Lessons Learned in Creating Spacecraft Computer Systems: Implications for Using Ada[tm] for the Space Station

by

James E. Tomayko

Senior Computer Scientist
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213*

Abstract

Twenty-five years of spacecraft onboard computer development have resulted in a better understanding of the requirements for effective, efficient, and fault tolerant flight computer systems. Lessons from eight flight programs (Gemini, Apollo, Skylab, Shuttle, Mariner, Viking, Voyager, Galileo) and three research programs (Digital Fly-By-Wire, STAR and the Unified Data System) are useful in projecting the computer hardware configuration of the Space Station and the ways in which the Ada programming language will enhance the development of the necessary software. This paper reviews the evolution of hardware technology, fault protection methods, and software architectures used in space flight in order to provide insight into the pending development of such items for the Space Station.

1. Introduction

During the 25 years since the first flights of manned and unmanned spacecraft carrying onboard computers, the tasks assigned to the machines have grown in complexity and pervasiveness until now it is impossible to consider designing a spacecraft without including substantial computing power. As with any mission critical component, the reliability of computers has to be ensured. NASA's efforts to use computers onboard spacecraft resulted in the development of various methods of fault tolerance. Development of computer systems for unmanned and manned spacecraft have largely followed separate tracks. Systems onboard manned spacecraft used increasing numbers of redundant processors as the primary method of protection. Those on unmanned spacecraft, though redundant, were more innovative in terms of distributing tasks and processing power. The Space Station project provides an opportunity to merge the two tracks, taking from the manned programs experience with using high level languages,

---

*The author is on leave from The Wichita State University, Wichita, Kan.

This work was partially sponsored by the Department of Defense, with most research done under NASA contract NASW-3714.

software synchronization, and large-scale software development, and from the unmanned programs use of distributed systems and microprocessors. This synthesis creates a system that lends itself to the use of Ada as the onboard software development language if the problems of implementing the language on distributed systems can be solved.

2. A Taxonomy of Spacecraft Computer Systems

A review of previous onboard computer systems is in order to provide a basis for discussing a computer architecture for the Space Station. Since all previous systems have used redundancy in some form for fault tolerance, a taxonomy can be established by considering the nature of the various redundancy schemes. Four types of systems can be identified: simplex, multiplex, functional distribution with full redundancy, and functional distribution with virtual redundancy. Both simplex and multiplex schemes have examples in both the manned and unmanned programs, while the latter pair of types presently have only unmanned spacecraft systems as members.

2.1. Simplex Systems

Simplex onboard computer systems are identified by the absence of redundancy. They are also characterized by being part of a single subsystem of the spacecraft, specifically the guidance and navigation subsystem on manned spacecraft and the commanding subsystem on unmanned spacecraft. If the simplex computer system failed, its tasks would be suspended when possible, or taken over by a backup with reduced functionality. Crew and spacecraft safety would be maintained, but mission objectives would be compromised. Three simplex systems were developed in the 1960s: the programmable sequencer onboard the later Mariner missions, the Gemini Digital Computer and the Apollo Guidance Computer.

2.1.1. Mariner's Programmable Sequencer

Prior to the Mariner Mars 1969 flyby missions, unmanned interplanetary spacecraft carried hardwired sequencers. Essentially these sequencers monitored a counter that was constantly updated by pulses from a clock. When an appropriate time interval had elapsed, some spacecraft activity would be initiated. For example, after the cruise period to a planet, at a time precalculated and put into the sequencer's logic, the spacecraft would orient itself and activate experiments to be done during the encounter with the planet. This meant that very accurate preflight navigation calculations had to be made, and that the sequences could not be changed after liftoff.

Mariner Mars 1969 was to be a double flyby of the Red Planet. If the spacecraft could be fitted with programmable sequencers, then the targeting and camera aiming of the second spacecraft could be changed to follow up on discoveries made by the first flyby. For instance, if a particularly interesting terrain feature was found, the second spacecraft could have its encounter sequence reprogrammed from the earth to obtain more imaging in that area. With a fixed sequencer this would have been impossible. Accordingly, a programmable sequencer with 128 words of memory was included. Later expanded to a 512 word memory, this machine controlled two Mars flyby missions, two orbiters (1971), and the Venus and Mercury flyby mission in 1973. The latter demonstrated the flexibility of the machine because the mission was so complex one software load was too small to do the job. Therefore, a series of complete

software loads were prepared and sent up to the spacecraft as the mission progressed [Hooke 1973]. Subsequently, in flight reprogramming became a planned and common feature of interplanetary missions, greatly reducing memory requirements and increasing flexibility.

