

Techniques for Simplifying Operations Using VML (Virtual Machine Language) Sequencing on Mars Odyssey and the Space Infrared Telescope Facility¹²

Dr. Christopher A. Grasso (Stellar Solutions)
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
M/S: 301-250D
Pasadena, CA 91109-8099
303-641-5926
cgrasso@stellarsolutions.com

Abstract—VML (Virtual Machine Language) is an advanced procedural sequencing language which simplifies spacecraft operations, minimizes uplink product size, and allows autonomous operations aboard a mission without the development of autonomous flight software. VML is used on Mars Odyssey and Space Infrared Telescope Facility, and is slated for use on Mars Reconnaissance Orbiter. The language is a mission-independent, high level, human readable script. It features a rich set of data types (including integers, doubles, and strings), named functions, parameters to functions, IF and WHILE control structures, polymorphism, and on-the-fly creation of spacecraft commands from calculated values.

The ground component of VML consists of a mission-independent compiler, a data-driven command generator, and an execution tool, all of which run under Unix. The compiler translates human readable source to a binary format. The data-driven command generator translates mission-specific spacecraft commands for the compiler from human-readable text to binary. The offline sequence execution tool runs sequences at speeds several thousand times real-time, and provides debugging features, integrated reports, and interactive execution options. These tools allow iterative development of sequences with a turnaround time of seconds rather than the hours or days typical with full-up lab testing.

Parameterization and use of reusable functions called blocks onboard the spacecraft has several advantages over ground expanded sequences. Mission safety is enhanced, since blocks receive more scrutiny up front during development. Development of sequences is simplified, since blocks provide a rich set of behaviors that can be invoked. The review and test process for the invoking sequences is simplified, since the behavior of the blocks is well understood. Mission costs for autonomy are reduced, since responses to conditions are coded into blocks and upgraded without changing flight software. Uplink load is reduced, since the blocks physically reside onboard the spacecraft.

VML use on Mars Odyssey and the Space Infrared Telescope Facility (SIRTF) has allowed spacecraft

operations teams to place autonomy aboard deep space missions. For instance, the Mars Odyssey team developed VML blocks which could react to autonomously detected unexpected blooming of the Martian atmosphere during aerobraking end-game to raise the spacecraft orbit to a safe altitude, without ground intervention. SIRTF is using VML functionality to gather more data during the mission by detecting when the facility has settled after a slew, rather than using worst-case settling times. SIRTF also uses VML to dynamically build spacecraft commands, dramatically reducing the size of uplink products and allowing the mission to live within its communications allocation.

This paper discusses techniques for parameterizing routine operations using onboard blocks. The relationship between one-use sequences and reusable blocks is discussed. Reduced development effort due to iterative block development is outlined. The ability to migrate to the spacecraft functionality which is more traditionally implemented on the ground is examined. The implications for implementing spacecraft autonomy without the need for expensive flight software agent development is also discussed.

Table of Contents

1. INTRODUCTION: SEQUENCING
2. VML SEQUENCING MODEL AND CAPABILITIES
3. PARAMETER USAGE IN BLOCKS
4. BLOCK LIBRARY DEVELOPMENT
5. UPLINK PRODUCT SIZE REDUCTION
6. MARS ODYSSEY AEROBRACING
7. MIGRATING AUTONOMY TO MRO WITH VML-2
8. CONCLUSIONS

1. INTRODUCTION: SEQUENCING

Deep space missions require some means for performing commands on a timed basis. The execution of timed spacecraft commands is known as *sequencing* [1].

Sequences are typically represented in a planning interface within the ground system, translated to an uplinkable form, radiated to the spacecraft, loaded by some means, then

¹ 0-7803-7651-X/03/\$17.00 © 2003 IEEE

² IEEEAC paper #155, Updated September 19, 2002

executed. Sequence execution results in the issuance of commands to the spacecraft in some timed order.

Modern spacecraft with real-time operating systems and preemptive task scheduling implement sequencing as a software component. The features implemented in the generic flight software sequencing capabilities profoundly affect the complexity of operating the spacecraft, the size of the team necessary implement sequences, the operations able to be undertaken, the frequency of uplink, and the size of uplinked products. Virtual Machine Language (VML) sequencing carries a number of distinct advantages over more traditional sequencing architectures, in both capability, personnel time, and cost.

This paper provides a brief overview of VML components presented in an earlier paper [2]. It then goes on to present techniques for simplifying operations and minimizing uplink size made possible by VML capabilities. Examples of these techniques are given for the Mars Odyssey and Space Infrared Telescope Facility (SIRTF) missions

2. VML SEQUENCING MODEL AND CAPABILITIES

The VML Flight Component (part of the flight software) follows a paradigm called *procedural sequencing*. At any particular time, exactly one instruction is considered to be "next" on a sequence engine. This allows named sequences which can be called using parameters, easy creation and evaluation of logic constructs, and an implicit ability to branch and loop. Parallelism is achieved by instantiating a fixed number of sequence engines, and explicitly loading and running sequences as threads on those engines. These kind of sequence engines are called virtual machines. They resemble a CPU which can interpret instructions, with memory, dynamic data storage implemented as a stack, and an instruction pointer (see Figure 2-1). Some number of machines are instantiated for the mission. These machines limit the number of threads of execution which can operate in parallel.

