IAA-0603P

AUTOMATED CONSTRAINT CHECKING OF SPACECRAFT COMMAND SEQUENCES

Joan C. Horvath, Leon J. Alkalaj, and Karl M. Schneider
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109

Joseph M. Spitale
California Institute of Technology
Pasadena, CA 91125

Dang Le
Telos Corporation
4800 Oak Grove Drive
Pasadcma,CA91109

ABSTRACT

Robotic spacecraft arc controlled by onboard sets of commands called "sequences." Determining that sequences will have the dc.sired effect on the spacecraft can bc expensive in terms of both labor and computer coding time, with diffc.rent particular costs for different types of spacecraft. Specification languages and appropriate user interface to the languages can be used to make the most effective use of engineering validation time. This paper dc.scribes one specification and verification environment ("SAVE") designed for validating that command sequences have not violated any flight rules. This SAVE system was subsequently adapted for flight use on the ']'OP};X/Poseidon spacecraft. The relationship of this work to rule-ba.wd artificial intelligence and to other specification techniques is discussed, as well as the issues that arise in the transfer of technology from a rc.search prototype to a full flight system.

1N1'ROI)LJCH'1ON

A Taxonomy Of Robotic Spacecraft

Robotic spacecraft have completed a variety of complex missions. All of the.sc spacecraft have the characteristic that they must be remotely commanded. Table 1 lays out the differing commanding drivers for three categoric.s of spacecraft: the earth-orbiting spacecraft, the planetary orbiter, and the. planetary flyby mission. Each type of mission has some overlap with each of the other two types, and a given mission might not fit any of the stereotypes Mow exactly. However, some general characterizations arc true: planetary missions, duc to their long one-way light times, tend to require that they be able to "take care of themselves" for a longer period than a comparable earth orbiter. as well as simply surviving for years until they reach their prim target. '1'heir signal strength will be lower duc to distance and to the higher cost of injecting a large antenna into an interplanetary trajectory, which in many cases will lead to lower bit rates.

Planetary missions usc the Deep Space Network (DSN), which has a scheduling system very different from the earth-orbi[cr Tracking and Data Relay Satellite S ystem ("1'1>1<SS). '1 'DRSS schedules tend to be more dynamic than DSN schedules, duc to the need to accommodate Shuttle operations as well as planetary missions.

}'lane.tary spacecraft controllers usual] y need to explicitl y manage three time systems with varying offsets: earth receive time, ground transmit time, and spacecraft event time, which can

differ by tens of hours for far-distant spacecraft. Earth orbiter tools usually do not need to make this distinction. From deep space, "earth" is close to a point target, and usually a planetary spacecraft will not need to "know" which antenna site on the ground it is using. Earth orbiters need to point at their ground or space antenna, since Earth looks big from Earth orbit.

Mapping or orbiting spacecraft, whether around earth or around another planet, will tend to be more repetitive in their actions than will a one-opportunity flyby spacecraft. This will lead to a requirement for differing optimization for tools for orbiters and flybys. The single-opportunity nature of flyby encounters also of necessity leads to a differing attitude about risk than for an orbiter.

The large number of satellites in Earth orbit has led to the development of a reasonably good characterization of this environment; planetary spacecraft largely do not have this luxury (yet). This means that planetary spacecraft may have more unknowns in an anomaly analysis.

For all these spacecraft, however, frequently the first sign of trouble is that the spacecraft ceases to communicate. The desire to avoid this state leads to a desire for sequence validation tools. The tools discussed in this paper are designed to make it easier to capture knowledge about the spacecraft's constraints in both nominal and anomaly conditions to make it less expensive and more reliable to fly any of these types of missions. A tool to help ensure quality of sequences for all the categories of spacecraft will be fast, to allow for quick turnarounds for volatile environments; will be easy to use, to allow all three. types of spacecraft users to input their requirements; will be useful both in a TDRSS or DSN-relate.d scheduling environment, and will not be tied to an optimization of any particular frequency of events.