Backup to the programmable sequencer was the same hardwired sequencer used in the early Mariner missions. If the programmable sequencer had failed, then the mission could continue, but only with preprogrammed sequences. Switching to the backup resulted in reduced functions. A similar situation existed in the two simplex manned spacecraft systems.

2.1.2. The Gemini Digital Computer

The Gemini program was more than a two-man follow-on to the Mercury spacecraft. It was a test bed for guidance and navigation techniques considered essential for the Apollo lunar landing program. Two of the more difficult of these were rendezvous and computer-controlled reentry. A small onboard computer custom-designed and programmed by IBM Corporation provided real-time calculations of maneuvers for the astronauts. During a rendezvous operation, required velocity changes would be displayed and the astronauts would fire thrusters and maintain attitude during powered maneuvers. The spacecraft had lifting capability sufficient to adjust the landing point within a rectangular footprint 500 miles long and 40 miles wide. The computer was programmed to target within the footprint. Each major function was contained in separate single software modules. By using a rotary switch and the start button, a program could be selected. When the machine was not needed, such as during coasting in orbit, it could be shut off.

If the computer failed, its tasks would either be abandoned or done by less effective means. A rendezvous could be canceled. Computer controlled reentry could be replaced by pilot control, such as on the Mercury missions. Either way, crew safety was maintained, but mission objectives were not accomplished.

2.1.3. Apollo's Simplex Systems

NASA contracted with the Instrumentation Laboratory (now the C. Stark Draper Laboratory) of the Massachussetts Institute of Technology for the Apollo guidance system. A computer first built for the Polaris submarine launched ballistic missile was redesigned as the Apollo Guidance Computer. Software for the computer functioned as a priority-interrupt system with some cyclic characteristics. Jobs were scheduled and monitored by an executive program. Code was executed by an interpreter. A typical software load consisted of several dozen "programs" which could be activated by the crew. Key mission phases such as lunar orbit insertion, landing, lunar orbit rendezvous, and entry into the earth's atmosphere were computer intensive activities.

The Apollo was a two-part spacecraft: command module and associated propulsion, and the lunar module. Each module had a computer, with, of course, different applications programs, but the same interpreter and executive. If the command module's computer failed, the mission would be aborted and return to earth would be handled by doing maneuver calculations on the ground and sending instructions to the crew. If the lunar module's computer failed, it had an onboard backup. The backup computer was a small device built by TRW Corporation that could guide the ascent portion of the lunar module to a rendezvous with the command module. That was its sole function, so a computer failure during lunar descent would have caused an abort of the landing attempt.

The obvious shortcoming of simplex systems is that a single computer failure severely damages the mission. This is apparent even in the systems described here, even though their use was limited to one subsystem. As NASA entered the 1970s, spacecraft that depended on computers for more than one function were being designed. On those spacecraft, computer failures would greatly affect crew and spacecraft safety. The first method of reducing the impact of such failures was the development of multiplex systems with full redundancy.

2.2. Multiplex Systems

Three spacecraft designed in the first half of the 1970s used fully redundant computers. The Viking unmanned Mars orbiters and landers and the Skylab orbiting space station both had duplex systems, while the Space Shuttle orbiter and its aircraft predecessor had more than two computers. The introduction of redundancy as a method of fault tolerance necessitated the addition of management software absent from the simplex systems.

2.2.1. Viking and Skylab: Dual Redundancy

The Viking missions to Mars were, in many ways, the most complex unmanned flights yet attempted. A two part spacecraft was placed into Martian orbit, whereupon the orbiter portion began a search for a landing site. When one was chosen, the lander portion descended to the Martian surface. Each part of the spacecraft functioned for years, the orbiter mapping the planet and conducting experiments best done from space, the lander doing chemical and biological analyses of the Martian soil and sending detailed images of the surface back to earth. Both the orbiter and lander had dual computer systems. Each could support its part of the mission independently, or could work cooperatively on separate tasks. The orbiter computers were primarily a replacement for the programmable sequencers carried on Mariners, with the same command and control functions. The lander computer had to control the descent and later the operation of the surface station. The Jet Propulsion Laboratory, which built the orbiter, designed a special purpose processor for its spacecraft. The lander, built by Martin-Marietta Corporation, used an existing Honeywell computer.

