You *Can* Handle the Truth: Simulation Architecture for Multiple Truth Engines

James A. Hadley, Steven E. Elrod Timothy E. Etters
The Boeing Company
Kent, WA
james.hadley3@boeing.com, steven.e.elrod@boeing.com, timothy.e.etters@boeing.com

ABSTRACT

High-fidelity training simulation architectures integrate all aspects of the mission system software including the system emulation, sensor/communication models, and truth engines. While this practice is beneficial for updates and maintenance, it creates difficulties when attempting to repurpose the training system to other platforms, develop new models, or reuse models with other truth engines. Additionally, managing multiple projects that require extensive integration efforts of models and truth engines requires time and money that are ill afforded on tight production schedules.

The modeling and simulation team for Boeing Surveillance and Engagement supports software test and training simulations for six intelligence, surveillance, and reconnaissance (ISR) platforms. To support its customers and requirements, the team developed a simulation architecture that separates truth, simulation models, and mission system software into different modules. These modules are tightly integrated, but loosely coupled. As a result, both mission software testing and training system configurations are reconfigurable with multiple truth engines.

This paper reviews key principles on developing a reconfigurable simulation that interfaces with multiple mission systems, expansion of sensor/subsystem models, and creating a common interface for truth data. It also provides lessons learned on how to manage the architecture to ensure future flexibility without having to create ad hoc, single-use solutions. The paper provides guidance on avoiding heavy simulation integration periods from project to project and creating simulation architectures capable of flexibility, modification, and expansion.

ABOUT THE AUTHORS

James A. Hadley is currently a training systems analyst with The Boeing Company in Kent, WA. He holds a Masters degree in instructional technology and is currently an Education Ph.D candidate at Capella University. Over the last ten years, Jim has developed instructional simulations for the U.S. Navy, Department of Energy, and the pharmaceutical industry. His research interests include instructional design, learning theory, and instructional product development.

Steven E. Elrod is an Associate Technical Fellow with The Boeing Company. He specializes in simulation software and software product line architecture for the Surveillance & Engagement organization. Steve has a B.S. in Electrical Engineering from the South Dakota School of Mines & Technology (1981) and an M.S. in Electrical Engineering from Stanford University (1985).

Timothy Etters is the first line leader for the AWS 40/45 AWACS Simulation IPT. The Simulation team provides software for use with mission computing for desktop, labs, and trainers, and provides high fidelity sensor simulations for Boeing surveillance aircraft. Tim grew up in southern Oregon, he graduated from Whitworth University in Spokane, WA, with a B.S. in Computer Science and Mathematics as a double major.

You *Can* Handle the Truth: Simulation Architecture for Multiple Truth Engines

James A. Hadley, Steven E. Elrod, Timothy E. Etters
The Boeing Company
Kent, WA
james.hadley3@boeing.com, steven.e.elrod@boeing.com, timothy.e.etters@boeing.com

INTRODUCTION

Flexibility in a simulation comes at a price. As designers anticipate future improvements, use cases, and upgrades, they must define a simulation architecture that can accommodate expansion, reconfiguration, and modification. In doing so, they create rigidity and standards into the design, rules that must be followed in the future to ensure that flexibility does not result in redesign.

The modeling and simulation designers at Boeing's Surveillance and Engagement division faced a difficult task of designing a simulation software set that could satisfy a multitude of different needs and customers. Rather than identifying all requirements upfront and developing a monolithic simulation set for meeting those requirements, they developed a flexible architecture that allowed them to more easily adjust to changes in the future, and above all, provide the ability to connect to the various truth engines used by their customers.

A truth engine is a software application that generates input data to simulation models. Truth may be a tactical picture developed by flight instructors, environmental data for sensors, or a central source for time to several systems. In this situation, the team worked with many different truth engines for their Intelligence, Surveillance, and Reconnaissance (ISR) mission system software models. Each truth engine had its own output format, different features, and varying levels of fidelity. With all the different sources of truth the team was required to operate with, they needed to develop a simulation architecture that could accommodate the differences and future changes.

