MULTIMISSION HIGH SPEED SPACECRAFT SIMULATION FOR THE GALILEO AND CASSINI MISSIONS

Alan Morrissett
Kirk Reinholtz
John Zipse
Gerald Crichton
Jet Propulsion Laboratory
Pasadena, California

Abstract

A simulation system has been developed which is capable of bit level simulation of spacecraft data systems. Object oriented techniques and an embedded interpreted language have been employed to produce a highly configurable tool for control and viewing of spacecraft states. Parallel processing computers have been used for running simulations to achieve execution performance of up to ten times real time, which allows for effective utilization of the simulator in testing spacecraft command sequences before they are committed to operation. Elements of simulations can be reused as-is in the construction of new simulators.

1. Background

Spacecraft exploration of the solar system requires communication over long distances, with communication delays on the order of hours. Commands sent to a spacecraft must meet a high standard of accuracy, as an error in a command can not be rapidly detected and corrected and could potentially mean loss of the spacecraft. Construction and verification of command sequences has traditionally been a labor intensive and painstaking procedure.

Since the advent of microprocessors in spacecraft data systems, it has been considered too complex to perform a rapid simulation of the data operations of a spacecraft bit for bit in order to test and validate planned event sequences to be carried out in space. However, advances in computing technology have provided machines with previously unimagined capabilities which have made this job feasible.

We have built a prototype multimission spacecraft simulator which has been used for the Galileo and Cassini missions to demonstrate the feasibility of bit level simulation, and are currently building a production version for each mission. The Galileo spacecraft, which incorporates a data system consisting of six RCA 1802 microprocessors running at 200 KHz and two spacecraft data buses running at 400 KHz, is now on its way to Jupiter for a two year orbital tour beginning in December 1995. The Cassini spacecraft is currently being designed and will have a data system consisting of four 1 MIP 1750a microprocessors connected by 1 MHz 1553b data buses. It will perform a similar mission at Saturn.

A previously published paper[1] covered work which established the feasibility of bit-level simulation of spacecraft systems. This paper discusses a new object-oriented architecture and implementation for bit-level simulation which gives a multi-mission modeling capability, flexible scheduling of element execution, the ability to establish hoc views of simulation components, and significantly higher levels of adaptability and maintainability.

2. Approach

The spacecraft data systems to be modeled typically consist of a number of CPUs interfaced via multiple high speed buses and often several marl peripherals. After considering several approaches, it was determined that this system was best simulated on a parallel architecture host. The spacecraft data system typically consists of about 10 independent processes being executed simultaneously. This is optimally simulated using 10 or more host processors (at least one host processor per simulated spacecraft processor.) The host processors each simulate for an appropriate time slice and then synchronize to assure that they stay in relative time step. Spacecraft processor communication is done via buses. The simulation treats buses as separate processes which transfer data between memories and processors. As long as the simulator time step between synchronization is kept small enough (typically 100 Hz), the simulated processes, including bus transactions, can occur in any time order within the time step and maintain fidelity.

The multiple processes on the host computer require a high level of interaction. Several methods were

Copyright c 1993 by the American Institute of Aeronautics and Astronautics, Inc. No copyright is asserted in the United States under Title 17, U.S. Code. The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental purposes. All other rights

explored to maximize performance. Initial prototypes were developed on distributed memory systems interfaced through serial buses. Data transfer rates and latency were critical and eventually it was found that a shared memory host architecture provided the optimal performance. An 8 processor Silicon Graphics computer with 40 MHz R3000 processors and 2 stage cache interfaces to shared memory was used for prototypes. There it was possible to run a simulation 10 times faster than real time and synchronize processes at 75 Hz in simulated spacecraft time (~750 Hz in real time.) This provided the performance needed to build a tool adequate to support spacecraft operations. Future implementations will use shared memory machines with faster CPUs, cache, and memory access. This will be needed to provide the same level of support for newer, faster spacecraft.

3. Simulator Architecture

3.1 Simulation Elements

Our implementation consists of objects which model the large grain hardware components of a spacecraft, e.g. 1750a processors or 1553b bus controllers. It was observed that these components exhibited a natural packaging, with a small number of well defined interfaces. For example, a processor usually contains at least one read/write interface to an address space. The processor is modeled as an object which includes an embedded address interface object, and performs reads or writes to addresses via this embedded object. The address interface object is connected at run time with another address interface object, which forwards a read or write operation into its owner element for action.

