LA-UR-98-1132

Approved for public release;
distribution is unlimited.

Title: Simulation Visualization through Dynamic Instrumentation

CONF-980412--

Author(s): Keith R. Bisset

Submitted to: 12 Annual International Symposium on
Aerospace/Defense Sensing, Simulation, and Controls

April 13-17, 1998, Orlando, FL

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

Los Alamos
NATIONAL LABORATORY

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. The Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Simulation visualization through dynamic instrumentation

Keith R. Bisset

Los Alamos National Laboratory, MS F602, Los Alamos, NM 87454

ABSTRACT

The goal of the instrument composition system is to allow a simulation user to dynamically create instruments as a simulation executes. Instruments can include graphical displays, data collectors, and debugging aides. Instruments are made up of small building blocks which can be easily combined into larger, more complex instruments. Through the use of an Attribute Server (a distributed publication/subscription mechanism), the actors and instruments in a simulation can interact without direct knowledge of each other. Instead, each actor publishes the attributes which it has available. An instrument subscribes to the attributes in which it is interested, and is notified whenever the value of one of these attribute changes. An instrument can also publish attributes for use by other instruments. Since the Attribute Server is distributed, the publisher of an attribute need not execute on the same machine as the subscriber. This allows CPU intensive data visualization to execute on separate machines from the simulation, minimizing the impact on the simulation.

Keywords: simulation, data visualization, instrumentation, TCL/TK, scripting, distributed computing

1. INTRODUCTION

The goal of the dynamic instrument composition system is the easy creation and modification of instruments (including visual, computational, debugging, and interface) for the Los Alamos National Laboratory prototyping effort for the Joint Warfare System (JWARS) program, using the Integrated virtual environment simulation (Ives) framework. Ives is a composable, object-oriented simulation framework, implemented in C++. Ives is based on the notion of an object repository. Prototypical simulation objects (e.g., entities, terrain, weather models) are created and placed in the repository. A prototypical object is generic in nature. It is a template from which actual instances can be created, by supplying information specific to the object (e.g., name, position, fuel level). Simulations are then composed (i.e., built) by coping objects from the repository into a simulation, specifying the attribute values of the objects and creating specific instances of the objects. Composed simulations are considered simulation objects as well, allowing them to be stored in repositories. As used in this work, an actor is analogous to a simulation object. An attribute of an actor is a portion of the state of the actor, such as velocity or status. An actor developer is the person who implements the models to be simulated. An analyst uses these models to answers questions of interest.

A JWARS prototyping effort which used a previous incarnation of Ives is the basis for the work described in this paper. The JWARS scenario consists of Joint Surveillance Target Attack Radar Systems (JSTARS), KC-135 tankers, unmanned aerial vehicles (UAVs) and associated ground station modules (GSMs), Transported Erector Launchers (TELs), and battalions. The JSTARS look for TELs and battalions, and are refueled by the KC-135s. When a battalion is observed by a JSTARS, the information about its location is sent to the GSM, which dispatches a UAV to the battalion's location for reconnaissance. A proof-of-concept instrument composition system was built using the JWARS scenario as a test case. The system described in this research is used to explore the possibilities of dynamic instruments and decide on desirable features of such a system.

It is often difficult to know *a priori* what kinds of data displays will be useful. A method to allow analysts to develop data displays (i.e., instruments) independently of the simulation framework is needed. These instruments take the form of small building blocks which can be assembled into a wide variety of instruments of varying complexity. The instruments are built using the Tool Command Language (TCL), a simple but powerful scripting language, or C++. Once an instrument has been created, it can be placed into a repository as a prototypical instrument which can be used as a building block for constructing other instruments, or as a fully configured instrument stored as part of a simulation, so that the instrument is created whenever the simulation is executed.