	Earth Orbiters	Planetary Orbiters	Planetary Flybys
nit rate	High	Lower	Lower
Lifetime requirements before prime mission	Zero (checkout only)	Years	Years
Repetitiveness of observation scheme	High to low	High to low	Low
Light Time	Negligible	Long	Long
Signal strength	High	LOW	LOw
Space Environmental knowledge	high	Low	Low
Tracking network	TDRSS (or none)	DSN	DSN
Schedule volatility	nigh	l&w	Low
First usual symptom of failure	Silence	Silence	Silence

Table 1. Some characteristics of different types of missions.

Spacecraft commanding

Currently, spacecraft software and commanding systems are built in a complex hardware and software context. (Figure 1.) Flight software is defined here as the lowest-level hardware

management soft$_{war}$e, that manages analog inputs from various mechanical devices on the spacecraft, m anages low-level data management and bus interactions on the computing hardware, and so on. On top of this software runs an operating system to manage the interaction of the "programs" that will cause the spacecraft to perform its science and engineering tasks. (The distinction here between "flight software" and "operating system" is somewhat arbitrary and semantic, and different projects and researchers split the difference at varying points. The exact distinction is not relevant for future discussion here.) Above this 10 W-ICVCI management layer is a set of commands that are the first level a routine user of the system will see. These commands are usually relatively low-level actions -- to slew a platform so many degrees, for example, or to flip a relay. Two types of "programs" can be written using this command set: fault protection routines and sequences.

Fault protection routines are programs that monitor spacecraft state and activate themselves w] ıen some error state occurs. These can be anthropomorphicized as the "reflex actions" of the spacecraft. Sequences are the "conscious" science and engineering actions that the spacecraft needs to take to complete its missions. Both fault protection and sequences have tools that have been developed to assist in building the.sc programs, as well as a variety of simulators that examine interaction between sequences and fault protection and programs for tracking results of sequences like science data quality, spacecraft hardware configuration, planned tracking station usage, and so on. Different types of spacecraft operations specialists will typically deal with different "layers", as shown in the key of Figure 1.

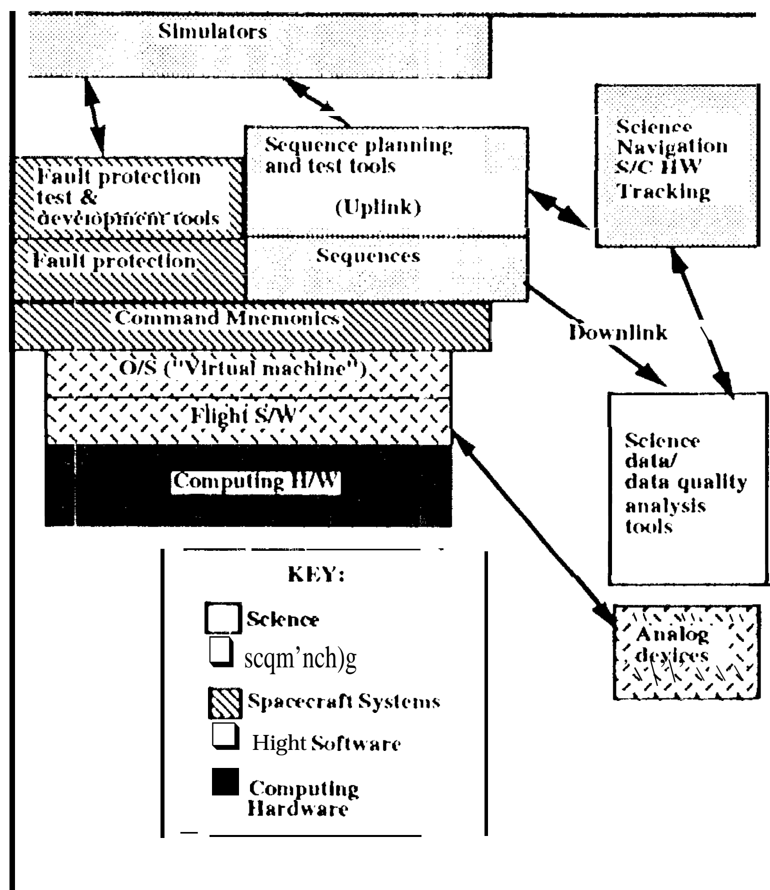

Figure 1. Software and hardware context for sequencing tools.

