Software Architecture of the NASA Shuttle Ground Operations Simulator--SGOS

R. P. Cook
Department of Computer Sciences, Georgia Southern University, Statesboro, GA 30460 U.S.A.
C. T. Lostroscio
NASA Mail Code YA-D8, Kennedy Space Center, FL 32899 U.S.A.

SUMMARY

The SGOS executive and its subsystems have been an integral component of the Shuttle Launch Safety Program for almost thirty years. It is usable (via the LAN) by over 2000 NASA employees at the Kennedy Space Center and 11,000 contractors. SGOS supports over 800 models comprised of several hundred thousand lines of code and over 1,000 MCP procedures. Yet neither language has a for loop!! The simulation software described in this paper is used to train ground controllers and to certify launch countdown readiness.

KEY WORDS: NASA, Shuttle, simulation, ground operations

INTRODUCTION

A recent project (KLASS), which was sponsored by the Education Office of the NASA Kennedy Space Center, to create a Shuttle-Launch edu-tainment application for 12 to 14-year olds presented an opportunity to document the architecture of the Shuttle Ground Operations Simulator (SGOS). The Kennedy Launch Academy Simulation System (KLASS) was created by porting SGOS and its SimAPI library.

The first author ported the SimAPI front-end of SGOS to Windows and Apple's OS-X; the second author was one of the system's architects and has ported the SGOS executive to Linux. The paper presents a rare glimpse into software that has helped to certify over a hundred Shuttle launches.

Simulation has always played a vital role in the space program. SGOS is used to test new launch control (firing room) software, to train launch teams, and to certify launch countdown readiness. The devices participating in the simulation are encoded as continuous and discrete mathematical models that are accurate real-time emulations of the physical orbiter, external fuel tank, solid-rocket boosters, payload interfaces and ground-support components. The simulation is plug-compatible with the I/O of the firing room.

SGOS was first written (over twenty-five years ago) for a mainframe using a mixture of FORTRAN and assembler. In 1995, a four-year project was undertaken to port SGOS to the C language running on a more versatile modern architecture. The Executive was ported to a VME single-board computer (running Sun Solaris) to control the hardware interfaces to the firing room. The Executive and all other support software was also ported to Sun Solaris workstations. Table 1 lists the major subsystems of SGOS.

SGOS Subsystems	
RSI	Real-time Simulation Interface provides the hardware and software to emulate Shuttle data links with modeled data
Execution	Supports the execution and manipulation of SGOS models and Model Control Procedures (MCPs)
Build	Supports the conversion of the SGOS modeling language and MCPs into executable objects
Support	Provides tools for creating, controlling, debugging, and analyzing models

Table 1. List of SGOS Subsystems

More recently, Cook and Lostroscio ported the model front-end library, the support tools and the Executive to Linux, Windows and Mac OS-X for the KLASS project. KLASS enables students anywhere in the world to collaborate (using the Internet) to count down a Shuttle launch. Further, teachers can control the execution of KLASS models to introduce problems that have either occurred in past launches or to create new problems (based on learning objectives) for the students to solve.

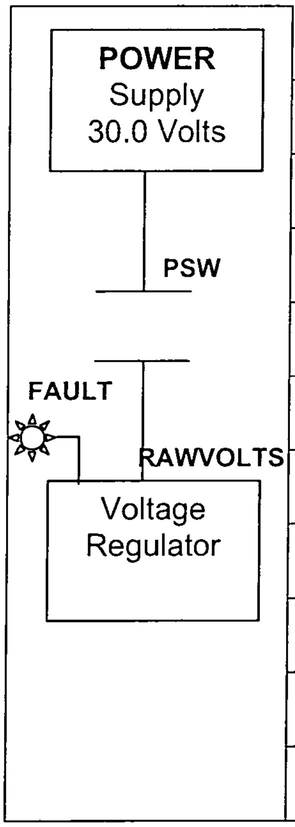
The paper introduces the operation of the Real-time Simulation Interface (RSI) and the Simulation Executive. The operational difference is that the RSI is connected to the firing room via the MIL-STD-1553 protocol and is clock-driven to run the simulation in "real" time. In Remote Terminal Mode (non real-time), the Simulation Executive runs the models at CPU speeds with simulated clocks. The paper covers the modeling language and support system (SimAPI) in more detail than the Executive. The reason is that those components will most affect KLASS users.

