COMPOSABILITY AS AN ARCHITECTURE DRIVER

David R. Pratt, L. Charles Ragusa, and Sonia von der Lippe
Applied Software Systems Engineering Technology Group
Science Applications International Corporation
Orlando, Florida

There are two main issues that must be addressed when building a composable simulation system. The most obvious is identifying the functional granularity of the composition. The choice for granularity defines the modules of the systems and therefore their building blocks. Compatibility across the functional composition boundaries can best be thought of as syntactical consistency (e.g., when two systems can exchange the agreed upon data in a clear and unambiguous manner). The semantic granularity of the system, the second and initially often overlooked issue, is where things like thoughts, concepts, and level of interactions begin to separate the system components. These divisions go a step further to start to define groups of modules that "make sense" together. Even though all the components might be able to exchange data with each other, the key is to exchange meaningful information in a flexible and timely manner. It is this capability the leads to a truly composable system.

There are similar architecture-related discussions concerning the computational and communication model of the systems and the target languages to be used. In the ideal world, none of these other issues should affect the design of the system. In practice, however, these issues are often the overriding determinants. The implementation, communication, and computational models tend to limit the resource allocation available to the software components. These limitations feedback to the designers and further constrain the architecture.

This paper will examine composability from an architectural perspective with Computer Generated Forces (CGF) as the target domain. We discuss an inversion of the traditional approach of first bounding the problem by decomposing the documented requirements, designing an architecture based upon required model functionality, and then negotiating interfaces based on model algorithmic needs. While this approach has worked in the past, very often after the system is operational, a new model requirement is added to the system that causes a violation to architecture precepts. Over time, this has caused many a system to become bloated and brittle. Instead, we start by stepping back from any specific model algorithmic requirements. This allows us to develop a component architecture that characterizes the information flows between notional categories of system components and not the specific implementation nor the functionality of the modules of a component. Some of these information flows are time-critical and high bandwidth, while others are broadcast, low bandwidth, and not time-critical. By building a system that supports the necessary types of data flows between categories of modules, a generalized interconnection context is developed. Furthermore, since the various model components are based upon the data flows rather than specific algorithmic centric views, we can compose and extend them as needed by the specific system application.

Dr. David R. Pratt is a Chief Scientist/Fellow at SAIC ASSET Group. Prior to joining SAIC, he had been the JSIMS Technical Director and a tenured Associate Professor of Computer Science at the Naval Postgraduate School. Dr. Pratt received a Ph.D. in Computer Science in 1993 and a Masters of Science degree in Computer Science in 1988 from Naval Postgraduate School.

L. Charles Ragusa is a Software Engineer at SAIC ASSET Group. Prior to joining SAIC, Mr. Ragusa worked in technical sales. He holds a Bachelor of Science in Chemistry (with honors) from the University of Central Florida, and is working on a Bachelor of Science in Computer Science.

Sonia R. von der Lippe has over thirteen years experience in software engineering focusing on user interfaces, software process, and system and software architectures. At SAIC, Ms. von der Lippe is a Staff Scientist in the ASSET Research Programs Division and is the Principle Investigator for the Composable Behavior Technologies (CBT) project. Ms. von der Lippe received her Bachelor of Science in Computer Science from Clemson University in 1987.

COMPOSABILITY AS AN ARCHITECTURE DRIVER

David R. Pratt, L. Charles Ragusa, and Sonia von der Lippe
Applied Software Systems Engineering Technology Group
Science Applications International Corporation
Orlando, Florida

INTRODUCTION

There are two main issues that must be addressed when building a composable simulation system. The most obvious is identifying the functional granularity of the composition. The choice for granularity defines the modules of the systems and therefore their building blocks. Compatibility across the functional composition boundaries can best be thought of as syntactical consistency (e.g., when two systems can exchange the agreed upon data in a clear and unambiguous manner). The semantic granularity of the system, the second and initially often overlooked issue, is where things like thoughts, concepts, and level of interactions begin to separate the system components. These divisions go a step further to start to define groups of modules that "make sense" together. Even though all the components might be able to exchange data with each other, the key is to exchange meaningful information in a flexible and timely manner. It is this capability the leads to a truly composable system. This paper attempts to address this capability in two ways. The first is a brief tutorial on some means to achieve composability. The second is a description of a simple test bed system that used composability as its driving requirement.