Three goals are used to direct the design of the instrument composition system. The first goal is to minimize the impact of the instruments on actor developers. An actor should not need to know about the various kinds of instruments, or even if instruments are present in the simulation. This allows instruments to be added to and removed from the simulation without modifying the actors. An actor simply needs to provide (upon demand) any data which may be required. It is the responsibility of the instrument to present the data in a meaningful way, and to perform any necessary calculations to derive additional data.

There should also not be any special actions required by the actor, such as notifying the instruments when an attribute changes. This frees the actor developer to concentrate on the actor itself.

The second goal is to minimize the performance impact of instruments on the simulation. Data should not need to be produced by an actor when no instrument will actually use it. Complex instruments should be able to execute on a separate processor from the simulation, to reduce the computational load on the processor(s) on which the simulation is executing.

The third goal is ease of use by the analyst. It should be easy to add and remove instruments during the execution of a simulation. Existing instruments should be easily modified, and new types of instruments easily created. There should also be a library of components (2D and 3D graphs, tables, relational database interfaces, etc.) available for reuse.

The first two goals are achieved through the use of the attribute server, explained in detail in Section 2. The third goal is met by using TCL as the instrument implementation language. Instruments are described in Section 3. Ideas for future development are in Section 4.

2. ATTRIBUTE SERVER

The attribute server provides a distributed publication-subscription mechanism, allowing communication between objects without the object having prior knowledge of each other. In order to separate actors and instruments, all communication takes place through attributes, via the attribute server. An actor publishes any attributes (i.e., data values) that it wishes to make available to other objects in the simulation. An instrument can subscribe to one or more attributes to perform some operation on their values. The designation of actor and instrument is somewhat artificial, since a single object can be both a publisher of and subscriber to attributes. This is useful to create a computational instrument which subscribes to one or more attributes and publishes an attribute based on those attribute values.

An attribute can be read-only or read-write. The value of a read-only attribute can be changed by the simulation, just not through the attribute server. The writability of an attribute can change during the course of a simulation. For example, the fuel-level of an actor may be read-write during the initialization phase of a simulation so that its value can be adjusted, but read-only once the simulation begins. The fuel-level will change as the actor maneuvers, but cannot be changed by any instrument.

The attribute server is distributed, with an attribute server on each processor. These attribute servers communicate, making it appear to the actors and instruments as if there is a single attribute server covering the entire simulation system. Figure 1 shows the architecture of the attribute server. Each processor has two actors, two instruments, and one object which acts as both an actor and an instrument. Each object (actor or instrument), communicates with the local attribute server, but it appears to the object that it is communicating with one large logical attribute server.

An actor registers its attributes with the attribute server by providing information about each attribute. This information includes: the class the actor belong to, the name of the actor, the name of the attribute, the data type of the attribute (integer, real, string, etc.), the units of the attribute (feet, pounds, furlongs per fortnight), and the methods used to get and (if the attribute is writable) set the attribute value. Optionally the actor can provide information about the minimum and maximum values that the attribute can take on. This can be used by an instrument to display the attribute. For example, the speed of an actor can be displayed on a dial, using the minimum and maximum values provided to set the endpoints of the dial. These values can also be used by an instrument for input validation, prior to setting the value of an attribute.

Registration is usually done when an actor is created. When an actor is removed from the simulation, its attributes are removed from the attribute server. Other than the registration and removal of attributes, an actor can ignore the presence of attributes. When an attribute value is needed by an instrument, the appropriate method is called via the attribute server. This meets the requirements of goal one: minimize the burden on the actor developer.

The attribute server provides facilities for an instrument to list all of the attributes available in the simulation or in a specific actor, as well as search for attributes by name. An instrument can also ask the attribute server to be notified whenever an attribute is added to or removed from the simulation. This allows, for example, an instrument to keep track of the position of every actor in the simulation. The instrument will be notified, via a callback function, whenever a new actor is added to the simulation, via the attributes created by that actor, and when an actor is removed from the simulation.

