Proceedings of the 1st Human-Centered Transportation Simulation Conference, The University of Iowa, Iowa City, Iowa, November 4-7, 2001 (ISSN 1538-3288).

THE SCHEMA ARCHITECTURE AND DRIVING SIMULATOR APPLICATIONS

Hugh Sparks – MTS Systems, *hugh.sparks@mts.com*
Judy A. Carmein - Solidica
Bradford K. Thoen - MTS Systems
Allen J. Clark - MTS Systems

August 14, 2001

ABSTRACT

Schema is an object-oriented development environment for building large distributed control systems. It has been successfully applied to diverse applications including earthquake simulators, driving simulators, robotics, and manufacturing process controls.

Schema classes support multi-rate digital signal processing, model-reference adaptive control, system modeling, transparent inter-processor communication and construction of a graphical user interface. The development tools are based on an interactive programming environment that runs on most popular operating systems. The distributed real time environment works on a variety of multiprocessor architectures.

Driving simulator applications make demanding use of all Schema features because they integrate multi-axis motion control with real-time graphics, audio and scenario management on distributed systems. The paper describes the Schema development environment and reviews its application in several recent driving simulator projects.

THE SCHEMA ARCHITECTURE

The goal of the Schema programming system is to facilitate the rapid development of industrial control applications running on distributed processing hardware. Software tools included with Schema address all aspects of the application from the graphical user interface to device drivers for sensors and actuators.

Key features of Schema

Object Orientation

Schema uses classes and objects for all levels of the application from the GUI to embedded code for digital signal processing.

Interactive Programming

As far as possible, all aspects of the software can be examined and modified while the control system is running.

Integrated development environment

The tools used to develop the software remain part of the finished application. Should exceptional circumstances call for debugging, they can be activated to explore the running software.

Transparent access to all system levels

The idea of transparency in this context refers to the ability to inspect all aspects of the system during normal operation. This is the function of special diagnostic software in more conventional control systems. Diagnostic software typically exposes only a small subset of the information present in a complex system. By activating the development tools that are part of a Schema application, all levels of the software and all component state information may be examined.

Rapid development cycle

Using typical embedded software tools, a system is evaluated by first starting the software and using it to put the hardware into a state that demonstrates a behavior to be improved or altered. The system must then be shut down so code can be edited, compiled and downloaded. Finally the software is restarted and the system returned to the previously state so the change can be evaluated. This cycle is repeated thousands of times during the development of a large industrial automation system, so a substantial reduction in the time required to restore the system to a given state will greatly impact the total development time. In many situations Schema permits the running system to be modified, reducing the state restoration time to zero.

Evolutionary development

It has been observed that large software systems are not built, but rather "evolved." The Smalltalk environment was the first (1) to demonstrate that interactive object oriented programming could greatly facilitate an evolutionary and iterative approach to large system development. Schema implements an environment modeled closely after the Smalltalk, but adapted for the requirements of high speed signal processing and control.

Software prototyping

Schema can be used to rapidly create the graphical user interface of a large industrial control application. The ability to demonstrate a functional user interface early in the development cycle help users understand the features the finished system will perform. The ability to make rapid online changes to the prototype facilitates the negotiation process between developers and users. In some cases, users can modify the prototype to demonstrate their own requirements.

Portability

Schema is designed for portability at all levels. The entire graphical user interface of a Schema application will run unmodified on any version of Windows, Macintosh or Linux/Unix workstation. All of the signal processing and control tools are based on ANSI standard C. Only a small volume of hardware interface code depends on the particular embedded processor environment. Using the popular and portable VxWorks operating system minimizes dependencies at this level.

System modeling

The real-time control, signal processing and mathematical tools used to build controllers are also useful for constructing analog computer-like simulations. As part of the application prototyping effort, the graphical user interface can be connected to a model of the hardware system. If the hardware model is sufficiently detailed, it is possible to predict many attributes of the finished system. When the real hardware becomes available, the same prototype application can be connected to real hardware with no changes to the high level software.

Scalability

Schema supports parallel processing on common bus shared memory systems and on systems interconnected by a high speed LAN.

Parallel processing

The signal processing and control components of Schema communicate using data flow networks. The objects themselves execute concurrently under the control of a scheduler. Dataflow diagrams are a natural way to express parallel computation and can usually directly express the implementation of signal flow diagrams used by control engineers.(2) Schema dataflow technology has been used on large processor arrays. (3)

Transparent object communication

Schema objects communicate directly using messages or through data-flow connections that transport time sampled signals. Regardless of how objects are placed in the distributed processor network, the interconnections are described by the same software specification. The developer is free to design the object architecture and later explore optimum execution arrangements.

Object Oriented Programming Mechanisms

A detailed discussion of object oriented programming is beyond the scope of this paper, but a few key concepts must be defined to show how Schema implements its component framework. We will use terminology borrowed from the Smalltalk language (1) to illustrate these ideas.

Objects are data structures used to create models of entities in the environment manipulated by a software system. Objects can model physical things like motors, actuators and sensors. They can also represent software abstractions provided by the operating system like files, windows, and processes.

Classes are used to specify how objects are represented and how they behave. Every object is said to be an *instance* of a class. The internal representation of an object is determined by a set of named *instance variables*. The values stored in the instance variables of a particular object determine the state of that object. All instances of a given class have the same set of internal instance variables.

Metaclasses are a powerful feature of some object-oriented languages where classes are themselves objects. Every class is an instance of a metaclass and has a protocol like any other object. A metaclass allows each class to define unique creation and initialization methods for its instances. Objects-oriented languages that lack the metaclass protocol must introduce complex and much less flexible features to deal with these issues. (5)

Messages are used to manipulate objects. A message has a name called a *selector.* A message may optionally include a set of *actual parameters.*

A m*essage expression* is used to send a message to an object. Message expressions are the procedure calls of an object-oriented language.

If the variable *x* contains an object, the following message expression might be used to change the object's color to red:

```
x.setColor[red]
```

In this example, the object contained in the variable *x* is the receiver of the message. The message consists of the selector *setColor* and the single actual parameter is a variable *red* that contains some specification of a color.

Methods define the procedures of an object-oriented language. When a message is sent to an object, a method is executed to manifest some response or behavior. The association of selectors and methods is defined on a class by class basis. Any number of classes can define a method for a particular selector. Methods, like procedures in functional languages, may specify any number of formal parameters. The collection of methods and parameters defined by a class determine the *protocol* for objects that are instances of the class.

In the example message expression above, the class of the object stored in the variable *x* would define a method for the selector *setColor.* The method would have one formal parameter, perhaps *theColor,* and would contain a sequence of expressions needed to alter the color of the object.

Polymorphism refers to the non-unique association between selectors and methods. A class used to represent a display string on the user interface screen could define a method for *setColor* that changes the color of the string. A class used to represent the interface for a paint-spraying robot could define a method for *setColor* that determines which color to spray. The selector is the same in both cases.

Specifying a set of instance variables and a set of methods defines a new class. In addition, a new class may *inherit* instances variables and methods from an existing class. In this case, the new class is said to be a *subclass* of the existing class. The existing class is a *superclass* of the new class.

Binding time refers to how a method is found to implement the behavior invoked by a message expression.

In *early* or *static* binding languages, the variable that contains the message receiver object must be declared in advance so that it may only contain objects of a specific class. In this case, the compiler knows which method will be required and the generated code will be a simple procedure call.

In *late* or *dynamic* binding languages, the variable that contains the message receiver has no declaration about the class of object it may contain. In this case, the method must be located at run time by searching the receiver object's class for a method that implements the message selector.

The binding strategy used by a object oriented languages is a controversial subject. Smalltalk, Objective C, Python and various object oriented dialects of Lisp use dynamic binding. C++, Java, C#, Eiffel are examples of languages that use static binding. The advantages and disadvantages of early and late binding languages depend to some extent on the nature of the application domain. A consensus seems to be emerging that interactive prototyping environments are more effective with the dynamic binding strategy. (4)

Schema uses dynamic binding for several reasons:

Generic data structures – A key advantage of dynamic binding is the ability to create generic data structures and generic application frameworks that will work with objects and classes introduced after they are created. For example, a class *List* may be defined with methods to add, remove and enumerate the objects it contains. The *List* may be used to hold any collection of objects regardless of their class.

Abstract methods – Another advantage of dynamic binding is the ability to define abstract methods: A method can be defined that depends only on the protocol of the objects it manipulates, not on their representation. For example, a method to implement efficient sorting can be written and compiled once and for all time. It depends only on the requirement that the objects it sorts will understand a Boolean-valued magnitude comparison selector.