Simulation elements usually contain more than one interface, e.g. processors contain interrupt and other service lines. A simulation element can contain any number of interface objects, including multiple copies of a single kind of interface. An element is used during a simulation run by creating a new instance of its type and then hooking up each of its embedded interfaces to a complementary interface.

We chose C++ as the implementation language for simulation elements, because of its run time efficiency as well as for its support for object-oriented programming. A processor element, for example, may emulate a hardware instruction set and the code which implements this must be sufficiently fast to meet the performance objectives of the simulator. This code for the current 1802 and 1750a processor implementations consists of a fetch, decode, and execute loop implemented with C++ switch statements and reads/writes via the embedded interface objects.

Our system does not force the implementation of any element to be a bit-wise emulation. Because the elements are decoupled from each other and communicate only via their embedded interface objects, they can be done in any way that meets localized objectives for speed and functionality. There can be several element implementations for a single processing element in the spacecraft, and the choice of which to use can be made at run time. For example, we have two alternative models for the processor which operates the Attitude, Articulation, and Control Subsystem (AACS) in the Galileo simulator. Because this microprocessor (ATAC 16) is faster than the 1802 processors in the Galileo Command and Data Subsystem (CDS), a full bit level emulation of its operation limits the speed of the entire simulator. In some cases the simpler, functional simulation is sufficient and allows for a faster simulation run for the entire spacecraft system.

3.2 Connecting Simulation Elements

In order to construct a working simulator, a number of simulation elements must be created and each of their embedded interfaces connected to a complementary interface. These connections, which we refer to as "splices", are typed and the interfaces to be connected on each end must be of compatible type. A simulation element has no internal knowledge of what it's connected to (other than internal assumptions about what an interrupt line means, for example), so it is possible to connect any compatible object to its end. One use of this is to interpose a monitoring object between two elements that would normally be spliced directly together, which can perform extra services such as statistical analysis or graphical display. A use intended in the future is to provide a "tinker toy" like construction of spacecraft Systems from existing pieces, in order to research new spacecraft designs more cheaply.

To accommodate memory mapping, address ranges for processing elements must be able to be split among several recipients. Our most recent design allows this capability via specialized interface objects which examine addresses before routing them appropriately. This kind of embedded interface is implemented via C++ inheritance, so that non-memory mapped elements which use the simpler base class implementation do not have to incur the cost of memory examination and routing. The address ranges of memory mapped connections are established by parameters to splicing commands issued at start-up. A related problem is the mapping of addresses to different addresses as they are being forwarded to the recipient object. This is accomplished in a similar way, with an inherited interface object which can perform the necessary transform.

3.3 Command and Control of Elements

Our design philosophy for simulation elements was that they should contain a simple and minimal set of C++ member functions (methods) to implement their functionality, but contain a Turing equivalent language interpreter which would allow construction of arbitrarily complicated compound operations based on the atomic member functions. Wc felt (hat it would be bard to anticipate all of the functionality that might be desired from a simulation element, but that by providing a language which could access an element's basic operations wc could provide any feature needed without extensive redevelopment.

The language chosen for the embedded interpreters was Tcl ("tool command language"), a freely distributed embeddable interpretive language developed at the University of California at Berkeley[2]. This is a small but powerful and extendable language which interfaces wc]] with C/C++. It also has a superset graphical language, Tk, which allows for the easy creation of graphical user interfaces which can exchange Tcl commands with an application (see example in figure 1.)

All elements share a small set of common functions, which include an 'execute' command to cause Simulation for a specified length of time and a 'set' command for setting and retrieving the value of an element's internal variables. An element's class definition contains a declaration of all internal variables that will be available to Tcl via the 'set' command, which can include processor registers, memories, and other metadata which an element contains such as lists of bus transactions, etc. The declaration of commands and variables for Tcl occur in an element's C++ constructor, so objects which inherit from a base class object receive the same Tcl functionality without having to redeclare it, Individual element classes also extend their command set to provide functions particular to their operation.