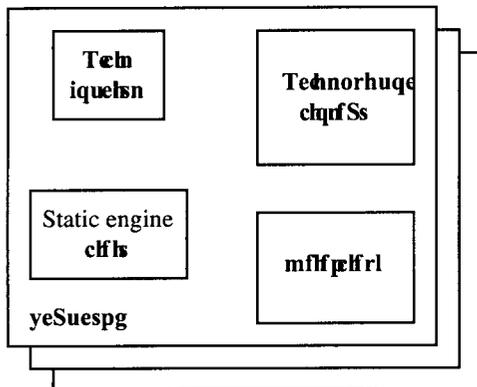


Figure 2-1: Virtual machine sequence engines

Each engine is used for two distinct purposes: storing sequences, and executing sequences. When a file containing a VML module is loaded into an engine, the named

sequences (called *functions*) within that module become available for running on any engine. They are invoked by name rather than in index. In some cases, the function is executed on the same engine in which it is stored. In other cases, the function is executed on a different engine than the one in which it is stored, as shown in Figure 2.2. This would be the case for a function calling another function in a library stored on a different engine.

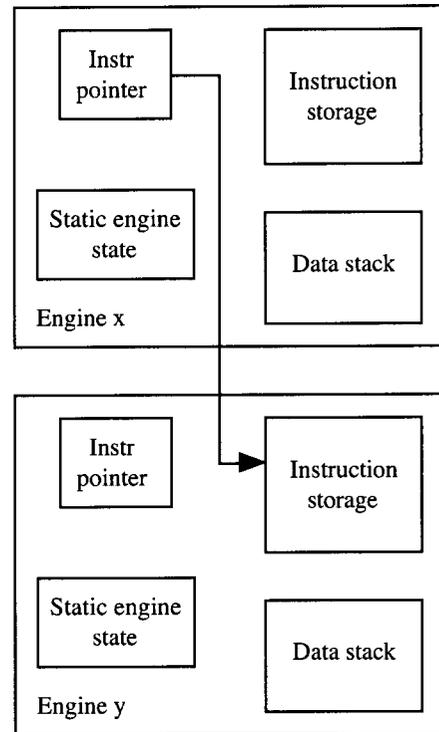


Figure 2-2: Function on engine x running library routine stored on different engine

The VML flight component runs as an embedded task under VxWorks or a similar real-time operating system within the flight software. It works in concert with the rest of flight software, dispatching commands to other flight software tasks in order to affect changes to the spacecraft behavior. The flight component has been developed in a manner compatible with JPL Category A SEI level 3 code, with appropriate methodology, documentation, review, and testing. It is available for use on any NASA mission under license from JPL.

The user creates functions as text using a text editor, or a front-end generating tool. The text is translated by the VML compiler according to generic VML constructs and mission-specific definition files for allowed global variable names and constants. It is then translated to a binary form compatible with the VML Flight Component. Discussion of the VML constructs is presented in their human-readable form here, rather than in binary form.

The procedural orientation of virtual machines allows sequences to be expressed using a number of high-level

language constructs. These constructs form a simple but powerful scripting language in which users can express desired spacecraft activities using named functions, with parameters, a variety of data types, and a rich set of operators.

Comments

Comments within VML code start with a semicolon (;) and continue with all characters until the end of the line is encountered. Comments are sprinkled throughout examples in the following sections.

Modules

A module is a container for one or more functions, along with optional persistent storage. It is bounded by MODULE and END_MODULE keywords.

Modules partition the functionality of the problem space into manageable chunks. For instance, a library module may contain several reusable blocks for controlling instruments and performing communications activities. A daily absolute sequence may reside as the only function in another module, since it is changed out with regularity.

One module exists per file input to the VML compiler. This same module is defined in a flight-compatible format in the VML compiler output file.

Functions: Blocks, Relative Sequences, Absolute Sequences

Function is the generic term for a sequence containing absolute and/or relative time tags. A function is an named executable chunk of VML which may accept parameters and define local variables. Functions may also return values to a calling function. A function starts with the keywords BLOCK or RELATIVE_SEQUENCE for relatively time tagged instructions, and starts with SEQUENCE or ABSOLUTE_SEQUENCE for instructions containing one or more absolute time tags.

A *block* is a function that is intended to be reused (and frequently is stored onboard the spacecraft). Blocks contain only relative time tags so that when they are executed none of the statements are late.

A function that contains only relative time tags between its statements and is not intended for multiple uses is called a *relative sequence*. Relative sequences contain only relative time tags so that they may be kicked off at any time without being late. Relative sequences are frequently autogenerated by ground-based tools for activities like aerobraking maneuvers or daily operations which are planning dependent.

The instructions in a function are bounded between the keywords BODY and END_BODY. The body of the function appears after all parameters, flags, and local variable declarations. A function returns the value UNKNOWN when it encounters the END_BODY. A function returns a specified value when it encounters any embedded RETURN statements.

Parameters

Parameter values are specified immediately after the function declaration as a series of zero or more INPUT or INPUT_OUTPUT keywords, each of which is followed by a locally scoped name. INPUT values are copies of values. INPUT_OUTPUT values reference a variable provided in the call and can return values. Example code is given below.

```
BLOCK acquire_star
  INPUT ra      ;right ascension
  INPUT dec     ;declination
  INPUT file
  INPUT acq_failure_delay
  INPUT_OUTPUT slew_time_result
```

Flags

Flags impose special behavior on a function. At present, only two flags exist: AUTOEXECUTE, which causes the function to automatically begin execution after loading, and AUTOUNLOAD, which causes the module the function is part of to be unloaded when the function completes execution. Flags appear immediately after the function's parameters, but functions with flags typically don't have parameters.

```
BLOCK deploy_antenna
  FLAGS AUTOEXECUTE AUTOUNLOAD
```

Variables

Several different scopes of variables exist in VML sequencing: local, module, and global.