Application frameworks – Application frameworks were introduced to make the graphical user interface for an object oriented program consistent, attractive, and easy to program. Dynamic binding makes it possible to use a compiled application framework with objects and classes that were never anticipated by the framework programmer. This is done by making extensive use of abstract methods in the framework class hierarchy. Schema extends the abstract framework concept to the real-time signal processor and control environment.

Transparent distributed object communication - An attractive feature for a distributed environment is transparent messaging: Consider a *List* class that implements an abstract method for sorting. It should be possible to sort objects on the list even if some of the objects reside on remote processors. To do this, it must be possible to write software that makes no distinction between local and remote objects when generating code for message expressions. Dynamic binding makes this feature easy to implement.

Proceedings of the 1st Human-Centered Transportation Simulation Conference, The University of Iowa, Iowa City, Iowa, November 4-7, 2001 (ISSN 1538-3288).

Schema Software Environments

Schema classes are constructed with two distinct but closely coupled programming tools: COOPS and Alltalk. COOPS (C Object Oriented Programming System) is based on ANSI standard C augmented with classes, objects and messages. The COOPS layer of Schema is used to implement a comprehensive set of tools for physical modeling, digital signal processing, and feedback control.

The Alltalk programming environment is used to implement the graphic user interface (GUI) for the application as well as all high-level application functions. The Alltalk layer of Schema also provides the interactive programming interface for COOPS.

Both Alltalk and COOPS contain features to implement communication between objects on multiple computers. A typical Schema application will have one workstation running an Alltalk GUI connected to one or more systems running COOPS control systems. Systems that require multiple user interface workstations may run several Alltalk GUI programs connected on a LAN.

COOPS (C Object Oriented Programming System)

COOPS implement all the Schema features required by the embedded real-time environment. It extends C by adding classes, metaclasses, methods and messages.

Key features of COOPS:

Portability

COOPS is implemented as an extension to ANSI standard C to achieve the greatest possible portability. COOPS will run under any operating system when used for simulations. For real-time controllers, COOPS depends on an environment with a hard real-time clock. COOPS can run on specialized digital signal processors with no other operating system software. On machines that support VxWorks, COOPS will use the portable clock and tasking features it provides.

Deterministic memory management

COOPS is part of an interactive programming environment. Because objects can be created, interconnected and destroyed while the system is running, it is necessary to employ dynamic data structures for all real-time components. The heap memory management supplied by the C runtime environment is not stable because it becomes fragmented after extended use. Also, heap algorithms frequently employ non-deterministic searches when allocating or deallocating memory. For these reasons, COOPS employs a dynamic memory system called a *Pile.* Objects are created by sending the *new* message to a class. Every class contains an initially empty list of available instances. When a class responds to a request for a new instance, it tries to supply one from the internal list. If the list is empty, the new object is allocated from a large linear memory array by advancing a free spaced pointer. As a result, objects can always be allocated or deallocated in a fixed time and there is no heap to become fragmented.

Hard real-time scheduling

COOPS provides rate monotonic scheduling (RMS), an optimal strategy for handling multi-rate systems. *(6)* Other object frameworks for real-time applications are discussed in (14). Conceptually, RMS assigns a priority to each process proportional to the required sampling rate. Higher rate processes have higher priority and can preempt processes running at slower sampling rates. COOPS avoids the use of processes by using an algorithm based on reentrant interrupts from a single clock. This technique avoids much of the overhead required by process context switching and allows COOPS to run on embedded systems that have no other operating system.

The clock provides the highest rate interrupt to the scheduler, which divides the rate down to any number of slower rates. Tasks that execute at the same rate are placed on a data structure called a *sequence,* which determines their order of execution. The scheduler manages the sequences and causes preemption when required.

Figure 1 show the conceptual arrangement of the scheduling system. Each task contains a message expression, usually directed to a method in a particular data flow object. By using tasks as an intermediate structure, it is possible to have data flow objects with multiple periodic behaviors, each with its own rate.

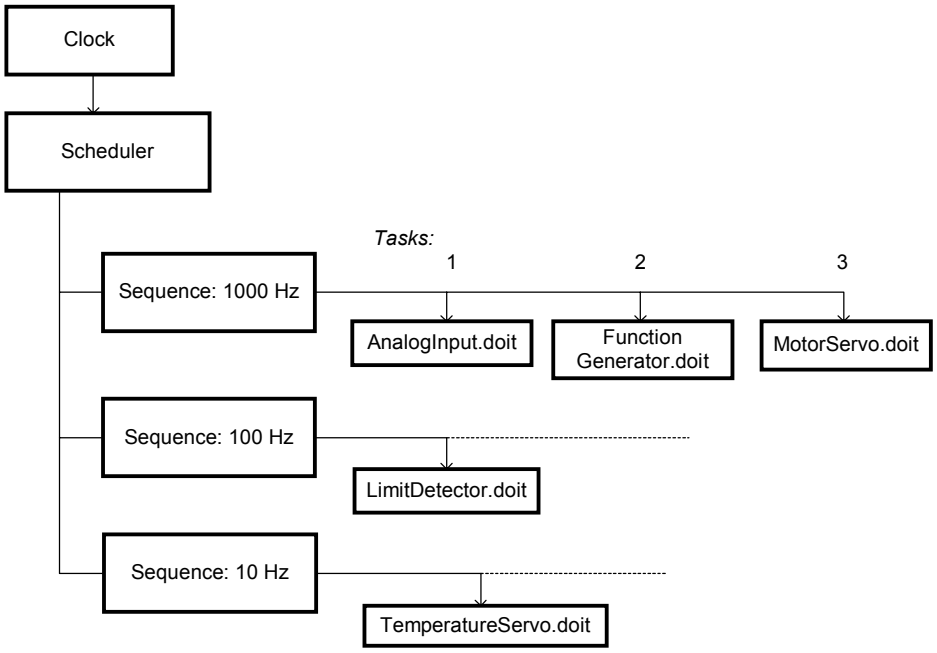

FIGURE 1 Rate monotonic scheduler components.

The COOPS RMS algorithm also deals with situations where a processor is overscheduled. The conventional RMS with processes will lock out objects with slower sampling rates when the system is overloaded. This would be unacceptable and often dangerous in mechanical automation applications. When COOPS is overloaded, some time is added to the clock tic, effectively slowing the system down enough to allow all objects to run at a slightly reduced sampling rate.

The clock used to control the COOPS scheduler can be adjusted explicitly while the control system is running. When the clock rate is adjusted, all periodic objects are notified that their sampling rate has changed. Each signal processing class implements a method that responds to the rate change message by recomputing all internal state variables that depend on the sampling rate.

Message expression syntax

COOPS messages are implemented as special generic functions. Using the color example discussed above, a message to change the color of an object stored in the variable *x* appears below:

```
setColor(x, red) ;
```

The first parameter to the *setColor* message is the message receiver. All additional parameters are the actual parameters of the message. In this example, *setColor* is a generic function. Methods for *setColor* may exist in several classes. The generic function will determine the proper method at runtime using the dynamic binding strategy.

Transparent distributed object communication

The syntax for messages is the same for local and remote objects. In the setColor example, the variable x may contain a local object or an object reference obtained from a remote system. If the object is non-local, the generic function will forward the message to the remote system using shared memory or the LAN. This process is completely automatic and transparent. No special message syntax is required to reach non-local message receivers.

Data flow programming

COOPS programmers implement signal processing and control software using dataflow networks. Dataflow networks are a natural description of many inherently concurrent programming problems. (12) Signal processing and control components are interconnected using software terminals. Any COOPS class can define named input and output terminals for its instances as shown in Figure 2. Terminals are created and stored in the object's instance variables like any other state information.

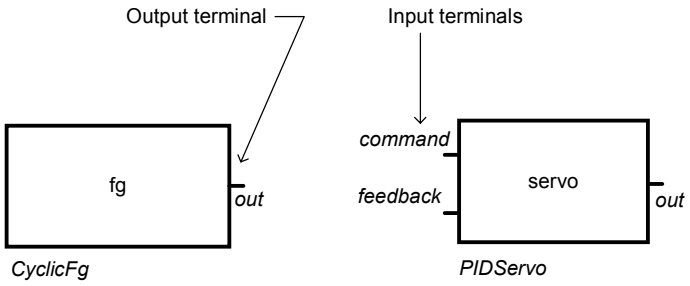

FIGURE 2 Data flow objects.

