Applying Simulation to the Development of Spacecraft Flight Software

Kirk Reinholtz
Jet Propulsion Laboratory*
California Institute of Technology
4800 Oak Grove Drive MS 303-310
Pasadena, CA 91109 USA
Kirk.Reinholtz@jpl.nasa.gov

Abstract— We describe how the application of simulation and emulation to the lifecycle of spacecraft software can improve quality and aid in schedule compression and cost reduction. We define various forms of simulation and emulation, describe their various uses over the software development lifecycle, outline our experiences with regards to what can go wrong and right, and discuss how one might insert the technology into a project lifecycle.

Table of Contents

1 Introduction

The development of spacecraft emulators[1] at JPL has been motivated by several needs that regularly arise during the development of spacecraft flight software.

Insufficient hardware breadboards During most project lifecycles there is a greater demand for breadboards than can be satisfied. Breadboard spacecraft are expensive, so this is perhaps inevitable. Simulators are often used in place of breadboards, to reduce the pressure on the project to supply breadboards.

Repeatability Simulators can be operated in a mode that supports fully synchronous and deterministic operation, so that two runs, given the same initial state and input timing, will yield exactly the same state trajectory and final state. This is extremely useful for "gold standard" testing, because no fuzzy compare is required: The final state is either exactly right, or its wrong. In contrast to this, hardware breadboards tend to not be fully repeatable, so a fuzzy compare is required. This generally either means time-consuming work, or the possibility of missed failures.

Performance Simulators often run faster than real-time, which is very useful when there is a large amount of testing or development to be done. Interestingly, this trend has continued even as spacecraft development schedules have compressed. Shorter design-to-launch cycles removes some of the advantage Moores Law confers upon the simulators, but technologies have been developed [2] that improved simulator efficiency enough to stay ahead of the flight hardware.

Visibility Spacecraft simulators have much better visibility than breadboards or flight hardware, and so they are generally a more productive debugging environment. Even esoteric cross-bus race conditions are easily detected on a simulator, where they would be difficult to identify on a hardware breadboard.

In this paper we describe a number of applications of software emulation (of spacecraft hardware) and simulation (of the universe visible to the spacecraft) over the lifecycle of a spacecraft: Codesign[3] to aid in making hardware/software trade decisions; Development of flight software on spacecraft software emulators to improve productivity and reduce development station costs; System test by executing the flight software on the software spacecraft emulator with a simulated universe around it; Operations verification by operating the simulated spacecraft using the primary operations tools and process; and end-to-end testing by inserting the simulated spacecraft far downstream in the operations tool chain so as to exercise virtually all of the operations tool chain.

Daniel Goldin, the NASA Administrator, recently stated[4] that simulation will become ever more important to achieving the NASA goals of reduced costs and greatly improved mission capability. One aspect of this vision has been demonstrated: It is practical to construct spacecraft simulations that can be operated just as real spacecraft as operated, so it is reasonable to exercise mission scenarios, or even complete missions, before

*The work described was performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration. U.S. Government work not protected by U.S. Copyright.

hardware is built and launched. In this way highly detailed design trades can be inexpensively performed.

Figure 1 outlines a typical spacecraft emulator and its simulated environment. It has several major components:

Link Simulator When doing end-to-end simulations, a link simulator is used to provide propagation delays and signal degradation. In other situations it is simply a conduit of uplink and downlink data.

Spacecraft This element simulates the spacecraft avionics.

Integrator Most spacecraft dynamics is simulated by integrating forces over time to compute current acceleration, velocity, and position. The forces may be a result of actuator activity (i.e. the firing of a thruster), or it may be celestial in origin.

Ephemeris The ephemeris is a database that gives the location of celestial objects at a given time. It is used to determine the location of celestial objects in order to generate scenes for sensors (i.e. starfields), and to compute gravitational forces on the spacecraft.