What is automated and why

To support the building of sequence programs, a variety of automation tools have bear developed. Figure 2 shows a generic sequence-building process, and shows the relative degree of automation of each of the processes for a typical spacecraft. Sequence requests are integrated more or less by hand, since this involves discussion, knowledge of both science and the spacecraft, and so on. After integration, the sequence "program" must be written. Usually, this program is written in "subroutines" (called blocks, or modules, or some other project-spc~ific terminology). For space.craft that are fair] y repetitive, this step will usual] y be more automated than for spacecraft that are one-encounter spacecraft.

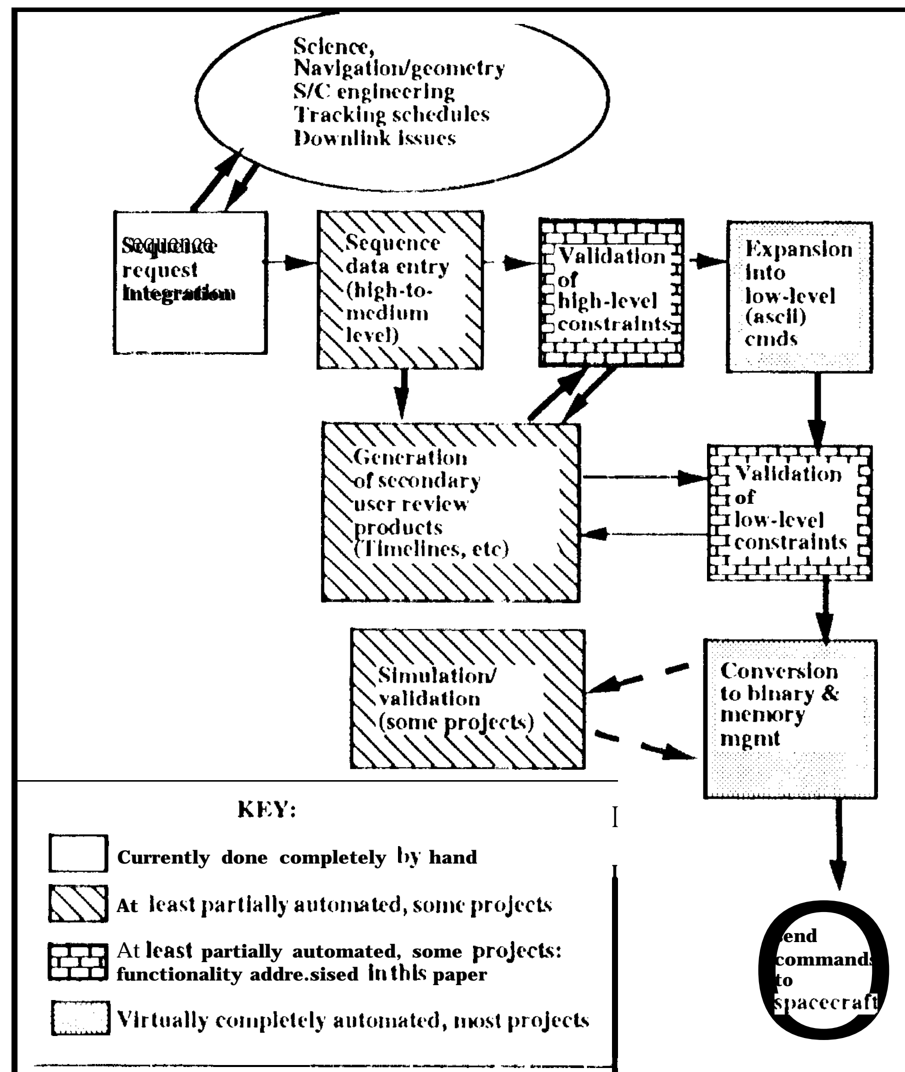

Figure 2. Software context of sequence validation tools.