Local variables are defined within functions, and are not visible by name outside the function. Each instance of an executing function contains fresh copies of its local variables. The local variables appear immediately after the parameter list and any flags in a function.

```
BLOCK acquire_star
  INPUT ra      ;right ascension
  INPUT dec     ;declination
  INPUT file
  INPUT acq_failure_delay
  INPUT_OUTPUT slew_time_result
  DECLARE DOUBLE slew_time
  DECLARE INT actual_acq_seconds
```

Module level variables are visible by name to all functions defined in the module, and have persistent data values until the module is unloaded from the engine.

Global level variables are visible by name to all functions, and to flight software. Storage in global variables is persistent. Global variables are used for event-driven sequencing, allowing sequences to respond to environmental changes in the spacecraft.

Variable types include integer, unsigned integer, floating point double, logical, time, and string. Variables may be assigned regardless of type: all assignments of different types result in meaningful values to the assignee. This runtime flexibility removes many constraints with which operators would otherwise have to deal, and results in less complicated sequences.

Time

Time may be specified in several formats: as absolute (wall-clock time), spacecraft time in seconds, and relative time in hour/minute/second form. Each instruction has a time tag which acts as a delay between its execution and the completion of the previous instruction.

Mathematical operations on time are available for calculating delay values needed in `DELAY_BY` and `DELAY_UNTIL` statements. `DELAY_BY` implements a relative time delay for a specified number of seconds. `DELAY_UNTIL` implements a delay until the given time has come to pass. `DELAY_UNTIL` is useful for inserting parameterized absolute time into a relatively time function.

Spacecraft commands

Spacecraft commands are issued by the sequence as either constant commands, or as dynamically built commands.

Constant commands are seen by the flight component as binary patterns to be forwarded without change to the command processing flight software. In the human-readable VML file, commands in a translated form are specified using the `ISSUE` keyword, which causes all characters to the end of the line to be passed to a mission-specific translation tool for embedding the corresponding binary in the uplink product.

Commands may be built on the fly by the VML flight component based on parameter and variable values. Any command defined in the system that can be interpreted by the flight software can be built with a special external call "issue_cmd". Dynamically built commands values are validated according to the same rules built into the ground system, thereby protecting the spacecraft from miscalculations. Invalid command parameter values result in a command error and will abort a thread of execution if aborts are enabled.

Operators

A wide variety of operators is available in VML. Arithmetic operators include absolute value, negation, addition, subtraction, multiplication, division, modulo, and power. Bitwise operators include and, or, exclusive or, invert, shift left, and shift right. Logical operators include and, or, exclusive or, and not. String operators include length, split left (returns substring from start of string up to and including given character position), and split right (trailing substring starting at given character position).

The most complex expression is one including a binary operator, e.g.

```
R00:00:00.1    max := 15.5 + try
```

Expressions with precedence may be incorporated on a future mission.

Conditionals

VML includes an `IF` construct which can be used for choosing a code path based on variable values. This selection allows logical evaluation of multiple conditions

using `ELSE_IF` and `ELSE` statements. The `IF` construct is particularly useful for reacting to parameter values passed into a function, calculated local variables, and global variable values.

```
R00:00:00.1    IF acq = -1 THEN
R00:00:00.1        CALL record_failure_tries, ra, dec
R00:00:00.0        DELAY BY acq_failure_delay
R00:00:00.0    ELSE_IF acq = 0 THEN
R00:00:00.0        DELAY BY 11.5
R00:00:00.0    ELSE
R00:00:00.1        slew_time_result := SPACECRAFT_TIME
R00:00:00.0    END_IF
```

```
R00:00:00.1    IF mode = "high_speed" THEN
R00:00:00.1        ISSUE ...
R00:00:00.1        ISSUE ...
R00:00:00.0    ELSE_IF mode = "mga_800" THEN
R00:00:00.1        ISSUE ...
R00:00:00.1        ISSUE ...
R00:00:00.0    ELSE_IF mode = "mga_100" THEN
R00:00:00.1        ISSUE ...
R00:00:00.1        ISSUE ...
R00:00:00.0    ELSE
R00:00:00.1        ISSUE ...
R00:00:00.1        ISSUE ...
R00:00:00.0    END_IF
```

Loops

A `WHILE` loop is available for structured conditional looping. This construct can be used to repeat a set of statements until a condition becomes `TRUE` or `FALSE`. This construct also allows repeating a set of statements a specific number of times using a counting variable.

```
R00:00:00.1    i := 1
R00:00:00.1    WHILE i <= 10 DO
R00:00:00.1        ISSUE ...
R00:00:00.1        ISSUE ...
R00:00:00.1        i := i + 1
R00:00:00.0    END_IF
```

Event-driven Sequencing

The `WAIT` and `TEST_AND_SET` statements are used to detect events represented by global variables.

The `WAIT` statement suspends operation of the function until its condition is met, then resumes execution of the function at the next instruction. This instruction is particularly useful for non-deterministic sequences which are related to real-time events: rather than assuming worst-case timing, the sequence can be designed to execute with a minimum of wasted time.