An instrument can access the value of an attribute in two ways. First, the attribute value can be retrieved directly from the attribute server. Second, the instrument can ask the attribute server to be notified whenever the value of an attribute changes. The instrument specifies how often, in simulation time, the attribute should be checked, and how large of a change in the attribute value is needed to trigger notification.

This method, called attribute server pull, works efficiently in most cases. It is called attribute server pull because the attribute server asks the actor for the value of its attributes (pull the values from the actors), and checks to see if they have changed.

Figure 1. Architecture of attribute server.

In some instances, however, this can lead to inefficiencies. Specifically, when the value of an attribute seldom changes, and an instrument needs to be informed immediately that a change has occurred. Using attribute server pull, the actor is queried repeatedly for an attribute value which is not changing.

Another method, actor push, is called for in this case. Using actor push, the attribute server does not query the actor, but waits for notification from the actor that an attribute has changed. The attribute server then notifies any instruments which are interested in changes to that particular attribute. This is more efficient, but places a higher burden on the actor developed. Actor must be sure to notify the attribute server whenever an attribute changes.

Through the use of the attribute server, the first two goals are satisfied. The actors and instruments are decoupled, so that they need not contain knowledge of each other. Performance goals are met by allowing the instruments to execute on different processors, not requiring data to be generated unless it is actually needed, and allowing actor push of data.

3. INSTRUMENTS

Instruments can be grouped into four general categories: visual, computational, debugging, and interface. Visual instruments graphically display attribute values. Computational instruments publish an attribute based on the values of one or more other attributes to which it subscribes. An example of a computational instrument would be a units converter. Debugging instruments are an aide in verifying the correct operation of a simulation. Interface instruments connect a simulation to external programs, or interact with the user. These categories are not meant to be exclusive or exhaustive, and only serve to demonstrate some of the diverse uses of instruments.

3.1. Computational instruments

Computational instruments allow instruments to manipulate the values of attributes and publish new attributes based on those manipulations. For example, an instrument can subscribe to an attribute in miles per hour, and publish an attribute whose value is the original value converted to feet per second. Two computational instruments have been built for the JWARS scenario. The first instrument simply takes a list of attributes, and produces the sum of the values of those attributes. It is used to monitor the amount of fuel transferred from the KC-135s to the JSTARS.

The second instrument, whose interface is shown in Figure 2, takes the location of two actors and computes the distance between them. Two actors are selected, and a name given to the output attribute. There is a single distance object in the

Figure 2. Distance instrument creation.

Figure 3. Attributes for distance object.

Figure 4. Clock instrument.

simulation, which publishes an attribute for each pair of actors selected. In the JWARS scenario, the distance between each JSTARS and its tanker is computed. Figure 3 shown the attributes of the distance object.

3.2. Visual instruments

A 2-D graphing instrument was developed. This instrument takes pairs of attributes. Each pair of attributes represents the x and y values of points on a line to be graphed. Whenever the value of an attribute changes, a new point is added to the appropriate line with the current values of the attributes. The graph in Figure 5 shows the distance values over time. The oscillations are due to the fact that the JSTARS flies a circular orbit, while the KC-135 remains on the ground. The distance drops near zero as the JSTARS is refueled. The right side of the graph shows JSTARS three and four relieving JSTARS one and two.

In addition to the 2-D graph instrument, two specialized instruments were developed for the JWARS scenario. The simulation clock (Figure 4) simply shows the current simulation time, in both digital and analog format. The *blip* display (Figure 6) shows the view of the JSTARS sensors, as reported to the GSM. The blips for the ground objects observed by the JSTARS are displayed by position (latitude and longitude). The view of the two JSTARS is differentiated by the shape of the markers on the display. This display is simply a modified 2-D graph widget.

Figure 5. 2-D graph instrument.

