D11

AUTOMATION OF CHECKOUT FOR THE SHUTTLE OPERATIONS ERA

Judith A. Anderson
Guidance, Digital & Software Systems Division
Kennedy Space Center

Kenneth O. Hendrickson
Applications & Integration Software Section
Kennedy Space Center

ABSTRACT

The Space Shuttle checkout is quite different from its Apollo predecessor. The complexity of the hardware, the shortened turnaround time, and the software that performs ground checkout have proven a challenging task to overcome.

Generating new techniques and standards for software development and the management structure to control it have been implemented. New challenges await those that have been solved.

INTRODUCTION

Testing of the Space Shuttle's many systems to assure that the Shuttle is ready to refly is a complex process. This paper will highlight some of the challenges in utilizing these computer systems in the testing of the vehicle in a timely fashion and how these challenges are being met.

HISTORY

During the Apollo program, the Saturn launch vehicle was heavily computer controlled (via the RCA 110A computers) and had virtually no cockpit control since the Apollo spacecraft was totally separate. The ground computer programs were primarily assembly language, with very low change rate. A very elementary user test language (ATOLL) was available for "linking" assembly language programs and to perform simple command verification sequences. This capability allowed KSC to automate the last nine hours of countdown to an almost hands-off point by the end of the Apollo program.

Where the Saturn automation was primarily a command-by-command serial sequencing function, the automation of the Shuttle checkout and launch preparation is a very complex scheduling exercise, with complex operations to be performed. The tools provided were: 1) the LPS system, with its GOAL user oriented language, capable of monitoring the measurements on the vehicle and in the ground systems, and (2) the on-board computer system (DPS), which provided the linkage to control, in a test environment, all the vehicle subsystem controllable during flight.

CHALLENGES

The complexity of the Space Shuttle has come to haunt us frequently in our efforts to automate our turnaround activities -- from the sheer numbers of measurements to be monitored and commands to be issued to the interrelationships of the subsystems. The flexibility of redundancy makes a highly reliable vehicle to fly, however, a difficult one to test. Our earliest efforts at automated testing were just to get the minimum amount of software written to get the job done -- we did not have time for any more. As we progressed thru the STS-1 flow, the need to do things faster, and more reliably became strong drivers. For manual operations, we found all the many ways one could do them -- many which would not work!

We also found during the early flows that we were spending large amounts of manpower sitting in the firing room waiting for a problem to occur. As problems decreased because of system maturity, we still needed the same number of people on console monitoring data. This was neither cost effective nor interesting. In order to have a cost effective system we had to reduce Firing Room manpower.

The shortening of the turnaround has been a constant driver to get things done faster and more efficiently. (Reference Table 1). The STS-7 turnaround was 63 days and our target for STS-30 is 28 days, a 60% reduction. This can be attacked several ways -- decrease test requirements, accomplish the same tests faster, and accomplish more testing in parallel.

TABLE I. FLIGHT vs. TURNAROUND FLOW LENGTH

FLIGHT	LENGTH
STS-1	2 years
STS-7	63 days
STS-8	49 days
STS-16	35 days (target)
STS-30	28 days (target)

SOLUTIONS

Our first approach at speeding operations up was to automate discrete activities. During early STS-1 it would take us over two hours to power up the Orbiter. This drove us to a 24 hour/day operation in just one week to avoid the overhead of the daily power-up time. We are currently powering up OV102 (STS-9) daily in less than 25 minutes. During this time period, approximately 500 commands are issued via computer and approximately 50 switches are thrown in the cockpit. How did we do it? First, each of the institutional support systems (EPDC, instrumentation, cooling and DPS) developed software to automatically perform each of their power up functions with a minimum of manual intervention. Control of these programs was then centralized at the integration console which cues each of the subsystem consoles when it is time to do a part of their activation. Procedural steps which must be run manually are tutorially presented to the console operator. This eliminates any unnecessary decision making in selecting the proper support LRU's for the day and "filling in the blanks". It also allowed all measurement/feedback information to be checked by the software.

Could these kinds of techniques be applied to other situations? Certainly, however, our normal testing situation is not as easily predefinable on a day to day basis as power up. Today we may want to test system A then B then C. Tomorrow we may need to test A in parallel with C and then do B. There are many drivers to the order of testing -- manpower available to do a job, Ground Support Equipment ready, compatibility of operations (downlist/downlink formats, GPC memory configuration, etc.) and the jobs scheduled due to unexpected drivers such as equipment failure and replacement.

In order to automate on a global firing room basis, we first have to automate individual subsystem functions. This is currently underway. To assure that these functions can then be integrated, a common set of groundrules (standards) have been developed to assure that subsystems can communicate with one another and with the integration console in a uniform manner.