Scene Generator Scene generators are used to generate views of the universe for input to sensors. For our purposes, a scene is not restricted to a scene like an image of a planet: it may be a scene of any field or particle.

Sensor Transfer Function This function takes a scene that is represented in a canonical first-principal form, and converts it to what will be sensed by the sensor. For example, the scene generator generates a "perfect" star field image (point sources with the proper intensity and perhaps spectra), which is then blurred, smeared, and filtered to generate the image that will be seen by the starfield camera.

2 What is simulation?

For the purposes of this paper, a simulator is a software program that acts like certain portions of the spacecraft hardware and the physics that acts upon the spacecraft during the mission. Such a simulator is used to develop and test spacecraft software components and systems without requiring access to real hardware or real physics. In the limiting case one end up with a simulator that obeys the *prime directive*, although many useful products can be built that do not have this property:

All possible softwares shall behave identically upon the simulator and the real hardware – With extreme performance[1].

[1]There are a couple of inherent limitations: There isn't real science data, and characteristics of the hardware that aren't known *a priori* aren't simulated.

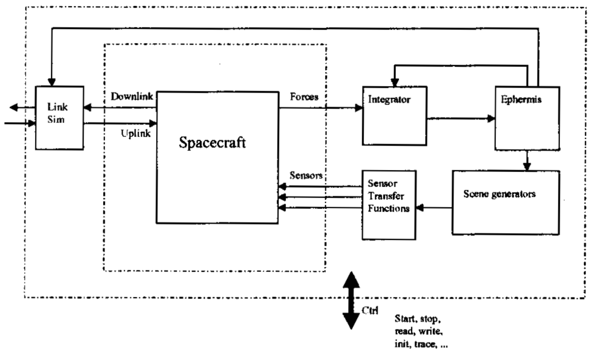

Figure 1: Overview of system

This prime directive came about in the early days of our spacecraft emulator, when it was used primarily as a software development platform. It is obvious that in that case, we needed to provide high fidelity in the sense offered by the prime directive. However, even today, where spacecraft simulators are routinely used as building blocks in larger systems and processes unrelated to the development of flight software, we still need exactly the same type of fidelity. This is because as a practical matter, if the flight software behaves the same, the spacecraft as observed from the outside behaves the same, which is exactly what you want for system-level uses of emulation.

3 Types of Simulation

Bit-level

A "bit-level"[2] simulator models hardware behavior with bit-level fidelity at the hardware/software interface [3], but uses unconstrained techniques to simulate the hardware behind the interface. Specifically, there is probably no gate-level modeling behind the interface, because that would be superfluous to the prime directive and would greatly reduce the performance of the simulator.

Certain spacecraft behavior that can be sensed by the software are dictated by physics and/or the environment of the spacecraft, and so, by the prime directive, these things are also simulated.

Instrument sensor values are not accurately simulated,

[2]History drives this terminology: These are behavorial models of the spacecraft hardware.

[3]The interfaces of modeled hardware must act as much like the real hardware as is possible. There is always a temptation to simplify the interfaces for the sake of easing the development of the simulator or of the flight software. This inevitably leads to a difficulties when the software is finally integrated with the real hardware. Because of this, we *STRONGLY* advise that no shortcuts are taken in the area of interface fidelity, unless a management plan is in place that provides a timely and controlled migration from lower fidelity to higher fidelity.

because that is generally impossible. We have found it reasonable to synthesize plausible values, correlated across sensors and closed with actuators, where necessary.

The extreme fidelity of this simulator uniquely positions it as a cost-effective substitute for an engineering model or breadboard. In particular, whenever the focus is on the software, or its interactions with the documented behavior of the hardware[4], you should consider the use of simulation. This is because you are using the EM or breadboard as a physical embodiment of the hardware specification as a substrate for execution of the software: you do not really need the literal hardware.