3.3. Debugging instruments

An interesting use for instruments is as a debugging aide. An instrument can be developed which monitors the simulation for unusual or erroneous conditions. When such a condition is observed, several different courses of action can be pursued. A window can be opened alerting the analyst to the problem. The simulation can be suspended to allow the analyst to examine the current state of the simulation, and perhaps to create additional instruments. These instruments can also be created automatically to monitor areas of the simulation in which the problem was observed. The entire state of the simulation can be dumped to a file for later analysis. Or the condition can be noted in a log file, the erroneous values corrected to some reasonable values, and the simulation continued. The data monitoring and correction can be as simple or complex as needed without changing the simulation framework or the actors, and can be selected and tuned as the simulation is executed.

3.4. Interface instruments

The entire runtime interface to the Ives framework is based on TCL instruments. The simulation control panel, shown in Figure 7, is actually an instrument. This control panel allows a composed simulation to be selected and executed, and additional instruments to be created. The simulation is controlled through a "state" attribute, which can take on the values INIT, RUN, PAUSE, and STOP. The simulation framework monitors this attribute and takes appropriate action when it changes. The map

Figure 6. JSTARS blip instrument.

display shown in Figure 8 uses the graphical input aggregate control (GIAC) and the G data system (GDS), which is a graphical user interface for military command and control simulations. The simulation framework communicates with GIAC/GDS through the attribute server. An interface instrument was developed which monitors the location, speed, and other attributes of each actor in the simulation. The instrument sends a message to GIAC/GDS whenever an attribute changes. GIAC/GDS then reflect the change on its display. Although Ives was not originally built with the use of GIAC/GDS in mind, the attribute server made it a relatively simple task to connect it to the Ives framework. The communication between the simulation and GIAC/GDS is currently one way, but can be easily extended to take advantage of GIAC's interface features.

A battle management language (BML) editing instrument, shown in Figure 9, was created to allow the viewing and editing of the BML of an actor. The editor also allows BML to be load from and saved to a text file. The BML displayed in the editor is for one of the JSTARS.

BML controls the actions of the actors. It is a superset of TCL, specialized for the actors being controlled. Six new commands were added to TCL corresponding to the actions that actors can take. These commands are move, surveil, report, resupply, form and relate.

BML is translated from a series of high-level commands to a small set of actor primitives (tasks). Each actor has a (possibly empty) initial script, and can receive BML commands during the course of the simulation. New BML commands can either supplement the existing commands, or override them. For instance, the command "move orbit \$orbit", will generate two tasks, the first to move to the starting location of the orbit, the second to fly on the specified orbit.

Figure 7. Control panel for Ives.

Figure 8. GIAC/GDS map display.

Figure 9. BML editor.

Tasks can either be sequential (task A must complete before task B starts), or concurrent (task A may start before or after the start of task B). This is accomplished by grouping tasks into blocks. A block may be either concurrent or sequential, and may contain other blocks.

Other types of interface instruments are possible. For instance, an instrument can store data into a relational database for current or later use. The data in the database will lag behind the simulation by some amount, depending on the database update speed and the rate of the data generated by the simulation, but would be easily accessible by existing database tools. This intermediate database completely decouples the generation of data from the display tools, increasing the possibility of reuse and allowing off the shelf products to be used. Data can also be sent to data visualization tools, such as Mathematica, where computationally expensive data manipulation and display can take place. Since the instrument need not reside on the same processor as the simulation, this manipulation or visualization need not impact the execution of the simulation itself.

3.5. Dynamic instrument creation

One of the main benefits to dynamic instrumentation is the creation of instruments as the simulation is executed in order to examine data currently of interest. The instrument composition system currently allows for four types of instruments to be created dynamically: a simulation clock, a 2-D graph, a textual history, and file output. Other types of instruments can be easily added. Appendix A shows the TCL code used to create some of the instruments shown in this paper.

The input attributes are selected from two menus, shown in Figure 10. The first menu (only partially displayed) shows the available actors in the simulation. Once an actor is selected, in this case JSTARS1, a second menu is shown, listing the attributes available for the selected actor. The units of the attribute are shown in parentheses after the attribute name.