SIMULATION MODEL

The complete model program size for a typical launch simulation is just over 100 megabytes. All components of a simulation, except user interaction commands, are statically linked to the executive. A simulation is comprised from databanks, model-control procedures (explained later) and systems. Systems can contain sub-systems, which can have "in" and "out" parameters. Sub-systems are coded from models.

A databank is an inverted list of sub-system and model variables, their data types, and the mapping to hardware. The latter information is only used for real-time emulation. The supported data types are **continuous** and **discrete**. Examples of continuous data are pressures, temperatures, currents, flow rates, and voltages. Models are programmed as unordered lists of segments. Each segment is a partially-ordered (explained later) list of statements.

Table 2 lists a model source file for a simple example. The voltage at the regulator is dependent on the state of PSW (power switch). RAWVOLTS will be either 30.0 (the voltage of the power supply) or 0.0 depending on whether PSW is closed (**ON**) or open (**OFF**). Comments are delimited by a pair of $ (dollar signs). Both the power source and the voltage into the regulator are physical processes so they are encoded as continuous variables. The **CMOD** keyword denotes a continuous model segment. The LOGIC FUNCTION SWITCH (**LFS**) is a library function that allows the modeler to assign one of two continuous values to RAWVOLTS based on whether the discrete value of PSW is **ON** or **OFF**. The DLM statement, on the other hand, assigns **ON/OFF** to a discrete variable (FAULT light) if the continuous variable RAWVOLTS is out of range.



DATABANK	DBANK;	
:SYS	VOLTREG;	
:INIT	D	FAULT /OFF/;
:INIT	C	POWER /30.0/;
:INIT	D	PSW /OFF/;
:INIT	C	RAWVOLTS /0.0/;
:CMOD		
RAWVOLTS	=	LFS(PSW,POWER,0.0);
CALL DLM(FAULT, RAWVOLTS, 29.0, 31.0, OUT);		
:SYSEND;		

Table 2. Example Voltage Regulator Model

The notation of the variables' data types (C/D) in the model initialization section is redundant as the types are also specified in the databank. The recommended coding style is to initialize all variables, which has the additional advantage of documenting their types on the program listing. The statements in the programming language will be discussed in more detail in the Build Sub-System Section.

RSI AND MODEL EXECUTION SUB-SYSTEMS

The RSI system[1] is built on a 21-slot VME chassis containing two Force Sparc CPUs for system control together with generic simulator cards (GSCs) that perform Shuttle and ground system data flow emulation. Each GSC is dynamically configurable to emulate one of a Shuttle downlink pulse code modulation (PCM) stream, a ground data bus (GDB), or a launch data bus (LDB). A DSP (TMS320) in each GSC enables the card to emulate data protocols for a variety of hardware via a MIL-STD-1553 protocol interface. Inter-processor communication utilizes a fiber-optic network connection and shared memory segments, which are implemented on reflective memory cards attached to each processor.

Prior to execution, all models, sub-systems and control procedures are translated to C-language statements. The current simulation has approximately 800 models and 1,000 control procedures. Procedures are written in a variant of the Ground-Operations Aerospace Language (GOAL). MCPs are used to configure sub-systems into predefined states and to introduce "ad hoc" failures for training purposes.

The most complex step in the translation process is to calculate the dependency chain for model segments (i.e. **CMOD**). A model, such as VOLTREG, with segments that have no variable dependencies or input parameters would be assigned Level Zero. Any segments in the same model, or in other models, that depend only on RAWVOLTS would be assigned Level One. The analysis continues until transitive closure is completed with no cycles.