A variety of `WAIT` statement constructs exists. The simplest waits for a new value to arrive in a global variable before proceeding. Condition checking can be applied if desired. In addition, the statement can wait until a value arrives which is different than the value of the variable at the start of the statement. An optional timeout to bound the worst-case behavior of the statement is available.

```
R00:00:00.1    v := WAIT gv_a
R00:00:00.1    v := WAIT gv_a > 4
R00:00:00.1    v := WAIT_CHAGE gv_a
```

```
R00:00:00.1    v := WAIT gv_a > 4 TIMEOUT R00:01:00.0
```

Because the continuation from the `WAIT` statement can depend on the value received, and because a wide variety of values could result in proceeding from a `WAIT` statement,

the value of the global variable which resulted in completion of the WAIT is returned for assignment to a local variable. This prevents a race condition between passing the WAIT statement and using the global variable value. Consider the following code fragment:

```
R00:00:00.1    v := WAIT gv_a > 4
R00:00:00.5    IF gv_a = 10 THEN
R00:00:00.1    ISSUE ...
R00:00:00.0    END_IF
```

If the WAIT statement is satisfied at time t with the value 10, but 0.1 seconds later a 3 is written by another sequence or by flight software, the body of the IF statement would not execute.

On the other hand, using a copy of gv_a placed in the variable v would be guaranteed to cause the IF statement to work correctly, even if the value of gv_a changed before reaching the IF:

```
R00:00:00.1    v := WAIT gv_a > 4
R00:00:00.5    IF v = 10 THEN
R00:00:00.1    ISSUE ...
R00:00:00.0    END_IF
```

TEST_AND_SET is used on a counting semaphore for managing shared resource. This is a classical real-time programming access problem [3]. An example use might be to enforce mutually exclusive access to an instrument suite by two separate, non-deterministic sequences. Using a check with a conditional followed by a subtraction leads to an intractable race condition whereby both blocks could complete the IF check before setting the semaphore with the blocking value. TEST_AND_SET allows an integer global variable to be checked and decremented in one instruction, preventing this race.

```
R00:00:00.1    v := TEST_AND_SET gv_a
```

Call: in-line function execution

A function may be executed in-line from another function using the CALL statement. The calling function is suspended, the caller is executed, and then the calling function resumes. The caller may pass parameters to and receive return values from the called function as appropriate using a RETURN statement. Relative time tags for the statement after the CALL indicate the amount of time from the completion of the CALL statement.

Calling does not start a separate thread of execution or use another sequence engine. Instead, resources on the calling engine continue to be used to maintain the thread of execution. Refer back to figure 2-2. This figure shows an engine using code that is stored on another engine (e.g. a master sequence calling a block in a library). The instruction pointer contains a value that indicates code residing on a different engine, but the instruction pointer itself resides on the same engine. The data stack accessed by that engine is always its own.

Calls may be nested arbitrarily deeply, limited only by data stack space on the calling engine. However, call depth greater than about three become difficult to evaluate, and can

make understanding the timing of the sequences problematic.

Spawn: New Thread of Execution

A new thread may be created to run in parallel with existing threads using the SPAWN statement. The spawning function may pass parameters to the spawned function, but no return value is possible. Unlike CALL statements, SPAWN statements complete on the same tick of the clock at which they are invoked. The spawned function is scheduled for evaluation on the next time tick.

Spawning is useful when an activity needs to be initiated that is functionally separate from the initiator, and contains no intrinsic ordering requirements relative to the initiator. For instance, a master sequence may need to initiate downlink at a certain time, but continue to manage instrument observations. If the downlink activities are consolidated in a block, the master sequence can simply spawn the downlink block, then continue on with its usual management tasks. This approach simplifies development of the master sequence by eliminating the interleaving of activities within the body of the sequence. It also allows the functionality of the downlink activity to be abstracted into a block, tested, then repeatedly used.

3.0 PARAMETER USAGE IN BLOCKS

Parameters allow organized information to be passed in to a function. Such functions are intended to be reused. The term for a reusable function is *block*.

Using parameters, functionality can be abstracted and named, and the behavior of the block altered using different parameter values. This bounds the problem domain of a specific block, and makes reviewing the products for problems easier. In addition, calls to the block become easier to review, as the repetitive steps have been abstracted, and only the values of the controlling parameters needs to be considered. This quality was put to use in Mars Odyssey aerobraking sequence generation tool, discussed later.

Parameters also allow a unique flexibility for changing the problem domain. Unlike sequence global variables, which have to be named at the beginning of the mission, parameter names are strictly local in nature. That is, the name is defined in the block. This allows the names and mix of the parameters of a block to be changed without requiring alterations to defined global variable names.

In addition, parameters allow secure transmission of values into a block. Each invocation of the block occurs in an uninterruptible fashion. The parameter values are copied as a snapshot in time, and passed to the invocation. Contrast this to the use of sequence global variables, where an implicit race condition exists due to the visibility of the variables to flight software and other blocks. At any point during the running of the block, a global variable value the block is depending on is subject to alteration, and considerable effort must be expended by designers and reviewers to guard against unintentional alteration.

Once the set of parameters to a block is defined, the interface to the block is consolidated. Use of sequence global variables within a block should be eliminated except for accessing event-driven variables which represent spacecraft state. This limits the side effects produced by the block to spacecraft commands, and removes a potential source of race conditions and logic problems that altering or using global variables would allow.

Parameters may be used to guide block execution logic and make selections for commands sent by the block, alter timing features of the block, or set spacecraft command parameter values of command sent by the block. Each of these uses shall be discussed in turn.

Execution logic

Coupled with the IF statement, parameters can easily determine which paths of code in a block are executed. By comparing parameter values to conditionals in an IF statement, a region of code is chosen for execution or skipped. Different IF constructs and parameter types are particularly good at different kinds of selection.

An IF statement can be as simple as the IF clause followed by an END_IF. This usage is good for taking optional steps within a block based on parameter values or calculated logical values. A logical value passed in can be tested to select a set of statements for execution.

```
BLOCK configure
  INPUT perform_stow
  ...
BODY
  ...
R00:00:00.1 IF perform_stow = TRUE THEN
R00:00:00.1   ISSUE ...
R00:00:00.0 END_IF
...
```

In the above example, a call from a function of the form
R00:00:00.1 CALL configure TRUE, ...
to the configure block would result in the execution of the guarded section of code.