Skylab's dual computers were also commercially available, coming from the IBM line of 4Pi processors that were derived from the 360 architecture. The Skylab computers were related to their manned spacecraft predecessors in that they were part of a single subsystem, in this case attitude control. This space station used a complex set of control moment gyros for stabilization and attitude maintenance. The computers were programmed to execute scheduled tasks cyclically, including a set of self-tests. Each cycle the primary computer would deposit a 64-bit status word to a special register in a common section of the system. This register and its associated logic were constructed of triple modular redundant circuits for reliability. If the secondary computer detected that the primary was failing its self-tests, it would take the status word from the common section before the failing computer could corrupt it, and shut down its partner. Such a failure never occurred during the lifetime of the Skylab, but a manual switchover was done to prove that the system was reliable no matter which machine was designated primary.

Even though the Viking and Skylab computers were fully redundant and provided a high degree of reliability, a dual system is insufficient for manned operations. If one half of a dual system detects a failure in the other half, it follows that the failing computer might well detect a failure in the good computer, and will try to shut it down. Also, there is a possibility that the computer detecting the failure is actually the one

failing, and that the detection is incorrect. An obvious solution to this dilemma is to add more computers, each running identical software---the solution chosen for the Shuttle.

2.2.2. Redundancy for Digital Fly-by-Wire Aerospacraft

NASA's Space Shuttle is different from all the other spacecraft so far discussed in that the onboard computers have tasks outside of a single system or small set of systems. The Shuttle computers control a large number of spacecraft functions, including such mundane items as the opening and closing of the cargo bay doors. Most importantly, the Shuttle has a digital fly-by-wire control system. This means that where mechanical linkages exist in conventional aircraft control systems, the Shuttle has electrical and electronic connections between the controlling devices, the computers, and the control surfaces. When an astronaut moves the hand controller in the Shuttle, signals are generated and transmitted to the computers, which then generate signals to the actuators at the control surfaces. Therefore, a software or hardware failure makes the control system inoperable, and even a short loss of control in a critical mission phase would be disastrous. Since early research showed that the most likely source of failure in an avionics system would be the computers, NASA chose to increase the levels of redundancy of the primary computer system to provide sufficient protection.

At first, the level of protection was what has been termed "fail-operational/fail-operational/fail-safe." If one computer fails, then the spacecraft is still operational, if a second fails, it is operational, but should return to earth because it has reached fail-safe level, at which another failure would mean serious danger. The fail-safe level escalated to three computers to avoid the stand-off situation. The sum of this is that five computers were necessary and NASA accordingly acquired five IBM AP-101 machines for each orbiter. Later adjustments to the design reduced the level of redundancy to fail-operational/fail-safe, but the fifth computer was kept on the spacecraft as a backup flight system that could be activated by the crew in case of a catastrophic failure of the primary. The backup can only control the ascent and descent of the orbiter, and by itself can not complete a mission.

Of central concern to the Shuttle designers was the development of a redundancy management scheme. Fortunately, NASA was already engaged in a research program that could shed direct light on the subject. The Dryden Flight Research Center at Edwards, California, had been conducting a digital fly-by-wire test program using a modified Vought F-8C aircraft. A single Apollo Guidance Computer was installed in the aircraft to provide flight control. An electronic analog system acted as a backup, but it never was needed. Dryden's research team realized that a simplex system would never be acceptable for routine use, so it was exploring a dual computer system when officials at the Johnson Space Center contacted them about installing three of the same computers to be used in the Shuttle in the F-8 and thus trying out methods of integrating multiple machines into an avionics system. Dryden agreed, and three AP-101s were installed and flown on the F-8. Several single computer failures occurred during flight, none of which endangered the aircraft.

The primary problem in managing multiple computer systems is failure detection. It was reasoned that if the software could be compared at regular intervals, then a failing computer would be obvious because its results would be different from the results of its partners. Comparing checksums consisting of the components of a number of parameters is a simple way of doing this; however, due to differences in the computer clocks, the machines would quickly reach the point where they were out of step, and anomalies would show up in the checksums even though the machines had not failed. To overcome this problem,

the machines had to be synchronized. Each time the software executes an input operation, output operation, or changes the module being executed, a three-bit discrete signal is sent on a dedicated bus to the other computers. The sending computer waits up to four milliseconds for its partners to check in with an identical signal. If the signals do not agree, of if the time limit expires, the computer which failed to check in properly is indicated to be failing, and the computer that detects this error goes on. Due to fear of generic errors, the computers are not capable of shutting each other off, only the crew can do that in response to the failure signals.