Further, each segment is analyzed for external dependencies. This results in a novel computation model in which only the segments that reference a changed variable are executed each model cycle. As a result, SGOS resembles a dynamic dataflow architecture.

The C modules are compiled to object code and statically linked with the Executive. The default execution step interval in real-time mode is 50 milliseconds (with a capability down to 5 milliseconds). During each 50ms cycle, stimuli to the model are processed from four different sources: Firing Room input, model-control procedures, user commands and time-based changes to continuous and discrete variables within each model.

All stimulated segments are queued in Level order at the beginning of each cycle. As each model segment runs, it may, in turn, generate changes to other model variables. This results in further segment queuing, which is also ordered by Level. Eventually, no more variables will change and no more segments will be queued. The cycle completes. All this happens every 50ms simulation cycle.

BUILD SUB-SYSTEM LANGUAGE SYNTAX

The GGOS Build Sub-system[2] consists of translators and tools that convert models and procedures to C source code compile and link a simulation, and that manage databanks. Every model has a unique system name. A partial listing of model syntax is documented in Table 3. Comments are delimited by a dollar sign ($) and can occur wherever white space is permitted except within strings or other comments.

Statements (in order)	Explanation
DATABANK <name>;	References the name space for this model
:SYS <name> [(<param-list>)];	Defines the unique system name for this model
[:INIT {C|D} <name> /<constant>/;]...	Initialize a continuous (C) or discrete (D) variable. The constant must match in type.
[:SUBSYS <name> AS <name> [(<param-list>)];]...	Import a parameterized duplicate of another model and assign a unique system name
[:DIM . <name> (<constant> [,<constant>]);]...	Array names must be preceded by a period. Bounds can range from 1 to 1024. The data type is double. Arrays are model-local variables and do not propagate changes.
{ {:BMOD <name> [,1]; | :CMOD <name>(<const>) [,1]; | :DMOD <name>; } <statements> }...	Boolean, Continuous, or Discrete Model segments. The [,1] limits execution to once per cycle. The constant qualifier on a CMOD is the response time (.1s to 10s). It is the maximum time until the segment is recalculated if an input variable does not change sooner.
:SYSEND;	Denotes the end of a program

Table 3. Model Program Syntax

A model is an unordered list of segment definitions. There are three different types of segments: Boolean (**BMOD**), continuous (**CMOD**), and discrete (**DMOD**). The execution of segments, and their associated statements, is initiated by changes to model variables. The segments can occur in any order in a program. The beginning of the next segment or the occurrence of a SYSE delimits the end of a segment.

The **BMOD** and **DMOD** segments can be best visualized as implementations of sequential or combinatorial circuits. They capture the cause and effect relationships among event occurrences in the simulation. The Boolean operators are & (and), + (or), - (not). The library functions BAND, BOR, BXOR perform a logical operation on two "continuous" expressions. The constants are ON, OFF, 1 and 0.

A **DMOD** segment has several differences with a **BMOD**. First, a **DMOD** can only contain discrete assignment statements. **DMOD** statements are executed selectively if any variable in that statement is triggered. Thus, a **DMOD** segment with n assignment statements is just convenient shorthand for n **BMOD** segments with one statement each. Table 4 lists the statements in the modeling language.