An IF clause followed by an ELSE clause is useful for two-value decision making within a block based on parameter values or calculated logical values.

```
BLOCK sci_configure
  INPUT configure_fast
  ...
BODY
  ...
R00:00:00.1   IF configure_fast = TRUE THEN
R00:00:00.1     ISSUE ...
R00:00:00.0   ELSE
R00:00:00.1     ISSUE ...
R00:00:00.1     ISSUE ...
R00:00:00.1     ISSUE ...
R00:00:00.0   END_IF
```

Parameters may also guide the number of times a certain set of statements is performed. Coupled with the WHILE construct, a parameter can be used to bound the execution of the number of repetitions desired. For instance, in the calibration block shown below, a parameter is passed in

during the call to determine the number of times the science instrument calibration command is sent.

```
BLOCK sci_configure
  INPUT configure_fast
  INPUT calibration_passes
  ...
  DECLARE INT i := 0
  ...
BODY
  ...
R00:00:00.1 i := 1
R00:00:00.1 WHILE i <= calibration_passes DO
R00:00:00.1   ISSUE SCI_CALIB 10, 12.2
R00:00:00.8   i := i + 1
R00:00:00.0 END_WHILE
...
```

Timing changes

VML provides two programmable delays: DELAY_BY for a relative delay of some number of seconds, and DELAY_UNTIL for a delay involving an absolute time tag. The statements can accept parameter values (or even calculated local variables based on parameter values) to allow a block to change its timing behavior in response to invocation changes.

For example, suppose the science configuration block has some amount of time that is supposed to elapse between each calibration pass completion and further calibration steps. The amount of delay could be controlled using a parameter and a DELAY_BY statement.

```
BLOCK sci_configure
  INPUT configure_fast
  INPUT calibration_passes
  INPUT calibration_delay
  ...
BODY
  ...
R00:00:00.1 i := 1
R00:00:00.1 WHILE i <= calibration_passes DO
R00:00:00.1   ISSUE SCI_CALIB 10, 12.2
R00:00:00.0   DELAY_BY calibration_delay
R00:00:00.1   i := i + 1
R00:00:00.0 END_WHILE
...
```

Another use might be found in a maneuvering block. Suppose the set of steps the spacecraft is supposed to take during ascent is dependent on the launch date of the vehicle, but the timing between those steps is fixed. Suppose further that there is an instantaneous launch window. Then the behavior of the block could be governed by passing in the known launch time as a parameter to the block, preventing it from running the useful portions before the appropriate time has come to pass. Such a use might appear as below. The block could be started from the ground and its status verified before launch, but all useful activities would be delayed until after launch has occurred for safety reasons.

```
BLOCK ascent
  INPUT launch_time
  ...
BODY
  ...
R00:00:00.0 DELAY_UNTIL launch_time
...
```

A spawn of the above block from the ground or from a one-use sequence might appear as follows:

```
R00:00:00.1 SPAWN ascent A2010-312T02:17:22.2, ...
```

Command values

Because VML can build commands on-the-fly, parameter values of a block can be directly substituted into commands issued on the spacecraft. This offers enormous flexibility to send commands without using lots of different paths through a block to select the correct constant command.

For instance, suppose a spacecraft block needs to maneuver the spacecraft in some manner that involves slewing to a known right ascension and declination. The flight software has the ability to slew the spacecraft if so desired. The values to use in this command have a very large set of potential states and combinations. By building the command directly, the behavior of the block is simple and succinct.

```
BLOCK acquire_star
  INPUT ra      ;right ascension
  INPUT dec     ;declination
...
R00:00:05.4 EXTERNAL_CALL "issue_cmd" "SLEW_TO",ra,dec
...
```

Non-numeric values can be used in command substitution as well. The string representing a file name might be substituted into a file load command that loads one of the VM block libraries.

```
BLOCK load_secondary_libraries
  INPUT sci_lib
  INPUT eng_lib
...
R00:00:00.1 EXTERNAL_CALL "issue_cmd" "VM_LOAD", 7, sci_lib
R00:00:00.1 EXTERNAL_CALL "issue_cmd" "VM_LOAD", 8, eng_lib
...
```

State values can also be used in command substitution. The string representing a state could be substituted into a command that selects a component. The block might appear as follows:

```
BLOCK contact_dsn
  INPUT antenna
  INPUT bit_rate
...
R00:00:00.1 EXTERNAL_CALL "issue_cmd" "DWN_SELECT", antenna
...
```

The call to this block from a one-use sequence might appear like this:

```
R00:00:00.1 CALL contact_dsn "high_gain", 120
```

Since the dynamic command building software checks that only valid strings result in command dispatches, the block contains the same checks as would be done by a ground expansion of the command. Note that VML also has provisions for checking the result of dynamically building and dispatching a command, and the block could be aborted if a bad command send was attempted.

4. BLOCK LIBRARY DEVELOPEMENT

Blocks residing together for reuse in one module are referred to as a *block library*. Typically, a block library includes parameterized blocks capable of the following:

- initiation and termination of communications

- safe mode recovery assistance
- maneuver operations
- science instrument configuration
- science instrument control

A mission may include one or more block libraries based on the need for specific capabilities during particular portions of a mission. For instance, a planetary mission may require different blocks in one library during launch, cruise, orbital injection, orbital maneuvering, and observation phase. A non-planetary deep space mission may be better suited to having separate libraries simultaneously loaded but with different functional content: engineering, science instruments, and fault protection.

