

THE ZIPSIM SERIES OF HIGH PERFORMANCE, HIGH FIDELITY SPACECRAFT SIMULATORS

W.K. Reinholz
W.J. Robison, III
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109

Abstract

This paper describes the architecture and applications of the ZIPSIM series of spacecraft simulators. ZIPSIM is the foundation upon which spacecraft simulators are built at either the register-transfer level or lower level of fidelity. A simulator for a particular spacecraft is built by combining mission-specific components with standard ZIPSIM components. ZIPSIM provides standardized capabilities including utilization of multi-processor hosts, common components (e.g. schedulers, graphical and command-line interfaces, and data displays), reusable models (e.g. processors such as MIL-STD-1750A), uplink and telemetry interfaces related to Mission Operations at JPL, standards for runtime establishment and operation of component interconnections, and a standard command language (Tcl/Tk) for configuring and operating a simulator. An automatic code generator based on the CLIPS expert system is used to generate some of the support code.

Introduction

Interplanetary spacecraft are expensive. A command error can cause the loss of hours or days of science data, or even damage or destroy the spacecraft. It is therefore important to validate commands before they are sent to the spacecraft to insure that they will not have undesired effects.

Static analysis is often used to validate commands. It is done either by manual inspection of the commands, or with automated tools that check the commands against a set of constraints (e.g. "don't point optical instruments at the Sun"). In either case, the analysis can only include known constraints. Due to the complexity of the flight software and hardware, there may be latent constraints that are not known until a series of commands violates one of them, perhaps during execution on the spacecraft. Further, there can be violations that static analysis cannot detect, as commands typically have computing constructs (variables, arithmetic, conditional branching) that can cause violations that can only be discovered by analyzing all possible paths through the command sequence.

Hardware testbeds can be (and are) used to validate commands. There are, however, important limitations to such an approach. The testbed hardware is probably not any faster than the flight hardware, so it is virtually impossible maintain a continuous test execution ahead of the spacecraft execution. It is very expensive to provide the same level of visibility in testbed hardware as is easily implemented in a simulator, so it's more difficult to check for

constraint violations. Finally, testbed hardware is expensive, so there tends to be few units available, leading to off-shift operations, availability problems, and other expensive and time consuming difficulties. In spite of all of these issues, though, only testbed hardware can detect violations of otherwise unknown hardware-imposed constraints, and so should be used to validate mission-critical commands and any other commands that are judged to be likely to violate such constraints.

High fidelity spacecraft simulation as described in this paper involves the execution of the actual binary flight software loads on CPU emulators, interacting with the remainder of the simulator just as it would with flight hardware. As such, both compute-based command errors and the violation of latent constraints can be detected¹.

The remainder of this paper describes the architecture and design of ZIPSIM Version 7, with a focus on features new to Version 7. [MORRI 93] describes the reasons for the undertaking of the project and the architecture of Version 6 of the simulator. Proof-of-concept research from which this project originated is described in [ZIPSE 91].

Overview of the ZIPSIM architecture

The architecture of ZIPSIM is based upon components.² There is a runtime library of components (e.g. CPU models, bus models, other spacecraft hardware models, simulation schedulers, data viewers, uplink and downlink interfaces) each of which has zero or more interfaces (all standardized) and can be connected at runtime to any other model that supports a compatible interface.

This architecture provides a number of advantages, the most important two being strongly-typed runtime component interconnections (called "Splices") and strong conceptual partitioning. The former insures that incorrect interconnections are rejected. The latter is a consequence of the fact that a components interaction with the system is

1. There are limits, though. Hardware properties not known to the model builder will not be present, and so latent constraints based on those properties will not be detected.

2. We differentiate this from "object-oriented" because the model interfaces do not imply a taxonomy - The component simply presents standard interfaces that can be connected to other, compatible interfaces.

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

fully specified by its interfaces, and thus the semantics of a component can be understood without having any understanding beyond the interfaces.

There are two parts to the design of a simulator for a given project. Any components not already in the library must be designed, built, and added to the library. The component interconnections must also be specified. A simulator is not fully designed until both of these have been done.