Thorpe, in his "Dial a War" paper, was one of the first to address composability in M&S system [1]. His concept was to bring together a collection of low-cost manned simulators, based upon the needs of the exercise, to train the crews in a team-training environment. The foundational idea here was to bring together those systems that are needed and no others. This system evolved into the Simulator Networking (SIMNET) program and proved to be the model of how most of us view composition to this day.

A refinement of the "dial a war" concept is the development of tools that reside on the network to support the management of the exercise. Motorola's ModIOS [2] typifies this type of composability. Fixed format messages are passed around on the network and provide the means for all communication. Still, the composability is largely based along functional lines. In this case, rather than warfare capabilities, it is in the role the system plays in managing the exercise.

In his discussion of composability in the Joint Simulation System (JSIMS), Butler develops a taxonomy and details how the program is meeting the composability requirements[3]. Using the taxonomy he developed, we are primarily interested in the plug and play composition at the component level.

Barham discusses the development of the Dynamic Simulation Environment (DSE) a system very similar to the concepts put forth in Butler's paper [4]. The difference being that Barham approach details a greater functional granularity of components. What is interesting in both cases, is there was a single infrastructure to support the multiple functional modules.

This paper takes a slightly different tack to composability. Rather than focusing on the requirements of the system in warfighter terms, as Butler does, or in the description, as Barham does, we focus on the operations and structure of the system in as it relates to composability. This results in a focus more on the "how" of the system vice the "what" of the system.

It is at this point where the system designer has to make a hard choice: performance or flexibility. Performance is measured in more than just execution speed or the number of entities on an individual box. Performance can also be measured in terms of development time and cost, and amount of supporting infrastructure required. It is not a black and white choice; there are a series of steps that can be taken to still have a very flexible system with good performance. The concept being that a flexible system will never perform as well as one that is optimized for a given task. With this in mind, the developers must ask themselves, is this system going to be used for more than one task? If this is the case, then the concept of amortizing the performance of the system over the range of applications comes into play and the concept of composability gains merit.

CAVEAT

Throughout this paper we will deal with syntactical composability. This is the ability to bring together parts of a simulation in such a manner that they interact with each other in a technically valid manner. We have purposely side stepped the issue of semantic composability. Semantic composability is when the components of

the simulation make sense to each other. From a syntactical point of view, a single F-16C tail number N12345 can do a wing over and drop two bombs on an infantry division. Granted it makes no semantic sense to have models of such differing fidelities operating together. Pulling together components that make sense is considerably harder and left as an exercise to the reader. Rather, we will focus on the being able to match inputs and outputs from the syntactical point of view.

KEY REQUIREMENTS FOR COMPOSABILITY

Composability, or the ability to build new things from existing pieces, has been one of the software engineering holy grails for many years. While most computer scientists prefer to think of composability as software reuse, the basic concepts are the same. The main difference being that we tend to think of software reuse as something we do once as we build a basically static system. Whereas, a true composable system is constantly pulling-in and shedding parts of the system, based on the current needs of the exercise. In this section we will discuss a few of the key requirements to allow this type of system reconfiguration.

Limited, Well Defined Interfaces

Perhaps the most crucial aspect of any system is that its interfaces be well defined. This is even more critical for composable systems where tools, not humans will be bringing the system together.[1] Not only do the interfaces need to be fully specified in the content of the interaction (for example, a method name and parameters) but the pre- and post-conditions also need to be fully specified. A key item here is not only do all the client interfaces, and the services it provides, need to be specified, but also all of the services that the subsystem uses must be clearly specified. It is not uncommon to try to pull-in a single module only to find out during link time that it contains numerous undocumented calls to other modules.

Current state-of-the-art in software development provides two basic types of interfaces: inheritance and Application Programmer Interface (API). While, in reality, most Object-Oriented (OO) systems have made use of both types of interfaces, we shall treat them separately. Inheritance is used in OO and framework based systems. By and large, this is the harder of the two systems. Since the functionality added to the system directly uses the services in the parent classes, the parents needed to be designed and in most cases implemented before the child can be constructed. Furthermore, the child can view, modify, or restrict the functionality provided by the parent. This capability requires that the parent classes are completed, or at least have reached a stable state, before the child classes are started. Concurrent development of the children and the parent is a recipe for significant problems.