When an output terminal on one object is connected to an input terminal on another object, the instance variable that describes the input terminal is modified so it points directly to the output terminal. Figure 3 shows a connection between two data flow objects, a function generator and a servo controller. The C expression that creates this connection is simply:

```
connect(out(fg), inp(servo)) ;
```

The expression is evaluated by sending the *out* message to the *fg* object, which will return the selected output terminal. The *inp* message is sent to the *servo* object, returning an input terminal. Finally, the *connect* message is sent to the output terminal with the input terminal as an actual parameter. The method for *connect* is implemented in class *OutputTerminal.*

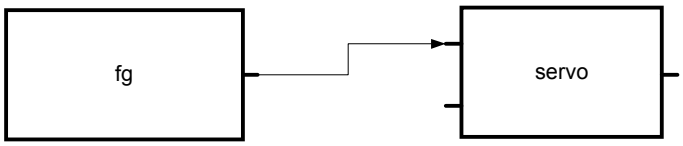

FIGURE 3 The logical effect of a connection.

Figures 4 and 5 show how the connection is implemented with memory pointers. A typical controller is built by creating a large network of interconnected. The rate monotonic scheduler arranges for the objects to run at the proper rate and in the required sequence. When an object is scheduled to run, it reads its input terminals, performs a signal processing operation and updates its output terminals.

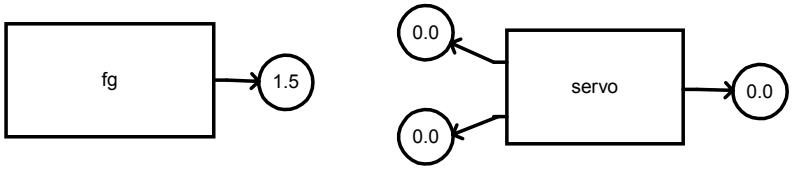

FIGURE 4 Memory pointers before connection.

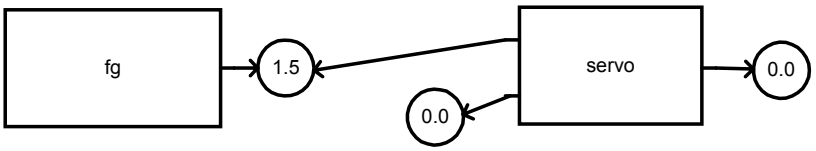

FIGURE 5 Memory pointers after connection.

Connections can be made or broken at any time while the control system is running. To undo the effect of the connection expression show above, the following C expression is used:

```
disconnect(out(fg), inp(servo)) ;
```

When terminals are disconnected, memory is allocated and attached to the free input terminal, restoring the memory that illustrated in Figure 4. The value stored in the new detached input terminal is copied from the old output terminal. This has the effect of freezing the last value sampled so that future executions of the disconnected object will see a constant value.

Interprocessor dataflow

Objects on different processors that share common memory can participate in a common data flow diagram. When connections are specified between objects on different processors, the data flow connections are automatically made using shared memory. Figure 6 shows the mechanism for an interprocessor connection. The program uses the same syntax to make local and non-local connections.

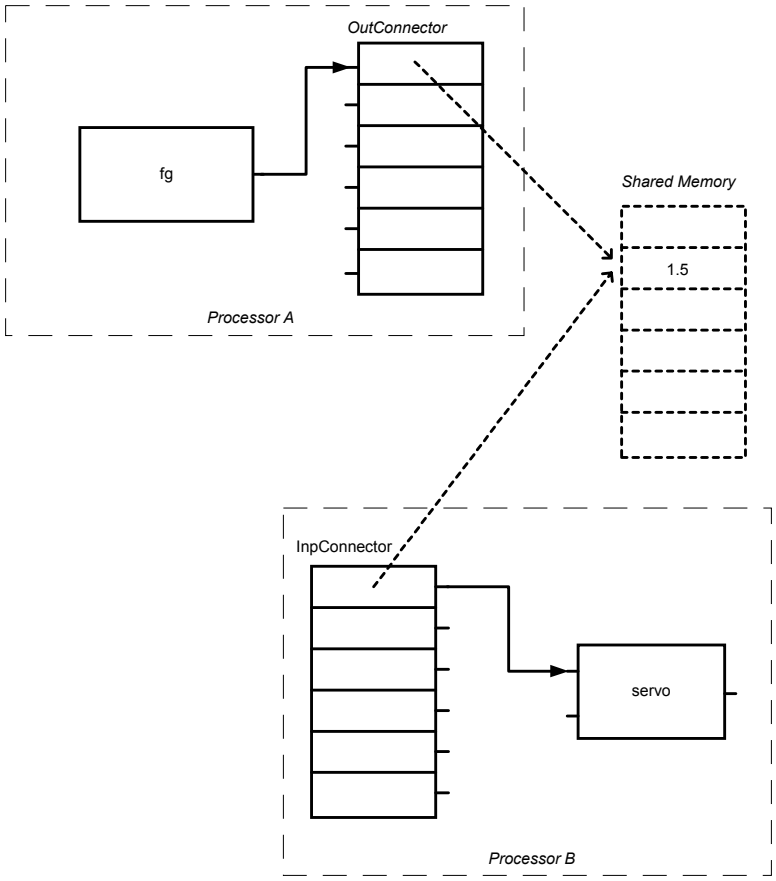

FIGURE 6 Connections between processors.

Schema will automatically recognize that a connection is directed to a remote object. The connection is first re-routed to an available slot on an *OutConnector*. The *OutConnector* will send a message to the *InpConnector* on processor B to complete the connection. Both connectors are configured to use a common shared memory location.

Connectors are themselves actively scheduled. When the scheduler runs the OutConnector on processor A, values stored on its input terminals are copied to a shared memory location. When the InpConnector runs on processor B, values are copied from the shared memory location and distributed to the output terminals.

Dataflow rate matching

When objects are connected that run at different sampling rates, COOPS provides rate-matching filters. A high-rate object connected to a low-rate object will use a low-pass filter to prevent aliasing. When a low-rate object is connected to a high -rate object, an up-sampling filter is used.

Dataflow synchronization between processors

When two objects are connected across processor boundaries, the connector objects will be scheduled independently, which sometimes gives unsatisfactory results. If the objects run at different sampling rates and their sampling rate difference is large, the matching filters discussed above will often correct disturbances caused by desynchronization. When the rate difference is small, the receiving objects must be scheduled to run after the sending object. To deal with this problem, a pair of synchronized connectors may be used. The classes are called *SyncOut* and *SyncIn.* They are substituted for the *InpConnector* and *OutConnector* described above. The memory management of the connection works as shown in Figure 6.

When synchronized connectors are used, the sender runs normally on a sequence controlled by its scheduler. The *SyncOut* connector is usually placed last on its sequence so it will run later than objects that generate output signals it will carry. When the *SyncOut* connector runs, it sends a trigger signal to its associated *SyncInp* connector using shared memory. The *SyncInp* connector is configured to control its parent sequence. The scheduler will defer execution of this sequence until the trigger arrives. All dataflow objects that depend on data from the *SyncInp* are placed later on the sequence, insuring that they run after their data samples arrive.

Dataflow in distributed systems

When processors are distributed on a network, data flow connections are made using reflective serial memory hardware such as the SysTran SCRAMNet (7). SCRAMNet connections are managed exactly like shared memory connections.

Time sampled data types

Data flow signals may be time-sampled integer, Boolean or floating point values. Vector valued connections are supported through terminal array types. Boolean-valued signals are used with logic gates and digital I/O hardware to perform the functions of a programmable logic controller. Many signal processing objects have a mixed set of terminal types.

Examples: A Boolean input terminal is often used to pause or resume the main signal processing path through a filter. Limit detectors have a Boolean output that becomes True when the input floating-point signal is out of bounds. Floating-point vector signals are used extensively in six degree of freedom robot and platform controllers.

The COOPS class hierarchy

A typical COOPS application contains hundreds of classes. The following outline displays only a representative subset. Additional class documentation is presented in (11).