Native Compilation

There are choices besides "bit-level" simulation described above. A defining attribute of a bit-level simulation is the use of an Instruction-Level Simulator (ILS) of the spacecraft CPUs that execute exactly the same binaries as does the real CPU. This provides the ultimate in fidelity, but it comes at a cost: it takes time to develop the ILS, and performance potential is limited.

Under some circumstances you can make slight modifications to the flight software source[5] and then directly compile the flight software for execution on the host processor. The compiled flight software is linked directly into the simulator (using all models other than the ILS as-is), for subsequent execution. Performance is thus greatly improved, but fidelity is lost and you are no longer obeying the "fly what you test, test what you fly" rule.

Virtual-machine interpreter

Spacecraft developed at JPL have often had a virtual-machine architecture: an instruction set suitable for commanding a spacecraft is developed, and the spacecraft is commanded in that notation via an interpreter onboard the spacecraft. If the instruction set were to be adequately defined, and little commanding or operations were performed via "back doors", then it may be feasible to simulate the spacecraft at the virtual machine level. Such a simulator would have extremely good performance. There has been no spacecraft instruction set developed to date at JPL that would be suitable for this level of simulation (at least not without a lot of caveats), but perhaps the time is right to develop such an architecture as the foundation of future spacecraft flight software. That architecture has performed well for many missions.

Future spacecraft are going to be goal-oriented, where a goal is a predicate applied to state that must be true over some period of time. Flight code[6] determines how to cause that goal to be achieved. It is not immediately obvious how this might be simulated without simulating the execution of all of the code that has anything to do with converting a goal into imperatives. If, however, this advanced architecture were to be implemented on top of a virtual machine that acts as the interface between goals and imperatives, it may be practical to simulate the actions of the spacecraft at the virtual-machine level, which should let us execute far faster than realtime. Unfortunately this will probably impose restrictions on how goals are achieved. It is not yet clear if this will be a fatal issue.

Scaffolding

Advanced software development often involves prototyping efforts, and flight software is no exception. There has in fact been a trend towards increased prototyping at JPL.

The prototyping efforts are usually focussed on small portions of the flight software, and generally need only an abstract simulation of those parts of the system that are needed for scaffolding purposes. Indeed, a detailed simulator would be a handicap in this application. Effort would be expended developing interfaces to the detailed hardware models that could otherwise be spent doing the prototyping itself.

Prototyping efforts are usually short-term efforts where agility of exploration is important. This means that simulators for this purpose must be especially easy to rapidly reconfigure or extend, so that the scaffolding does not get in the way of the prototyping work. At the same time, the scaffolding should reflect enough of the flight system that the prototype does not ignore inconvenient details that will later have significant impact.

The greatest challenge of this type of simulation is to find the right level of abstraction. If it's too abstract, the transition to reality will invalidate the results of the prototyping. If it's not abstract enough, time is wasted building detailed device drivers and doing other interfacing work not germane to the primary objective of the prototyping effort.

Dynamics

It is sometimes the case, especially during early software design phases, that one only needs a detailed simulation of the spacecraft dynamics: the particular avionics is not relevant. Work at this level is routinely performed during development of the attitude control software, where the focus is more on control laws than on hardware interface details.

JPL has developed a sophisticated dynamics modeling capability called DARTS[5], which can be used standalone as mentioned above, or can be used as part of a larger system. Most of the simulation systems mentioned in this paper use DARTS for dynamics simulation.

[4]Contrast this with the case where your focus is on the behavior of the hardware artifact itself, for example when searching for a gate latchup or timing glitch.

[5]Most importantly, the simulator needs to know when the FSW has read or written reactive memory, which means that e.g. device drivers must use a function call for all device i/o, rather than directly manipulating memory locations. We also need hooks for timing control, and of course certain FSW functions like dynamic code uplinking won't work properly.

[6]Perhaps very sophisticated code, involving the use of planning and scheduling algorithms, model-based reasoning, and other advanced algorithms.