This paper briefly describes the situation the team faced and the many facets involved in their solution. Next, it describes the architecture they created and how it accommodated their needs. Finally, the paper provides lessons learned about developing an architecture that can accommodate multiple truth engines.

THE SITUATION

The Airborne Warning Systems (AWS) mission computing group with Boeing's Surveillance and Engagement division develops software for multiple platforms and different customers for those platforms. For example, the group supports mission system software for the U.S. AWACS Block 30/35, U.S. AWACS Block 40/45, French AWACS Mid-life Upgrade, and other domestic and international platforms. While the mission computing characteristics are similar, the systems vary in their capability, fidelity, and output. Additionally, international variants of the aircraft require adherence to International Trade and Arms Regulations (ITAR) and export restrictions. All of these differences require unique software deployments and quality control standards.

The modeling and simulation team supports the software development testing group to perform system verification and validation (SV&V). They develop sensor, environmental, and ownship models to ensure mission computing behaves correctly and within the scope of platform requirements. The group also builds high fidelity communications models that accurately represent the tactical environment in today's network-centric battle space.

Several years ago the team won a contract to provide a simulation set for AWS mission system trainers. The trainers would support up to 15 operator workstations that could be reconfigured into multiple, independent scenarios. The customer required that the operational flight program (OFP) software (the same software that operates on the aircraft) run on the training device and that the team have sufficiently high fidelity models to support it. However, because the team typically supported SV&V activities, they did not have instructor operator software or truth engines that could accurately portray and support distributed training events as required by the contract.

The culmination of supporting multiple programs, ensuring high fidelity models (legacy and future), and developing simulations for both SV&V and training forced the team software architects to reconsider their product architecture. Because their role was supporting mission computing, they were assured the ability to integrate their product as the company acquired new platform contracts. However, the need to support multiple test groups with varying needs, and now the requirement to support training, ensured that the simulation set would need to interact with many different truth engines.

Truth Engines and Requirements

The truth engines used by the team provide information to the simulation about a tactical environment including ownship data, terrain, weather, entities, kinematic data, and also command and control capabilities. The engines support different types of simulated events such as stress loading the system with high entity counts, tactical coordination training, or simply testing system start up and operation. Many of the customers use their own truth engines for testing and the simulation team needed to be assured that their simulation set was compatible with each of them.

The team attempted to meet the following broad architecture requirements:

- Accommodate multiple truth sources and levels of truth fidelity within a single architecture.
- Allow sensor models to be easily reconfigured within the simulation to accommodate different platform configurations.
- Provide high fidelity sensor interfaces to the operational flight program.
- Deploy the simulation on multiple hardware configurations including desktops, high-end stress-load computing, and even the aircraft platform (airborne embedded training).

Using internal research and development funds, the team took 18 months to develop the requisite architecture and simulation set.

THE ARCHITECTURE

Given the new requirements and demands on the simulation system, the team emphasized rigidity in how simulation elements integrated, but also wished to create a flexible system to allow different pieces to interface with the simulation. This was not a contradiction of terms, either.

The underlying principles were based on separation of the three major components of the simulation: the truth engine, the simulation models, and the operational flight program (Figure 1).

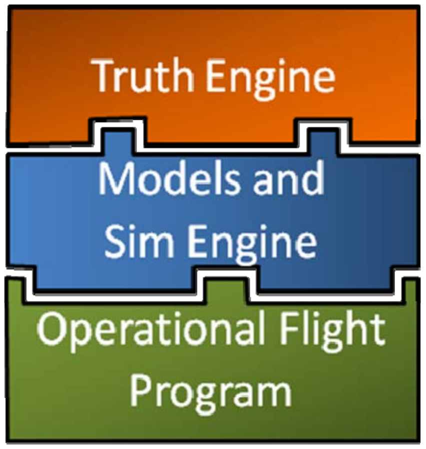


Figure 1. Separating the three major components of the simulation.