These "programs" must then be debugged (checked against constraints), compiled down into the low-level commands that run on the spacecraft operating system, and then checked again against command-level constraint% "I'his constraint-checked program is then made into binary and sent to the. space.craf[. Along the way, a variety of ancillary programs generate timelines and other human-readable products to assist in review. Some projects also will simulate the effects of a

sequence (and its interaction with fault protection) at some level. The response time and character of the tools for building, sequences reflect the expected volatility of the sequences (if everything is going to change a lot at the last minute, one might as well defer most detailed planning until then, for example) as well as most of the other characteristics in Table 1.

The constraint-validation tasks in Figure 2 are among the most time-consuming across all types of missions. If a means could be found to automate this part of the command-generation ("uplink") process in a manner most applicable to each of the categories of robotic spacecraft, the savings would be substantial.

‘1’111{ SAW; SYSTEM

Overview and chronology

We have defined a Specification And Verification Environment (SAVE) for spacecraft flight rules. Initially, it was developed as a prototype running on a parallel computer. This prototype was then adapted to run on a desktop workstation for actual flight use for the ‘1’OPI{X/Poseidon space.craft; this adapted version is called MSAVE (Mission Planning and Sequencing Subsystem SAVE). The implementation of this system has been extensivel y described elsewhere[1,2,3] and will be summari zed below.

Parallel processing heritage

The SAVE system was developed with the idea of running on a parallel computer so that complex systems could be validated interactively. This meant that the information in the models had to be readily separable into mostly non-interconnected chunks. This was achieved[2] by requiring that a command only "belong" to one model; a mechanism was developed for transmitting data between models when the need arose. Our first test case was obtained by "reverse engineering" several Galileo models and rules from a PL/I based older mainframe code. Good utilization efficiency of the parallel processors was obtained[1].

Constraint identification versus constraint satisfaction

The SAVE system identifies constraints that have been violated. It does not suggest means for resolving the constraint, as has been studied for a variety of artificial intelligence planning tools. The constraint-identification problem is much simpler (solution complexity is of the. order of the number of rules and models), whereas constraint-satisfaction and scheduling problems are for the most part np-complete. The. SAVE system can thus be thought of as a stepping-s[one towards distributed planning systems (which are under development for several applications)[4]: it forces the user to collect constraints on the system in an organized and complete manner, while avoiding the difficult problem of attempting to lay out all possible future paths in a replan.

Automating constraint checking while allowing the user to determine the fix for the constraint violation allows the computer to do what it dots best (sifting through large amounts of data and events) while allowing the human to do what she does best (handling exceptions). This approach remains scalable as the system gels larger and more complex, since the complexity of the mode.ls will rise more or less linearly with the complexity of the system.

Other specification systems that allow a user to specify constraints and desired system behavior with the intent of generating provably correct code have also been developed[5] and we will continue to explore ways of using the best features of language.s and underlying code generators to have as intuitive and portable a constraint-chec king and machine code generation system as possible.

Specification and verification languages

The "Specification" part of the SAW environment allows the user to specify the system behavior without specifying constraints. This behavior is represented using a finite-state machine approach similar to the statecharts introduced by D. Harel.[6] The specification can be simulated so as to validate system behavior. Every system is modeled as a set of states and transitions between those states. In the spacecraft models these transitions usually arc caused by commands (sometimes with a time delay).

Defining models of the spacecraft

SAVE uses a spreadsheet-style specification technique. The user develops models by laying out a spreadsheet (original] y, a commercial personal computer spreadsheet was used) which has states of the system cm onc axis and commands on the other axis (see Figure 4). The user puts an "action" into each cell of the spreadsheet that reflects how the system responds to that command when it is in that particular state. [1,2,3]

"Whenever" clauses

Flight rules arc expressed as relationships among state transitions. We use what we call a "whenever" clause to express a flight rule:

Whenever (a state -> a certain value)
if (some condition holds)
=> (a violation of the flight rule)

Where "->" implies "bc.comes" and "=>" means "generates". The "->" implies that the state is set to the value. of interest whether it already had that value or not. We also allow the operator "=>" in the syntax; this is read in that context as "changes to the value. " The two operators arc useful for different types of rules.