4 Java

If the flight software is written in Java, A number of good things occur, at least from the simulation viewpoint. First, one needn't construct a processor model for the simulator, because the Java bytecode interpreter performs that function. Second, in the early phases of the project one could simulate device i/o at outer driver level (Java class specification), rather than at hardware command-status register detail. If the interfaces are well specified, it is reasonable to expect that the migration to flight hardware would go much more smoothly than has previously been experienced.

We are assuming that there is no operating system under the Java interpreter. If there is then things get more difficult because it may not be possible to implement the VM without modeling a significant portion of the underlying operating system. We are also assuming that at least in the early stages of the project, bytecode execution is used rather than just-in-time compilation or direct compilation, either of which would necessitate the development of a model of the avionics processor, so that the resulting code could be properly executed.

5 Applications of Simulation

Coverification

Coverification[6] is the act of verifying the behavior of a system composed of hardware and software before you have real hardware, by executing the software against models (e.g. Verilog or VHDL) of the hardware.

Coverification is generally performed by running small fragments of the software against a gate-level simulation of the hardware. Performance factors often limit the scope of the coverification to a limited amount of code: almost certainly nothing near the whole system.

The spacecraft simulator can be used in a hybrid configuration to do high-performance coverification[7]. By performing most of the execution on the simulator models, and as little as possible on the vendor coverification platform, performance is only limited by the amount of hardware being coverified in a particular execution: All other hardware operates at simulator speeds.

Hardware/software codesign

Related to coverification is codesign[3]: The design space of the system is explored before committing to silicon. This is done by constructing a simulator containing behavorial models of the hardware design, and running prototype software on the simulator. Opportunities for improving the distribution of functionality are identified between hardware and software before committing to silicon.

The spacecraft simulator obviously supports codesign because it can be used to model hardware that doesn't yet exist. It can also directly interpret statecharts[7] as models of hardware components. Statecharts are well understood by many designers, and are easily and quickly modified as the design evolves, so the design iteration cycle is radically shortened: It may be practical to do several hardware/software design trades in a single day.

[7]This has not been demonstrated.

This concept isn't new to the spacecraft avionics design process: The Voyager COMSIM simulator was used almost thirty years ago to perform codesign. However, it is now much easier and faster.

Software development

Simulation-centric software development is the act of developing embedded software using a software simulation of the target for day to day development activities, rather than a physical instance of the target. There are many reasons why this is good (described in some detail later in this paper), but basically it's because of improved control over and visibility into operation of the software. It's also a cheaper and faster way to develop software.

By employing the improved control and visibility, one can increase productivity in many ways; automated testing, detailed performance measurements, improved debugging, and fancy breakpoints, for example. And since it's all software, the whole simulation can be checkpointed, so work can be suspended, then later resumed from exactly the checkpointed state [8].

Commercial tools (like debuggers and performance measurement tools) can usually be used in a simulation environment: They typically come with a customizable "connect this the breadboard somehow" component, and perhaps a bit of software to install on the target hardware. It is not difficult to use the adaptation methods to attach the product to the simulator, rather than real hardware. By the prime directive, the tool will work[9].

It is still a good idea to regularly exercise the development team on the target hardware, so as to minimize backend and integration surprises. Management decisions must be made with respect to development performance *vs* risk reduction.

Why it's good

A simulation-centric development environment is a developers dream. It directly provides the many advantages inherent of software over hardware, reducing costs, risk, and schedule.

Control The simulator can be single-stepped a μs at a time; provides breakpoints triggered by expressions of arbitrary complexity; can checkpoint *everything*; and can be dramatically reconfigured in a matter of

[8]We often do this in the Cassini High Speed Simulator(HSS): The simulator is initialized and run to some interesting state, then checkpointed. Multiple experiments are then executed, starting from that checkpoint. This can save many hours of debugging and testing.

[9]This isn't nearly the stretch it might at first seem. Such tools are designed specifically to adapt to a wide range of development environments.

moments. No more tedious days with a logic analyzer trying to find an interrupt problem or race condition.