A 2-D graph is created, as shown in Figure 11(a), by adding one or more lines to the graph. The example has four lines, each plotting the distance between a JSTARS and a KC-135 against simulation time. Attributes are shown as *actor:attribute*. The ES actor is the events scheduler, which is responsible for keeping time in the simulation. To add a line, as shown in Figure 11(b), X and Y attributes are selected from the available attributes in the system. When all of the necessary lines have been added, the graph is created and begins plotting the requested data, as shown in Figure 5. The creation of a textual history instrument, shown in Figure 12 and Figure 13, and a file output instrument are similar. The history instrument allows either the entire history of the values taken on by an attribute to be shown, or only the current values.

Figure 10. Example actor and attributes.

Figure 11. Creation of 2-D graph instrument.

Figure 12. Textual history instrument creation.

Distance: J1-KC1:	387226
Distance: J2-KC2:	200789
Distance: J3-KC3:	502475
Distance: J4-KC4:	299844

Figure 13. Textual history instrument.

Some deficiencies have been discovered in the attribute selection mechanism. Currently the analyst creating the instrument must ensure that the attributes selected as inputs match the expectations of the instrument being created. For example, while the textual history instrument can be used with any attribute, the 2-D graph instrument expects that the attributes are numeric and the blip instrument expects a list of positions. Selecting the correct types of attributes is even more difficult when using an instrument from repository, which is written by someone else and may have unknown assumptions.. A generic "impedance matching" mechanism is being developed which will allow an instrument to describe the types of attributes it can accept as input. This description will be compared to the actual attributes selected to ensure that only valid attributes are specified.

4. CONCLUSIONS

The ultimate goal of the instrument composition system is to allow an analyst to create new instruments without writing any code. This requires a library of instrument building blocks, and a visual instrument creation environment. Instruments can be taken from the library and assembled into more complex instruments. These complex instruments can be stored back into the library for latter use in creating even more complex instruments. An idea of how the environment would work is shown in Figure 14. Along the left side is a library of instrument building blocks. These instruments have been combined into a more complex instrument. This instrument first takes the velocity of the four JSTARS and converts them into feet per second. The converted values are then fed into a statistics instrument which computes their average, minimum, and maximum. These statistics are graphed over time and output to a text file.

It has been shown how instruments allow an analyst to examine a running simulation by extracting data which is of interest. The display of this data is separate from the simulation framework and is tailored to the needs of the analyst. In addition, the simulation can be easily attached to external programs, as shown by the GIAC/GDS instrument. Adequate performance is maintained through the use of an attribute server, which also serves to decouple the actors and and simulation framework from the instruments. Simple instruments can easily be combined into more complex ones (preferably using a visual environment), and new instruments are easily created using the TCL scripting language.

Figure 14. Example of visual instrument creation environment.

APPENDIX A. TCL CODE FOR INSTRUMENTS

A.1. File Output

```
set file_num 0
proc SetupFile {aids filename} {
    global file_num files file_aids
    set files($file_num) [open $filename w]
    set file_aids($file_num) $aids

    foreach aid "$aids" {
        set attrib [ASGetAttrib $aid]
        append str [lindex $attrib 2] ":" [lindex $attrib 0] " "
    }
    puts $files($file_num) "$str"
    ASWatchAttrib [lindex $aids 0] "UpdateFile $file_num" 60 0
    set old_num $file_num
    set file_num [expr $file_num+1]
}

proc UpdateFile {num} {
    global file_num files file_aids

    set aids $file_aids($num)
    foreach aid "$aids" {
        append str [ASGetValue $aid] " "
    }
    puts $files($num) "$str"
}
```