Foundation classes
- Object
- Class

Data structures
- List
- Dictionary
- Association
- Array

Rate Monotonic Scheduling
- Clock
- Scheduler
- Sequence
- Task
- DSPObject

Filters
- FIRFilter
- IIRFilter
- AdaptiveFIRFilter
- LowPass
- HighPass
- Notch
- LatticeFilter
- SampleHold

Rate matching filters
- UpSampler
- DownSampler

Servos
- PIDServo
- TVCServo
- PolePlacementServo

Data Acquisition
- DAQBuffer
- CircularDAQ
- ScopeBuffer
- MatlabDAQFile

Dataflow connections
- Terminal
- IntTerminal
- FloatTerminal
- OutConnector
- InpConnector
- SyncInp
- SyncOut

Interprocessor messaging
- Link
- Message
- Uplink
- Downlink
- UpSocket
- DownSocket
- UpShare
- DownShare

Transformations
- ScalarTransform
- VectorTransfom
- Matrix
- LinearSwitch
- LinearDemux

Function generators
- RampFg
- CyclicFg
- RandomFg
- RandomPeriodFg

Limit detectors
- PeakPicker
- FloatLimit
- IntLimit
- DynamicLimit

Logic Gates
- AndGate
- OrGate

Modeling
- Mass
- MassInverse
- Jacobian
- InverseTransform
- Actuator
- Inverse Actuator

Hardware interface
- DigitalInput
- DigitalOutput
- AnalogInput
- AnalogOutput

Alltalk

Alltalk implements the graphical user interface for Schema applications. Alltalk also supplies the interactive programming and debugging environment for COOPS. Alltalk implements classes, metaclasses, methods, and messages using a block-structured syntax similar to Modula II or Ada.

Key features of Alltalk:

Portable

Alltalk is implemented entirely in ANSI standard C to achieve the greatest possible portability. Alltalk runs under most versions of Microsoft Windows, including WindowsCE for handheld devices. Versions for Macintosh and all Linux and Unix workstations that use X-Windows are also available. An application written in Alltalk will run without modification on any supported system.

Interactive

Alltalk programs can be modified and extended at any time. There is no modal distinction between developing and running a program. Interactive programming in an object-oriented environment has proven exceptionally effective for rapid prototyping. (13)

Self reflective

All the programming tools such as browsers, inspectors & debuggers are written in Alltalk and are part of every application. Classes that describe the execution and compiler variables are part of the environment. This information is a key asset for implementing transparent communication between Alltalk objects on networked computers.

Deterministic memory management

Alltalk programs rely on dynamic memory allocation and deallocation to support interactive software development and modification. Alltalk also employs the Pile memory architecture used by COOPS. The performance gained by this technique is substantial: Languages such as Smalltalk that employ reference counting garbage collection use up to 30% of the processor to handle this feature. (8) Languages that use scanning garbage collection have behaviors completely unsuited to a real-time application: They stop responding to the user interface for unpredictable time intervals.

Message expression syntax

Alltalk messages are implemented using 'dots' to separate message selectors from objects. Using the color example discussed above, a message to change the color of an object stored in the variable *x* has this form:

```
x.setColor[red]
```

As in COOPS, methods for *setColor* may exist in several classes. The runtime binding strategy searches for the proper method staring in the class of the receiver object and working up through all the superclasses. The search is optimized by a cache, which greatly reduces the overhead for message binding.

The left to right order of Alltalk expressions make them more attractive for many expressions involving objects and messages. For example, the data flow connection expression discussed earlier take this form in Alltalk:

```
fg.out.connect[servo.inp]
```

This expression illustrates cascaded messages: the result of *fg.out* (an *OuputTerminal)* becomes the receiver of the *connect* message which has the input terminal returned by *servo.inp* as an actual parameter.

Most developers prefer to configure the entire COOPS network using Alltalk rather than programming in C language level in COOPS.

Transparent distributed object communication

Alltalk programs can interact transparently with objects created on other systems in the Alltalk or COOPS environment. When an object reference is exported, the local address is compressed to make room for a node number that uniquely identifies the system where the object was created. The resulting bit pattern is made an instance of class *RemoteObject.* Class *RemoteObject* implements a method for *undefined*, but no other methods. Consequently, all messages directed to an instance of *RemoteObject* are handled by the method for *undefined.* The message receiver's bit pattern contains the node number, which is used to find the appropriate *Link* to the remote system. Class *Link* contains all the mechanisms for formatting and transmitting a message to a remote processor. When the remote system receives the message, the receiver object is transformed back into a local address. Any actual parameters for the message are also transformed into local objects if their node numbers match that of the local system.

System State Preservation

Alltalk applications have automatic support for saving and restoring the state of complex object networks. Although Alltalk messages can directly manipulate remote objects, it is a common practice to introduce a set of local Alltalk classes that mirror those found in the COOPS environment. These mirror classes are called models, because they are used to construct a complete description of how the real-time network is configured. All messages that modify the state of a remote COOPS system pass through the models where information needed to restore the state of the remote system is cached.

When Alltalk is used to create a prototype GUI for demonstration purposes, the model layer remains part of the prototype. When users operate controls on the GUI, the models are modified. When no COOPS system is attached, the models simply record changes to the state information. When a COOPS system is attached, the changes are passed through to the remote system.

The entire hierarchical model layer may be written to a file using a general object pacification mechanism. These files are the 'documents' created by an Alltalk application. When the file is read, the state of all the model objects is restored. The models then update the remote COOPS system so they reflect the new state. This procedure is used record all the calibration, turning and configuration information for the COOPS network. Because the remote COOPS system usually a machine controller, it is often call the *machine.* Figure 7 illustrates the relationship between the view, model and machine environments.

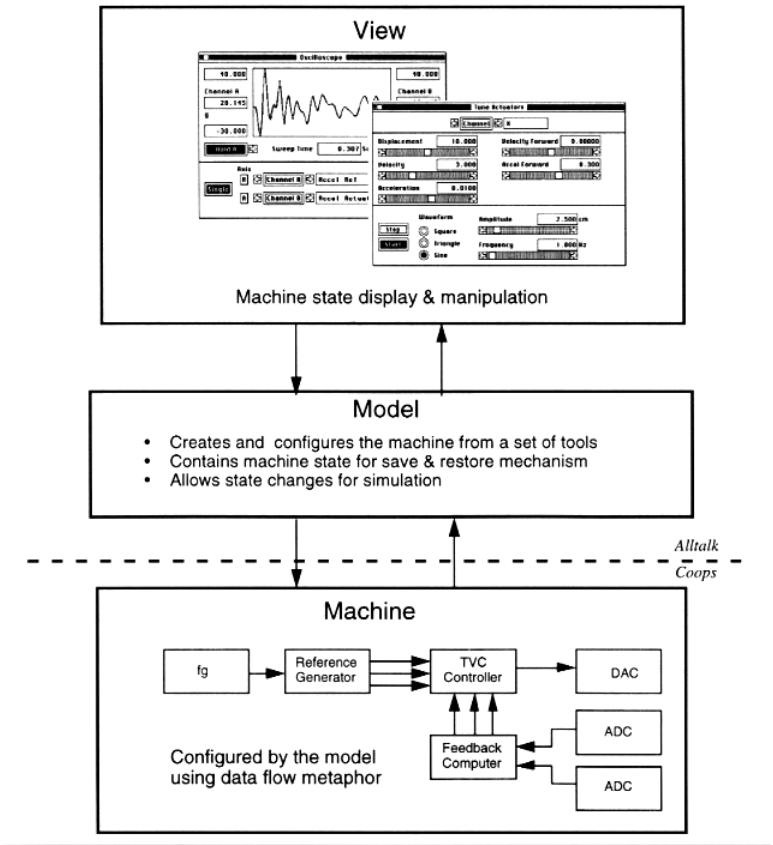

FIGURE 7 View-Model-Machine relationships.

International language support
Alltalk applications support dynamic language translation. A menu selection is used to choose one of several supported languages. All graphical user interface elements that contain text instantly change to display the selected language, resizing and repositioning themselves as necessary. This is a valuable feature when training because the teacher and student can switch between their native languages while the application is running.

Developers write applications in their native language. A data file is then created automatically that contains all GUI display text in the native language. A bilingual translator edits this file by adding appropriate translations for each phrase. Any number of translations can be created and included with the same application.

The Alltalk class hierarchy

A subset of the Alltalk class hierarchy is presented below to suggest the breadth of features provided:

Foundation classes
- Object
- Class
- Method
- Selector
- Context

Files
- Directory
- File
- StringFile
- BinaryFile
- SerialFile
- BinaryRecord
- Socket

Data structures
- Dictionary
- Association
- List
- Array
- ByteArray
- String
- WideString
- Literal
- WideLiteral
- RealArray
- IntegerArray

Programming tools
- Browser
- Workspace
- Inspector
- Listener

Windows
- Window
- ViewWindow
- Panel
- CrtWindow
- ModalDialog
- ModalConfirm
- ModalGetParam
- ModalNotifier

Menus
- Menu
- MenuBar
- PopUpMenu
- ComboBox
- RadioCluster

Graphics Support
- View
- Image
- FillView
- FrameView
- Event
- Control
- Font

Display text
- Text
- DynamicText
- Paragraph
- EditParagraph
- Translation

Buttons
- Button
- BoxButton
- NameButton
- ImageButton
- CheckBox
- RadioCluster

Signal manipulation
- NumberDisplay
- NumberControl
- SignalDisplay
- SignalControl
- BarSignalDisplay
- Scope
- Trace
- Knob
- CircularKnob

Remote object models
- Model
- ModelArray
- AliasModel
- Parameter
- RemoteParameter
- RemoteObject
- RemotePort
- Processor

Networked objects
- Node
- Link
- RemoteThing

Alltalk GUI Gallery

The following windows are examples of programming tools and user interface panels from a typical Alltalk application. (The NADS motion controller) These panels are presented to show how the Alltalk GUI looks to the end users and developers.