The division of these components was both practical and necessary for two reasons. First, the main software interfaces occurred between the individual components, meaning there was natural separation between the components where data needed to be shared. Second, the truth engines and the OFPs were both outside the control of the group. For example, the truth engines were developed by different groups or companies and the simulation team had little input in their design or modifications. The OFP capabilities were determined by the customer, requiring the simulation group to eventually model their characteristics. So the team had the most freedom to develop their architecture in between those components.

Major Components

Truth Engine Characteristics

Due to customer requirements and internal operations, the modeling and simulation team was required to use several different truth engines. Some engines were proprietary toolsets and could more easily be modified by the team if necessary. However, several engines were COTS/GOTS products or, in some cases, provided by subcontractors to the specific programs. At the time, a total of seven different truth engines needed to interface with the simulation.

Some engines were used to record and playback system performance to test system integration with low-fidelity input. Other engines provided robust kinematic data about the tactical environment with terrain and weather generation capabilities. Engines used for training systems also incorporate distributed protocols such as the Distributed Interactive Simulation (DIS) format.

While the truth engines varied, they provided the following characteristics, at widely different fidelity levels:

- Scenario data
- Ownship model data
- Synthetic environment
- Flight dynamics
- Entity attributes

All of these characteristics were managed by a truth engine executive that coordinated entities and terrain and maintains scenario time. The scenario files included information about the entities, such as their location and cued events. Depending on the truth engine, the ownship models varied from mere location, heading, speed, altitude information to specifics about navigation or sensor system malfunctions. The fidelity of the synthetic environment differed from engine to engine for maps, digital terrain elevation data (DTED), atmospherics, and weather. The flight dynamics model, or kinematics model, directed how different entities operated in the environment, such as turning rates for certain aircraft flying between way points. Finally, the entity attribute files provided specific information about each entity such as type, identify friend/foe (IFF) mode codes, radar cross-section, and weapons loadout.

Out of necessity, the entity attribute files are also shared directly with the simulation and sensor models because the data are directly applicable to certain sensors. Together, the truth engine executive provides data to the simulation/truth interface (see Figure 2).

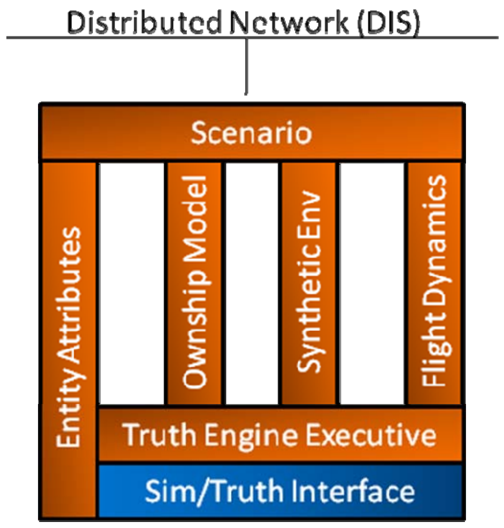


Figure 2. Common truth engine architecture.

Some truth engines allowed several live/virtual participants and made distributed network capabilities available to coordinate multiple ownship models and data link communications. Given customer and internal requirements, the team used the Distributed Interactive Simulation (DIS) standard to communicate via protocol data units (PDUs).

Because the simulation and sensor models required specific inputs, the team created specifications for the simulation/truth interface where truth engine data were translated. This interface is clearly defined and structured. This strict adherence to interface protocols was one of the most important characteristics of the architecture and is explained later.

Models and Simulation Core

The next component of the architecture focused on the simulation, sensor, and communications models. Because the mission system was the same software used on the aircraft, the models had to provide interface inputs that mimicked actual inputs from the real sensors and subsystems. Similarly, the models had to accept inputs from the truth engine that represented information from the ownship and tactical environment.