Most components contain a Tcl [OSTER 93] interpreter. Tcl is a freely available embeddable language and concrete interpreter implementation developed at the University of California, Berkeley. We augmented Tcl to make it multi-thread ("MT") safe and to add some features, in particular a C++ class wrapper and a fast "remote procedure call" that can execute commands remotely on named interpreters (many of the components within ZIPSIM contain named interpreters, which are used to manipulate the component) and return the result of the execution to the local interpreter. The ZIPSIM Tcl subsystem is used for many things, including model state examination and alteration, establishment of model interconnections, and model state monitoring and display (using Tk, a toolkit based on Tcl which can quickly create Graphical User Interfaces). As a general rule we use Tcl unless either performance or robustness concerns dictate the use of splices.

ZIPSIM Version 7 includes an automatic code generator ("ACG") based on the NASA CLIPS [RILEY 91] expert system. It's used to generate most of the code that can be deduced from concise specifications. This both makes it unnecessary for the developer to derive the needed code (or even to understand how to do so), and it makes it easy to alter the implementation because though a concept may appear in a hundred places, it need be altered in only a single place: the ACG.

Applications

ZIPSIM is used to create high fidelity spacecraft simulators. It is being used to create simulators for the Cassini and Voyager spacecraft. Version 6 was used to create a simulator for the Galileo project.

There are a number of applications of the ZIPSIM simulators. Some of the more important are:

- Unit-testing of flight software components prior to testing on flight hardware. This only requires the relevant processor models. ZIPSIM provides more visibility than the typical hardware testbed, and is a more friendly environment within which to perform such testing. It also greatly increases the probability that the module will quickly pass its tests on the flight hardware, which reduces costs as operation of the hardware testbed is typically quite expensive and availability quite limited.

- Integration testing of flight software prior to testing on flight hardware. This requires a complete simulator. Visibility and cost reduction is again the motivation for testing on the simulator prior to testing on flight hardware.
- Command sequence testing prior to testing the sequence on flight hardware. There are typically very few flight hardware testbeds available for sequence testing, and their operation is expensive. Testing sequences on ZIPSIM prior to testing them on hardware decreases costs by virtue of reduced use of hardware testbeds, and increases reliability by allowing testing that could not be scheduled on testbed hardware.
- Anomaly investigation. Should an error occur on the spacecraft, a ZIPSIM simulator can be used to investigate the error. The error may be a consequence of a hardware attribute that is not modeled by the simulator and thus may not occur on the simulator. However, spacecraft downtime is VERY expensive and so the ability to perform a number of parallel investigations on several instances of ZIPSIM in conjunction with hardware-based testing is very attractive. Further, ZIPSIM can provide more visibility than can be had from the testbed hardware, thus making it a more productive investigation tool.
- Generation of engineering telemetry predicts. ZIPSIM can be used to generate spacecraft engineering data predicts, which are compared with actual engineering data to detect operational anomalies.

Standard model features

Models within ZIPSIM are required to have certain features depending on the attributes of the model. For example, all state of the model must be visible, and if the model is a CPU then it must support certain debugging features such as instruction tracing and breakpoints.

The interfaces to the standard features are also standardized as much as possible. This increases the opportunity to reuse components, particularly those that do state display and monitoring, that operate upon certain types of models. For example, simulated RAM is always accessed in the same way. Therefore, the same display component can be used to monitor RAM contents regardless of the type of model that contains the RAM.

The following list shows the significant standard features of a model:

- Scheduler interface.
- Splice interfaces.
- Return list of all splice types and namespaces for given class of components, and for individual compo-

ment objects.

- Return list of all state items.
- Read and write all state items.
- Standard read and write formats for primitive state items such as integers, floats, boolean, yes/no, on/off, and arrays of these things.
- Set/clear breakpoint at specified CPU address and read/write/execute access.
- Enable/disable instruction trace by:
 - Address range.
 - Instruction opcode.
 - Instruction class (e.g. branching).

Tcl and Tk

Tcl ("Tool Command Language") is a small and simple language designed and implemented by John Osterhout at the University of California, Berkeley. Tcl has been released into the public domain. It is designed specifically for use as an imbedded control language.

Tk is a collection of Tcl and C code that together forms a window-oriented language that provides a simple and powerful method of quickly creating Graphical User Interface applications.

Tcl is not MT-safe. ZIPSIM is MT, so we put a C++ class wrapper around Tcl, called SimTcl, to provide a uniform mechanism for mutexing Tcl¹. This also makes Tcl fit better into ZIPSIM by virtue of its becoming a C++ class, rather than a collection of C functions. The remainder of this paper presents our use of Tcl in terms of SimTcl.

The following describes the major features of SimTcl.