The *Browser*, Figure 8, is used to inspect or modify any Alltalk class. The upper left window pane shows a list of all classes. The upper right pane displays all the method selectors implemented by the class. The lower large pane shows the source code for the class. More details about the Alltalk development tools are found in (9).

FIGURE 8 The Alltalk browser.

The *Inspector*, Figure 9, is used to examine the state variables of any object. Selecting any object displayed in the inspector view will 'open' that object to display a new view.

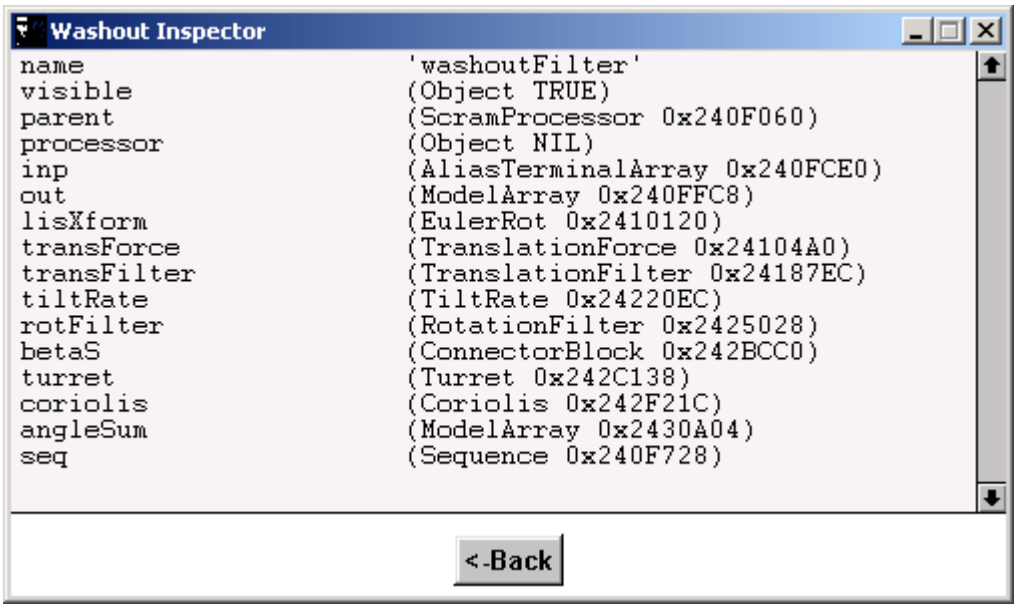

FIGURE 9 An object inspector.

The *ScopePanel*, Figure 10, is a simple two-channel oscilloscope. The component buttons, displays and text are all instances of other Alltalk GUI classes:

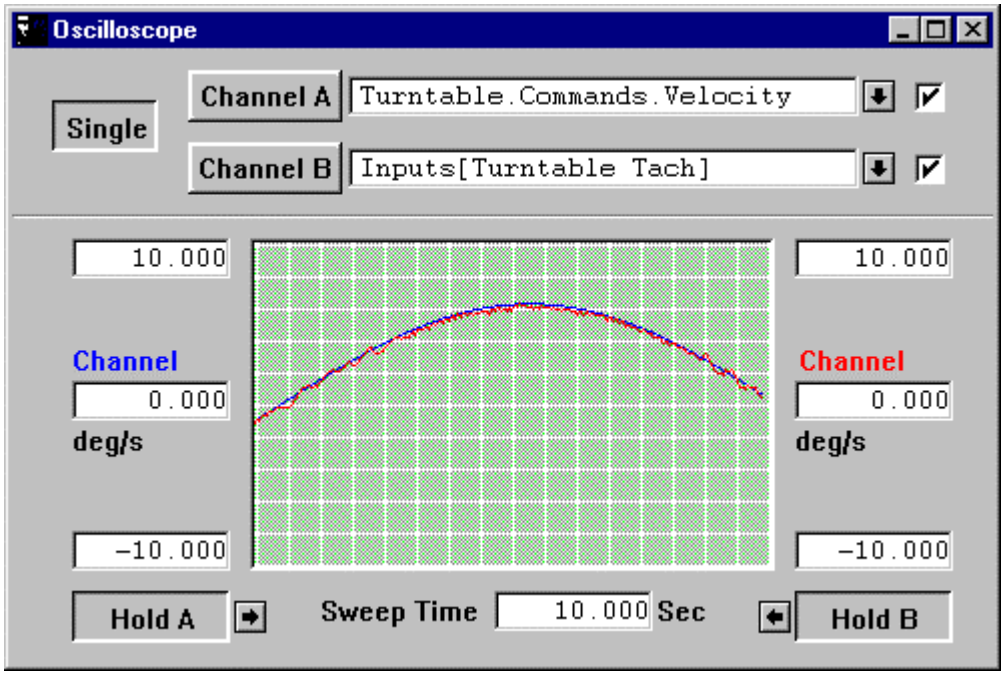

FIGURE 10 A two channel oscilloscope panel.

Transfer functions can be measured by specifying signals on any pair of terminals. The *Transfer Function Estimator*, Figure 11, uses an FFT to determine the magnitude and phase of the transfer function.

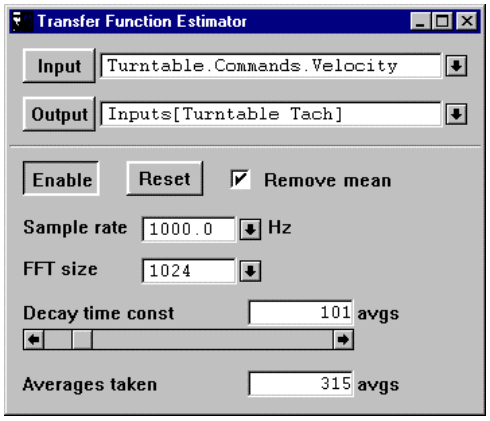

FIGURE 11 Transfer function estimator configuration.

Once configured, the transfer function can be observed on the *Frequency Response Function Plotter,* Figure 12. The plots can also show the power spectral density of the signal of the input or output signals. Other menu options (not shown) allow the user to display the transfer function of any digital filter in the system.

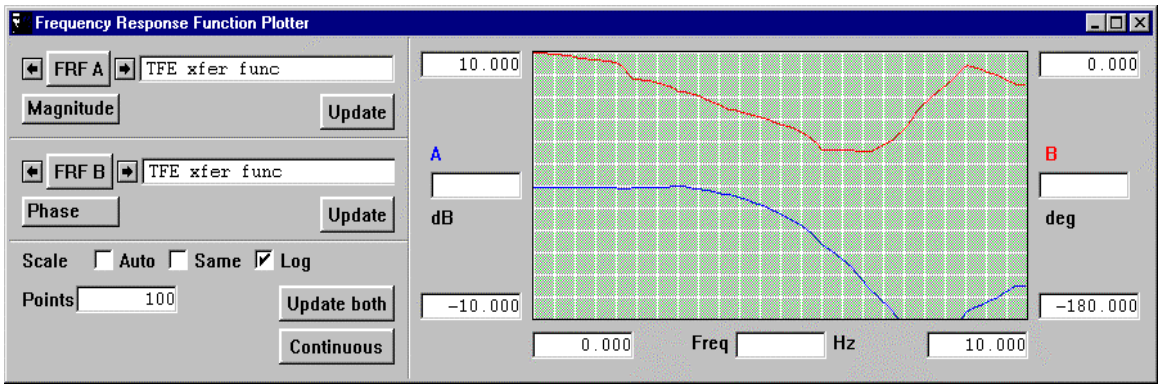

FIGURE 12 Transfer function display.

Washout filters are specialized high pass filters used in driving simulators to keep the motion base centered in the room while simultaneously providing the driver with appropriate inertial cues. The panel shown in Figure 13 is used to adjust the washout parameters for the TARDEC and NADS driving simulators. Details about the NADS washout filter and its adjustments are presented in (10).