In order to support multiple deployments, each model was designed with strict input/output parameters. This allowed interchangeability between models under different mission deployments. For example, the AWACS aircraft has several international derivatives, each with similar sensors and communications subsystems. Due to ITAR issues, these sensors or subsystems may be limited in capability. The models, therefore, had to be modified to meet restrictions and customer requirements. These changes would not necessarily affect the mission computing software, so the input/output had to be consistent.

The simulation core manages all of the data from the truth engine and attribute files to the models. This model coordinates time, sensor inputs, and navigation. It also provides data back to the truth engine for ownship, active sensor activity, and data link activity. Figure 3 shows a block-level representation of the model architecture.

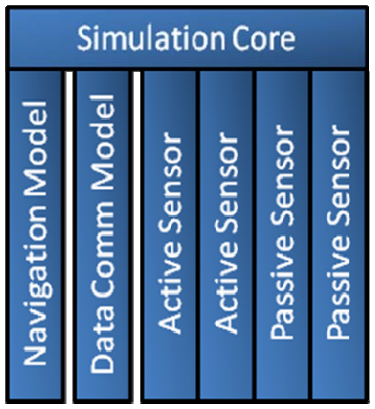


Figure 3. Block-level representation of the model architecture.

The simulation architecture and characteristics are explained in more depth in the following section.

Operational Flight Program

Due to ITAR and proprietary issues, the architecture of the OFP cannot be discussed in this paper. However, there are many benefits to using an OFP in a training simulator versus using a system emulation.

As military systems advance further and further into the digital age, the ability to integrate mission sensors, communications, and tracking tools into a single software set allows for better situational awareness of the tactical environment and easier maintenance or updates to the system. Many legacy military aircraft consist of individual sensors throughout the aircraft, each with their own processors, displays, and interface controls. Operators monitor sensors at a work station consisting of separate displays and indicators to analyze/identify/track contacts. Multiple operators or specialists relay sensor data to senior operators or mission crew members who build the tactical picture and make decisions about how to employ the platform.

Twenty-first century mission systems, however, specialize in integrating sensor data and leveraging digital tools and capabilities to automate many of the menial tasks, such as establishing a track on a contact, navigating routes, and analyzing signal data. Additionally, networked operational workstations allow mission crew members to share the same data and build a complete tactical picture. Operators still play an integral part, but their specialized skills are now focused on activities that the computing systems cannot do well such as threat analysis, tactics, and sensor interpretation.

In modern training systems, many DoD acquisitions are also leveraging the existing mission computing software to run natively on the training hardware. This approach ensures students have concurrency between the platform and the training device (a situation that is not common with legacy platforms). This improvement exposes students to the up-to-date graphical user interfaces, system behaviors, and actual algorithms used within the mission system rather than emulations of the systems that lack fidelity (Turner, Barnes, & Woodall, 1996).

The major benefit of using the OFP on the same simulation model used for both SV&V activities and for training is that the simulation models are validated early on in the development process. Additionally, the simulation can be deployed in desktop configurations early on for train-the-trainer activities and courseware teams can also use the systems for obtaining media and system functionality information.

The simulation software, then, “plugs” directly into the mission system software of the OFP, meaning it uses the same data interfaces as those used on the aircraft (see Figure 4). The sensor models provide high fidelity input that can be used for either testing or training. Again, due to proprietary reasons, the architecture of the OFP cannot be shown in this diagram.

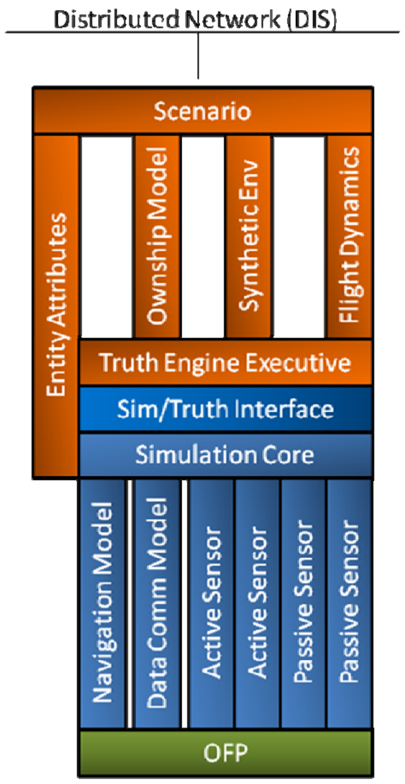


Figure 4. Block diagram of truth engine, simulation, and OFP components.