Rule and model compilation

Once a user inputs a rule or model, this new information is compiled into code in the C programming language, and this new C module is linked in with the existing models and the general simulation and utility libraries. Figure 3 shows the overall architecture of the SAVE system. Once the code is linked in with the standard core. routines, this software system becomes the "verification" executable code. 'i'his is the part of the system below the dotted line in Figure 3.

The "Verification" part of SAVE allows a user to check constraints on the state space. These constraints can be behavioral, imposing some ordering on events, or the constraints can be time-dependent. Flight rules may translate into one or several actual state constraints. Validating a sequence of commands involves reading in a sequence, which is interpreted as a set of events to the models. "l'hat is, with each command the models change state according to their specification. Whenever an event occurs that changes a state variable for which a constraint exists, this constraint is checked; if the constraint is not satisfied, the user is notified.

The oniy host-dependent parts of SAVE arc those shaded in Figure 3. The core routines arc shown partially shaded, since some additional routines arc brought into the core to handle parallel processing hosts versus single workstation hosts.

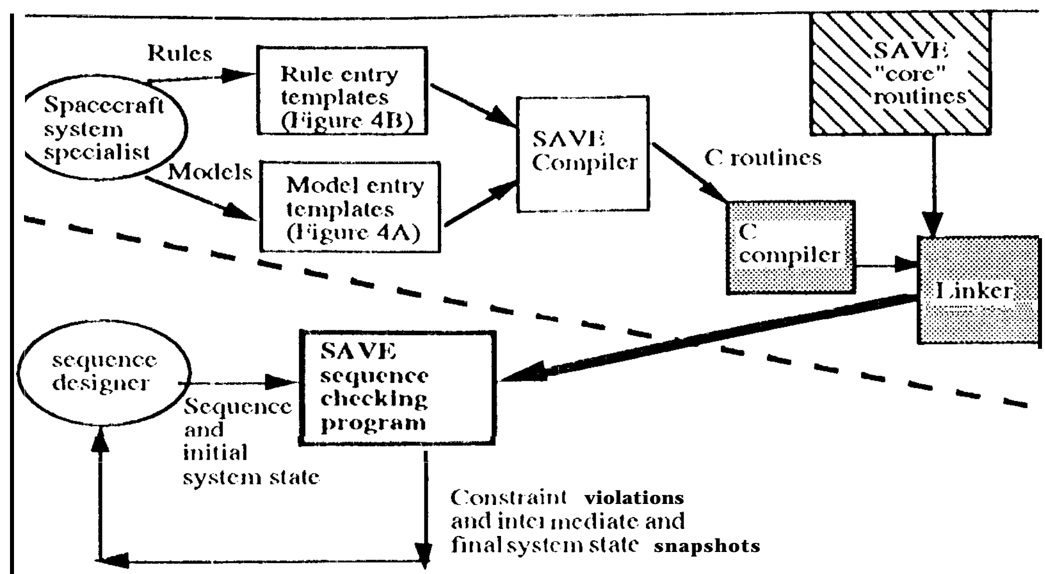

Figure 3. Overall architecture of SAVE.

TOPEX/POSEIDON MSAVE SYSTEM

TOPEX/Poseidon is an earth-orbiting system that is mapping the Earth's ocean surface. Some time after launch, it was decided that it would be desirable to start automating the checking of low-level constraints. The SAVE research prototype was becoming mature at this time, and it was decide to modify this system for use on TOPEX. The modified system, which runs on a single processor workstation (as opposed to a parallel computer), is called MSAVE.

New requirements on MSAVE

There were several new features required for TOPEX that were not applicable for the initial research prototype. The first requirement was a graphical user interface (Figures 4A and 4B). This interface allowed the user to rapidly enter and debug rules and models. The second was the ability to read TOPEX-formatted sequences, which are different from the Galileo prototype format.

Users also needed snapshots of state information (analogous to debugging dumps in conventional programming) so that they could determine why state transition errors had occurred. Since errors in a sequence are taken seriously, users also wanted printed "no errors were found in this sequence" message outputs when no constraints were violated.