Statements	**Explanation**
<C-name> = <C-expression>;	Continuous variable assignment
<D-name> [(<time-delay>)] = <D-expression>;	B/D Boolean assignment with an optional delay value (.05s to 10s). A <time-delay> is a comma-separated list of two constants. The first constant specifies the time delay for an OFF to ON transition. The second constant specifies the ON to OFF delay.
CALL ASRT(<C-name-1>,<C-name-2>, <constant>,2);	The <constant> is the number of array elements. Store the permutation vector for a sorted <name-2> in <name-1> without changing <name-2>.
CALL DLM(<D-name>, <C-name>, <C-expression-1>, <C-expression-2> [, { IN/OUT }] [, <time-delay>]);	Set a discrete model name to indicate that a continuous model name is IN or OUTside of specified limits. The <time-delay> is applied as in discrete assignment.
CALL FLOW (<n-constant>, <m-constant>, .G, .HP, .P, .Q, .A,.B,.RA,.RB,.R);	Calculates pressures for interior nodes and flows for a fluid network. n is the total number of nodes and m is the number of interior (unknown) nodes. G (nx29) contains the G numbers. Any desired valve action should be included (see LFS) in the equations that load the non-zero (active) slots in G. HP (nx29) is initialized with head pressures. P (n) is loaded with the known pressures in the last n-m slots. Q (nx29) contains the results of the output flow calculations. The last five arrays (mx29, m, nx29, nx29, and m) are for intermediate results.
CALL INTGRL (<C-name>, <C-expression>, DELTA=<C-expression>, LOW=<C-expression>, HI=<C-expression>);	Any assignment to <C-name> when an interval is inactive will schedule the segment containing the integral for the next cycle. The second argument calculates the integration rate in units/second. It is multiplied by the interval since the last **CMOD** execution to derive the next increment for <C-name>. If the increment is less than DELTA, the interval is deactivated.

	The value of <C-name> is clamped within LOW and HIGH.
CALL PROP (<C-expression>, <constant>, [<C-name>,<constant>]^5);	The propagation delay statement creates a circular buffer of size <constant> to hold (value,timestamp) pairs. Up to five (name,interval) arguments may be specified. When each interval expires, the corresponding <C-name> is set to the value with the timestamp closest to now-interval.
CALL UPDATE (<C-name>, <C-expression-1>,<C-expression-2>);	This statement implements temporal iteration as a substitute for a **for**-loop. The statement iterates at the interval specified for its **CMOD**. The first expression is assigned to <C-name> at every iteration until **abs**(newC-oldC) is less than or equal to the second expression.
<int> CONTINUE;	Defines a label number, which must be unique.
DECODE(<name>,<C-expression-1>, <C-expression-2>);	Sets <name> to the value of bit <C-expression-1> in the value of <C-expression-2>. Bit zero is the 2^0 position.
DIF '[' <D-expression> ']' <statement>	Execute <statement> if <D-expression> true.
GO TO <int>;	Transfer control to label <int>. Loops are not allowed.

Table 4. Segment Statement Syntax

Names (or identifiers) must be capitalized alphanumeric and are exported from the model by default. Prepending an "XX" to a name defines segment-local variables. Sub-system names can be accessed with a qualified reference. A colon (:) is used as the delimiter.

The arithmetic operators are (* / + - and ** for exponentiation). The relational operators are: (.LT. .GT. .EQ. .NE. .LE. .GE.). There is a library of math functions as well as two continuous-expression functions: CLAMP and LFS. CLAMP takes three arguments and returns (1st<2nd)?2nd:(1st<=3rd)?1st:3rd. The LFS function uses its D-expression first argument to select a return value from its second (ON) and third (OFF) arguments, which are C-expressions.

BUILD SUB-SYSTEM PROCEDURE TEST LANGUAGE

Testing ground operations for launches was an important activity, even before the existence of SGOS. The early Shuttle ground-test language was called GOAL, Ground Operations Aerospace Language. This is the language that was, and still is, used by firing-room personnel to code their support procedures.

When SGOS was implemented, one of the requirements was to develop a model-test interface that was similar to GOAL. Thus, other than model and variable names, the syntax for the test language, which encodes MCPs, is unique. For example, GOAL supports dimension tags (Volts, Meters) on constants. Table 5 lists example statements to illustrate some of the features of the test language.