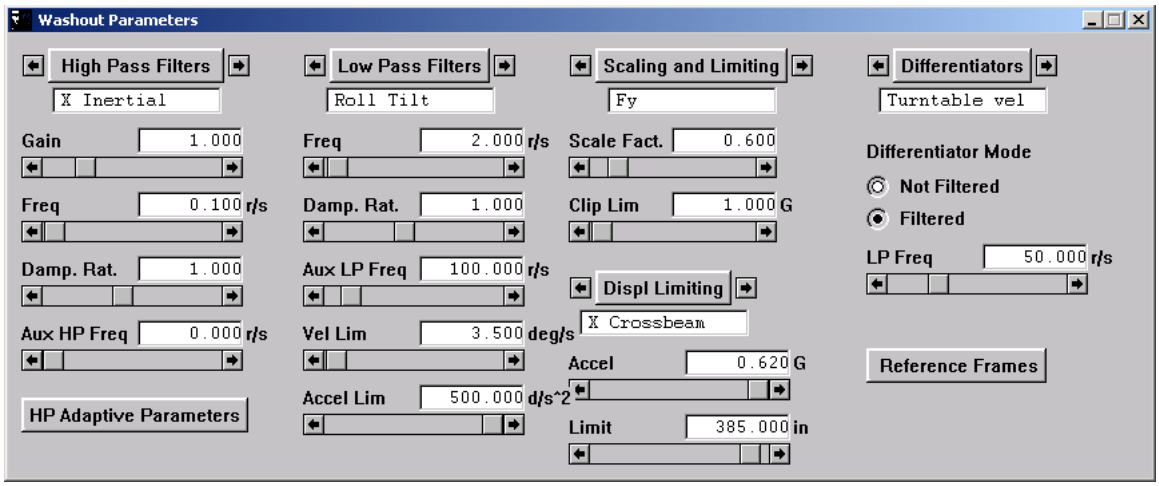

FIGURE 13 Washout filter tuning.

DRIVING SIMULATOR APPLICATIONS

The motion control problem for driving simulators has many demanding requirements where Schema has proven effective:

Driving simulators are large real-time applications that must integrate large numbers of sensor and actuator channels on physically distributed computers. Because Schema supports distributed dataflow processing, the hardware for hundreds of signal converters may be located on multiple computers placed close to the actuator arrays they control.

Accurate high-bandwidth servo control on multiple synchronized degrees of freedom are required to implement the motion subsystems. Schema uses closed loop servo control in degree-of -freedom space, rather than on individual motors or actuators. Kinematic transformations to and from actuator space are computed continuously in the controller. This strategy has numerous advantages for tuning and operating the motion base. (15)

Minimum phase delays between driver initiated command and motion response.
The payload is stabilized with a multi-variable feedback controller. A parallel feed-forward model predicts the required actuator and motor commands. This strategy has proven very effective in minimizing phase delays in the control system. (16)

The controller must deal with non-linear plant characteristics: The moving configuration of actuators and payload determines a dynamically changing plant transfer function. The feed-forward part of the Schema controller contains a non-linear inverse model of the actuator-motor-payload system. This inverse model consists of a rigid body component and a hydraulic component. The rigid body inverse model predicts the forces required to move the payload allowing for all inertial effects and the variable orientation of the actuators. The inverse hydraulic component predicts the valve openings required to obtain the required actuator motion.

Safety and reliability are critical because simulators have human drivers. Schema addresses safety and reliability in several ways: Limit detectors monitor all safety critical parameters in the controller. Responses to out-of-bounds conditions on any signal are configurable to provide a warning, arrest the motion of the platform, or perform a complete power shutdown. Integrated time domain and frequency domain analysis tools make it possible to explore and verify controller stability throughout the envelop of motion.

The most important contribution to safety is the stability of the software component base. Because Schema controllers are constructed using classes that have evolved through use in diverse applications, they have become highly reliable. Conventional single-step source code debugging is rarely done because code defects at this level have been eliminated. The process of programming in Schema consists of constructing the appropriate data flow diagram and tuning the resulting system while examining signals. Graphical oscilloscopes, meters and signal analysis tools are the most commonly used tools: The debugging and development process resembles electronic prototyping where the circuit and component values can be modified while the circuit operates. A detailed analysis of software and hardware safety issues for a driving simulator is presented in (17).

Driving simulators that use Schema

Schema has been used for several recent simulator projects: TARDEC, NADS, Ford, and VTI. All of these systems have embedded COOPS motion controls integrated with Alltalk. The TARDEC system also uses COOPS for high-level system configuration and data flow between subsystems.

The following sections provide a brief overview for some of these applications.

The TARDEC/TACOM tank simulator

The tank-driving simulator at TARDEC/TACOM is a large Schema application. There are two motion base systems: one for the tank driver, Figure 14, and one for the turret crew, Figure 15. The turret motion base supports a real tank turret. When simulating a tank, the system operates both platforms in parallel as one vehicle. It is also possible to configure either motion base as an independent vehicle.

Each motion controller contains:

- 64436 Objects
- 514 Data flow objects
- 3608 Terminals
- 5027 Parameters

The COOPS object network runs on two PowerPC processors: One for servo control and one for the washout filter and safety monitor. Additional information on the architecture and operation of the TARDEC motion controller is available in (18).

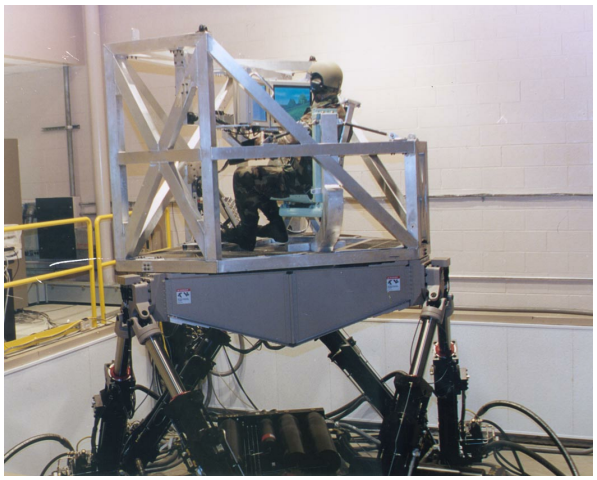

FIGURE 14 The TARDEC tank driver's motion base.

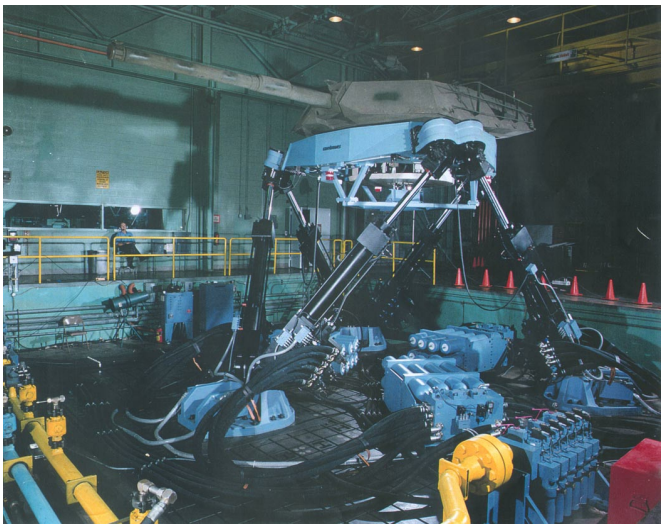

FIGURE 15 The TARDEC tank crew motion base.

The TARDEC simulator uses COOPS for high-level system configuration and scenario management as well as dataflow between the subsystems. The motion controller hosts are Windows NT machines. All other subsystem hosts are Unix workstations. Figure 16 shows the simulator architecture.

The simulator can participate in DIS Net battle simulations with crews driving on remote simulators. Each participant can see the other tanks and try to blast them out of existence. (Simulated shocks and explosions only.)