Basically, multiplex systems provide fault tolerance by layers of redundancy. The disadvantages of this are that entire systems must be replicated at least three times and more reasonably four times to provide the reliability needed by mission critical systems. In a computationally intensive environment, such as that on the proposed Space Station, so many processors would have to be replicated that the increase in power consumption and other resources devoted to the computers would be prohibitive. Other forms of reliability insurance developed for the unmanned flight programs may provide more sensible solutions for the Space Station.

2.3. Functional Distribution with Full Redundancy

NASA's longest lived interplanetary spacecraft are the two Voyagers launched in 1977 and still working successfully, as proved by the recent flyby of Uranus. The Voyagers carry a functionally distributed set of three pairs of redundant computers. Probably most of the reason why this computer configuration was chosen is the structure of the Jet Propulsion Laboratory. Different sections of the Laboratory contribute different components to a spacecraft. In the case of Voyager, the section that builds the command system reused the computer developed for the Viking orbiter with an almost identical software structure. The attitude control system developers used a speeded up version of the command computer and the flight data system had a newly developed machine. Each of the three groups independently determined that the inclusion of a computer system was the best way for the specific tasks involved to be accomplished.

One change caused by adopting functional distribution was the need to communicate with other computers instead of hardwired logic devices. Most intercomputer communication consisted of commands and signals relating to internal tests. More complex communications were required by the next level of unmanned spacecraft systems.

2.4. Functional Distribution with Virtual Redundancy

The next major interplanetary spacecraft designed after Voyager was Galileo, a Jupiter atmospheric probe and orbiter. Galileo carries a dual computer system for attitude control and pointing that uses an off-the-shelf microprocessor, the ATAC-16, and is programmed in a high level language, HAL/S. Its command and data system also uses commerically available microprocessors, six RCA 1802s in two strings of three. This system was derived from research sponsored by the Jet Propulsion Laboratory concerning reliable computer systems for unmanned spacecraft.

Beginning in the early 1960s, the Laboratory sponsored the design of a computer called STAR (for *S*elf *T*esting *a*nd *R*epair) that consisted of collections of multiple copies of each major component [Avizienis 1968]. For instance, memories, input/output devices and the like were triplicated. A special piece of

hardware called the Test and Repair Processor, or TARP, had five copies. When the computer was operating, one of each subcomponent and three TARPs were powered up and running. If the TARPs voted that a component was failed, they activated one of the spares. If the vote had not been unanimous, the dissident TARP would be shut off and another activated. In this way no more than the minimum number of components would be powered at any given time. The weakness of this scheme is that a failure of the switches used to turn off bad components and turn on good ones would negate the fault tolerance. However, the concept of a single computer with virtual redundancy survived to the next round of research.

Research initiated after STAR led to the construction of the Unified Data System in the early to mid 1970s [Rennels 1978]. Here the emphasis was on several processors working cooperatively. Certain processors, called High Level Modules, would communicate only with other processors, called Terminal Modules. The Terminal Modules would deal with spacecraft systems or the outside world. Conceptually, by carrying several High Level and a larger number of Terminal Modules, each communicating by means of multiple busses, and sharing numerous memory modules, the system could function with a variety of combinations of modules, memory, and bus connections. This way a single processor failure would result in a change of the configuration, but no degradation of performance unless a number of different failures occurred.

Designers of the Galileo command and data system did not fully adopt the concept, even though they adapted the terminology. Two 1802s are assigned as High Level Modules, four are Low Level Modules. Several memories and redundant busses are part of the system. However, it is fundamentally separated into two redundant strings. Even so, the software is constructed in what are termed "virtual machines" and is distributed over the several processors. From the Unified Data System and the actual Galileo software some hints for a possible Space Station computer architecture can be derived.