Testing

One of the biggest issues that arose in transferring this technology into flight use was determining the proper level of test for the core code, which had not been developed in the traditional flight system. We had to determine the proper level of documentation for the code that had not been developed in a traditional waterfall development cycle and then find ways of consistently determining ways of testing to requirements.

MSAVE rules are compiled into C, versus being interpreted, for speed generally and efficient use of a parallel processing computer in particular. Thetthe entire program is the.n relinked and a new executable is generated. A long philosophical debate ensue.d about when "new code" was being delivered to the project. Part of the point of MSAVE was to allow easy modification

and addition of rules. Hence, we had delivered extensive rule and model modification capability. A variety of special case.ss and guidelines were developed to govern the amount of retest required when a rule or model was added or modified.

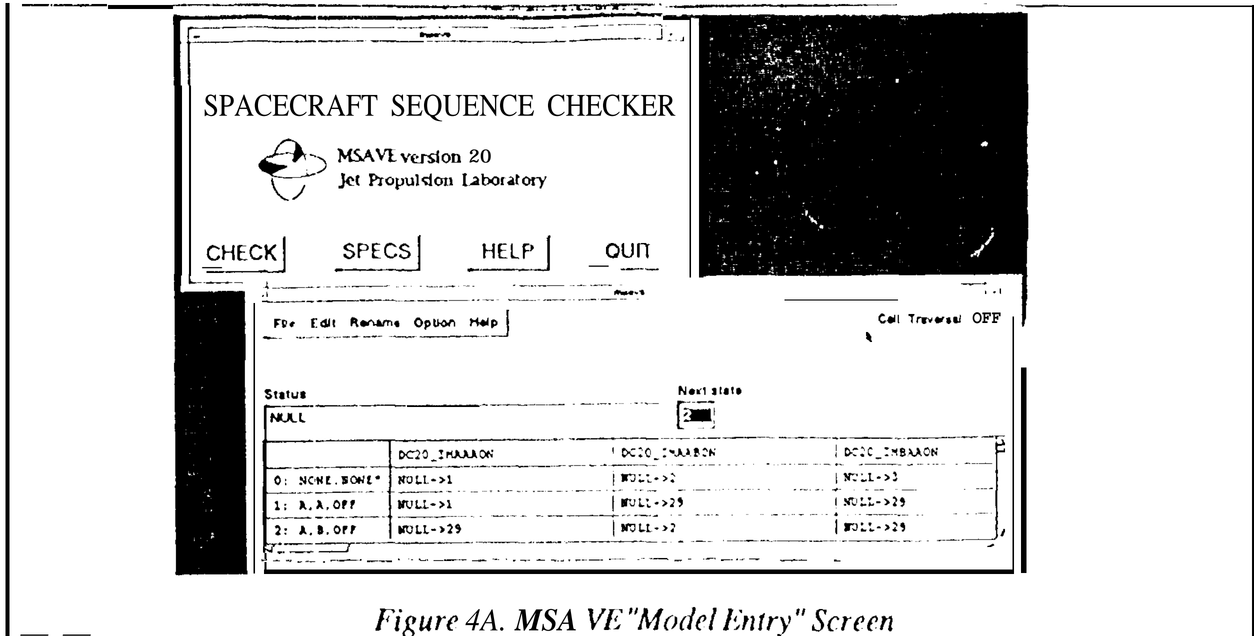

Figure 4A. MSA VE "Model Entry" Screen

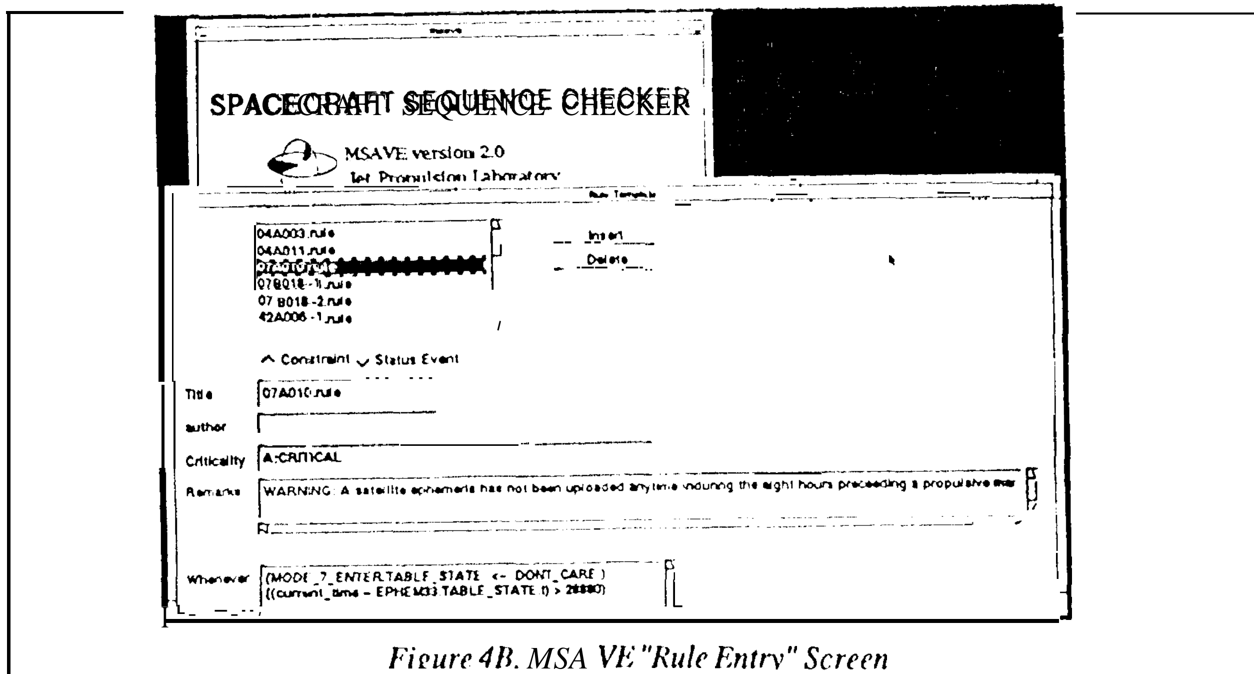

Figure 4B. MSA VE "Rule Entry" Screen

nitial user reaction

Previous constraint checking software had been slow since either it ran on an older mainframe computer or because it was using an interpreted language, whereas MSAVE is compiled. MSAVE checks a TOPEX week-'long sequence in a matter of seconds. This initially caused a reaction among some early test users that the code was not actually doing anything.