3.4 Simulator construction

The simulator operates in an interpreted fashion in constructing a spacecraft mode] to perform a particular simulation. The main() function for this program is very simple:

```
main(int argc, char * argv[])
(
        Executive exe();
        exe.startup(argc -1, argv + 1);
        return(0);
)
```

A single object is initially constructed, the "executive", which is responsible for interpreting further commands to create objects, splice them together, and execute simulation activities. The executive tries to find a name of a start-up file from its argv[] arguments, from a file called ".fastsim" in the current directory, or from the standard input, in (hat order. 1((hen waits in a service loop for further commands from a user interface element, if one has been created, or from the standard input if one has not.

The interpreted nature of system construction allows for a great deal of flexibility. It is possible to create a standard simulation from a batch file, or to create a graphical user interface (GUI) which offers choices for configuration options to allow a user to bring up a customized simulator.

Commands to the executive are Tcl commands which arc special to its class: 'new' for invoking element constructors, 'splice' for connecting element embedded interfaces together, etc. Another command which is special to the executive is 'send', which allows a Tcl command to be sent to any object that has been created. This allows script driven ad hoc queries and computation to be performed at any time, giving a large amount of adaptability and flexibility to the simulation system. 'Scud' commands arc passed over a command splice to a receiving object's interpreter, and the response to the command read back and despatched appropriately. A command splice to objects is automatically created by the executive during execution of the 'new' command.

2.5 Execution Scheduling

It is possible to operate a simulation by sending Tcl 'execute' commands Via the executive's 'send' commands, but in practice (his is uot fast enough for production simulation runs. Instead, a 'scheduler' element is created which can form execution splices to simulation elements, WhiCh add minimal overhead in invoking element execution member functions.

The scheduler is responsible for enforcing rendezvous points during execution. During parallel execution of a simulation, a number of machine cycles for different elements may be executed simultaneously and asynchronously, and if any interaction between elements occurs at this time it will most likely not reflect the synchroneity that occurs on the real hardware. Rendezvous provide synchronization and a merging of threads so that element interactions can occur reliably. Rendezvous times can be sc(to any value, from as little as onc simulation clock cycle 10 any arbitrarily higher value. A onc clock cycle rendezvous ensures perfect fidelity to the spacecraft hardware, but the overhead from this number of rendezvous greatly reduces performance. Setting larger rendezvous values allows us to achieve higher

performance, but there is an upper limit to the size of a rendezvous before the simulation fails 10 mirror the actual hardware. In the Galileo spacecraft, most transactions between spacecraft components occur at a "real time interrupt" (I-WI) which occurs every 1/1 5 second, but a number of sub-RTI transactions also occur. We have found that rendezvousing at 1/5 of an RTI gives the largest time slice that will work, but which still allows us to attain a ten times real time simulation.

Rendezvous points are enforced by the scheduler even when execution commands are sent which would otherwise cause a rendezvous point to be overrun, For example, a user might choose m single step the simulator through an interesting portion, then ask it to jump ahead one RTI. The scheduler issues commands to elements to execute for a particular number of clock cycles, and reads the return value of the call to find the actual number of cycles executed (an element may execute less than the number of requested cycles if it cannot finish an atomic operation,) It then computes the correct number of cycles m send to each element on the next command based on the running total of executed time for each element and the length of time to the next rendezvous.

The scheduler runs from a master scheduling list, which can be composed of any mixture of elements or other scheduling sub-lists, Each list is defined to be serial or parallel, which specifies whether or not parallel computation can be performed on the list elements. Some parts of a simulation must occur serially in order to operate correctly; for example, the delivery of an RTI signal must occur after all computation in a time slice has completed, and so the element delivering this signal is in a serial list following other element execution. Most of the simulation time, however, is spent in parallel execution of spacecraft elements.

Our simulation parallelism is based on symmetric multiprocessing, so that all threads have access to the same address space. Because all of the executing spacecraft elements are implemented as C++ objects they operate on their own encapsulated data, and use no data locking or mutexes to avoid conflict. A faulty scheduling list has the potential m cause non-deterministic behavior if it schedules elements in parallel which call through other elements executing concurrently, but our scheduling lists to date have been simple and we haven't had any problem with collision. I lowever, we plan to install switchable mutexes in the future so that scheduling list integrity can be checked at will.

A simulation may be run on a single processor or multiple processor machine with no change in code or configuration. On a a single processor machine, a parallel scheduling list simply executes sequentially.

