SANDIA REPORT

SAND2005-3819
Unlimited Release
Printed July 2005

Umbra's System Representation

Michael J. McDonald

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2005-3819
Unlimited Release
Printed July 2005

Umbra's System Representation

Michael J. McDonald
Intelligent Systems and Robotics Center
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1004

Abstract

This document describes the Umbra System representation. Umbra System representation, initially developed in the spring of 2003, is implemented in Incr/Tcl using concepts borrowed from Carnegie Mellon University's Architecture Description Language (ADL) called Acme. In the spring of 2004 through January 2005, System was converted to Umbra 4, extended slightly, and adopted as the underlying software system for a variety of Umbra applications that support Complex Systems Engineering (CSE) and Complex Adaptive Systems Engineering (CASE). System is now a standard part of Umbra 4. While Umbra 4 also includes an XML parser for System, the XML parser and Schema are not described in this document.

1 Introduction

Umbra's System.tcl library implements a System class definition. The need for a System representation grew from the need to represent or model a variety of complex systems in support of System of Systems analytics as well as CSE and CASE at Sandia. The design and specification of System closely follows design concepts borrowed from Carnegie Mellon University's Architecture Description Language (ADL) called Acme. This paper provides an overview of the relevant pieces of Acme, compares it with concepts in Umbra, then describes and provides examples of the Umbra System class.

It is noteworthy that Acme, like other ADLs, was developed to provide a formal basis for description and analysis of the architectures of component-based systems. Typically, the focus is upon software system architectures. By contrast, Umbra simulations include working models of complex component-based systems which can, in turn, often be described using ADLs. Unlike mainstream ADL work, the focus of Umbra simulations is not typically constrained to software systems. In particular, Systems typically analyzed using Umbra include physical elements or components that are not typically discussed in the ADL literature.

1.1 Acme Overview[1]

Acme is an ADL developed at CMU for modeling complex software architectures. Acme is built on an ontology of seven types of entities for architectural representation: components, connectors, systems, ports, roles, representations, and rep-maps. Of the seven types, the most basic elements of architectural description are components, connectors, and systems.

- *Components* represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include such things as clients, servers, filters, objects, blackboards, and databases.
- *Connectors* represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. Informally they provide the "glue" for architectural designs, and intuitively, they correspond to the lines in box-and-line descriptions. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. But connectors may also represent more complex interactions, such as a client-server protocol or a SQL link between a database and an application.
- *Systems* represent configurations of components and connectors.

Components' interfaces are defined by a set of ports. Each port identifies a point of interaction between the component and its environment. A component may provide multiple interfaces by using different types of ports. A port can represent an interface as simple as a single procedure signature, or more complex interfaces, such as a collection of procedure calls that must be invoked in certain specified orders, or an event multi-cast interface point.

---

[1] This section is strongly based upon documents available at http://www-2.cs.cmu.edu/~acme/

Connectors also have interfaces that are defined by a set of roles. Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as the caller and called roles of an RPC connector, the reading and writing roles of a pipe, or the sender and receiver roles of a message passing connector. Other kinds of connectors may have more than two roles. For example an event broadcast connector might have a single event-announcer role and an arbitrary number of event-receiver roles.

Acme supports the hierarchical description of architectures. Specifically, any component or connector can be represented by one or more detailed, lower-level descriptions. Each such description is termed a representation in Acme. The use of multiple representations allows Acme to encode multiple views of architectural entities. It also supports the description of encapsulation boundaries, as well as multiple refinement levels.

1.2 Acme and Umbra Compared

Umbra applications can be readily modeled using the Acme representation. With a few notable exceptions most Umbra features directly correlate to key Acme concepts. The most notable exception is the use of the term connector which is elaborated below.

Figure 1 shows how a typical two-module Acme system might be illustrated for the purpose of these discussions. The interior components drawn encapsulate some functionality and are interfaced through ports. Components are associated through connectors. These connectors may have two or more rolls, with each roll associating a particular port to the connector. Together the components are drawn in a representation. A system is formed by instancing this representation and encapsulating it as a component by its own right. To hide interior detail, this component's ports are bound to various internal ports through a binding process.