The block development process can be performed iteratively. Because blocks have a well-defined scope and a parameterized interface, the process of coding can be broken down among team members working in parallel. Blocks are treated as software, with a light weight iterative development process involving an architectural breakdown, some specification of purpose and requirements (perhaps in a block dictionary), iterative development of VML code, and iterative testing.

The architectural phase involves making decisions about the need for blocks to do certain tasks, and the name of these blocks. If multiple libraries are present on the mission, the library or libraries to which the block belongs must be determined. Block inclusion when there is one library is trivial. Block inclusion when there are multiple libraries must be explicitly decided, as only one copy of the block can be present within the VML flight component at any instance in time. If a block is needed on a per-phase basis, it should be included in those versions of the block library corresponding to phases where the block is needed. If it is needed across several instrument libraries which may be loaded simultaneously, it should be included instead in a common library divorced from the instrument libraries.

Once entries exist to track each required block in a block dictionary, requirements regarding the functionality of that block are levied. A first pass at parameters to specify the interface for the block is made, including entry conditions regarding allowed values. Basic descriptions of the block's purpose and operating constraints are made.

Next, the block undergoes iterative development by a specific developer. The developer uses an editor to produce a VML source file, which is fed to the VML Compiler to produce a binary. The binary is then executed in a workstation program called Offline Virtual Machine (OLVM).

OLVM allows the user to interactively control the clock, examine variables, step through statement execution, collect human-readable reports showing the execution time of each command in the sequence, and generally debug the logic of the block. OLVM allows the block to be run at several hundred to several thousand times real-time, on the user's own workstation. This tool is a very inexpensive alternative to using a flight-like software testing laboratory with an

flight-like CPU: no special personnel are required, no unique resources are scheduled, and the cycle time between detecting an error and fixing it is measured in seconds rather than days. And, since OLVM consist of the flight code with auser interface wrapper, the fidelity of behavior between OLVM and the flight code cycling on the vehicle approaches 100%.

During execution, the user has the option of using a capture file to record all keystrokes in to OLVM, and all output generated. This capture file is human-readable, but can also act as a script to drive a rerun of the testing performed. This means that unit test scripts can be developed interactively, and kept for later retest. Once ablo ck is considered to be complete to some level of functionality, the block and its testing script can be stored in a revision control system for later extraction and testing.

As the code for a block is completed, it is subject to a design review involving the developer, the operations manager, representatives from systems, and representatives from flight hardware and flight software subsystems. The experts evaluate the timing of commands shown in the logs produced by the developer using OLVM, and consider the content of each command executed. Changes are fed back to the block developer, who repeats the development cycle until all requirements are verified. The interactive test scripts produced during the process are available to retest the block quickly in case of further changes.

As a final validation, blocks are run in a flight-like way in the expensive software test lab and on the vehicle. At this point in the process, however, the blocks tend to work correctly, as they have been repeatedly rung out for errors on the workstation tools.

Test teams can be as small as two or three members due to the fast turnaround nature of the VML development process. Mars Odyssey's sequencing team numbered three, and developed roughly forty blocks. Work tends to be closely coupled with requirements, debugging is simple, and required changes can be fed back very rapidly without the need for expensive software test facilities. The process is also scalable to larger groups (as used on SIRTf) thanks to the ability to easily distribute the OLVM test environment.

5. UPLINK PRODUCT SIZE REDUCTION

The combination of parameterization, variables, a large set of data types, and dynamic commanding makes it possible in some cases to reduce the size of uplinked products over traditional ground-expansion of sequences. One case is that of the Space Infrared Telescope Facility (SIRTf)

Instrument design on SIRTf requires that large commands (hundreds of bytes) be transmitted frequently over a serial line. The exact byte patterns would have to be embedded repeatedly in a controlling sequence performing observations, leading to very large ground-expanded blocks which exceeded available uplink contact time through the Deep Space Network.

During most of the commanding, however, most of the parameters in the instrument commands stays the same. The observatory has the equivalent of a series of modes the instrument needed to be in, with compatible sets of command parameters being repeated transmitted to the instrument. So, rather than ground expand the instrument commands, blocks set the observatory to particular instrument modes, represented by global variable values. Other blocks accept parameters for the instrument command values that vary, and dynamic commands are built from these parameters and the modal global variables. The controlling sequence then invokes a block with a few parameters, initiating a cascade of activity which results in sending commands to the instruments.

The uplink size reduction for this situation is potentially very large. A simple mathematical expression represents the uplink load for any VML file.

$$bytes = \sum_{i=1}^n bytes(instr_i) \quad (1)$$

For a fully ground-expanded sequence with no logic, no use of parameterized blocks, and no use of global variables for state, equation (1) is dominated almost entirely by the command instructions. It can be estimated by the following.

$$\begin{aligned} bytes &= \sum_{i=1}^n (bytes(time) + bytes(opc) + bytes(const) + bytes(cmd)) \\ &= (n)(2 + 1 + 2 + cmdsize) \\ &= (n)(5 + cmdsize) \end{aligned} \quad (2)$$

For a hypothetical command size of 200 bytes, and 1000 spacecraft commands, the number of uplink bytes is therefore

$$\begin{aligned} bytes &= 1000 * (5 + 200) \\ &= 205000 \end{aligned} \quad (3)$$

without considering framing overhead or insertion of error detection. These will scale linearly with the size of the uplink product and therefore will cancel out when examining the percentage savings.