3.6 Simulation Monitors

'Monitors' are simulation elements which are not part of a spacecraft model, and which serve as companion elements that watch and report on the state of a spacecraft element. Like all elements, they contain a Tcl interpreter which can provide any desired functionality by loading the proper script, Monitors can be created on the fly, and can be added to or deleted from scheduling lists as necessary. A monitor is normally spliced to its companion's Tcl command interpreter, which provides access to all of a spacecraft's internal state via 'set' and other commands. In some cases, a higher performance interface is needed for examining memory locations in the companion object and a memory splice is made to provide this.

One usage of monitors is for debugging; the monitor is set to watch for any computable state specified by the logic in its script. On detection of an anomalous state, it can send a warning message or a request to halt back to the executive. It can also be used to send periodic values back to the executive which can be forwarded to a user interface for display in any desired way, e.g. strip chart, dial, text, etc.

A capability that we intend to implement is to checkpoint the state of spacecraft elements to disk at regular intervals. Upon detection of an anomalous condition, the executive will restore the most recently checkpointed slate vector and then single step forward until the anomaly is again encountered. This will allow monitors to do sampling at longer intervals while still providing for a capability to pinpoint an exact anomaly state, without significantly degrading simulator performance.

4. Simulator Performance

A simulation speed of ten times real time has been achieved with the Galileo simulator, using a Silicon Graphics 41)/480, an eight node multiprocessor machine using MIPS R3000 40 MI Iz cpus. We have achieved up to six times real time using a single processor Silicon Graphics Indigo with a later generation processor, the MIPS R4000 50 MI Iz cpu. We believe that with additional analysis and performance work that we will be able to achieve even better performance from the multiprocessor simulation.

A significant part of the overhead cost of multiprocessing is due to rendezvous costs. We experimented with a number of different ways of coding rendezvous and determined that rhe fastest was to have threads spin on a shared lock variable in user shared address space. This spin-lock was designed to cause a minimum of bus transactions, which would otherwise. contend for bus

resources with other processes still performing useful simulation work, This scheme keeps barrier costs within the order of microseconds, and was found to be faster than any of the system provided rendezvous methods. With faster hardware for rendezvous support wc may be able to increase performance in this area.

Preliminary analysis indicates that time slices for any particular spacecraft element tend to vary substantially, depending on what code execution is being simulated during the time slice. During execution of parallel scheduling lists, execution times of the threads vary in a stochastic way, so tha some of the elements arc always waiting for the curren ly slowest one to finish. Wc feel that with dynamic scheduling we nay be able to achieve belter performance by performing useful work with the CPU cycles current wasted in spin-locks while waiting for the slowest element to complete execution.

< Future Directions

There arc a number of enhancements that wc can make within our existing design, and a number that wc would like to evolve the design reward in the future. Wc plan to make a number of changes in the coming year to make the Galileo simulator much closer to its ideal, which will mainly involve writing of additional Tcl and Tk scripts within the existing framework. Possibilities include making the processor register and memory windows writable as well as readable in order to allow easy modification of spacecraft stale, alarm displays that illustrate what condition has been triggered by monitors, production of specialized GUIs for particular subviews of the system such as a scan platform or AACS, active displays which show the amount of idle time on a processor or tbc amount and direction of bus traffic, etc.

Although wc have already demonstrated the substantial performance advantage of executing tbc simulator on shared memory parallel processor platforms, wc believe that wc haven't yet demonstrated the full potential of the host hardware, In particular, the simulator currently spends over half of the total available CPU cycles in spin-locks. The use of dynamic allocation of simulation elements to physical processors may improve the performance in this area,

A goal for the future is to make a tool for designing spacecraft processing systems, in which the designer could select components from menus and configure them graphically. Embedded interfaces could be drawn within the graphical representation of an object, and 'wiring' of the connections done via drag and drop bet ween interfaces.

A possible architecture for this is onc in which the graphical view or views of an element arc encapsulated within i(s object definition, rather than specified by procedural scripts as wc do now. It is a future point of investigation at to whether Tcl/Tk is tbc best way to accomplish this or *not.*

Summary

Wc have delivered an initial production version of the Galileo simulator, and arc currently implementing a production version of a Cassini simulator. Our current implementation already provides speed, flexibility, visibility, and case of usc advantages over the existing hardwarc simulator for Galileo. Further enhancements to the user interface will give visibility and control of systems that flight software developers bad not previously imagined.