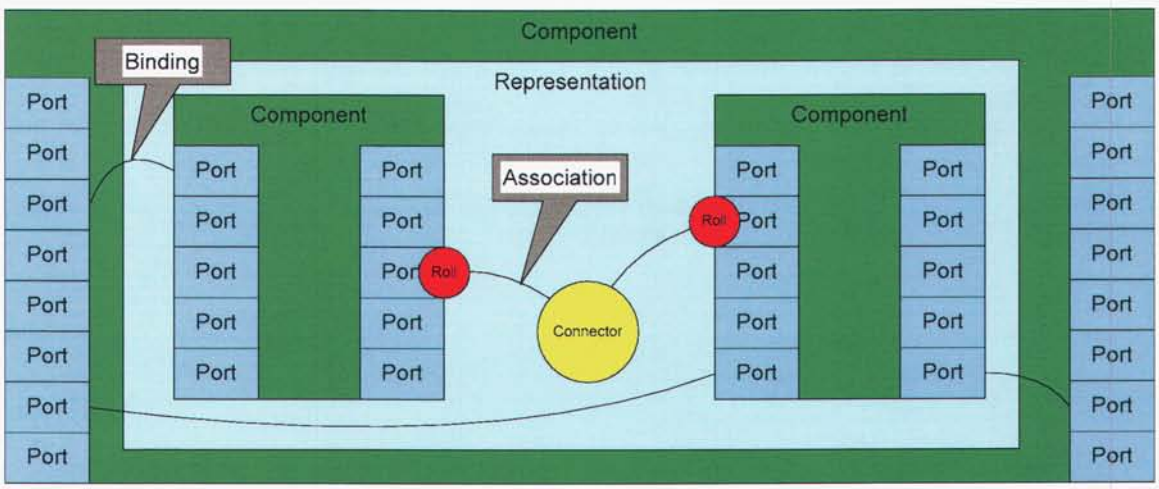

Figure 1: Illustration of an Acme System and its key attributes.

Figure 2 shows a conceptual Umbra System. Key attributes shown in the diagram are the Umbra modules, which encapsulate computation, the Umbra input and output connectors, the connections made between modules, and the Tcl interface points.

Umbra modules are analogous to elemental Acme components. (Typically, they are implemented as C++ classes and instanced through the interpreter.) Umbra modules have four interfaces that would be modeled as Ports in Acme. These are the input and output connectors (which support continuous data flow) and the Tcl method and callback interfaces (which serve as input and output messaging interfaces).

It is easy to confuse Umbra and Acme's respective uses of the word connector. In Umbra, the connector is the interface point. Connections are made between connectors. In Acme, Ports are the interface points, connectors are used to generally implement the connections. Acme's richer representation allows Acme connectors themselves to have computational roles and can be modeled in a hierarchical fashion. Presently, Umbra does not implement features that use this abstraction.

Umbra has three features that would be modeled in Acme as connectors. These are normal (feed forward) and feedback connections between Umbra connectors and the Tcl, procedures that get used to implement application features that use the module's messaging interfaces. In the case of connections, no computation is performed and the connection nearly approximates an Acme association. In the case of the messaging interface, a substantial amount of local computation is performed. This computation can be thought of as the computation within an Acme connection. It is noteworthy that in the case of event callbacks, Umbra uses the term associate to link various callback interface points to specific Tcl procedure calls.