The critical points of this architecture are between the Sim/Truth Interface and the Simulation Core where truth data are passed back and forth. Also, note that the Entity Attributes interface directly with the Simulation Core as well as the Truth Engine Executive. The Entity Attribute file aligns with the scenario files and passes critical sensor data to the sensor models.

In test or training situations involving weather or terrain, the truth engine manages the synthetic environment, entity locations within the scenario, and the ownship model, and then, through the Sim/Truth Interface, determines if weather or terrain create occlusion or degradation impacts on the sensor models. For example, if the DTED information indicates that a mountain range is blocking an entity from the ownship line of sight, that information is passed through the Sim/Truth Interface to the Simulation Core to either indicate signal loss or degradation to a particular sensor.

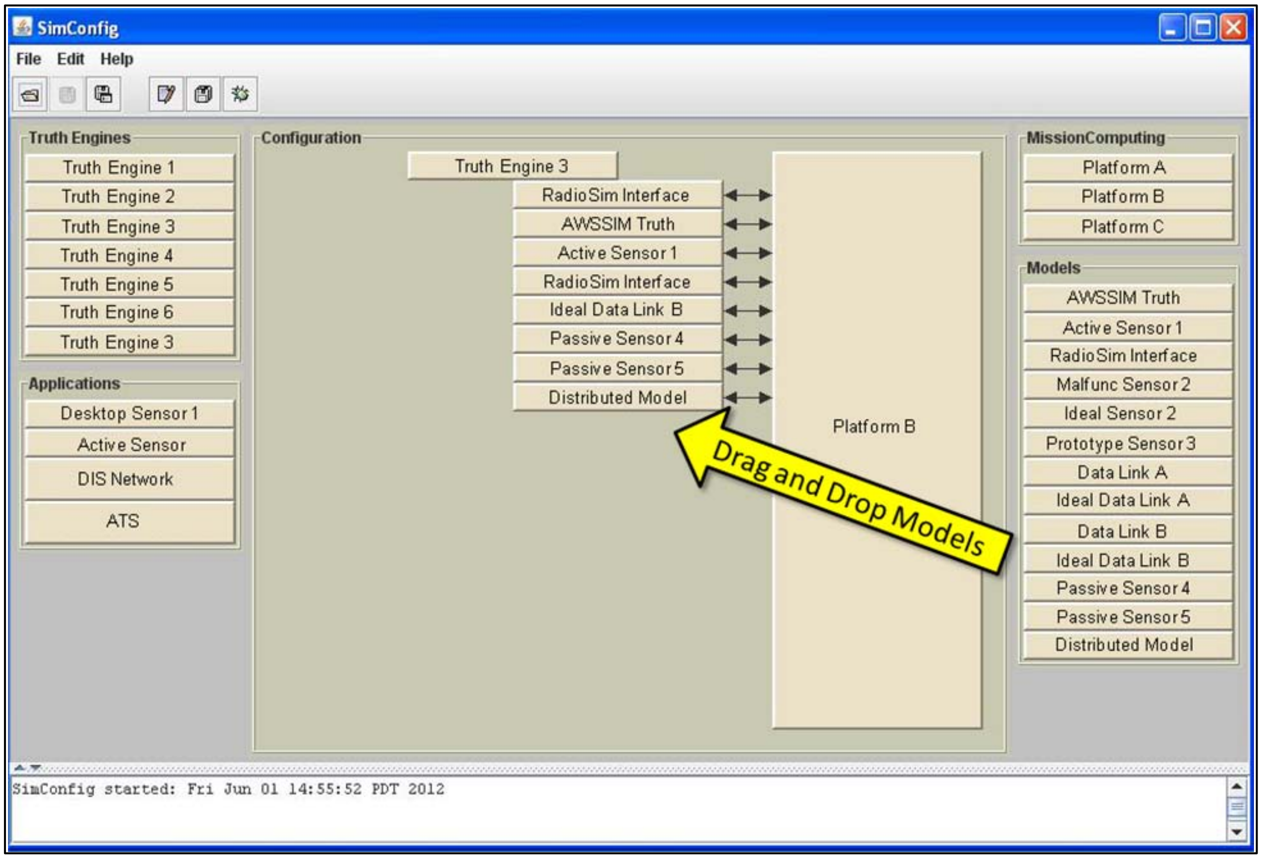
While the overall architecture is extremely complex, with many interfaces, there are deliberate separations between the components to ensure flexibility and future modification. The following section covers the characteristics of the architecture.