Further use for other missions from Voyager to MESUR is also being considered. Wc feel that the flexibility of this implementation will allow us to produce simulations of other spacecraft that give high capability at a relatively low cost.

Further development of the simulator into a design tool may make possible tbc development of new architectures for spacecraft data systems that would not otherwise be tried because of expense and risk. If so, the technology of spacecraft design could be evolved more rapidly.

References

[1 John E. Zipse et al, *A Multicomputer Simulation of the Galileo Spacecraft Command and Data Subsystem,* Proceedings of the Sixth Distributed Memory Computing Conference, IEEE Computer Society, 1991.

[2 John K. Osterhout, *An Introduction to Tcl and Tk,* Addison-Wesley, 1993.

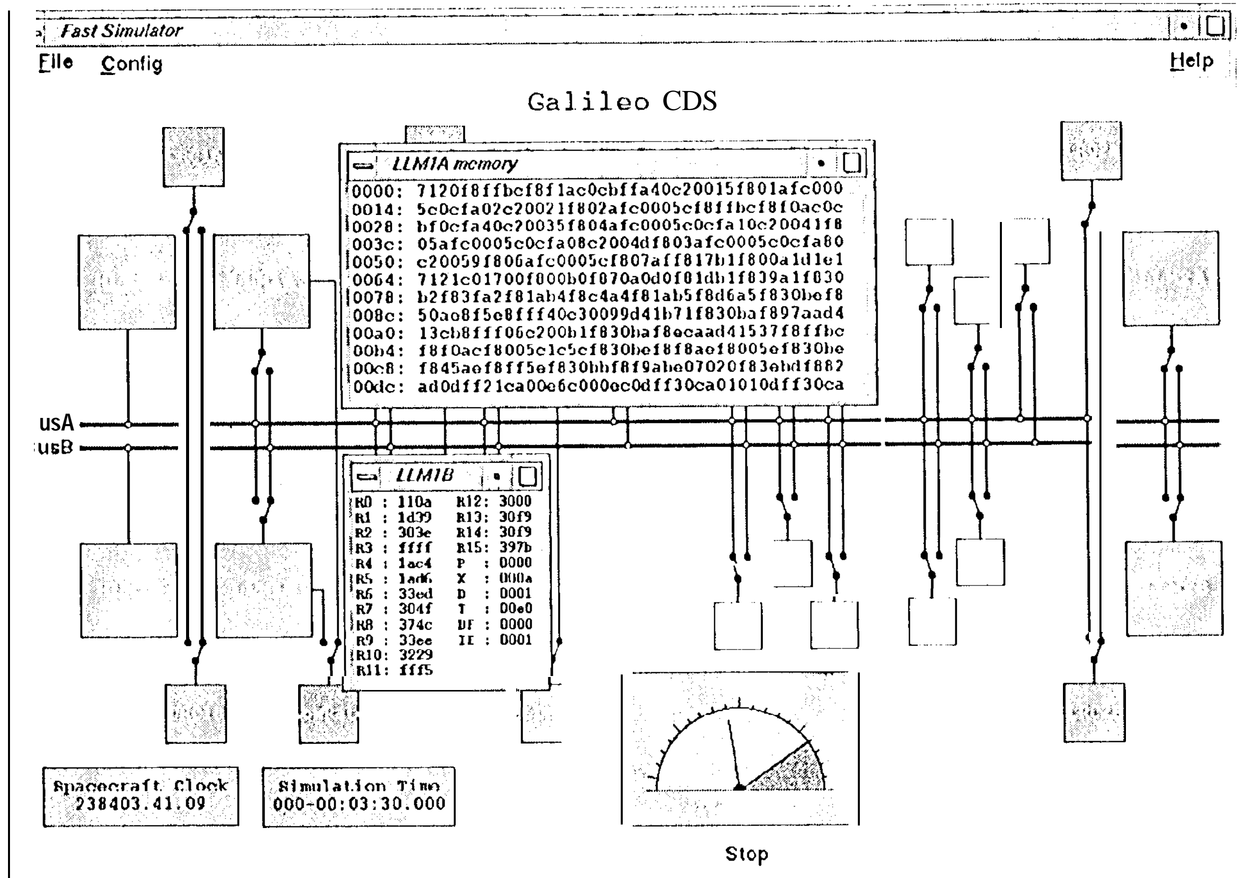

Figure 1. Tk user interface with a memory and a register display active.