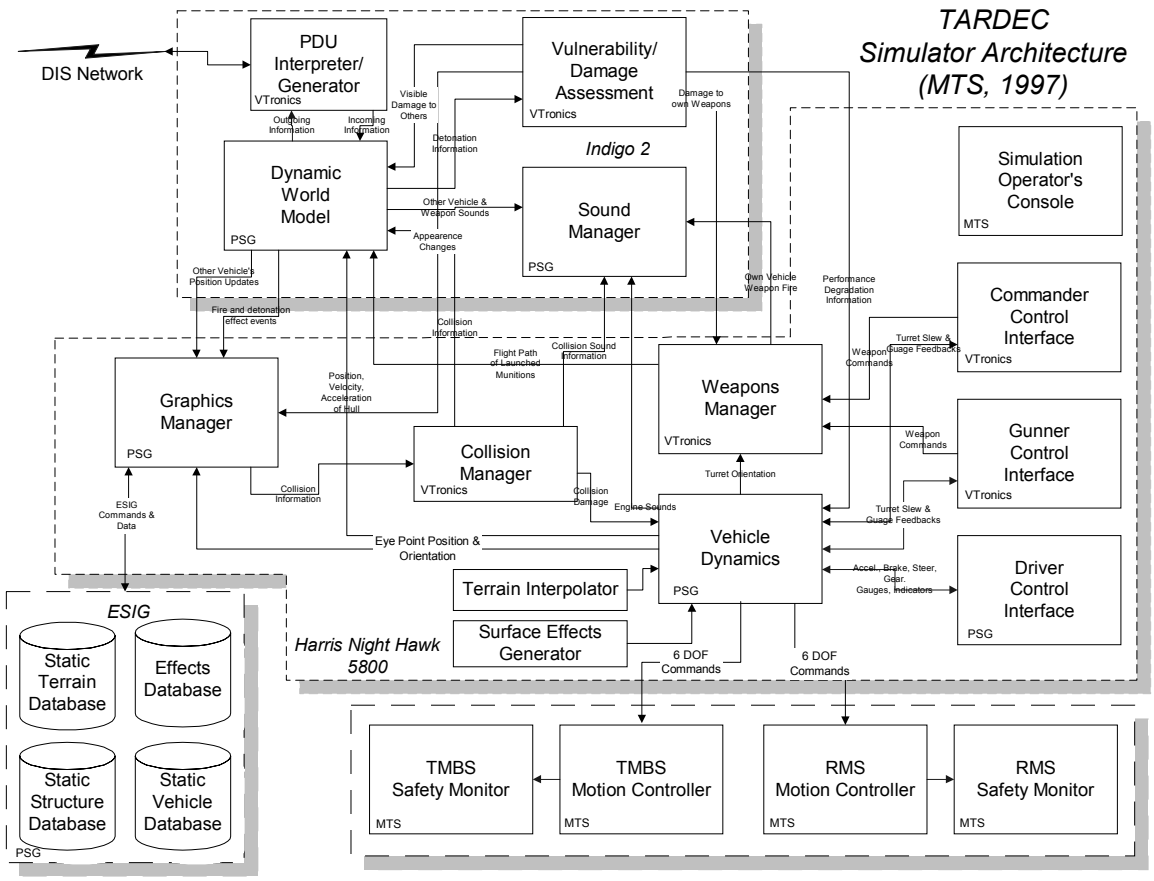

FIGURE 16 TARDEC driving simulator architecture.

Proceedings of the 1st Human-Centered Transportation Simulation Conference, The University of Iowa, Iowa City, Iowa, November 4-7, 2001 (ISSN 1538-3288).

NADS Motion Controller

The National Advanced Driving Simulator has a Schema controller for a hexapod motion base mounted on an X-Y carriage. (See Figures 17 and 18.)

FIGURE 17 NADS Testing at MTS.

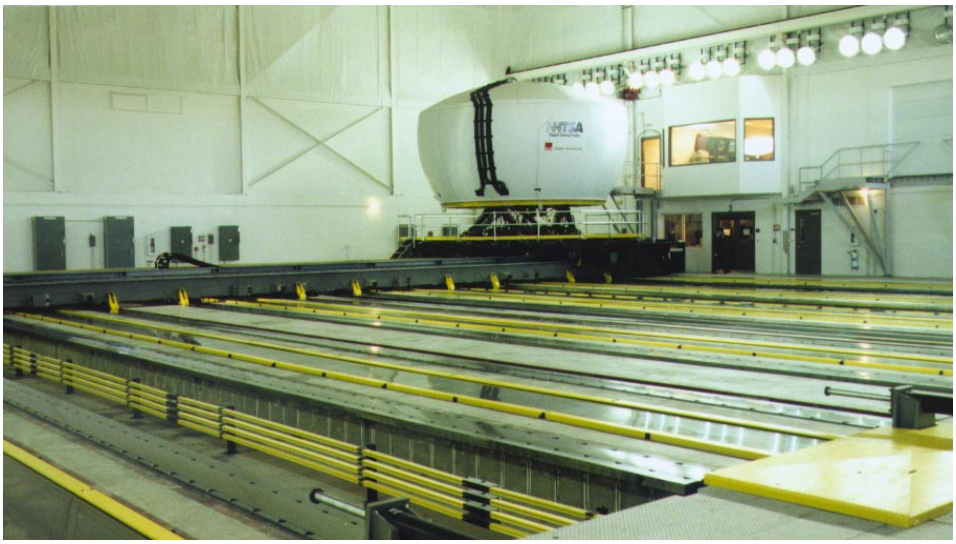

FIGURE 18 NADS at the passenger dock.

The NADS motion controller runs COOPS on distributed PowerPC processors with an Alltalk GUI running on a Windows NT host. Four PowerPC subsystems are placed near the motors or actuators they control, all subsystems are linked using SCRAMNet shared memory.

1. Command generator and washout filter
2. Hexapod controller
3. X crossbeam controller
4. Y carriage controller

The Schema environment for the NADS motion controller contains:

- 72,245 objects
- 650 Data flow objects
- 3500 Terminals
- 4375 Parameters

Scenario management and system configuration is handled by software from TRW.

The washout filter for NADS is designed to apportion all translations to the X-Y carriage. Tilt coordination and vertical motion are handled by the hexapod. A turntable on the platform is used for yaw motion. Figure 19 shows the top-level dataflow diagram for the washout filter. Figure 13 shows the control panel used to adjust the filter parameters.

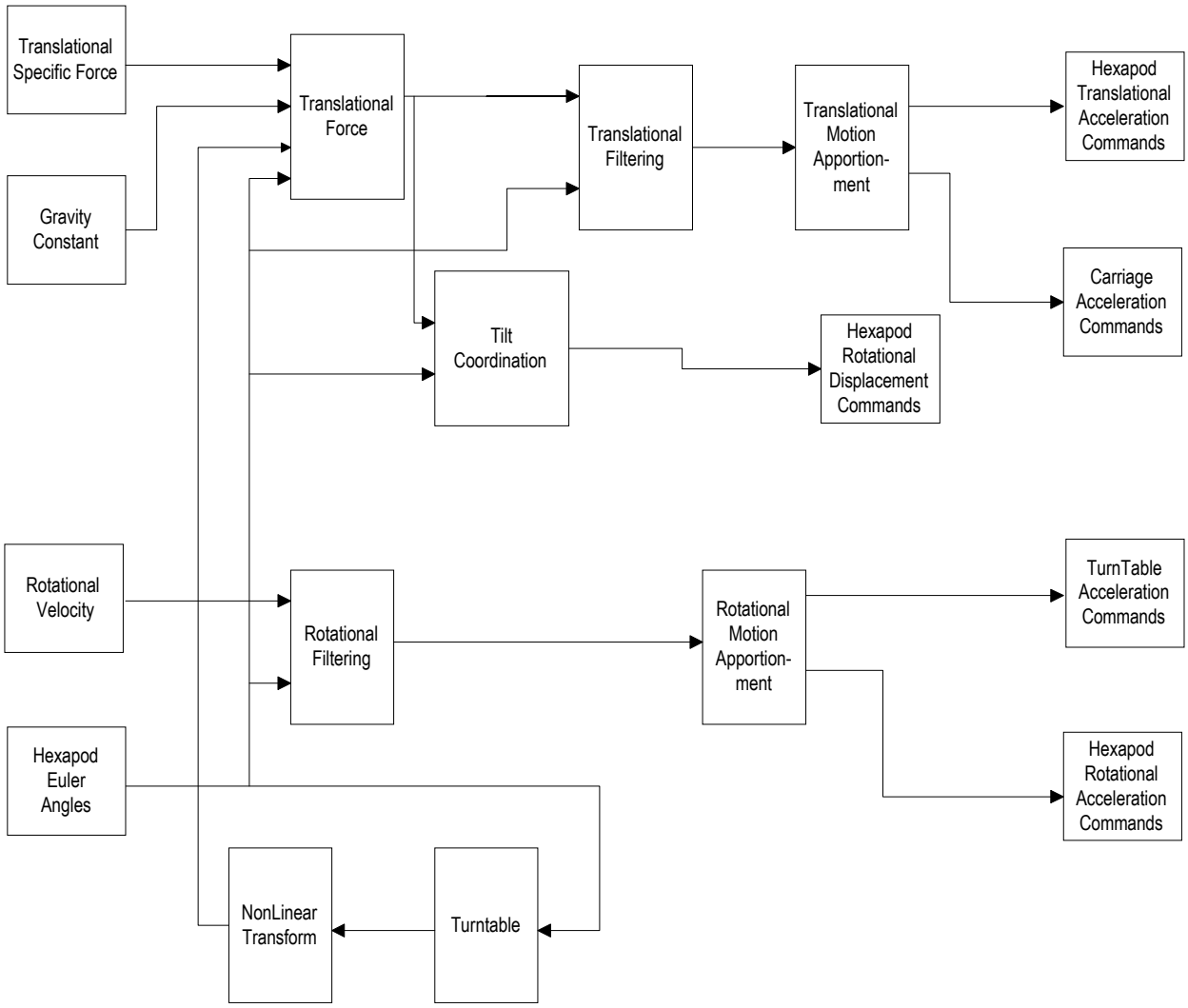

FIGURE 19 The NADS washout filter.