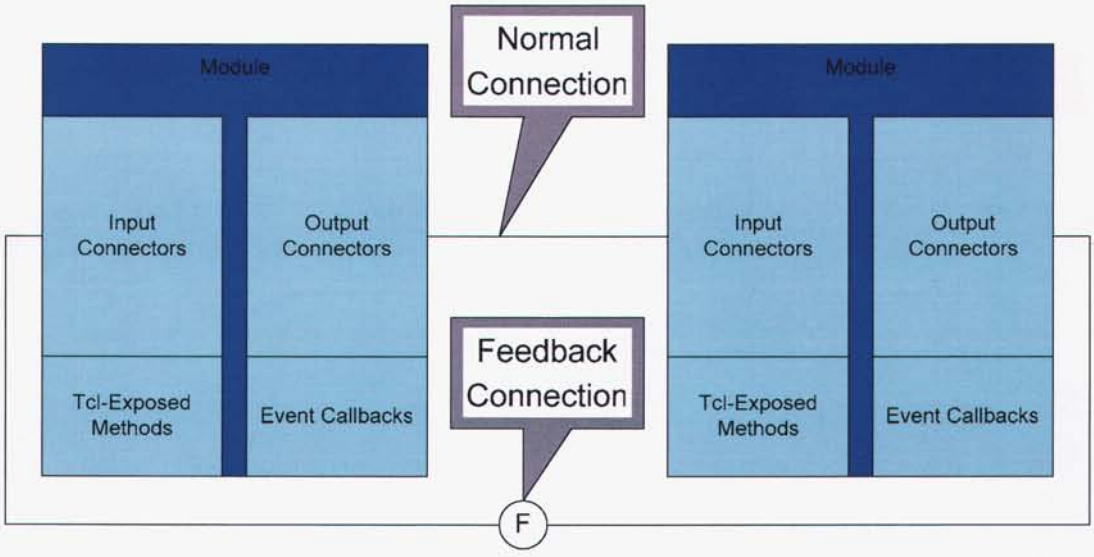

Figure 2: Conceptual illustration of an Umbra module pair combined to simulate a system

1.3 Umbra's System Class

Umbra's System class tries to strike a balance between the very general representations within Acme and the special needs in Umbra. In its implementation, Systems are

represented as incrTcl[2] classes. System classes can also be thought of as Acme System representations. Likewise, System class instances are components that can be treated like any other component.

Table 1 below groups and compares interface methods in the System class against those from Umbra modules. In cases where there is strong overlap, the System class supports methods with the same behavior and syntax as those of the Umbra module interface. It is noteworthy that many Umbra applications and the Umbra user community's discussions are tied to Umbra's use of the word connector to refer to a specific type of port. The System design preserves this preference but also introduces the use of port where it has distinguishing characteristics. In some minor cases, syntax used within Umbra collides with Tcl namespaces. In particular, the set method is problematic when used as an incrTcl class method. In this case, the term port (as in portValue) is introduced to move System closer to the Acme representation.

Table 1) Functional Grouping of Umbra's module and System interface elements

Functional Group	**Umbra Module Method**	**System Method**
System Instantiation & Maintenance	classType	classType
	delete	itcl::delete instance
		deleteClassInstances
		cleanupClass
Port, Connector, and Parameter Query and Management		portNames
	connectors	connectors
	connections	connections
		binding
	get / set	portValue
	connected	connected
	parameters	
	connect	connect
	feedback	feedback
	unconnect	unconnect
Component Queries and Manipulation		componentNames
		component
		componentP
		bind
		component
		sendComponent
Condition and Event Handling	offChange	
	offConnect	
	offSet	
	offUpdate	
	onChange	
	onConnect	
	onSet	
	onUpdate	
	offChange	

[2] IncrTcl is a lightweight object oriented extension to Tcl. Current Tcl releases include incrTcl as a standard component. As a result, documentation for incrTcl can be found within Tcl documentation sets.

1.3.1 Example System

As an example system, we consider ClockSystem which is included with the basic Umbra installation and defined in ClockSystem.tcl. ClockSystem is diagrammed in Figure 3. This System's ports include a dtIn and a dtOut. The dtIn port is bound to a SimClock modules's scale input connector while the dtOut port is bound to the same module's dt output connector. A pair of System methods are defined for ClockSystem, realTime and simTime can connect and disconnect the system's DigitalFilter module as shown in Figure 3. (Note, the switch is shown for conceptual purposes. Typically connectors or associations are not described using this convention.)