STANDARDS

A set of groundrules were established to design the application software. Early efforts concentrated on standardizing the man-computer interface. Standards were developed for the use of color on the CRT's and on how the data was to be displayed. Standards were developed on how the engineer would use the CCMS keyboard to communicate to the application software. Later standards were developed that provided rules on how the software structure was to be designed. This standard identified program types and the relationship of each type to the overall design of the

software set. Application sets were defined along engineering subsystems, e.g., Hydraulics, Environmental Control and Life Support, Electrical Power Distribution and Control, Data Processing, etc. These Application Sets were assigned to a physical Firing Room console and teams were established by console to produce the documentation and software.

Each Console Set Working Team is comprised of contractor and NASA system engineers from each member Application Set, software specialist engineers, technical documentation, and quality control personnel. The teams are responsible for producing console application software requirements, software design specifications, and development and implementation of the software itself. The teams meet on a regular basis to coordinate requirements and implementation details. This highly orangized activity is opposed to the methods used earlier where a system engineer had a broad, general set of requirements which he went off and coded to. Since most programs were simple, stand-alone programs, this method was satisfactory. The increased level of organization and coordination was driven by the increased complexity and interdependency of the software which was required. Because this software was now going to be used at multiple sites (VAFB), and because it would probably be with the Shuttle program longer than its designer, documentation became more important.

In addition to the console set working teams, someone had to assure the consoles would communicate properly with one another, and that subsystems required to support other subsystems were aware of it. This function is provided for by the Software Automation Subpanel. This group's primary responsibility is the integration of the automation effort within the Firing Room.

STRUCTURE STANDARD

The Software Structure Standard establishes a software design that separates the overall software function into primarily three groups. Display Driver programs are the primary man-machine interface. The operator uses these Display Driver programs to initiate software functions and also to view data on the engineering system. While looking at an overview display of a Shuttle system, the operator may move a cursor to a target on the CRT which causes the overview display to terminate and another Display Driver is to be performed which displays a particular subsystem in greater detail. This Display Driver may have cursor targets that, when selected, cause a particular command to be issued. The command feedback is displayed, allowing the operator visability into system response to the command.

Sequencer programs are designed to automate a particular function. There are Sequencer programs that power-up a particular hardware system on the Orbiter. There are sequencers that perform detailed LRU checkout. In general, a sequencer requires no manual control except to perhaps supply program options, or respond to errors. A sequencer provides only limited operator interface capability. Instead the sequencer interfaces with Display Driver programs to display messages or to present prompts to the operator. The Sequencer program is also responsible for recognizing and reacting to system anomalies.

The third class of programs are those that bind the other program types together and provide the continuity between one function and another. The main program in this type is called the System Scheduler. The purpose of this program is to validate all requests to perform a function against functions already in progress and the current hardware configuration. It also establishes a relative priority among functions and will interrupt one task to execute a task of a higher priority. The System Scheduler is the hub of inter and intra console software communication. When one program needs to communicate to another it sends the request to the System Scheduler which will validate the request and relay it to the proper receiving program. This same scheme works when one Application Set needs to send data to or request data from another Application Set. This standardized communication scheme provides the linkages that form an integrated Application Set and ultimately an integrated Firing Room software design.

CONSOLE STATIONKEEPING

In order to solve the problem of decreasing the number of engineers required on console during relatively quiet periods while other subsystems are testing, we developed a concept called

"Stationkeeping", (frequently referred to as "babysitting"). Depending upon the system being station-kept, the level of monitoring of functions and automatic response varies. All systems have a set of measurements which are monitored for anomalous conditions. In general, these measurements are monitored against limits in the Front End Processors. When a limit is violdated, an interrupt is sent to the GOAL program at the console. In response to this interrupt, the program then evaluates a set of related measurements to determine what, if any, the failure was. A message indicating the failure is then sent to the operator. In the case where no operator is present, the message is routed to the Integration console for display to the operator there.

What happens in response to an error? Here again, this is system dependent. In DPS, any failures which degraded the testing support (such as a GPC failure) caused data collection to be automatically initiated and a proposed plan for recovery to be displayed to the operator. If the console operator selects to perform the recovery plan, all steps which can are automatically executed. Any steps requiring manual actions are presented in a tutorial fashion. This software, in essence, is a canned "expert system engineer" who knows what to do ahead of time in all predictable failures. There undoubtably will be cases which the software was not designed to cover. When these occur they will be added into our software, thus teaching our "expect" something new.

The concept of stopping to provide the console operator with an option to perform the recovery sequence or not is not used in all cases. In many instances, because of possible hazards involved or potential hardware damage, recovery is invoked automatically. Loss of cooling is an example where steps are automatically taken to restore cooling to the vehicle without operator intervention.