(Creation of some artificial test cases with abundant error messages later cured this impression.) We also had used terminology familiar to those in the computer science community for items on pull-down menus instead of using terminology traditional to the sequencing community. This also slowed wider user acceptance.

Long-term program usage

MSAVE 2.0 is now in use on TOPEX and has not had any major failure reports since its delivery about six months prior to this writing. It is routinely used in mainline flight operations to support checking the rules[7] shown in Table 2. Criticality "A" rules are the most critical; some of them took many "whenever" clauses to implement because they involved complex timing management. "constraints" were checks that were never made into flight rules, but that were useful consistency checks and items that the sequence team felt they wanted flagged, A total of 32 model tables were built to simulate the behavior of the spacecraft associated with checking these rules and constraints.

Criticality	Number of rules	Number of "whenever" clauses needed to implement rules
A	8	69
B	3	4
C	1	2
Constraint	16	16

Table 2. Relationship between "whenever" clauses and rules

Knowledge capture

One of the good side effects of the MSAVE program was that it forced explicit codification of behavior of many TOPEX subsystems at a time (just after launch) when many of the subsystem experts were getting ready to move on to other things. The user-input format is understandable and usable for many engineers with minimal explanation, avoiding the cost of an intermediate rule programmer, as well as minimizing the loss of information that can occur during encoding.

IMPLICATIONS OF THIS WORK FOR FUTURE MISSION OPERATIONS

How can the sequencing experience teach us to write more error-free software?