Under VML, physical values that would be held constant in the instrument commanding regime are loaded into sequence global variables, and the values are substituted on-the-fly into dynamically built commands. The block for creating and executing the command is held onboard in a block library, and is not subject to uplink load: once onboard, the instructions do not have to be retransmitted. Instead, a series of calls to this construction block is made from a master sequence, and consists of a block name followed by a series of parameters:

```
R00:00:00.1 CALL instrexec 12, 5, 2, 4
```

The instruction breaks down as a time tag (2 bytes), a CALL opcode (2 bytes) and a name (1 byte for offset in a table), for 4 bytes. Each call parameter requires 3 bytes of overhead plus a 4 byte value for a total of 7 bytes.

$$\begin{aligned} bytes &= callbytes + 4 * parmbytes \\ &= 4 + 4 * 7 \\ &= 32 \end{aligned} \quad (4)$$

So, since equation (1) is again dominated by the spacecraft commanding instructions, 1000 instructions becomes

$$\begin{aligned} \text{bytes} &= 1000 * (32) \\ &= 32000 \end{aligned} \quad (5)$$

Taking the ratio of the ground-expanded case versus the dynamically-issued case results in a savings of

$$\begin{aligned} \text{savings} &= (\text{groundex} - \text{dyncmd}) / \text{groundex} \\ &= (205000 - 32000) / 205000 \\ &= 84\% \end{aligned} \quad (6)$$

The SIRTf numbers differ slightly from the above simple case, but the results were similar. The reduction in uplink brought the spacecraft closer to its target DSN uplink allocation, without developing complex instrument flight software additions or new sequencing flight software requirements.

6. MARS ODYSSEY AEROBRAKING

The Mars Odyssey aerobraking experience is a good example of using parameterized blocks to simplify operations in a repetitive but challenging mission phase. Unanticipated events can require a rapid response in order to maintain safe operations, or even to survive. Communication delays and processing with distant spacecraft can exacerbate the effect of unanticipated threats to spacecraft safety, as the ground is seeing a snapshot of state minutes in the past. Due to its flexible logic, VML is well suited to respond to threatening events.

Aerobraking involves the use of a planet's atmosphere to alter the spacecraft's orbit [4]. After a burn to capture into a highly elliptical orbit, the spacecraft periapsis is lowered into the rarified atmosphere in order to use drag to alter the orbit. Successive passes through the atmosphere reduce the amount of energy of the orbit, and thus the apoapsis of the orbit is lowered. After the orbit is sufficiently lowered, a burn is performed to raise the periapsis of the spacecraft out of the atmosphere, place the spacecraft in a stable orbit.

In order to maximize drag while maintaining control authority, the solar array of the Mars Odyssey spacecraft was presented perpendicular to the direction of travel. The period of Odyssey's orbit around Mars was reduced from eighteen hours down to two hours.

The series of steps for a drag pass is illustrated in figure 6-1. At the beginning of the pass, the spacecraft turns to aerobraking orientation, terminating contact with earth. Mars Odyssey has a hook for the solar array in order to lock it into a mechanically stable orientation. The solar array is placed into the hook. Then the spacecraft begins to slew to maintain the solar array at a known perpendicular orientation to the direction of travel, maximizing drag during the pass and lowering the apoapsis of the orbit. The spacecraft passes through the densest portion of the atmosphere, then exits that portion. The solar array is unhooked and reacquires the sun. Odyssey then slews to earth and initiates contact.

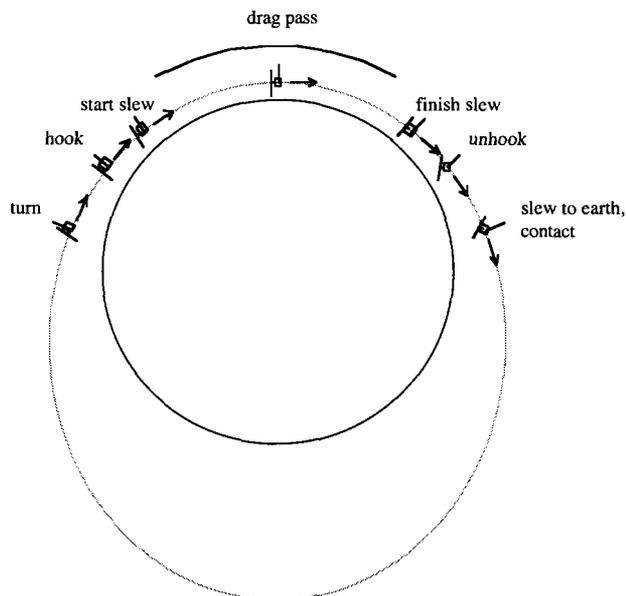


Figure 6-1: Mars Odyssey aerobraking steps

Nominal aerobraking passes

Because of the frequency of the activity and its repetitive nature, the aerobraking activities were abstracted into a named block with parameters. The block implemented each of the steps required for the pass. The block also included parameters for time delays for placing the solar array in the hook, turning to drag attitude, acquiring attitude data when exiting the drag pass, slewing to earth, and completing packet retransmission. The duration of the drag pass and the number of drag pass playbacks of data were also parameterized.

This block was invoked from an automatically generated relative sequence, which was replaced every few passes. The number of passes increased in frequency over time as the orbit was lowered and the orbital period shortened. One to two invocations of the aerobraking block occurred in the autogenerated sequence at the start of aerobraking, whereas six to nine occurred during end-game. By kicking off the parameterized aerobraking block, the team had smaller products to check which contained only the important information that would change from pass to pass.

In order to increase the resilience of the process, the autogenerated sequence which called the aerobraking block covered more passes than was required. That way, when the next version of the sequence was available, it would be started before the previous version had run out, and terminated the previous version. This allowed extra passes to be available for use in case communications problems or other difficulties prevented timely delivery of the invocation for the next drag pass.