Once we developed the concept of stationkeeping software for systems when engineers were not going to be on station, it was just an extension to also use this same software when the engineer was on station. This helps in assuring the appropriate data is taken when a problem occurs and that the correct steps are taken to correct a problem. This allows less experienced engineers to become console operators. The stationkeeping concept has also been extended to systems such as hydraulics to provide their "incident prevention" software which causes emergency hydraulics power down whenever anything is detected which indicates the system is incorrectly configured or something critical has failed which could result in hardware damage.

In the case of DPS, in order to have Integration console do their stationkeeping a number of support functions had to be performed from the Integration console (i.e., format changes, launch data bus switching, I/O resets, etc.). This was easily implemented using the communication techniques described above. As the system matures, additional capabilites will be added to the Integration console menu of DPS functions to increase the amount of time stationkeeping can be active from the Integration console.

Although our stationkeeping software is still under development, we have already begun to reap the rewards. DPS stationkeeping software went on station during STS-6. Approximately 80% of the flow is now done with no one at the DPS console. This solves many problems:

- o More cost effective utilization of manpower.
- o Improved morale by decreasing shift work.
- o Allow engineers to work more interesting tasks.

THE CHALLENGE IS NOT OVER

The solutions that have been outlined have a common denominator. They all require highly integrated and complex software. Early efforts at automation isolated top level functions from one another to minimize the amount of interaction between software elements. This methodology worked fine but it would not support an environment where multiple semi-independent operations

Firing Room environment is exactly what is needed to produce a turnaround concept that minimizes human intervention and decision making. Now that we have solutions for our past challenges, new ones confront us in our efforts to control this huge ball that has begun to roll called "automation". The software development tools that we have used in the past were fashioned after our level of sophisticated software which in most cases was crude. The new challenge that we face is to produce the software development tools and techniques that will keep the automating ball rolling in the right direction and speed so as not to swallow up those of us in its path.

For the most part, mathematical models of the various Shuttle and ground support systems were used to verify the checkout application software. Our simulation capability is called the SGOS (Shuttle Ground Operations Simulator) system. It consists of math models executing under a real-time operating system in one of our ground data processing computers and another computer that supports the Orbiter and ground data links, buffering, and data conditioning between the Firing Room and the real-time operating system. To the Firing Room personnel and their software executing in the consoles, a high fidelity math model will provide measurements and react to commands identically to its hardware counterpart. The math models would adequately allow the engineer to debug and verify his mostly manually controlled programs, but they were not to the level of fidelity to simulate a total system response to a stimuli. The need to have high fidelity integrated models of hardware was an obvious priority when we began our automation effort. Once high fidelity models were produced, we quickly ran into the limits of the real-time simulation capabilities. Unlike other NASA centers, we don't have dedicated computers for our math models to run in. Instead we designed a real-time simulation operating system that would time-share the computer resources with many other operations. Because of this constraint, the problem of increasing the simulation capability without taking over the whole computer and still maintaining real-time response proved quite challenging. The end result of this challenge is affectionately known as "Big Sim". The system has just been released for Firing Room use. It triples our model capacity while spreading out its operating system responsibilities so as not to significantly increase processor usage. With this increased capability, we are now able to integrate enough system models to simulate a total Shuttle at the Pad with the required ground support equipment. For the first time we will have the capability to provide launch countdown simulation with high fidelity math models. "Big Sim" is a necessary and welcome addition to our expanding collection of software development tools.

Software development is a lengthy and time consuming process. Requirements must be generated, software specifications must be developed, and the programs themselves have to be coded, debugged, and verified. Each of these steps have to be reviewed and approved. The whole process is complicated even more by our overall objectives to significantly increase the level of integration between systems. The automation effort requires a substantial committment by NASA and its contractors to supply the necessary manpower to implement these concepts that have been discussed. This software development effort coincides with our requirement to shorten Shuttle turnaround and to process multiple Orbiters, ET, and SRB's in parallel. All this must be done within the current manpower ceilings in order to remain cost-effective. How do we do it?

We are currently working on a software system that will take a software specification as input and generate an application program. The system is called LAP (Launch Processing System Automatic Programmer). The specification document written in a user oriented language is processed against the rules of our Application Software Structure Standard. The output will be an application program meeting the software specification requirements and also conforming to the Structure Standard. This automatically generated program should be much easier to debug because only high level logic needs to be checked instead of a module by module debug. This system is in the very early stages of implementation, so it is too early to tell how efficient the end product will be. Because few of our application programs have been optimized for speed, we do not expect the inherent degradation of performance usually associated with adding another layer of software between the programmer and computer to be much of a problem.

We have accomplished a lot in our automation efforts to date, but the job is far from complete and we continue to meet challenges on a daily basis.