3. Computer Architecture for the Space Station

The Space Station will be different from any previous manned spacecraft in terms of its computational needs. In fact, it will be much closer to an unmanned spacecraft. This is primarily because the guidance and navigation tasks on a Space Station are minimal compared to what a spacecraft like the Shuttle requires for active flight control. However, considerable computational capability in the areas of data acquisition and analysis, attitude control, life support, and spacecraft health monitoring will be necessary. With this variety of tasks, it is logical to imagine that the final configuration of the computers onboard the Space Station will be a distributed system, with physical processors embedded in the hardware built to accomplish each function. Thus the Station's computer systems will resemble the functional distributions used on Voyager and Galileo, rather than the centralized systems used on the Shuttle.** Questions of redundancy can then be handled at the local level. Some systems such as life support are so critical as to require fault protection to the same degree as flight control, and will require multiple dedicated processors for redundancy. Other systems can be virtually redundant in that their tasks can be transferred to another processor in another system in case of a failure. Perhaps a common pool of processors can be made available to host tasks offloaded from failed machines. In any case, the intent of a hardware

**Although the Shuttle has "local" computers on the main engines and on payloads, the Data Processing System, with its multiplex configuration, does all other computational operations.

architecture for the Space Station should be to provide fault tolerance relative to the importance of the systems, and to avoid carrying large numbers of resource-hungry multiplex systems.

4. Implementing Space Station Onboard Software with Ada

Since Ada has already been designated as the development language for the Space Station, its strengths and weaknesses in implementing software for its potential computer architecture are of interest. Ada's strengths in developing this type of system lie in its inherent ability to handle concurrency, both in terms of data sharing and synchronization, and in hiding mechanisms of concurrency in the programming language. Fundamentally, the entire software load for the Station could be created as a set of tasks, some of which will run individually on separate processors, some of which will share a single processor, but all of which can be considered as part of a library of related programs. This was impossible in previous distributed systems in which the software for each computer was written separately in different languages, sometimes in a mix of high level and low level languages, and interconnected with great difficulty. The chief weakness of Ada at this point in time is not the language itself, but the lack of implementations of it that make use of its full range of features, particularly those most applicable to the Space Station.

4.1. Ada Features Most Useful for Distribution and Fault Tolerance

Since the original purpose of designing Ada was to serve the development of large and real time systems, several features of the language are directly applicable to programming the heterogenous machines on the Space Station.

4.1.1. Tasks

Using Ada, programs can be made up of a variety of units, including *tasks*. A task is a program unit that runs in parallel to other tasks, and to the main program, which is implicitly also a task. Moreover, it can run either interleaved with other tasks in one physical processor or as a single process on a machine in a multicomputer system. Tasks on the Space Station would have varying degrees of interaction. For instance, a task monitoring spacecraft health would periodically wish to receive signals from processes throughout the Station in order to make sure everything is still functioning. These messages would be far less frequent than three computers running identical tasks as part of a mission critical, locally redundant, synchronized subsystem. Regardless of the level of communication, the information to be exchanged can be abstracted in the task body, hiding the complexity of the interior of a task from programmers working on associated tasks.

4.1.2. Rendezvous

Previous parallel computations in spacecraft shared information by message passing or common data pools. On Voyager, messages are sent between the command computer and attitude control computer as single units. On the Shuttle, the high level language HAL/S provides for the declaration of common data shared by several scheduled parallel processes. Ada provides for common data using pragmas for shared information, but the most common form of information exchange on the Station would probably be message passing, usable for simple data exchange or for synchronization. Message passing is

implemented in Ada using the rendezvous, in which a task will be blocked while attempting to send or receive a message. When both sender and receiver reach the point in their respective task bodies where they are ready to do the exchange, a rendezvous occurs, data is transferred, and both tasks continue. These rendezvous can take place between widely distributed tasks.

4.1.3. Exceptions

One Ada feature critical for Space Station systems is the ability to gracefully handle predictable errors. Even though most Space Station subsystems could have short duration failures without endangering the crew, actively handling the failures as opposed to reacting to existing conditions is almost always preferable. Exception handlers can be part of each task, and, used creatively, can eliminate complete shutdowns of subsystems.

4.1.4. Modularity

Since the Space Station is expected to operate over a long period of time, with many changes in its component modules, the software used on it must be easily modifiable. Ada's ability to separately compile tasks that have been added or modified and include them in the existing software load is a significant advantage. NASA has made good progress in reusing software in preparing Shuttle flight loads. Consciousness of reusability can be easily transferred to the Space Station project since the development language directly supports such techniques through the use of generics.

4.2. An Example: Implementing Shuttle-Like Computer Failure Detection in Ada

As an example of tasks, rendezvous, and exception handling, the Ada code in Figure 1 on the next page implements the Shuttle computer failure detection and synchronization scheme in a two processor system.