Applications Program Interfaces (APIs) are used primarily in component-based systems. These are normally implemented as a series of procedural calls. Unlike inheritance, the client has no knowledge of the implementation or structure of the component providing the service. This allows a certain amount of "black box" development. As long as the procedure's signatures do not change, the user is buffered from any internal changes to the implementation of the component. This allows for a certain amount of concurrent development.

Both types of interfaces have to abide by a subtyping relationship. A subtyping relationship is defined as the child can be used anyplace the parent is used with no change in apparent functionality or in the interface. Basically, this requires that the children's interface be a superset of the parent class. This is not to say that the child can not have a different implementation or add new functionality, it can. The simplest way to implement this in an inheritance-based system is to have all of the communications go through the base class. In the case of the component based system, any new component must have, at a minimum, the same interface as the component originally designed to be used.

Abstract Representation of Simulation Elements

Most architects would not think of designing a warfighting simulation system that had a different software representation for each entity in the battlespace, yet think very little of breaking out each type of entity, or at best, each domain, into separate components and developing specialized routines to handle them. By taking a step back from the warfighting entities in a simulation, we notice that there are certain constructs and actions they all share. By dealing with the abstract representation of the entities, we can freely change the instance or type of entity we are dealing with.

Documented Repository

From experience, the major hurdle to code reuse is the ability to find out what code is out there and what does it do. It is not uncommon at the start of a program to hear about all the code reuse that is going to take place during the development phase. Yet, in reality, not much

[1] The key difference between manual composability and tool-based composability is the amount of additional data the user can infer about the modules. Most, if not all, tools do not have the ability to draw inferences about the module. Thus, tool based composability is a higher standard to achieve.

really happens. It is not that the developers don't want to reuse code, although that is sometimes the case. Rather, it is very often harder and more time consuming to try to understand what is going on from poorly documented source code and out of date design documents than it is to just start from scratch. That is assuming that the developer can find the code in the first place. To compose a system, the components have to be documented and readily accessible.

This is even more so the case when we are dealing with tool-based composition. In situations such as this the components and the information about them have to be understandable to that tools building the system. This is normally thought of in the sense of building a repository that contains not only the software component, but also metadata about the component. Joint Simulation System (JSIMS) is taking this approach with the JSIMS Modeling and Simulation Resource Repository (JMSRR). The Java Beans component architecture is a different approach that requires each Bean to make its interfaces visible via a process of introspection and reflection. This allows builder tools to compose new application from the existing Beans with out modifying the underlying code.

Stable Infrastructure

Building a composable system is much like building a house. In both cases most people are more concerned with what the system looks like than what it is built on. Yet, if both do not have a firm foundation, they will soon crumble. For a composable software system, the foundation is the architectural infrastructure. This is not to say that the computational infrastructure itself can not change. There is nothing preventing any piece of the system from being replaced. Rather, the interfaces, expectations, pre- and post-conditions, and the resource allocations must be stable and understandable. If the architectural infrastructure changes components might be rendered obsolete or be forced to change their interfaces.

Very often a stable infrastructure is thought to contain a stable and reliable hardware and operating system combination. In reality, what is needed is a stable virtual machine with defined interfaces. There is a large temptation to implement a system in such a way that there are hidden system optimizations based upon the developers' knowledge of the underlying operating system and hardware. While this might improve performance for a given combination, it might, and often does, limit the performance and/or portability on other platforms. This is another case where by violating the limited and well defined interface principle, the system composablity now excludes the computational platform.

TYPES OF INTERACTIONS

The proof of an interoperable system is its ability to support interactions amongst the differing components in a consistent manner. For our purposes, an interaction is comprised of three actions. The first is the necessary pre-conditions, what situation has to be in place for the interaction to transpire in a valid manner. For example, before a Height Above Terrain (HAT) query can be processed, it is a pre-condition that the terrain database be opened first. The second phase is the interaction itself. In the example, it is the method call (the computation of the value) and the return of the entity's altitude above ground level. The final phase is the post-condition. This is any change in state of the system. In the height above terrain call example, there is a null post-condition – the state of the system has not changed. However, in the case of opening the terrain database, the post-condition is that now a database is open for processing. Complex interactions can be broken down and expressed as a series of simple interactions with the pre-condition that the previous interaction has taken place.