- RPC - Each SimTcl object has a name. There is a remote procedure call that will execute a command on a named SimTcl, returning all results (error code, return value, and error trace) of the execution of the remote procedure. The called SimTcl has access to the name of the calling SimTcl.
- Inheritance and Virtual Functions - Each SimTcl object has zero or more SimTcl parent objects. When there is an attempt to execute a command that is not

1. We considered making Tcl MT-safe by modifying the Tcl source code, but upon analysis it became clear that doing so reliably would be quite a bit of work, for little benefit. (we don't use Tcl for time-critical jobs, so the performance gain possible by using fine-grain locking doesn't appear to cover the cost of implementing it.) We could go to fine-grain locking later, if necessary, with no change to the interface.

defined in the local SimTcl, SimTcl tries to execute the command in the parent SimTcl objects until either the command is executed or all parents have been tried. This allows inheritance. The search is recursive, so if the parent doesn't implement the command, SimTcl will try the parent(s) of the parent, and so on. Proper use of this mechanism can be enforced by use of lists of commands that may be called by child SimTcl object. RPC ensures that the identity of the originating SimTcl is available to the procedure that is finally executed, so it can gain access to namespace of the originating SimTcl.

- SimTcl has some C++ stream operations for evaluating a string, returning the results of the evaluation, and reading and writing Tcl variables. The intent is to minimize the string manipulation that a user must perform by allowing the natural use of C++ streams to conjure operations on the SimTcl. Otherwise the user will have to do C++ I/O stream operations to create command strings, send the string to the SimTcl, then delete various remnants as necessary, all error-prone operations that tend to cause memory leaks and use of stale pointers.
- SimTcl has operations for mutexing an individual interpreter. They are used to insure that non-atomic operation sequences (e.g. evaluating a procedure that sets a variable then reads the resulting variable) can be guaranteed to perform without disruption. Otherwise another thread could come through the SimTcl and alter the results of the sequence.
- SimTcl comes with a coding standard for writing command procedures and Tcl code.

Most SimTcl objects within ZIPSIM lie within a simple inheritance tree: There is a static interpreter at the top of the tree. Below that is a static interpreter for each model class. Finally, each model has an interpreter that is created when the model object is created. A procedure call executes on the interpreter making the call, if it exists there, then goes to the interpreter shared by all objects of that type, and if not there, it goes to the base interpreter. Procedures that may be used by all objects are placed into the base interpreter. Procedures common to all objects of a given class are put in the class static interpreter. Each object can have unique procedures if necessary.

Splices: Plug-and-play component interconnection

This section describes splices: the method used to dynamically connect together components in ZIPSIM.

A splice is a connection between two components. The components connected by the splice are called the master and the slave. The master component originates calls for service by the slave. The slave services the request. The

splice is processed on the process thread of the master unless the developer writes explicit code to make it otherwise. A given component can be both a master and a slave, or in fact several masters and several slaves.

A splice master or slave must implement the semantics of a given splice type, but the implementation of the required semantics is the responsibility of the component developer. For example, the same RAM model may provide 8 bit and 16 bit splices, perhaps with several types of relationships between 8 bit and 16 bit addresses. The mapping from those splices to the internal storage of the RAM is strictly up to the designer of the RAM model. The intent is that the splice capability provides nothing more than an interface contract between the two splice endpoints. How either implements the contract is an issue internal to the individual designers.

Splices as well as components are established during program execution, rather than established during program compilation. This provides a tremendous amount of flexibility, because ZIPSIM can be configured for a particular simulation run by editing a runtime configuration file, rather than re-compiling the simulator. Splices and components can be manipulated during an execution, for example, to insert a monitor into a splice between two models to trace their interactions.

Splices are implemented using an object called a splicebuf. Essentially, all masters of a given splice type have pointers at the base class splicebuf for the given splice type, and for each slave there is a derived splicebuf that contains a pointer at the slave. A virtual function in the splicebuf forwards the operation from the base class, which all masters can point at, to derived classes, each of which can point at one of the slave classes. The implementation does not use any casting or other escapes from the C++ typing system.

The following pidgen C++ code fragment outlines the structure of splices. T is the type of the splice. M is a master for splice type T, and S is a slave types for splice type T. f() is a typical member function for the given splice type. Many tactical details are omitted.

```
class T{
    int slave_chan;
    virtual int f(int a)=0;
};

class M{
    T* T_p[];
    int f(int a,int master_chan){
        return T_p[master_chan]->f(a);
    };
};
```

```
class S{
    int f(int a,int slave_chan);
};

class TS:public T{
    S* S_p;
    virtual int f(int a){
        return S_p->f(a,slave_chan);
    };
};
```

Splice performance is in some cases critical. Though the implementation shown above does not impose a large overhead, there is a cost to the pointer indirections and the virtual function.