A review of the NADS controller is presented in (15) and (16). The architecture and operation of the controller is covered in great detail by (10).

Ford VIRTEX Driving Simulator

The Ford "Virtual Test Track Experiment" (VIRTEX, Figure 20) uses a Schema motion controller running on a single PowerPC processor. Ford Research Laboratory developed all other aspects of the simulator software.

FIGURES 20a, 20b Ford VIRTEX motion base and dome.

VTI Driving Simulator

The Swedish National Road and Transportation Research Institute, Figure 21, has developed and operated a pivoted motion base for several years. Recently, the institute has added a linear degree of freedom based on the NADS motion base technology using a motor driven metal belt and a carriage with hydrostatic bearings. This system uses Schema only for the linear controller.

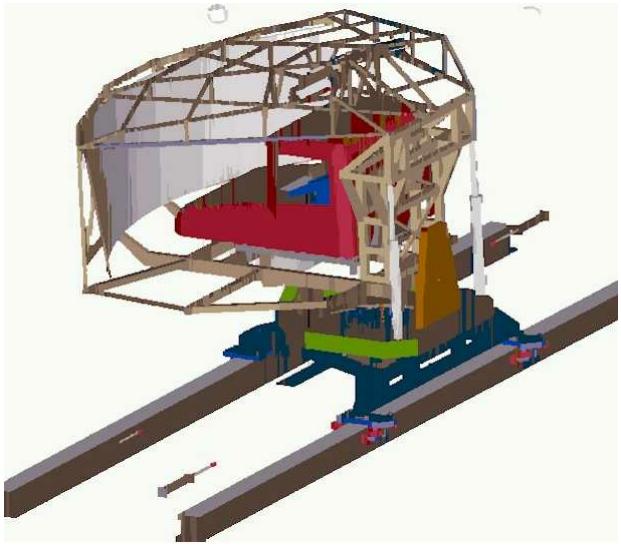

FIGURE 21 VTI Conceptual drawing with cab.

The new VTI simulator is currently being assembled and tested in Sweden, Figure 22.

FIGURE 22 VTI Linear motion base.

FUTURE WORK

Data flow diagram editing

Currently, scripts written in Alltalk create Schema dataflow graphs. This is somewhat archaic because previous MTS data flow systems used integrated graphical diagram editors. (12) The next major release of Schema will support interactive editing of the diagrams.

Integration with Matlab/Simulink diagrams

Many end-users want to incorporate Matlab or Simulink models with COOPS controllers. A currently funded development program will add this feature to our next generation earthquake simulators. Upon completion, this capability will be part of the generic Schema toolkit.

Interconnected rigid bodies (joints)

The forward and inverse rigid body models used with the hexapod controllers can handle arbitrary actuators connected to a single rigid body with arbitrary inertia. In order to handle interconnected rigid bodies, joint models for common connections are being developed for Schema. Several researchers have described frameworks that deal with kinematics and dynamics in an object-oriented framework. (19)

Flexible body models using modal analysis

Vibration modes are currently handled ad-hoc by adding hand-tuned resonators and anti-resonators to the control loop. This is currently done in degree-of-freedom space and is only accurate for small displacements from the home position of the payload. By measuring the normal modes of the payload and incorporating them into the generic inertia model, we hope to obtain better compensation for all payload orientations.

Improved hydraulic controls

The inverse hydraulic models used in the current driving simulators and shock controllers (20) have been extended to include several non-linear effects due to oil compression flow, large actuator displacements and asymmetric actuators. Updated models and inverse models have been developed with Simulink and will be part of future Schema systems.

Large scale distributed systems using Internet II

The University of Nevada Reno has obtained an NSF grant for the NEES program: "Network for Earthquake Engineering Simulation". Several features will be added to Schema for this program including broadband audio and video channels for teleoperation and many new error recovery and fault tolerance features.

ACKNOWLEGEMENTS

To realize the vision of a comprehensive framework for industrial control software would not be possible without a large community of developers and end users. The Schema architecture has matured through over 100 entity-years of labor. The authors would like to acknowledge the large contribution of these Schema developers:

Mark Brudnak (TARDEC), Mark Fullen, Tom Hessburg, Dean Hystad, Peter Jeeps, Rod Larsen, Jim Langseth, Gary Laughlin, Jon Lonstreth, Karen Nohr, Victor Paul, Paul Randal, Rich Romano (Real Time Technologies), Jim Rosenow, Craig Schankwitz (University of Minnesota), Marlin Sunderman, Scott Zwetler

All contributors are current or former employees of MTS System Corporation unless otherwise noted.

REFERENCES

1) Goldburg, A. and Robson D., *Smalltalk: The Language and Its Implementation*, Addison-Wesley, 1983.

2) Sparks, H., *Object Oriented Dataflow Programming Techniques for Industrial Automation*, Proceedings of Control Expo 90, Chicago, 1990.

3) Chatham, B. and Sparks, H., *Butterfly HOSE: Graphical Programming for Parallel Systems,* Abstracts of IEE and USENIX Fifth Workshop on Real-time Software and Operating Systems, Washington, 1988, pp. 75-79.

4) Kiczales,G. J. Rivieres, & D. Bobrow, *The Art of the Metaobject Protocol,* MIT Press, 1991

5) Freburger, K. *RAPID: Prototyping Control Panel Interfaces*, Proceedings of OOPSLA 87, ACM 1987, pp. 416-422.

6) Klein, Mark, Thomas Ralya, Bill Pollak, Ray Obenza, Michael Harbour, *A Practitioner's Handbook for Real-Time Analysis,* Kluwer, 1993

7) *SCRAMNet Network: Simplicity and Speed*, SYSTRAN Corporation, 1997.

8) Krasner, Glen, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1984.

9) Sparks, H., *Introduction to Programming with Alltalk,* MTS Systems Corporation, 1991.

10) *NADS Motion Subsystem, Operation Software Manual*, MTS System Corporation, 2001.

11) Sparks, H., *MTS COOPS: C Object Oriented Programming System Product Overview,* MTS Systems Corporation, 1995.

12) Sparks, H., *Uniform Dataflow Software System for Global CIM Applications*, Proceedings of CIMCON 90, National Institute of Standards & Technology, 1990, pp. 353-372.

13) Goldburg, A. *The Influence of an Object-Oriented language on the Programming Environment,* Interactive Programming Environments, McGraw-Hill, 1984, pp. 141-174.

14) Buhr, Peter A., Harji, Lim & Chen, *Object-Oriented Real-Time Concurrency*, Proceedings of OOPSLA 2000, ACM, pp. 29-46

15) Carmein, Judy A. and Clark, Allen J., *New Methods for Increased Fidelity and Safety of Simulation Motion Platforms,* Proceedings of the AIAA Modeling and Simulation Technologies Conference, Boston, 1998, pp. 295-304.

16) Clark, A., Sparks, H., Carmein J., *Unique Features and Capabilities of the NADS Motion System*,(CD-ROM), Submission ID 254-O, Proceedings of the 17th International Technical Conference on the Enhanced Safety of Vehicles, Amsterdam 2001. NHTSA.

17) *The Ride Motion Simulator Safety Assessment Report,* MTS Systems Corporation, 1998.

18) *TARDEC RMS Motion Controller Users Guide,* MTS Systems Corporation, 1998.

19) Lee, Ji Y, Kim, Hye J., and Kang, Kyo C., A Real World Object Modeling Method for Creating Simulation Environment of Real-Time Systems, *Proceedings of OOPSLA 2000, ACM, pp. 94-104.*

20) Hessburg, T., Krantz, D., *Shock Test System Performance Prediction and Feed-Forward Control using High-Fidelity Nonlinear Dynamic Hydraulic System Modeling*, Proceedings of the 68th Shock and Vibration Symposium, Baltimore, 1997.