Visibility The full state of the system, at any instant, is easily examined, checkpointed, logged, or modified. Formal analysis for deadlock potential or race conditions is feasible. Cache patterns can be analyzed in detail. Abuse of the hardware (e.g. use of uninitialized variable, or failure to save all state during an interrupt, or setting both of the "never set this bit and that bit at the same time" bits) can be detected online and *in situ.*

Replication The marginal cost of additional simulators is essentially zero, so every developer can own a full development system. No more graveyard and weekend shifts on the testbed as schedule crunch builds.

Extensibility The behavior of the simulator is easily extended, especially with the use of modern scripting subsystems[10] that give the sophisticated power user access to the full feature set of the tool.

Early availability A simulator generally has a shorter development cycle than a hardware testbed of comparable fidelity, and so can be available earlier in the project life-cycle. Developers can begin work on a full-fidelity platform, rather than cobbling together *ad-hoc* scaffolding for their individual areas of concern. Hardware/software design trades can be explored before hardware is taped out.

Automated control Hands-off and repeatable testing is encouraged because it's easy and natural to control the execution of the simulator via other software. Nightly integration testing is reasonable. Testing staff and schedule is greatly reduced.

Modularity A simulator can be constructed in a highly modular fashion, so that more urgently needed models are built first and available to the users before the entire simulator has been built and validated.

No Hardware No fried boards. No emergency Fed-X for replacement parts. No days of debugging looking for a weak component or intermittent connection. Developers stay focused on the job of developing.

Adoption impediments

Most spacecraft and instrument flight software development still takes place in hardware-centric development environments. We have observed a number of reasons over the years, which can be generalized to the usual problems of bringing an unfamiliar technology into a process.

Abstraction can be a tough sell. The essence of simulation is *abstraction*: Hardware and environmental details are carefully reduced to the essentials relevant to the project phase at hand. A reasonable manager considering the adoption of simulation technologies will ask how one can know if the abstractions were performed correctly. This topic is addressed in another section of this paper. We can't argue that simulation should always be trusted. There are critical areas where it isn't worth the risk. However, one must trade schedule compression against risk, and consider that schedule compression in one area leads to more time for analysis in another, which could improve the overall project risk/cost position. And finally, as advocated in detail elsewhere in this paper, we believe it prudent to do regular ground-truth runs on the target hardware, just to make sure nothing slipped through.

[10] rogriguez:1995

Adoption of new tool is difficult. A simulator is a new tool to learn. If simulation is to be successful, each developer must be competent in the operation of the tool. Developers, especially the seasoned ones, won't acquire this knowledge and skill without good reason: they've seen many a tool come and go, and rightfully don't want to invest effort into yet another dead end. To mitigate, the simulator should be designed to allow low-cost buy-in – It must be available at the right time, easy to install, and the payoff must be obvious to the user that has knowledge (which must be easily acquired!) of just a few basic commands. We also recommend that some sim developers join the development virtual team, so expertise is always at hand and is clearly focused on the concerns of the developers.

Simulation must be introduced early. There is a natural time for the introduction of simulation into a project, and if simulators aren't provided in harmony with that timing, the developers will do something else. Once that something else is done[11], it introduces a project-wide habit that will not converge to a single product and will probably last the life of the effort[12].

Front-loading a schedule hurts. For whatever reason, software development schedules and budgets are almost always back-loaded. This makes it difficult to fund front-loaded work like the development of simulators and other productivity tools. Unfortunately, simulators must be developed early and systematically (i.e. they front-load the schedule and budget) if they are to provide their maximum payoff.

"I've got testbeds." Virtually every project builds a few testbeds. It's reasonable for project management to ask why it should sponsor simulators in addition to the testbeds. By now it must be clear

[11] Usually either home-grown simulation or testbed-based development. It's not that the developers will do a bad job of these things: They may do exceptional work (In fact, several of the HSS principals were at one time just such developers). The problem is that you don't want *n ad hoc* simulator or environment development efforts within a project, because that'll always cost more time and money than one may anticipate, and probably won't be a managed line-item.