Statements	Explanation
APPLY ANALOG 8. Volts TO <UPLNFR01>;	Allows an analog stimuli to be sent to the model under test.
WAIT 5 SEC OR UNTIL <T23> .GT. 5. GAL;	Delay the test procedure no longer than the specified interval or until the test condition is true.
VERIFY <VSICHK> IS ON WITHIN 3 MIN ELSE PRINT TEXT(FLIP SW1 WHEN GREEN LIGHT) AND GOTO S 20;	Similar to WAIT but a failure clause can be added. Compound statements are supported using the AND connector.
EVERY 5 SEC CONCURRENTLY PERFORM PROGRAM(FOOBAR);	Similar to a fork/exec in UNIX. In this case, the action would occur every 5 seconds.
FAIL <TY73> TO 3.7 PSI FOR 2. SEC;	Freeze the value of a continuous variable for 2 seconds then resume normal calculations.

Table 5. Example Test Language Statements

BUILD SUB-SYSTEM PORTING THE SimAPI INTERFACE

The Sim Application Programming Interface (API) is a collection of C .h files that can be used to communicate with the SGOS executive (via its SimMaster interface), to perform simple operations on models (start or kill), and to stimulate and monitor changes in model variables under program control. The TCP socket interface is used to implement the functions. As a result, multiple clients can execute on a shared computer or on any other computer connected to the Internet. Since the SimMaster and RSI sub-systems use shared memory as an inter-process communication mechanism, the SimAPI implementation also uses shared memory. The benefit is that all clients on the same computer will "see" only a single copy of the model variables that are being monitored.

As stated earlier, the SimAPI library was written for Sun Solaris and then ported to Linux, Apple OS-X and Windows. The ports became progressively more difficult from Linux to Apple to Windows. In order to minimize the use of conditional compilation (#ifdef) preprocessor directives, OS functions with different semantics (or that did not exist) were renamed to create a virtual machine (VM) interface for the library. For example, the **recv** function from socket.h was renamed to **socket_recv** so that different system semantics could be isolated. Table 6 lists some of the problem areas supported by VM functions for each platform.

Linux	**Apple OS-X**	**Microsoft Windows**
Missing high-res time	Different high-res time API	Different high-res time API
Conversion to Big Endian for net data transmission		Conversion to Big Endian for net data transmission
	Structure packing different from all other systems	
Missing streams (fattach)	Missing streams (fattach)	Missing streams (fattach)
	No semaphores	Different semaphore API
	Different shared memory functions	Different shared memory functions
		Different socket semantics
		Missing fork, gettimeofday
		Missing SIGPOLL
		Missing nanosleep
		Missing getopt, getpagesize
		Missing UNIX types (pid_t)

Table 6. System Porting Difficulties

The two most difficult conversion problems on Windows were the handle semantics for sockets and the absence of a **fork()** system call. SimAPI requires socket handles (fds) to be small integers. The solution was to "stub" the socket calls with routines that allocated fds in the range 3-31. The missing **fork()** (luckily only used once) was replaced by isolating the child process' code in a procedure that accessed a copy of all global variables through a **struct** argument.

The solution achieves the desired notational opaqueness. On all systems, the parent must initialize the **struct**. The initialization is redundant in Linux and OS-X since under normal circumstances the child would have a copy of the global variables. Whereas in Windows, the **fork()** stub creates a thread to execute the child process's code.

BUILD SUB-SYSTEM PROGRAMMING WITH THE SimAPI INTERFACE

The SimAPI library was used to implement the KLASS (Kennedy Launch Academy Simulation System) project. The application was created by writing a GUI front-end to a variety of simplified launch-countdown models and displaying compelling graphics of Shuttle and ground components as the countdown progresses. Table 7 lists the methods in the interface together with a brief description of each.

The primary design requirement for the library was that the SimMaster could never be blocked, which in turn means that clients cannot be blocked. Since all client-server communication is implemented with sockets, this dictated that clients and the server use a socket **select** for I/O. Even though TCP socket output is asynchronous, if the output fills the connection's window size, the sender will block. Further once clients register an interest in a model's variables with the SimMaster, it can be continuously generating changes. A second design consideration was to coalesce the network connections for clients of the same model on the same host in order to minimize I/O by the SimMaster.