CHARACTERISTICS/BENEFITS OF THE ARCHITECTURE

In order to meet the heavy requirements the team created for itself, the simulation needed to have certain characteristics. One key need was to develop a capability to reconfigure between the truth engines, models, and the OFP. Next, the simulation architecture needed to be scalable to various hardware deployments, plus resolve time synching and multiple thread processes. Finally, the developers needed to define how different truth engines would interface with the simulation.

Reconfigurability

The goal of a flexible architecture is the ability to reconfigure the system in many different ways. For example, as new customers purchase a platform, they select sensors to meet their mission needs. Some sensor models may already exist, while others must be created or modified. In the case of international customers procuring derivative aircraft, system capability or fidelity may be limited by export restrictions. In all of these situations, the ability to develop a library of



```
SimConfig started: Fri Jun 01 14:55:52 PDT 2012
```

Figure 5. Screen capture of the simulation configuration tool (image has been altered for export purposes)

models is a cost-effective way of reusing or repurposing.

To achieve this goal, the team developed a simulation reconfiguration tool, essentially a software application allowing users to select a platform, add sensor and subsystem models, and then choose a truth engine (see Figure 5).

The tool allowed users to modify the I/O scheme they were using, such as switching between a 1553 data bus interface or a desktop Ethernet connection. These options supported different hardware configurations where the simulation would be deployed. Additionally, the tool provided settings for software testing such as log services, measuring instruments, and selecting the fidelity level of models (ideal scenarios, radar constant false alarm rates, etc.). Finally, the tool provided inputs for interface configuration, frame rates, IP addresses, ports, and exercise IDs. This tool is essential, and upon configuration creates a script that identifies how all the elements of the simulation will work in concert with each other during runtime.

Tightly Integrated, Loosely Coupled

The phrase “tightly integrated, loosely coupled” might be best be explained with the analogy of Henry Ford’s use of interchangeable parts in automobiles (McGregor, Northrop, Jarrad, & Pohl, 2002). Individual parts have a specific function, such as an oil filter or a carburetor. However, parts from vehicle to vehicle are interchangeable because the vehicle frame and design provide for it. Rather than building customized solutions from one vehicle to the next, strict specifications for placement, inputs, and outputs for parts accommodate easy removal and replacement of parts without modification to the structure or part.

This same concept applies to the simulation and modeling architecture. Individual models follow strict parameters in relation to receiving simulation core inputs and mission system outputs. Tightly integrated refers to having standard interfaces, while loosely coupled refers to easy removal/replacement of the models. Likewise, the models interact with only a few other components in the system. They have a single function that is independent of the rest of the system with discrete connectivity to other components.

Scalability

Building full-scale simulations requires vast amounts of hardware to house databases, process data, and display tactical situations. Those configurations are difficult to scale down and operate as a part-task trainer or even a desktop trainer. The simulation team knew upfront that the simulation would need to be deployed in various use cases. These use cases included:

- Hardware-in-the-loop
- Stimulated to real aircraft hardware
- Desktop testing stations
- Interim training devices
- Full-scale training devices
- On-aircraft embedded training

The team faced several challenges in meeting this requirement. With several systems needing to synchronize, they wrestled with resolving time issues. Also, because of the multiples uses of the simulation, from load testing to functional testing, they needed a solution for data processor management.