ZIPSIM uses two standard techniques for increasing the performance of splices where necessary: The master may be allowed to access the slave directly; and the splice may implement iterating operations (e.g. bulk data moves, rather than word-at-a-time explicit iteration by the master). The latter is especially important, as it maintains the abstraction of the splice which is broken when the master is allowed direct access to the slave. Yet it can reduce the overhead of the splice to essentially zero because the overhead of the splice is amortized over many data moves. Our experience so far shows that memory-oriented operations and bus operations are the only splice types that push the performance of the splice implementation, and both tend to be bulk-data oriented. The semantics of the bulk operations are written such that they are exactly identical to explicit master iteration, so the slave is free to implement the bulk operation directly or decompose it into an iteration. The developer of the master thus need not consider the type of the slave - bulk operations can be used where they make sense to the master, implemented where they make sense to the slave, and higher performance will then be automatically supplied where possible.

The automatic code generator generates most of the code necessary to implement splices, and in some cases (where no data transforms are applied by the splicebuf) it can generate all necessary code.

Automatic code generation

Much of the code that must be written to support the construction of a new component or a new splice type can be derived from a few attributes of the component or splice type. ZIPSIM Version 6 required that the developer write the code manually, assisted by a (rather involved) checklist and a few macros. We found several potential areas for improvement:

- Component developers needed skills beyond the domain of the component being developed.
- Developers repeatedly had to move up the learning

curve of the process of component or splice creation, only to move down after the job was done.

- Maintenance was complicated because changes that impact the code derived via the checklist required the understanding and modification of each instance of code built via the checklist.
- Each instance of the code written via the checklist introduced another opportunity for coding errors.

All of these areas are addressed by our Automatic Code Generator (ACG). The ACG takes as input a concise specification of the component being built, including its splice types and associated namespaces, and the type of the component. The type of the model (e.g. model, cpu_model) in turn implies certain interfaces, which are automatically included.

The ACG potentially emits a number of files, including a class specification for the component, a file of member function definitions related to individual interfaces, a file containing member function definitions related to splice establishment, and files containing other sets of member function definitions.

The ACG is implemented using the CLIPS [RILEY91] expert system. CLIPS is a rule-based expert system. It is based on an engine that matches a set of facts with the left side of "if" statements, and if the condition is satisfied, the right side of the "if" statement is executed, which typically results in the addition, deletion, or modification of facts.

The ACG uses two phases to generate output files. The first phase converts the input specification into CLIPS facts, using only the contents of the specification file. This phase performs no deduction. The second phase applies a number of rules to the facts, which generates a number of other facts and may also cause facts from other specifications to be loaded. For example, when a component specifies a particular interface, the facts for that interface are read and duplicated (perhaps with some string substitution) into the facts for the component. Finally, after all facts are generated, a series of rules are applied to the facts that cause the output files to be generated. The ACG typically fires a few hundred rules and typically executes in two to five seconds on a 50 MHz Sparc platform.

There was some initial concern about the robustness of the use of expert system technology to generate code, as it is difficult to demonstrate the correctness of rule-based expert systems, even with substantial testing. However, our experience indicates that errors in the rules or facts causes the emission of code that will not compile and link, and so the error does not propagate into executable code.

User Interface

ZIPSIM is designed to run with multiple user inter-

faces(UI). ZIPSIM Version 6 had two user interfaces: a character-based command line interface (CLI) and a graphical user interface (GUI). The CLI is well-suited for batch mode execution or execution on machines without graphics capability. The GUI is intended for use during interactive-intensive execution, much like a debugger.

The intent in ZIPSIM Version 7 is to make the UIs completely interchangeable. In principle, anything you can do with one UI can be done with any other. This implies a standard way of communicating between the UIs and the simulation itself.

In ZIPSIM, the UIs are implemented as a separate process which communicates with the simulation process through pipes, sockets, the Tk "send" procedure, or some other inter-process communication mechanism. If the UI is a GUI, the process is based on Tk. Otherwise, the UI is based on Tcl.