SOCKET **ConnectToSimMaster**(serverIP, port)	Open a client connection to serverIP/port. Send the hostname/username to authenticate and wait for response. Return the socket handle.
DisconnectFromSimMaster(socket)	Close the socket handle, which will propagate an error return to SimMaster.
{SOCKET, int, ModelTable[]} **GetRunningModels**(serverIP, port, shmID)	Call ConnectToSimMaster to get a socket handle. Send a command to get count and descriptors for all running models if shmID==-1; otherwise just model shmID.
int **PrintModelTableEntry**(modelTableEntry)	Print **shmID name loadTime owner health termPort**
ModelTableEntry **LoadModel**(smSD, directory, name, parameters)	Send SimMaster a load/directory/name/parameters command. The response is a model descriptor.
int **KillModel**(smSD, shmID)	Send SimMaster a kill/shmID command message.
{tSD, Tracker} **ConnectToModel**(serverIP, port, shmID)	Invoke GetRunningModels to get a SimMaster socket and the model descriptor. If the Tracker thread/process does not exist, create a Tracker segment named serverIP-port-shmID, fork the Tracker process, and wait for it to initialize. Finally, open a client connection to localhost/tracker-port and return socket and Tracker descriptor to caller.
int **DisconnectFromModel**(tSD, tracker)	Close shared memory, sockets and tracker.
int **PrintTracker**(tracker)	Print the model-table entry associated with the tracker. Print **processID trackerPort useCount nameCount**(^{n}) Print {name index C|D I|O|-- [INIT] [FAIL] value}^{n}
{value, initialized, failed, C|D} **GetVariable**(tracker, variableName)	Lookup name in the tracker's name table. Loads and stores are encoded as index-value pairs, not names.
WaitHandle **OpenChangePoll**(modelTableEntry, count, names[])	Call ConnectToModel twice. By convention, the first socket handle is for data, the second handle is for commands. For all names, send a register command to the Tracker command port.
int **CloseChangePoll**(waitHandle)	Deregister names, close sockets, close tracker, deallocate waitHandle.
{count, changeIndex[], changeValue[]} **ChangePoll**(waitHandle, timeout)	Use the socket **select** function to wait for a message or a timeout. If changes are received, return the list.
int **SendCommand**(modelTableEntry, command, name, value)	Call ConnectToModel twice. Send Tracker the command. **set** <name> <double value>; **fail** <name> <double value>; **reset fail** <name>; **reset all**; **stop**; **step**; **resume** <double value>;

Table 7. Sim Application Programming Interface

For these reasons (and a few others), the Tracker concept was evolved. There is one Tracker process (or thread) per model per host. Each tracker opens one port to the SimMaster and one connection port to itself. The socket **select** method (with a timeout) is used to query both original Tracker connections in addition to new connections generated by **ConnectToModel**. In the API description, all client-server communication is via Trackers, except for **LoadModel** and **KillModel**.

SUMMARY

The SGOS executive and its subsystems have been an integral component of the Shuttle Launch Safety Program for almost thirty years. It is usable (via the network) by over 2000 NASA employees at the Kennedy Space Center and 11,000 contractors. SGOS supports over 800 models comprised of several hundred thousand lines of code and over 1,000 model-control procedures. Yet neither language has a **for** loop!! NASA hopes that the KLASS project will be used to educate and excite the next generation of scientists and engineers in the areas critical to the new NASA Exploration Mission.

REFERENCES

C.T. Lostroscio and G.S. Estes. 'KSC real-time simulation interface (RSI)', in KSC Research and Technology Annual Report, NASA Technical Memorandum 211179, 122-123, 2002.

'Shuttle ground operations simulator (SGOS) user guide, build sub-system', NASA KSC Document #84K08140, Revision 4, Feb. 2004.