Time

One of the main difficulties when coordinating between separate and reconfigurable simulation elements is how time is handled. Some components, such as the OFP or sensor models, require consistent and accurate time representation with sensitive margins for frame overruns. Time may be defined by the truth engine as scenario time, or time may be defined by the operating system based on a hardware timer. However, when the elements operate over multiple pieces of hardware, all the elements must synchronize. Otherwise delays and frame overruns occur, causing the simulation to represent inaccurate data.

When the team initially developed the architecture, time was defined by several time events. This created difficulties with time references. Finally, the team resolved the issue by centralizing control of the single timer event within the simulation coupled with prox’ied control of OFP time. It is the master time/event manager.

Multithreading

Just as managing time is essential between multiple elements, processing becomes important when dealing with potentially different data loads. The team needed to be able to control and adjust the data threading model within the simulation core. For functional testing at a desktop, the simulation may run twenty applications with only 50 entities in a single sequential thread. However, when doing stress load testing across multiple machines with 2,000 entities, it was necessary to manage the thread model with multiple cores. The thread model provides mapping to different hardware configurations such as running the truth engine on one machine, using hardware in the loop, and even running simulation models across multiple machines. The team included capabilities within the simulation configuration tool to control the multithreading model.

Because of the different deployment configurations, the team also needed the ability to either deploy the software as a single executable file or as multiple executables launched on different machines. These elements were defined in the simulation configuration tool and executed following the configuration files specification.

Developer Kits

Truth Development Kit (TDK)
Because of the strict I/O parameters for the models and the simulation engine, the team was able to develop a development kit specifying how truth engines would interface with models. This documentation has proven essential when new projects emerge with either customer-mandated truth engines or using highly specialized engines to test mission system software.

One such occasion occurred when the team began development for an update of a legacy platform. The government customer was upgrading their OFP but wished to continue using the same truth engine vendor. The team subcontracted the vendor and provided them with the detailed specifications in the developer kit. As a result, the team was able to develop a completely integrated solution even though different parts of the system were designed by different groups.

Model Development Kit (MDK)
The team also created a model development kit for guidance on developing different models that could be added to a model library. The kit is beneficial for expanding the simulation capabilities for different platforms. For example, the organization needed a new radar for a ground-based command and control system. Knowing they would use the simulation architecture, developers from another organization followed the MDK to create the needed radar model. As a result, it easily integrated into the existing simulation, saving a lot of time and effort trying to define interfaces.

Multipurpose

The benefit of the architecture in the end was that it allowed the simulation to be used in multiple environments. To date, the architecture has been leveraged for development activities, functional testing, stress-load testing, training devices, demonstrations, mission planning, and concept of operations planning.

Another benefit is that testers operate the same user environment from program to program. Despite having multiple platforms, the testers are familiar with the tool sets that are used. Similarly, they are able to reuse scenario files from program to program with only minor changes. The overall benefit is having individuals move across programs but do not have to be retrained or ramp-up on a completely different set of standards.

LESSONS LEARNED

While the team developed a successful architecture and continues to improve it, there were many lessons learned in the process that other organizations can take advantage of if they initiate a similar architecture.

OFP versus Emulation for Training Simulations

Use of the OFP in a training simulation has many advantages. It ensures an accurate system representation for training. Updates to the aircraft software can be done concurrently in the trainers, thus ensuring operators are always training with the most recent version.

The difficulty with using an OFP, however, is that it requires very close integration with the organization that developed it. The modeling and simulation team was already embedded with the mission systems software group and could easily coordinate changes and updates. However, this is not always the case. With many legacy platforms, the aircraft software may be built by one company, upgraded by another, and the trainer built by a completely different organization. Modeling and simulation groups may run into proprietary information issues or simply have difficulty gathering accurate system performance data. In these cases, the organization may have no other choice than to build an emulation of the system and base fidelity on training requirements rather than system specifications.