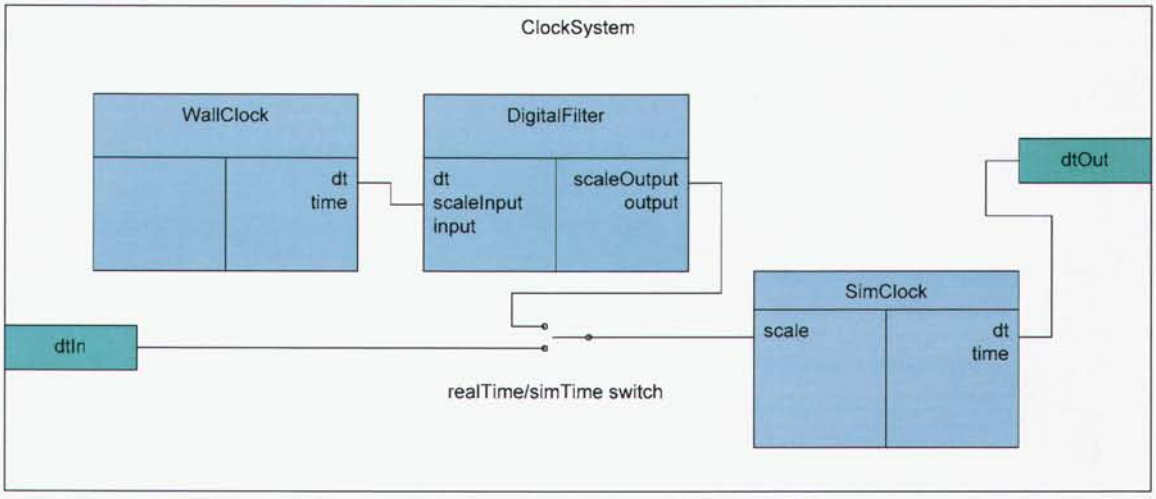

Figure 3: ClockSystem Diagram (from example ClockSystem.tcl)

1.4 System Instantiation & Maintenance

As with typical Umbra modules, systems are instantiated by following the name of the system with the name of the instance. In this case, the system name is actually an incrTcl class and the instance is an incrTcl instance. In some cases, a variety of messages may be printed during module creation. The instantiation command then returns the name of the module just created.

1.4.1 Instantiating and Deleting System Instances and Classes

```
System instanceName
```

Creates an instance of a base system

```
itcl::delete instanceName
```

Note: this is different than the standard way Umbra modules are deleted. Here, the itcl system is responsible for initiating the deletion whereas Umbra modules have a method to delete themselves.

```
deleteClassInstances System
```

This procedure deletes all modules within a class (but leaves the class defined).

```
cleanupClass className
```

This procedure deletes all modules within a class as well as the class itself. This method is particularly useful for clearing memory prior to redefining a system class.

1.4.2 Example Conventions

The examples that follow were generated by copying output from the Umbra console window. Commands are indicated using the percent sign prompt (characters leading up to the percent key are deleted). Values are offset and colored blue for clarity. System messages, which may not be part of the release version of Umbra, are retained and colored red. Comments about the example are offset and italicized.

1.4.3 Examples of System Instantiation and Deletion

This example constructs a basic System

```
% System foo
foo
```

This example constructs several ClockSystems

```
% ClockSystem cs
```

Constructing System instance cs

```
cs
% ClockSystem cs2
```

Constructing System instance cs2

```
cs2
% ClockSystem cs3
```

Constructing System instance cs3

```
cs3
```

This example deletes all System instances and, in the process, deletes all classes derived from System (including the ClockSystems defined above).

```
% itcl::delete object cs
```

Deleting system cs

```
% deleteClassInstances System
1
```

After rebuilding the classes

```
% cleanupClass System
```

cleanupClass: Deleting class System
Deleting system cs3
Deleting system cs2

1.5 Port, Connector, and Parameter Query and Management

In this implementation ports mainly refer to interface elements that correspond to Umbra module connectors. Within Umbra, Systems have input and output ports. These ports are bound to the connectors and subsystem ports.

In its implementation, ports are not actually connected to Umbra module connectors or other system ports. Rather, the connect and feedback methods use the port bindings to find and connect individual modules to one another. In the case of hierarchically defined systems, these methods recursively search downward until they can individually reach and connect each Umbra module. The unconnect method uses an inverse process to unconnect each module.

1.5.1 Basic System Methods

`attribute attName`

Queries the System to determine whether attName is a defined attribute of the System. Returns the attribute value or null if the attribute isn't defined (or if its value is null).

`attribute attName attValue`

Sets the System attribute attName to attValue. attValue can be any Tcl atom, list, or string.

`attributes`
`portNames`

Returns a list of the names of the ports as they are defined at the system interface.

`connectors`

Returns a discriminated list of port names. This method corresponds to the default Umbra module connectors method.

`connections portName`

Returns a list of connections (as opposed to bindings) associated with the given port name. This method corresponds to the default Umbra module connections method.

Caveat: As this command returns the actual connections, the modules named within the return value are names of actual Umbra modules, as opposed to names of the systems within which the modules are contained.

`connected portName`

Not implemented. This method will return 1 if the port is connected and 0 if it is not connected.

`binding portName`

Returns the internal binding for portName. For further elaboration, see the section on System Modification and Manipulation.

`portValue portName`

Returns the current value of the connector on any internally bound component.

Caveat: When two or more connectors are bound to the same port (only possible with input ports), this method returns the value on the first port. This value can be misleading if the system is not connected at the input.

`portValue portName newValue`

Sets all internally bound connectors at portName to the newValue.