An interaction can itself be broken down into several components. The three we are most concerned about when dealing with composability are the type, timing, and format components. In this section we will briefly discuss the types of interactions, the next two sections deal with the timing issues and format.

Service Request

By far the most common type of interaction is the service request. A component is asking another to perform an action. For example, a call to the print module to display a line of text on the screen. The caller is requesting that the print service perform an action. Notice that it does not tell it how to do it, nor does it really care. Rather, it is simply asking for a service to be performed.

Status Reporting

A status reporting interaction is a very one-sided interaction. A component publishes an event, change of state, or even its unchanged current state. It does so without knowledge of how the information will be used by the receiver or expectation of any action to be performed. The receivers of this interaction may or may not take action on receipt. The Distributed Interactive Simulation (DIS) Entity State Protocol Data Unit (ESPDU) is a classic case of a status reporting interaction. The status reporting mechanism also supports the anonymous receipt of messages and events, thus al-

lowing for a decoupling of sender and receiver, a la the Observer pattern.[10]

Meditated Interaction

The final type of request is the meditated interaction. In reality it can be either the service request or status reporting interaction. The difference is that the request is sent via a level of indirection to a "middleman" who then routes it to the appropriate recipient. With the growth of middleware products, such as CORBA and COM/DCOM, this type of interaction has become increasingly popular. The fundamental reason for this is the location of the requester and service provider are now abstracted out to a point where nether needs to know where the other one is located. Furthermore, the middleware keeps track of the components, the services they provide, and their physical location. Thus, it acts as a rudimentary infrastructure and communication infrastructure. This level of abstraction aids composability by not having the component developer needing to worry about the where or how the service requested will be provided. But of course there may be some important performance implications, which may be greatly impacted depending on the degree of coupling and frequency of interaction. This leads to the next topic, temporal requirements.

TEMPORAL REQUIREMENTS

There is a saying amongst hardcore software engineers who have seen their elegant designs run pillaged: "Many a system design has been sacrificed on the altar of performance." All too frequently, short cuts in the name of performance are taken in the translation of the design to code. Granted, an extra vehicle might get squeezed on the platform, but at what cost? The flexibility and modularity of the system is often compromised. This results in a system that is, in the long run, less composable and harder to maintain. In this section we will take a look at two of the most common breaches of a good modular and composable design. The section then concludes with the thinking that tends to lead to big monolithic single platform designs.

Direct Variable Access

Just about every OO book talks about the use of accessor functions to ensure encapsulation. By providing a function that is part of the defined interface to get and set the component's variables, composability is increased by providing the user with a more abstract view of the underlying representation. Yet, for the sake of speed, trying to avoid the overhead associated with a method invocation, variables are made public and can be changed directly. If the underlying representation was to change, say by using a new module to replace the existing one, access to variables of the same name and type must be provided.

C++ gets around the need for speed argument by providing the developer with the **`inline`** keyword. What this does is, while it appears to be a method invocation in the source code space, the compiler optimizes the code and places the body of the method in the command stream. As a result, at execution time, the variable is accessed directly. This provides the best of both worlds.

Direct Method Invocation

A lessor problem is the desire to call a method with an explicit set of parameters rather than dealing with an abstract interface. There are two basic cases where this comes into play. The first being a direct call to a service versus going though the middleware layer. Calls to the Synthetic Natural Environment (SNE) are typical of this case. The users feel that there is such a high bandwidth requirement that any additional overhead must be eliminated. Thus, a direct method call is used. The vast majority of the service calls tend to be non-observable to other entities and do not cause a state change.

The second case is a more problematic. Rather than sending a message via the inter-model communication construct, one entity invokes another entity's methods directly. This creates significant problems when a new component needs access to the information contained in the interaction. A very simple case of this is a command that is passed between two units. If the message was passed via a direct method invocation, the data collection system has no way of knowing when the message takes place. The use of the direct method invocation, particularly for command and control data, severely limits composability.

