code is loaded on the test bench. This is the most accurate form of testing but also comes with the drawback that it is the most expensive.

B. Functional Test Bed

In this category of testing, a hybrid solution is utilized. Some system hardware is still included while sensors are generally represented by models. Actual flight software binaries are run on an emulator. For this purpose, real-time constraints are somewhat relaxed. This solution still provides good fidelity and is generally a median cost solution.

C. Engineering Simulation

In this category of testing, a complete model based simulation is utilized. All system components are represented by component based models. The flight software is recompiled for the target platform of the test environment. Some risk is taken with respect to the flight software not operating in its target environment. When rehosting software to a new target platform, the possibility of injecting errors greatly increases. This is generally the least expensive solution for performing flight software testing.

D. Special Purpose Models or Test Tools

In this category of testing, small segments of the flight software is the key focus. Algorithms are compiled as components in the test environment. Input test data is fed into the algorithms and output results are analyzed for the adjustments to the algorithms, or to report any anomaly that may occur.

VIII. Lessons Learned

With the diminishing costs of PC based hardware, today's simulations are limited not by the complexities of the models involved, but rather by the inflexibility of legacy applications and vendor specific APIs. One of the main lessons learned while developing REATSS was the importance of a flexible service oriented architecture. In creating a test environment capable of meeting the robust requirements of the customer, it became obvious that a mechanism for integrating a diverse set of simulation components that would come in many different formats was paramount. The solution was to focus on a component interface that abstracts the actual model implementation from its interaction with other components. In doing so, it became very easy to integrate or change a model and not have to recompile the entire simulation due to a single change. This is the essence of object oriented design, and a testament to the benefits of its ideals.

IX. Conclusion

Several reusable key technologies were developed through REATSS: a reliable and deterministic communications link, a rapidly reconfigurable system architecture, and a semantic metadata driven system of systems management and composition. These technologies are extensible to other problem domains requiring a system of systems architecture, a service oriented architecture (SOA), etc. The reusable key technologies are: a rapidly reconfigurable system architecture, a semantic metadata driven system of systems management and composition, and a reliable and deterministic communications link.

Leveraging existing technologies as well as the development of new capabilities, REATSS provides the NASA IV&V facility the ability to perform flight software testing on actual flight software binaries in a simulation environment that is tailored to the spacecraft platform. These capabilities are not limited to NASA's needs or their target hardware, but rather are easily extensible to applications requiring testing of software where purchasing actual hardware is not cost effective, such as military aircraft, onboard ship computers, and missile guidance systems.

References

[1]Object Management Group, Inc., "Data Distribution Service for Real-Time Systems Specification," OMG ptc/04-4-12, 2004.

[2]Real-Time Innovations, Inc., "NDDS 4.0 Architectural Overview," September 2005.

[3]Simulation Interoperability Standards Committee, "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template (OMT) Specification," IEEE Std 1516.2-2000, 2000.

4Object Technology International, Inc., "Eclipse Platform Technical Overview," July, 2001.

5Virtutech, Inc., "Full System Simulation: Using Simics Instead of Hardware to Develop and Test Software," February, 2005.