Caveat: As with normal Umbra modules, this setting will only take effect if the connectors referenced are not connected to other modules.

`connect portName moduleName portName`
`connect portName moduleName connectorName`
`feedback portName moduleName portName`
`feedback portName moduleName connectorName`

Modules within System instances can be connected to other system instances or normal Umbra modules using the same syntax as the normal Umbra connect method.

Caveat: When connecting a system instance to an Umbra module, the system instance must be listed first. This is because Umbra modules do not know about system modules.

`unconnect portName`

This method, unlike the Umbra module method, disconnects all internal connections found through the port binding hierarchy.

1.5.2 Attribute setting and querying

```
% System s
s
% s attributes
% s attribute foo 1
1
% s attribute foo
1
% s attributes
{foo 1}
```

1.5.3 Connection Query Examples

In these examples, cs is a ClockSystem system instance as defined in ClockSystem.tcl

```
% cs portNames
dtIn dtOut
% cs portValue dtIn
0.0
% cs portValue dtOut
0.0
% cs connectors
{Input dtIn} {Output dtOut}
% cs connections dtOut
null
```

The following connections query exposes an internal connection that gets made when the clockSystem is put in realTime mode (the default).

```
% cs connections dtIn
{connect cs.clockSystem.filter scaleOutput}
% cs simTime
0.001
% cs connections dtIn
null
```

1.5.4 Connection Management Examples

In this example, a normal Umbra module is created and connected to a ClockSystem instance.

```
% DigitalFilter filter
% cs connect dtOut filter scaleInput
```

Making Umbra Connection: filter connect scaleInput cs.ClockSystem.simClock dt

This next example connects two ClockSystem instances. Initially, the system produces an error because the internal connector that the port references is in use at the time the connection is attempted. (Specifically, dtIn is being used internally by the wallclock.) See diagram provided above for further elaboration.

```
% cs connect dtOut cs2 dtIn
```

Making system connection: cs2 connect dtIn cs.clockSystem.simClock dt

Making Umbra Connection: cs.clockSystem.simClock connect dt cs2.clockSystem.simClock scale

unable to connect cs.clockSystem.simClock->dt with cs2.clockSystem.simClock->scale

Putting the clockSystem into simulated time mode disconnects the internal connector and allows it to be used by the external system.

```
% cs2 simTime
0.001
% cs connect dtOut cs2 dtIn
```

Making system connection: cs2 connect dtIn cs.clockSystem.simClock dt

Making Umbra Connection: cs.clockSystem.simClock connect dt cs2.clockSystem.simClock scale

Here is a similar example using the feedback.

```
% cs2.clockSystem.simClock connections scale
{connect cs2.clockSystem.filter scaleOutput}
% cs2 simTime
```

0.001

```
% cs feedback dtOut cs2 dtIn
```

Making system feedback connection: cs2 feedback dtIn cs.clockSystem.simClock dt

Making Umbra Connection: cs.clockSystem.simClock feedback dt cs2.clockSystem.simClock scale

1.6 Component Query and Manipulation Functions

For some applications, it's important to directly interact with or inspect the state of internal modules. A set of component query functions are available for inspecting and interacting with components within a system.

Unlike Umbra modules, System modules can be extended dynamically by adding components and establishing binding. By extension, internal component interfaces are exposed through a system sendComponent command. These capabilities have been implemented to allow systems to be defined and manipulated dynamically. Dynamic system creation is particularly useful for defining systems through XML documents.

In its implementation, Systems store named lists of components. Within Umbra, each module must have a unique name whether it is associated with a system or built as a stand-alone module. Rather than require that user applications be directly exposed to specific module names, query and manipulation functions are available for interacting with internal components by using a standard name. In Acme, this indirection is a form of binding.

1.6.1 Key Limitations

Dynamically created systems cannot have unique methods. This is a limitation of the incrTcl implementation. This limitation is similar to the fact that Umbra modules with specialized methods must be created through a programming language (i.e., C++). By extension, systems that are complex enough to require their own methods should be defined through a programming language (i.e., incrTcl).

1.6.2 Caveats

A careful balance must be struck between creating systems directly (in code) and creating or extending systems dynamically. Applications should not overly rely on system modification and manipulation mechanisms as doing so can negate the benefits of having system definition formalisms in the first place. At the extreme, overuse of these methods could cause the system formalism to be more a barrier to effective programming than a boon in usability.