Same Host Interactions

It is widely known that there is at least an order of magnitude difference in the speed of communications between two entities communing locally versus over a network. This led us to the conclusion that we want to have everything running locally to minimize communication delays. The problem with this is, while computers are getting faster, they still have a bounded amount of computational ability. Given the complexity and scale of our current and projected set of models, they may not be able to run on a single platform. For this reason, we must look towards shedding some of the load onto other processors. If the components are developed in such a way that they rely on the sender and receiver of the interaction operating in the same mem-

ory space, we have limited composability to an extent that we are just building a large monolithic system.

Networked Interactions

Network interactions tend to optimize around low bandwidth discrete interactions with large periods of computation between them. Even given this limitation many systems are being designed to require multiple computers. For example, there is a logical separation between that red-force controller and blue-force controller, not much information is passed between them and both systems require a degree of computational effort. In many cases, you do not want to have one seeing what the other one is doing. If this is coded into the system in such a way, where the red and blue controllers have to be on separate computers communicating over the network, composability has been limited.

MESSAGE BASED COMMUNICATION

As we have discussed above, as a means of improving composability, we are trying to get away from the practice of using direct method invocations. Since the interactions still need to take place, we are using an explicit message-passing scheme. This allows the use of both directed messages, the receiver is identified, and multicast, the message is sent to all who have subscribed to a selected group, messages. By making the message-passing scheme explicit, it is visible and passed through the middleware so it can be logged and distributed to all interested parties. As with all of the topics discussed above, the message-passing scheme has an effect on composability as well. In this section, we will briefly discuss the message format issues as they affect composability.

Prior to discussing formats, the issue of separating the content from the transport should be discussed. The SIMNET protocol was defined down to the Ethernet frame-level. The format was very tightly coupled to the transport mechanism. This was done for the sake of efficiency, but as SIMNET networking code was ported to off-the-shelf UNIX machines, this introduced the complexity of having to run as the super-user to read the raw Ethernet packets off the wire. While the design promoted composability, the implementation of the message passing design, which in fact introduced system-level composability to the modeling and simulation community, was partially sacrificed at the alter of performance.

Fixed Format Messages

Of the two types of messages, the fixed format messages are the simplest to deal with. As the name implies, the format is fixed and defined *a priori*. Like all stable and fixed interfaces, it is normally a straightforward manner to program to the known interface. By virtue of it not changing, it is simple manner to compose a new system out of two or more components that implement the interface. The fact that the format is fixed is also a limitation. If there is additional data that must be added to support a new interaction between two components, the standard must be violated or revised to accommodate the new interaction. Depending on the rigor and narrowness of the definition, the fixed format could lead to a very large number of message formats.

Tuple Based Messages

A tuple is a structured set of attributes. A tuple-based message passing system does not really have a fixed format. Rather, the content of the message is based upon the need of the module. Unlike the fixed format messages, a wide variety of data can be passed between components. The advantage to this type of message passing scheme is that the middleware layer doing the transport is completely unaware of the structure of the data being transferred. This allows the same infrastructure to be used for a wide range of application with very little restriction on the components. This flexibility does come at a significant cost. Commonly implemented as "name-value" pairs, or Lisp-like "properties," the tuple needs to be able to be decoded by the receiver. Since the format is not fixed, the interfaces must be negotiated to ensure the sender and receiver can both understand what is being sent. With tuple-based message passing, it is a rather simple matter to implement syntactical interoperability. Achieving semantic interoperability between independently developed components is much harder, and more cumbersome, than it is in the case of fixed format messages.

SIMULATION TEST BED COMPONENTS

Up until this point, all of the discussion has been academic in nature. To determine the viability of the concepts presented and the complexities involved on focusing composability as an architecture driver, the authors developed a simple Java based implementation of a CGF system. The remainder of the paper will present selected portions of the implementation to date and what we have learned.

System Overview

In addition to the composability guidelines discussed to this point, there were two decisions we made early which were pivotal in the formulation of the system. The first is, the system will be a Discrete Event Simu-

lation (DES) based system. This was a logical outgrowth of our desire to have a fast as possible and scaled real-time mode. This influenced just about every facet of the system. The other was that we were going to implement the system in Java on a WinTel platform. Recognizing that making this decision up front was a violation of most good software engineering practices; we wanted to take advantage of some of the features the language supported. The hardware selection was driven by availability. Care was taken to ensure that neither one of these decisions place restrictions on composability or the validity of the prototype implementation.