[12] We have seen disaster-class events force first-principal reengineering of development processes, which makes for open minds and a natural second chance for introduction of simulation, but you can't count on that happening.

that there are two general reasons for doing so: The marginal cost of simulator copies is near zero, so you needn't have developers sitting idle waiting for testbed time; and even if you have many testbeds, the simulator provides a better development environment, which tightens schedule and schedule variance. These lead to significant productivity improvements and thus schedule compression and cost reduction. Note that we *DO NOT* advocate simulation-only development. It isn't yet realistic to expect anything other than the real hardware to provide the ground truth. However, a sim is sufficient for many purposes, improves productivity, and much cheaper.

Operations verification

The purpose of a spacecraft is to return science data. Spacecraft cost amortized over mission lifetime can easily be in the neighborhood of US$100K per day, so anything that reduces the quality or quantity of data, even for a few minutes or by a few percent, is quite costly by this cost model.

Spacecraft on the drawing boards today will be rather robust in the face of potentially damaging commanding: They are being designed with onboard intelligence that will maintain the fundamental health of the spacecraft, and continue to work on mission goals, even in the face of commands that attempt to to otherwise. This is good, and will help prevent disastrous consequences of erroneous commanding. However, the simple existence of the erroneous command implies that something went wrong in the operations process. That in turn means that somebody probably isn't going to get the results they anticipated[13].

But no matter how that erroneous command is handled by the spacecraft, science return has to be reduced: The craft will generates a workaround, but it is unlikely that it'll come up with the original intent of the author of the command.

A spacecraft emulator can be used to validate commands before they are sent to the real spacecraft, as outlined in a paper devoted to this subject[8]. In this way one can confirm that the command causes the anticipated behavior without impacting the spacecraft should something go wrong.

Spacecraft are gaining in their ability to operate autonomously, which means that they are increasingly responsive to their environment. A prudent risk management plan must ask how "close" the spacecraft will come, during operations, to being unable to respond properly (this is another face to the "lost science return" model of failure). This question can in principle be answered by using a spacecraft simulator to explore the behavior of the spacecraft over a number of perturbations of the "nominal" scenario while gleaning proximity to failure information from the software internals during the many scenarios. This type of verification is discussed in detail in a previous paper[9]. Another approach, based upon analysis[10], is also possible. Each method has strengths and weaknesses, and both will probably be necessary.

[13]Hopefully an informative error message and minimally reduced science return, rather than a "blue screen" spacecraft.

A simulator is the best way to do do perturbation testing, for several reasons: The test will probably require visibility into flight software internals in order to infer the proximity to failure; and it will probably take a large number of executions to acquire sufficient information. The best way to do this is to run many experiments in parallel, and run each as fast as possible.

End-to-end test

An end-to-end simulation is easily performed by operating the simulated spacecraft as though it were the real spacecraft: real operations tools and processes are used as though a real spacecraft were being operated. Only the low-level uplink and downlink interfaces at the Deep Space Network (DSN) need special configuration. All other tools can be operated nominally. Given sufficiently clever scene generators, even science analysis tools could be exercised in this manner.

6 CASE STUDIES

Galileo

The Galileo mission launched in 1989 to study Jupiter. It consists of a large (5000 pound) spacecraft with ten instruments, and a probe with an additional six instruments. In April of 1991 it suffered an onboard hardware failure when the large high-gain antenna failed to deploy.

The antenna deployment failure necessitated switching the science data link to the much slower low-gain antenna. This, in turn, required a significant unscheduled upgrading of the flight software to use sophisticated data compression and data management techniques in order to achieve the science objectives of the mission.

Until the antenna failure, software development was for the most part performed on a single hardware breadboard of the Galileo spacecraft. Development was paced by this limited resource because there is only one, and it is expensive to operate it. The large software upgrade required to recover from the antenna failure was well outside of what could be accomplished using the testbed-based methods used until that time.