1.6.3 Methods

`componentNames`

Returns list of component names. These are consistent, easier to remember externally referenced names, as opposed to the names of the internal modules.

`component componentName`

Returns name of actual or internal module referred to externally by the high level name. In the case of Umbra modules, these are the actual instance names.

`component componentName moduleInstanceName`

Adds moduleInstanceName to the system with and names it componentName for external referencing

`component componentName delete`

Deletes the named component from the system and deletes the Umbra instance for it as well

componentP *internalName*
Returns

1 if the named component is a component of this system

2 if the name is the name of a component in the system

0 if it's neither a component or the name the system uses

`binding portName`
`bind portName`

Returns the internal bindings to portName

`bind portName [Input|Output] bindings`

Creates an input or output port to connector binding to portName

`sendComponent componentName message`

`invokeMethod componentName method`

This method sends message (Tcl method) to the internal component named componentName. Note that componentName is the externally referenced name and not the actual moduleInstanceName. This external referencing allows the same "sendComponent" command to be sent to several Systems regardless of their composition.

Note: sendComponent and invokeMethod are just two different wordings of the same function call. Eventually, we will remove one or the other. Vote your preference.

Caveat. If the component is a System instance, this method assumes that the System instance has a message that can receive the same command. If it does not, that call will likely flag an error. For this reason, sendMessage is more appropriate.

`bindMessage portName bindings`

Not implemented. Creates message binding named portName. As with connector ports, message bindings are actually lists of methods. For some systems, the same method needs to be sent to several internal components. This mechanism will provide that capability.

`sendMessage portName arguments`

Not implemented. Sends the message (which in Acme parlay is also a port) with given arguments to the internal components.

1.6.4 Example 1

In these examples, cs is a ClockSystem system instance as defined in ClockSystem.tcl

```
% cs componentNames
wallClock simClock filter
% cs component wallClock
cs.clockSystem.wallClock
% cs componentP wallClock
0
% cs componentP cs.clockSystem.wallClock
1
```

1.6.5 Example 2

The code for the following example is in AnimateSystem.tcl. The system, diagrammed in Figure 4, includes a VectorInterpolator and a geometric object (a scene model) with joints set up to drive it in XYZ. The system is diagrammed below.

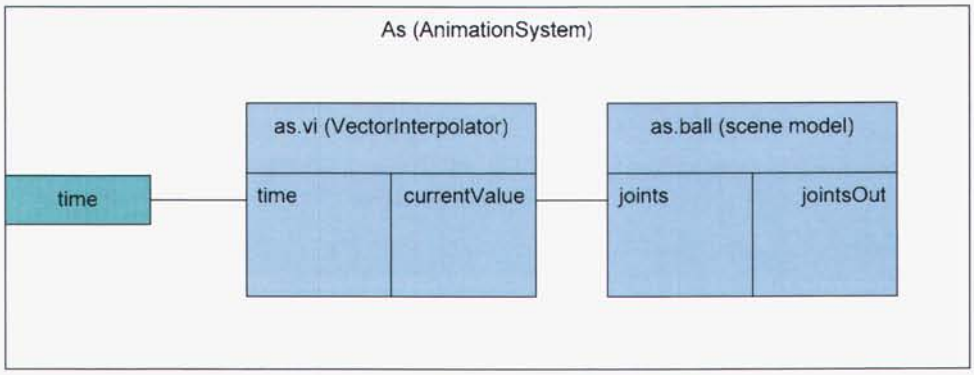

Figure 4 Animate System

The first step is to create the separate modules

```
% VectorInterpolator as.vi
% as.vi events 0 "0 0 0" 10 "0 0 3" 20 "0 4 4" 50 "5 5 5" 100 "0 0
0"
0 {0 0 0 } 10 {0 0 3 } 20 {0 4 4 } 50 {5 5 5 } 100 {0 0 0 }
% scene model as.ball
% as.ball sphere 1
% as.ball makePosJoint {0 1 2}
% as.vi connect currentValue as.ball joints
```

The second step is to create and populate a system with these components and then add the needed bindings.

```
% System as
as
% as component replayer as.vi
as.vi
% as component geom as.ball
as.ball
% as bind time Input "{as.vi time}"
{as.vi time}
```

Finally, the system is connected to the simClock, the simClock is reset, and (optionally), the system run to see the simple animation.

```
% as connect time simClock time
```