Scheduler

The scheduler is the heart of the system. Comprised of three parts, the clock, events, and event queue, the basic design was based upon the fundamental DES infrastructure design presented in [5]. The class structure for the clock and scheduler is shown below (see Figure 1). The clock is an abstraction of the system clock. Additional functionality, such as start, stop, and pause has been added to allow complete control of the time. A scaling factor has also been added to allow for scaled real-time operation.

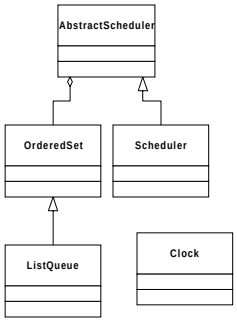

Figure 1. The Scheduler and Clock Classes

The abstract scheduler is not entirely necessary, but it allows us to change the implementation scheduler class without impacting the rest of the program. As with most DES systems, events are inserted into the ordered list based upon the time they are to be executed. The simulator then loops through the ordered list processing the first event until the queue is empty.

The event hierarchy is shown below (see Figure 2). DESComparable is an interface to ensure all events can be ordered by time. DESEvent is the main event class. It contains the two call back methods that pass the processing thread to the appropriate instanced class. The RealTimeEvent checks the clock to determine the current simulation time. It then sleeps until the correct amount of time has elapsed. This, in effect, slaves the DES system to the scaled real-time clock. The TupleEvent is a call back that contains a message from another source.

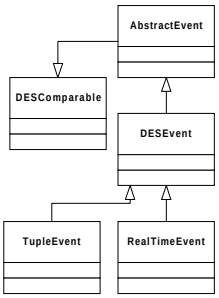

Figure 2. The Event Hierarchy

Data Distribution

Initially, we planned to use JavaSpaces for the embedded Data Distribution Management (DDM) [6]. However, we found a lack of examples and viable implementations. This led us to the use of IBM T-Spaces [7]. At the abstract level, JavaSpaces, T-Spaces, and the High Level Architecture (HLA) [8], all provide the same tuple-based DDM services. This allowed use to encapsulate the services. The DDM hierarchy uses a subtype relationship, which allows the using components to deal only with the methods in the base DDM class (see Figure 3). This allows the supporting infrastructure to evolve independently over time. All incoming messages are placed in the event queue for processing.

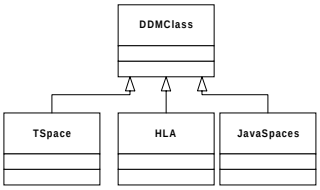

Figure 3. The DDM Hierarchy

Synthetic Environment

Much like the DDM hierarchy, the Synthetic Environment was implemented using an abstract class to hide the real implementations. The ModTerrain project provided insight into the basic set of primitive functions that could be supported across a range of terrain databases [9].

Model Representation

The model hierarchy is shown below (see Figure 4). There are three features of the model representation that promote composablity. The first is the use of a standard model template for all models. This allows us to treat all models using the same interface. In our case, a call back from an event processed by the scheduler calls the models.

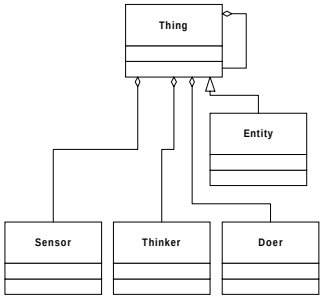

Figure 4. The Model Hierarchy

The second major feature relating to composability is the use of the Composite pattern [10]. As shown in the figure, an instance of the Thing class can be composed of 0..n Things. By doing this we can build a unified structure with the instances of the Thing classes as the interior nodes representing military units. The Entity class represents the leaf node in the Thing hierarchy and the physical devices in the battle space.

The remaining composability driven feature is the use of the robotics view of an entity for composing a Thing. Recently, other researchers have applied the same principle in their developments to promote composability [11]. By dealing with a generic Sensor, Thinker, and Doer, we are able to abstract out the basic interfaces within a Thing, so we can mix and match instances of the derived classes to suit our composability needs.