5. Summary

Ada has many characteristics that support the development of software that implements fault tolerance schemes developed for previous spacecraft. Also, the ability to run on distributed systems essentially transparently to programmers working on the Space Station software means that a variety of redundancy configurations can be used. This, of course, depends on continued research and development concerning implementing Ada. Although some attempts have been made to implement Ada on several cooperating processors, the nuances of doing so are still not all understood [Ardo 1984]. Also, the progress of Ada development environments, though picking up steam, is still behind original expectations. Adoption of existing Ada development technology by the Space Station project coupled with support of efforts designed at multiprocessor implementations provides the safest route to completion of Space Station software in the early 1990s.

```
-- The task specification states which entry points and parameters
-- will be honored by the task.

task SYNCHRONIZE is
   entry IO_EVENT (I_OR_O: in OPERATION_TYPE);
         -- where I_OR_O is a value indicating whether the sending task
         -- running on the host processor has just done an input or output.

   entry SYNCH_SIGNAL (SIGNAL: in OPERATION_TYPE);
         -- where SIGNAL is a value indicating whether the sending task
         -- located on the other processor has just done an input or output.
end SYNCHRONIZE;


task body SYNCHRONIZE is

   WAIT_INTERVAL: constant DURATION := MILLISECONDS*4.0;

begin
 loop
        -- this task activates in parallel to the others and runs forever.

  begin
     accept  IO_EVENT (I_OR_O: in OPERATION_TYPE) do
        -- the task is blocked at this statement waiting for one of its
        -- fellow tasks on this processor to send word that it has done
        -- an input or an output. When a value for I_OR_O arrives, then
        -- it waits for up to 4 milliseconds for the other processor to
        -- check in using this select statement:

        select
                accept SYNCH_SIGNAL ( SIGNAL: in OPERATION_TYPE) do

                   -- check to see if the signals match, if not,
                   -- raise an exception.

                        if I_OR_O /= SIGNAL then
                           raise OUT_OF_SYNCH;
                        else
                           --continue
                        end if;
                end SYNCH_SIGNAL;

        or

                delay WAIT_INTERVAL;

                raise TIME_EXPIRED;

        end select;
      end IO_EVENT;

    exception
       when OUT_OF_SYNCH =>
          -- handle exception caused by a mismatch of action codes.

       when TIME_EXPIRED =>
          -- handle exception caused by a failure to synchronize in time.
  end;
 end loop;

end SYNCHRONIZE;
```

Figure 1

Bibliography

[Ada 1983]*Ada Programming Language*, ANSI/MIL-STD-1815A, 22 January 1983.

[Ardo 1984] A. Ardo, "Experimental Implementation of an Ada Tasking Run-time System on the Multiprocessor Computer Cm*", *Proceedings of the First Annual Washington Ada Symposium.*

[Avizienis 1968] A. Avizienis, "An Experimental Self-Repairing Computer," NASA-TR-32-1356, Jet Propulsion Laboratory.

[Gehani 1983] Narain Gehani, *Ada, An Advanced Introduction* (Prentice Hall).

[Habermann 1983] A. Nico Habermann and Dewayne E. Perry, *Ada for Experienced Programmers* (Addison-Wesley).

[Hooke 1976] A. J. Hooke, "In Flight Utilization of the Mariner 10 Spacecraft Computer," in *Journal of the British Interplanetary Society,* Vol. 29, April, 1976.

[Rennels 1978] David A. Rennels, "Reconfigurable Modular Computer Networks for Spacecraft On-board Processing," *Computer*, July, 1978.

Notes on the Author

James E. Tomayko is a Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, a federally-funded research and development center. He is on leave from a faculty position in the Computer Science Department of The Wichita State University. Recently Dr. Tomayko completed a three-year study of NASA's use of computers in space flight operations to be released by the Agency as a book. Articles related to the subject of the present paper have already appeared:

- "NASA's Manned Spacecraft Computers," *Annals of the History of Computing*, Volume 7, #1, January 1985, pp. 7-18.
- "Achieving Reliability: The Evolution of Redundancy in American Manned Spacecraft Computers," *Journal of the British Interplanetary Society*, Volume 38, #12, December, 1985, pp. 545-553.
- "Digital Fly-By-Wire: A Case of Bidirectional Technology Transfer," *Aerospace Historian*, Volume 33, #1.

Dr. Tomayko is a National Lecturer for the Association of Computing Machinery, and has given over 50 talks on manned and unmanned spacecraft computer systems within the last 18 months.