Send the geometry (geom) subcomponents a message to change color.

```
% as sendComponent geom "color {.4 .3 .6}"
```

Now reset the simClock and watch the system animate by running Umbra (click run button).

```
% simClock reset
```

At this point, the system contains several modules that can be discovered through Umbra's modules command:

```
% modules as
```

as.ball
as.vi

Because the system owns the modules, the modules will get deleted if the system gets deleted. However, No modules get listed because they are all deleted.

```
itcl::delete object as
```

Deleting system as.System

```
% modules as
```

2 Conclusion

This document described the Umbra System representation and compared key design elements with Carnegie Mellon University's ADL called Acme. As a compact way to organize system definition code, the Umbra System representation is recommended for all serious System of Systems, CSE, and CASE modeling work. Work is ongoing to convert Sandia legacy models to use the Umbra System representation. Feedback has been favorable. An initial extension to this concept has been an XML Schema and parser which are under development. These developments are expected to further firm up the use of the Umbra System representation.

Key Words

ADL	Architecture Description Language
Acme	A simple, generic software architecture description language (ADL) developed at Carnegie Mellon University that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools.
Complex Adaptive Systems	A second-order operational model for the complexity paradigm that addresses adaptively or the ability for the system to change its behavior over time in response to evolving system dynamics. Examples include economies, ecologies, weather, traffic and social organizations including military and security organizations.
Complex Systems	A first-order operational model for the complexity paradigm. Complex Systems is a new field of science studying how parts of a system give rise to the collective behaviors of the system, and how the system interacts with its environment.
Component	A constituent element, as of a system. In Umbra, the terms module and component are used nearly synonymously. In Acme, the term component dominates.
Connector	See Port.
Event	A message (e.g., via a method call) sent to an object (e.g., an Umbra module) that allows the object to execute work.
IncrTcl	IncrTcl is a lightweight object oriented extension to Tcl. Current Tcl releases include incrTcl as a standard component. Documentation for incrTcl can be found within Tcl documentation sets or from http://www.activeState.com which distributes Tcl.
Module	1. A standardized, often interchangeable component of a system or construction that is designed for easy assembly or flexible use. 2. A portion of a program that carries out a specific function and may be used alone or combined with other modules of the same program.
Ontology	An explicit formal specification of how to represent the objects, concepts and other entities that are assumed to exist in some area of interest and the relationships that hold among them.
Parameter	A data value associated with a system or system component. In Umbra, parameters can be values set by the application or can result from computation. Parameter values typically change infrequently.
Port	An Umbra construct that allows modules to share information in an organized fashion. (Sometimes referred to as connectors.)

System	A collection of Umbra modules that represents a system function such as a unmanned ground vehicle (UGV), unmanned air vehicle (UAV), and unattended ground sensor (UGS). System is often referred as a meta-module.
System of Systems	A system where each component is a system (including one or more components) and the sub-systems components behave interdependently. Also, the large-scale integration of many interdependent, self-contained systems in that together satisfies a global need.
Tcl	Script-level programming language used in Umbra. (Commercially available at http://www.activeState.com).
Umbra	Developed by engineers in the Intelligent Systems and Robotics Center at Sandia National Laboratories in Albuquerque, NM, Umbra is a powerful software framework that engineers and programmers can use to develop simulations and analyses, intelligent system and robot controls, 3-D graphics modeling, and new applications. Umbra supports trade-off analyses of complex robotic systems, device, physics, and component concepts.

Distribution:

1	MS1002	Steve Roehrig, 15200
1	MS1002	Philip Heermann, 15230
1	MS1004	Elaine Hinman-Sweeney, 15231
5	MS1004	Michael McDonald, 15231
1	MS1004	Fred Oppel, 15231
1	MS1004	Brian Rigdon, 15231
1	MS1004	Patrick Xavier, 15231
1	MS1005	Russ Skocypec, 15240
1	MS1005	Alan Nanco, 15240
1	MS1010	Kelly Hays, 15233
1	MS1125	Dan Morrow, 15244
1	MS1176	Robert Cranwell, 15243
1	MS1188	John Wagner, 15241
1	MS1188	Matthew Glickman, 15241
1	MS1188	Carl Lippitt, 15241
1	MS1188	Eric Parker, 15241
1	MS1188	Steve Tucker, 15241
1	MS0123	Donna Chavez, LDRD Office, 01011
1	MS9018	Central Technical Files, 8945-1
2	MS0899	Technical Library, 9616