To ensure composability and future growth, no two Thing instances communicate directly. Rather, the sender sends a tuple-based message to the instanced DDM. The DDM then turns it over to the scheduler to be placed in the event queue. At the appropriate time, the event is processed and the receiver is alerted via a TupleEvent call back. This process is shown below (see Figure 5).

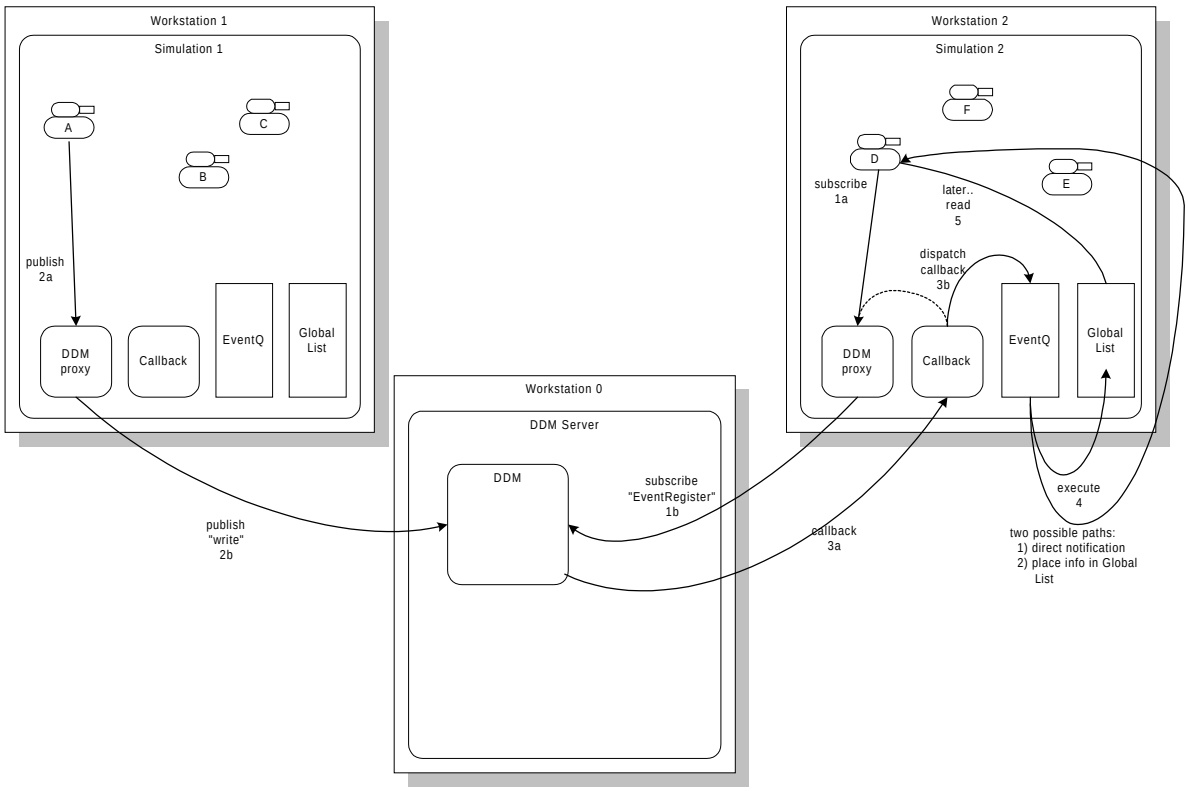

Figure 5. Inter-Thing Communication

Sensor
The Sensor class represents the sensory inputs from the simulated world. By providing a focal point for all inputs, all other objects have a well-defined interface to send messages. The sensor class also contains the algorithms that make determinations as to intervisibility and detection.

Doer
In robotics parlance, the Doer class is an implementation of an effector. The Doer class contains the physical model of the Entity class. In the case of a Thing, the Doer issues orders to the subordinate Thing instances via the event queue. The abstract representation of the physical properties allows the use of multi-fidelity models.

Thinker
The thinker is the cognitive controller of the object. By using a common interface, we can substitute a simple re-active state machine or a complex agent based goal-seeking engine with no change to the Sensor or Doer components. In reality, this trivializes a very hard problem of subtyping.