We developed[1] a complete software simulation of the Galileo spacecraft, with fidelity sufficient that the flight software binary code (exactly what's uploaded into the spacecraft RAM for execution) would execute exactly as it would on the spacecraft and with performance about *ten times faster than realtime*. We also built models of the various sensors and instruments with fidelity sufficient to maintain nominal readings so that the software would not enter fault handling conditions.

The simulators were used as software development targets, and for software testing. During some peak periods, there were over a dozen instances of the simulator executing tests in parallel, and a staff of dozens of engineers using them as development and test targets. The software was tested on the hardware testbed only after being developed and tested on the simulators.

Simulation directly enabled a much larger scope of changes to the flight software, and so a much greater recovery of science return than could have been possible using only hardware testbeds for development and testing.

Cassini

Cassini is another large mission, launched in 1997 and destined for Saturn.

In this case, spacecraft simulation is used primarily for purposes of verification and validation during operations: Commands that are to be sent to the spacecraft are first sent to the simulator, in order to confirm that the commands behave as expected and that there are no unexpected resource other other interactions. Certain critical commands are also confirmed on a hardware testbed, but as with Galileo, this is expensive and time-consuming.

The core simulation technology was the same as used for Galileo, but GUI interfaces and operations concepts required significant modification to suit this different application of the tool.

Voyager

The voyager mission consists of two spacecraft launched over twenty years ago (1997) and now outside of our solar system. A simulator had been built for Voyager. The simulator, called COMSIM, executes on a legacy mainframe that was scheduled for decommissioning. COMSIM was written in a mix of various FORTRAN dialects and assembly language, and was heavily dependent upon vendor-specific operating system characteristics.

The charter was to develop a replacement for COMSIM that would execute on modern workstations, so that the decommissioning could be performed on schedule without damaging Voyager operations. Three approaches were at first considered: (1) port it; (2) write an instruction level simulation (ILS) of the legacy system or a binary translator[11]; or (3) rewrite COMSIM using our newer simulator framework. Option (3) was selected, because (1) appeared to be impractical and (2) was likely to result in excessive performance degradation.

There were two major challenges. First, the COMSIM source code reflects many discoveries made over the years about the voyager hardware behavior as parts degraded and failed, so we couldn't derive specifications from requirements: we had to reverse-engineer the existing product, which was a lot of work. Second, COMSIM generates large reports that contain spacecraft and COMSIM data and are processed by various tools generated over the years. Most of the tools operate directly on printer-image text files and depend upon spacing, page length, numeric formats and many other syntactic details, all of which had to be exactly duplicated so that the new simulator would operate within the existing processes and tool chains.

7 History of bit-level simulation at JPL

Bit-level simulators have been in use at JPL for over thirty years. The first was part of the STARSYSTEM development environment that was created at JPL in the late 1960's as part of a program to develop fault-tolerant computers for space flight applications. The simulator was used to do design trades and as a software development target, since only one breadboard of the JPL-STAR computer system was at first built.

The STARSYSTEM, including the simulator, evolved into the development environment for the Viking spacecraft. A minimal simulation of the environment was added to the simulator at this time. This system was called VIKINGSYSTEM. The simulator was enhanced to support other processors onboard the Viking spacecraft, and came to be called COMSIM.

COMSIM was then modified for the Voyager project, and was recently replaced by the later generation of simulators described in this paper because its host platform is being decommissioned and porting was deemed more expensive than replacing it.

Finally, the simulators discussed in this paper carry the tradition of bit-level simulation to contemporary projects.

8 Verification of the Simulator

A significant issue that comes up when writing simulators is that of verification: How do you know that the simulator behaves identically to the hardware that it's supposed to be simulating? We've found several methods useful, outlined in this section.