Sequences are just software, with the important distinction that the consequences of a bug that gets through to the spacecraft can be catastrophic. Tools and techniques developed for the specialized purpose of validating sequences should generalize to the problem of validating software, especially for real-time systems. We intend to pursue these analogies with a variety of collaborators. We are particularly interested in the feature that SAVE was explicitly designed to scale well to the case of large systems, a regime of software development and validation that has so far largely resisted practical validation tools. We would like to see SAVE-like tools used at some of the other levels shown in Figure 1, particularly in the fault-protection program development arena.

In the sequence validation realm per se, however, we also are encouraged by the fact that this methodology does not really favor any of the classes of spacecraft in particular, and seems to be generally applicable and intuitive for each of the styles of sequencing programming. Interestingly, for TOPEX (which is of the repetitive-orbiter category) some of the constraints implemented are not expected to occur often -- in fact, some models were put into software

precisely because the rule-triggering event was so rare that there was concern that by the time it did occur no one would remember that this event had implications for sequence building. Most of the rules and constraints, however, were rules that potentially were violated dozens or hundreds of times per sequence, since each sequence consists of about 100 orbits of the earth and some actions are taken either every few orbits or several times an orbit. SAVE can be useful both in the realm of capturing knowledge for single-encounter, long-cruise spacecraft (where spacecraft knowledge might be lost to turnover by the time the spacecraft gets to its target) and for facilitating fast turnaround for volatile, yet repetitive scheduling situations for earth orbiters.

Ideas for facilitating technology transfer in and out of flight projects

The SAVE/MSAVE experience has also been of significant interest because it is one of the more successful examples of a research system making the transition into routine flight operations. There were a variety of factors that facilitated this. One of the major ones was that one of the authors was responsible both for the SAVE prototype and part of the TOPEX sequence generation software and was "bilingual" in terminology used by research computer scientists and sequencers. It is important that there be crossover between the two communities for transfer like this to work. Often, research groups are looking for "real problems", which flight communities have in abundance. Frequent informal contact between the two groups might be of long-term benefit to both. This will require creating incentives in research communities for computer scientists to spend time on a flight project in an operational capacity (currently, this looks like a hole in the researcher's publication record) and finding ways for operations people to participate in research tasks. None of these have easy or obvious implementations, but the potential benefits are very large.

ACKNOWLEDGMENTS

The authors would like to acknowledge the support of the JPL Director's Discretionary Fund and the TOPEX/Poseidon project at various stages of this work. Joseph Spitale supported this task as a California Institute of Technology Summer Undergraduate Research Fellow. Thanks are due to a variety of people from the TOPEX project who assisted with testing and requirements review. The work described in this paper was performed by the Jet Propulsion Laboratory, California Institute of Technology, under contract to the National Aeronautics and Space Administration.

REFERENCES

1. Horvath, J. C., Alkalaj, L. J., Schneider, K. M., Amador, A. V., and Spitale, J. N., "The 'Instant Sequencing' Task: Toward Constraint-Checking a Complex Spacecraft Command Sequence Interactively." Presented at and in proceedings of NASA SpaceOps92, Pasadena CA November 1992.
2. Alkalaj, L. J., "Towards a Specification Language and programming Environment for Concurrent Constraint Validation of Spacecraft Commands," JPL Internal Report, July 1992.
3. MSAVE Team, "Mission Planning, Sequencing and Scheduling Subsystem: User's Guide, MSAVE 2.0". JPL Internal Report, August 1993.
4. Lansky, A.L., "Localized Planning with Diversified Plan Construction Methods." NASA Ames Tech Report FIA-93-17, June 1993.
5. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., and Underwood, I., "Deductive Composition from Astronomical Software Libraries." To be presented at Twelfth Int'l Conference on Automated Deduction (CADE-12), Nancy, France, June 28-July 1, 1994.
6. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, A., Shtull-Trauring, A., Trakhtenbrot, M., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." IEEE Trans. on Software Engineering, 16(4):403-414, April 1990.
7. Horvath, J.C., "TOPEX/Poseidon Project Flight Rule Model, Constraint, and Status Implementation Document." JPL Internal Report, July 1993.