User Interface

The use of Java on the user interfaces classes was the one place we ran into trouble with the inheritance model. Java does not support true multiple inheritance, and this required that we implement the abstract Input and Output classes as interfaces. The Plan Review Display (PVD) class inherits from both the Input and Output classes (see Figure 6). The abstraction of the input/output classes allows for the use of a low resource text interfaces, a voice interface or a graphical user interface. Using the tuple-based DDM, the interfaces need not be on the same machine as the simulated entities.

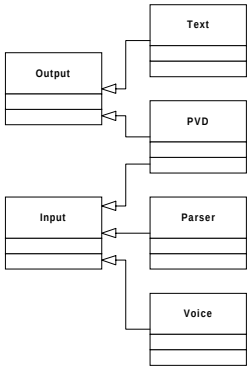

Figure 6. The Input/Output Hierarchy

It has been the experience of the authors that the user interface is the most subjective and individualistic portion of the system. By providing an abstract interface, we are allow the user to develop their own, either from scratch or derived form one provided, without impacting the rest of the system or requiring the user to fully comprehend the inner-workings of the system. Once again, this was done to promote composability.

RESULTS

We are currently in the process of implementing the prototype system. While not complete, early indications are that the system will meet our composability expectations. The combination of component-based development, object-oriented inheritance to provide a common interface, and the use of an abstract tuple messaging system have proven to be quite powerful enablers. As stated earlier, we are absorbing a performance penalty for this flexibility, but as we are frequently reminded, computers are getting faster.

CONCLUSIONS

Designing and building a composable system is not impossible; it just runs counter to our experience and traditional motivations. It does require a certain amount of discipline to avoid cutting corners for the sake of performance. In many ways, this is the hardest aspect of the process. Likewise, forcing the developers to think at the abstract level of base classes and components proved difficult. The use of terminology and the use of diagrams to show the high-level interactions overcame this to some extent.

The old pithy saying "nothing is ever free" applies to a composable system. The initial system will cost more, be more complex, and be less efficient than one that is designed to fit a narrow niche. It is not until the second, third, or maybe even the n^{th} composition does there start

to be a return on investment. As an interesting note, the Frameworks literature has its famous "Rule of 3" where you can't justify a frameworks-based implementation unless you can demonstrate 3 examples that have used and have refined (or will refine) the framework. However, with the cost of computing power going down and the cost of software development increasing, the emphasis is shifting from computational efficiency to developmental efficiency. It is for this reason that composability will become an increasingly more important requirement of the systems.

ACKNOWLEDGMENTS

The authors would like to thank Anthony Courtemanche for his comments and suggestions on the early version of this paper and Susan Smith for editing the final version of the paper.

REFERENCES

[1] Thorpe, Jack A., “The New Technology of Large Scale Simulator Networking: Implications for Mastering the Art of Warfighting,” Proceedings of the 9th Interservice/Industry Training System Conference, Nov. – Dec. 1987.

[2] ModIOS homepage site http://www.mot.com/GSS/SSTG/CSD/C4I/ADSS.html

[3] Butler, Brett "Composablity in JSIMS,", Proceedings of the 1998 Interservice/Industry Training System Conference, December 1998

[4] Barham, Paul et. al.," Dynamic Simulation Environment," Proceedings of the ITEC 99 Conference, April 1999.

[5] Altman, Michael “Writing a Discrete Event Simulator: ten easy lessons,” http://www.labmed.umn.edu/~michael/des/

[6] JavaSpaces homepage at http://java.sun.com/products/javaspaces/

[7] IBM's T-Space homepage at http://www.almaden.ibm.com/cs/TSpaces/

[8] High Level Architecture homepage at http://hla.dmso.mil/hla/

[9] Jackson, Leroy, Dale Henderson, and Shirley Pratt. “ModTerrain White Paper.” 16 February 1999.

[10] Gamma, Erich, et. al. "Design Patterns - Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[11] Lubetsky, Ben and Karen Overfield, " Development of Common Design Framework for Commercial and Military CGF Applications," Proceedings of the Eighth Conference on Computer Generated Forces and Behavioral Representation, May 1999.