The UI can act in a passive or active mode in the collection of information to display. In its passive mode, a component sends data to the UI. The UI accepts the data and updates the display, if necessary. In its active mode, the simulator queries the component's Tcl interpreter for data. The component's interpreter returns the requested data to the UI's interpreter which will update the screen. The active mode requires the UI's screen update procedure to call all components to gather the information needed to update the screen.

Both methods have advantages. The passive method is better for less frequently changing values, or for catching events which are important to see. The UI process could become a performance bottle-neck if a value changes very rapidly and informs the UI every time that it changes. This is the method of choice if the state change must be seen.

The active method is better for rapidly changing values and for user-initiated queries of component information. A particular value may change so rapidly that it changes faster than the eye can detect. This will slow down the simulation, trying to provide information that no one can use. The active method is also better for information requested by the user from the UI. If asked, the component must respond with the current value of the information in question.

All UIs, whether GUI or CLI must recognize the same commands from the simulation process, and all UIs must request information from the simulation components in the same way. This suggests at least that the ACG can generate stubs and possibly full procedures for accepting and requesting data from the components. The ACG knows the characteristics of each component and can therefore add the code necessary to display and modify that component's state. For example, if a component has an internal

RAM space, a memory viewer would be automatically added to the UIs, and a procedure would be automatically added to the model to read and write to its RAM. Each UI would automatically know how to access the component's RAM space, and the RAM space would automatically have procedures to read and write to the RAM space.

Multi-processor capability

ZIPSIM uses a flexible task scheduling system which allows multiple processors to work on separate parts of the simulation in parallel. At runtime, a configuration file creates the components and splices the components together as specified for the simulation. Scheduler components are created which control in which order and on which processor the components are executed. The scheduler is given a list of components to execute. It is also told to execute these components in serial or in parallel. The schedulers can be told to execute other schedulers. It is possible to build arbitrary hierarchical trees of schedulers which can call other components, including other schedulers which execute their components in either serial or parallel.

In ZIPSIM Version 6 when running on multiple processors, the head scheduler executes components in serial. One of those components is a scheduler which in turn executes a number of components in parallel. Those components are schedulers which execute a number of components in serial. This general mechanism will be used in ZIPSIM Version 7.

Future work

There are a number of capabilities that could be added to ZIPSIM to decrease the time and cost required to build a simulator. This section outlines some of the possible extensions to ZIPSIM that would provide quicker model development.

- Use the ACG to generate Tcl-only components with procedures to access all master splices, procedures that are called for all accesses to slave splices, and a procedure to advance time in the component. This "generic" model could be used to quickly generate prototype models, as the semantics of the model would be written in Tcl, rather than C++. Such a model would be easily modified, even by the user of the simulator.
- Give components access to CLIPS, perhaps under a Tcl-only component. Some models can be specified fairly easily in CLIPS-like notation. We would like to build a model that contains CLIPS, so that such models could be rapidly prototyped using the powerful expert-system notation of CLIPS.
- Design a state-machine notation that can be converted into executable input to ZIPSIM (e.g. Tcl, C++, or

CLIPS). Some models can be specified fairly easily as state machines. We would like to build a notation (perhaps the extension to the notation used in SCR advocated in [FAULK89]) and a toolset that compiles the notation directly into an implementation. This would once again enable rapid development of models compatible with the notation, and reduce the likelihood of software errors.

Summary

ZIPSIM is a framework for building high performance spacecraft simulators. It has a number of interesting features, including a component-based architecture, extensive use of Tcl/Tk, and automatic code generation.

Biography

Mr. Kirk Reinholtz is the Task Manager and Cognizant Engineer for the ZIPSIM Multimission Spacecraft Simulation project Jet Propulsion Laboratory. For the past five years at the Jet Propulsion Laboratory, Mr. Reinholtz interests have included the design of spacecraft flight software, formal methods, and simulation. He received his BS in Computer Science from the California State Polytechnic University, Pomona.

References

- [FAULK89] S.R. Faulk, "State Determination in Hard-Embedded Systems", Ph.D Thesis, U. of Carolina at Chapel Hill, 1989.
- MORRI 93] A. Morrisset et al, "Multimission High Speed Spacecraft Simulation for the Galileo and Cassini Missions", AIAA Computing in Aerospace 9, 1993.
- [OSTER 93] J.K. Osterhout, *An Introduction to Tcl and Tk*, Addison-Wesley, 1993.
- [RILEY 91] G. Riley, "CLIPS: An Expert System Building Tool", *Proceedings of the Technology 2001 Conference*, NASA, 1991.
- [ZIPSE 91] J.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.