End-game aerobraking passes: autonomous pop-up

The last few orbits (or end-game) of aerobraking is particularly sensitive to unexpected drag events, as the natural period of the orbit without the final periapsis raising burn is as short as 24 hours. Unexpected atmospheric blooming can dramatically increase the density of the atmosphere through which the spacecraft flies, causing a larger than expected decrease in orbital altitude and exceeding thermal limits on the solar array due to excess atmospheric friction. Notification of this condition to the ground, and issuing commands, is subject to light speed delays (30 minutes in the case of Mars Odyssey). In addition, the real behavior of the spacecraft grows more and more difficult to model as the spacecraft orbit shrinks. Navigation solutions take two to four hours to produce, which is longer than the last few orbits of the aerobraking process. For these reasons, some sort of onboard detection and response to unexpectedly large aerobraking during end-game was required on Mars Odyssey.

Two separate elements were required for Mars Odyssey: some means of detecting the bloom's effect on the spacecraft, and some means for initiating a burn of the engines to raise the periapsis to a safe altitude.

Detection was performed onboard using flight software to check IMU data [5]. The flight software calculated an estimated time of periapsis during each pass, and placed this value into a sequence global variable. This time shift was used to advance or delay the kickoff of the aerobraking block in a normal pass, allowing better accuracy for the pass. In addition, this flight software could note to high level fault protection that an unusually large shift of the periapsis had been detected. If fault protection determined that the spacecraft was in the end-game, it invoked a separately developed pop-up block which would autonomously burn the engines to raise the spacecraft out of the atmosphere.

The presence of the pop-up block, coupled with the flight software detection mechanism, would have allowed Mars Odyssey to reach a safe condition without the intervention of the ground even in the event of a communications dropout, should an unexpected blooming event have occurred.

7. MIGRATING AUTONOMY TO MRO WITH VML-2

Limited autonomy proved so useful that the next Mars orbital mission, the Mars Reconnaissance Orbiter, will fly with the enhanced VML-2 flight component. This flight component will allow even easier development of limited autonomic functions on the spacecraft.

Autonomous response requires two main components: a logic and decision making feature and access to state information. The logic and decision making features available in VML already provide enough capability to select and execute courses of action. What is needed is access to spacecraft state to provide data on which to base decisions. Ideally, access to the telemetry measurements

would provide a uniquely complete snapshot of the spacecraft's state.

To this end, the VML-2 flight component to be flown on MRO is being upgraded with accessors into the telemetry system. These accessors provide the latest data pushed by the flight software to the sequence engines, and are viewed as read-only global variables. Rather than explicitly designing visibility to certain states and requiring the flight software components to place these values into global variables, all such state will implicitly be available for use should a sequence need to use it.

This approach provides a great deal of flexibility for dealing with problems during the mission. Since telemetry provides almost all the information available to the ground-based operations team, simple decision making can be made onboard where necessary without incurring round-trip light-speed delays. The proximity of the decision making to the state detection makes it possible to handle a whole range of physical effects with short time constants. Small amounts of autonomy developed by the spacecraft operations team can supplement basic flight software capabilities.

8. CONCLUSIONS

The parameterization and block libraries made possible by VML simplify spacecraft operations by allowing functionality of the spacecraft to be abstracted. Uplink product size is minimized by the ability to call blocks that implement most of the command steps. This block is well-suited to a development process including review and test, using inexpensive runtime tools for most of the block development cycle. The block library approach also allows some autonomous operations aboard a mission to be implemented without the development of autonomous flight software.

Procedural orientation allows sequencing to be approached as a structured programming problem, which in turn allows higher quality products to be produced by smaller operations teams. The use of rapid check-out tools like Offline VM reduces the modification cycle time of sequences, allowing the operations development team to produce products on an accelerated schedule.

Simple autonomy depends on access to data. To this end, VML-2 will include the ability to read telemetry points from within a sequence in order to use this data in the decision making logic with blocks. This will enable specifically targeted in-situ decision making capabilities without the development of flight software agents.

REFERENCES

[1] D. Kirkpatrick, "Spacecraft Subsystems: Telemetry, Tracking, and Command", *Space Mission Analysis and Design, 3rd Edition*, pp 381-394, edited by J. R. Wertz and W. J. Larson, New York, Microcosm Press and Kluwer Academic Publishers, 2000.

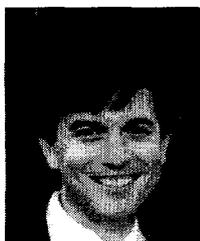
[2] C. Grasso, "The Fully Programmable Spacecraft: Procedural Sequencing for JPL Deep Space Missions Using VML (Virtual Machine Language)", *2002 IEEE Aerospace Applications Conference Proceedings*, March 2002.

[3] Carlton, *Real-time Programming*, New York: Springer, 1988.

[4] B. Mase et. al., " ??? MGS paper", *2002 AIAA ????* Conference Proceedings, ??? 2002.

[5] J. Chapel et. al., "Aerobraking Safing Approach for 2001 Mars Odyssey", *2002 American Astronautics Society, ????* 2002.

Dr. Christopher A. Grasso is a flight software consultant for Stellar Solutions working with the Deep Space Mission Systems directorate of the Jet Propulsion Laboratory. He has developed telemetry, i/o, and sequencing flight components for seven JPL deep space missions, and a series of ground sequencing components.



He earned a PhD in Electrical and Computer Engineering from the University of Colorado in Boulder for work on provably correct real-time system. He now holds an adjunct faculty position at the University of Colorado to teach spacecraft software systems to undergraduate seniors.

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