A.2. Blip instrument

```
proc SetupBlip {aid1 aid2} {
    global blip_aids
    set blip_aids(1) $aid1
    set blip_aids(2) $aid2

    ASWatchAttrib $aid1 "UpdateBlip 1" 60 0
    ASWatchAttrib $aid2 "UpdateBlip 2" 60 0
    set blip .blip
    toplevel $blip
    graph $blip.graph
    pack $blip.graph
    $blip.graph yaxis configure -min 43.5 -max 45.5
    $blip.graph xaxis configure -min 14.5 -max 21.5
    $blip.graph configure -bufferelements false
    $blip.graph element create blip1
    $blip.graph element create blip2
    $blip.graph element configure blip1 -linewidth 0 \
        -symbol circle \
        -fill red \
        -outline red \
        -label "GSM 1" \
        -pixels .03i
    $blip.graph element configure blip2 -linewidth 0 \
        -symbol splus \
        -fill red \
        -outline red \
        -label "GSM 2" \
        -pixels .03i
    Blt_ActiveLegend $blip.graph
    Blt_ZoomStack $blip.graph
}

proc UpdateBlip {num} {
    global blip_aids
    set aid $blip_aids($num)
    append blipname blip $num

    set graph .blip.graph
    set blips [ASGetValue $aid]
    set blip_list [split $blips '\n']
    set data ""
    for {set i 0} {$i < [llength $blip_list]} {incr i} {
        set blip [split [lindex $blip_list $i]]
        set type [lindex $blip 0]
        set lat [lindex $blip 1]
        set lon [lindex $blip 2]
        lappend data $lon $lat
    }
    $graph element configure $blipname -data $data
}
```

A.3. BML Editor

```
proc EditBML {oid} {
    set obj [ASGetObject $oid]
    set objname [lindex $obj 0]
    set aid [lindex [ASGetAttribByName $objname "bml"] 4]

    global file_name
    set bml ".bml$aid"

    toplevel $bml
    wm title $bml "$objname BML Editor"
    set bmlstr [ASGetValue $aid]
    set file_name [lindex [ASGetAttrib $aid] 2]
    append file_name ".bml"

    frame $bml.buttons
    button $bml.buttons.file -text "File" -command "FileBML $aid"
    button $bml.buttons.accept -text "Accept" -command "SetBML $aid"
    button $bml.buttons.reset -text "Reset" -command "destroy $bmlstr; EditBML $aid"
    button $bml.buttons.cancel -text "Cancel" -command "destroy $bml"

    text $bml.text -yscrollcommand "$bml.scroll set" -width 0
    $bml.text insert end "$bmlstr"
    scrollbar $bml.scroll -command "$bml.text yview" -width 10

    pack $bml.buttons.file $bml.buttons.accept $bml.buttons.reset $bml.buttons.cancel
    pack $bml.buttons -side top
    pack $bml.text -side left -fill both -expand true
    pack $bml.scroll -side right -fill y
}

proc SetBML {aid} {
    set bml ".bml$aid"
    ASSetValue $aid [$bml.text get 1.0 end]
    destroy $bml
}

proc FileBML {aid} {
    global file_name

    toplevel .name
    frame .name.n
    label .name.n.label -text "File Name:"
    entry .name.n.entry -textvariable file_name

    frame .name.b
    button .name.b.accept -text "Read" -command "ReadBML $aid"
    button .name.b.reset -text "Write" -command "WriteBML $aid"
    button .name.b.cancel -text "Cancel" -command "destroy .name"

    pack .name.n.label .name.n.entry -side left -padx 3 -pady 3
    pack .name.b.accept .name.b.reset .name.b.cancel -side left -padx 3 -pady 3
    pack .name.n .name.b -side top
}

proc ReadBML {aid} {
    global file_name
    set bml ".bml$aid"
    set f [open $file_name r]
    while {[gets $f line] >= 0} {
        $bml.text insert end "$line\n"
    }
    close $f
}

proc WriteBML {aid} {
    global file_name
    set bml ".bml$aid"
    set f [open $file_name w]
    set bmlstr [$bml.text get 1.0 end]
    puts $f "$bmlstr"
    close $f
}
```