1. **Gold-standard comparison** It is sometimes the case that there is an existing breadboard. A gold-standard test is performed by extracting detailed traces of memory, bus, and i/o register activity from executions of the hardware. The same tests are then executed on the simulator, and compared the traces. It is often necessary to deal with nondeterministic behavior of the hardware by extracting subtraces and comparing them individually. We have done work on using dynamic slicing for this purpose.

2. **Use the hardware verification suite** Flight hardware is extensively verified, and much of the verification is done via software that executes upon as well as verifies the avionics. Those same tests are used to validate the simulator.

3. **Verify against the specification** Sometimes there is no hardware verification suite, so tests must be derived from the specifications. Spacecraft hardware is generally well specified, so this is tedious but fairly reliable. It is usually fairly straightforward to write an extensive test suite.

It's interesting to note that most of the simulator verification tasks actually become more difficult if the models aren't full-fidelity. They won't match the flight hardware, so you can't do gold-standard testing; and similarly they'll fail most of the hardware acceptance tests. In each case it will take a large amount of error-prone manual effort to explain all of the differences without missing any errors in the differences. If the models are full-fidelity, any difference is bad and that's that.

9 Summary

Bit-level spacecraft simulation has been used over the full spacecraft lifecycle for decades at JPL. It has repeatedly been demonstrated to save money and reduce schedules.

References

[1] A. Morrissett, K. Reinholtz, J. Zipse, and G. Crichton. Multimission High Speed Spacecraft Simulation For The Galileo and Cassini Missions. In *AIAA Computing in Aerospace Conference 9th, San Diego, CA, October 19-21, 1993*, pages 544–549. American Institute of Aeronautics and Astronautics, October 1993.

[2] K. Reinholtz. High-performance software emulation of 1750A processor. In *Digital Avionics Systems Conference(DASC), 16th, Irvine, CA, Oct. 16-20, 1997*, pages 8.2–1–8.2–7. IEEE, October 1997.

[3] R. Ernst. Codesign of Embedded Systems: Status and Trends. *IEEE Design and Test of Computers*, 15(2):45–49, April 1998.

[4] Daniel S. Goldin, Samuel L. Venneri, and Ahmed K. Noor. Beyond Incremental Change. *Computer*, 31(10):31–39, October 1998.

[5] J. Biesiadecki, D. Henriquez, and A. Jain. A Reusable, Real-Time Spacecraft Dynamics Simulator. In *Digital Avionics Systems Conference(DASC), 16th, Irvine, CA, Oct. 16-20, 1997*. IEEE, October 1997.

[6] A.S. Berger. Co-verification handles more complex embedded systems (reprinted from electronic design, september 15, 1997). *ELECTRONIC DESIGN*, (S):9–+, 1998.

[7] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[8] K. Patel, K. Reinholtz, and W. Robison. High Speed Simulator - A simulator for all seasons. In *International Symposium on Space Mission Operations and Ground Data Systems - "Spaceops 96", 4th, Munich, Germany, Sept. 16-20, 1996*, pages 749–756. AIAA, September 1996.

[9] K. Reinholtz and K. Patel. Testing autonomous systems for deep space exploration. In *1998 IEEE Aerospace Conference, Aspen, CO, Mar. 21-28, 1998*, pages 283–288, March 1998.

[10] Michael Lowry and Daniel Dvorak. Analytic Verification of Flight Software. *IEEE Intelligent Systems*, 13(5):45–49, September 1998.

[11] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary Translation. *Communications of the ACM*, 36(2):69–81, 1993.

Kirk Reinholtz is a team lead at the California Institute of Technology, Jet Propulsion Laboratory. He was lead architect and developer of the flight software for the Mars Observer Camera, after which he helped form the team that now develops spacecraft simulators at JPL. He is currently developing a series of advanced simulators for an upcoming series of spacecraft. He has a BSCS from California State Polytechnic University, Pomona, and an MSCS from the University of Southern California.

P2009 Version 1.9 December 8, 1998