Groups that intend to develop this type of architecture should conduct cost benefit analyses to determine whether using an OFP is feasible or even possible. They should determine upfront risks to gaining data, establish proprietary information agreements with the original equipment manufacturer (OEM), assess timelines to having technical data, and identify subject matter experts on both the customer side and OEM side who can interpret data or gather missing information. If these issues are not clearly established and understood, then a system emulation may be a lower risk option.

Product Line vs One-Off

A major difficulty in any software organization is the compromise often made between developing a one-off product and a product line software package. On one hand, the software team has a client or set of clients making very specific demands of the product. They want certain features and capability to meet their needs. The software team attempts to meet these requirements and satisfy customer demands.

However, over time the team may have many similarities from product to product. In these situations the organization can benefit from a product line approach where they attempt to build up the features and capabilities into a single product. This makes the company more profitable and efficient by not having to expend effort to "recreate the wheel" each time. The product is therefore cheaper and can be fielded more quickly.

The difficulty with a product line approach, though, is getting a commitment from leadership and other developers to make it work. Because customers levy different requirements on a program, the programs are typically only interested in satisfying their own requirements. They may not care that XYZ program has a similar but slightly different requirement. A product line perspective and commitment means that leaders attempt to compromise between programs while still satisfying (or negotiating) with their customers. This is not easy to do. Additionally, program leadership should be willing to migrate from older systems to updated product line systems.

Finally, the commitment to develop a product line also means funding updates and improvements. The modeling and simulation team was initially supporting programs from a reactive point of view, meaning they responded to requests from the programs. Eventually, though, they requested internal development funds to develop a new simulation set that would support the programs rather than be reactive to them. Funding is also needed to maintain the product and make needed improvements. Securing that funding and demonstrating a return on investment is essential to the success of the team and the effectiveness of the software.

Different Programs

Because the simulation set supports so many different programs, it has created a difficult situation with meeting different development schedules. One program may require frequent tests and need weekly software updates, while another program is less rigorous and requires only monthly updates.

Rather than being at the whim of each of the programs the team must be proactive in how it manages its own software builds. This includes developing daily and weekly builds made available to the programs and managing requirements internally. They also prioritize fixes, bugs, and updates in a way that supports the product line approach and can provide maximum benefit to all the users.

Regular coordination with the programs is essential. The team actually has bi-weekly meetings with program leads to inform them of new functionality and changes they will be receiving in their next build. A similar meeting is done on a bi-weekly basis within the team to ensure everyone knows what improvements are coming.

CONCLUSION

Since first implementing the new architecture over four years ago, the simulation and modeling team has a much greater ability to meet the demands of current and future programs. By separating the major components and emphasizing a product line approach, team members and sub-teams are now able to specialize in certain areas. For example, the team currently is divided into a modeling group, infrastructure group, and test/integration group. Because of the standards established early on, the team is more able to provide accurate estimates for new work and can easily predict how new sensor models, truth engines, and OFPs will need to interact with the overall system. This greatly decreased time and money on a program and significantly decreased risk.

ACKNOWLEDGEMENTS

Special thanks to all the members of the AWSSim team for their hard work and dedication. Thanks to Paul Phillips for bird-dogging our paper and providing valuable feedback. Also, a special vote of gratitude for the testers and program leads who provide input to improve the simulation. And finally, thanks to our families for supporting us through the long hours and arduous times.

REFERENCES

McGregor, J.D., Northrop, L.M., Jarrad, S., & Pohl, K. (2002). Initiating software product lines. *IEEE Software 19*(4), p. 24-27.

Turner, C., Barnes, K., and Woodall, K. (1996). Use of operational flight programs (OFPS) on maintenance trainers. *Proceedings of the Interservice/Industry Training, Simulation, and Education Conference 1996*. Orlando, FL.