Non-Linear Sequencing

Erann Gat
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91109
gat@jpl.nasa.gov

Abstract—Spacecraft are traditionally commanded using linear sequences of time-based commands. Linear sequences work fairly well, but they are difficult and expensive to generate, and are usually not capable of responding to contingencies. Any anomalous behavior while executing a linear sequence generally results in the spacecraft entering a safe mode. Critical sequences like orbit insertions which must be able to respond to faults without going into safe mode are particularly difficult to design and verify. The effort needed to generate command sequences can be reduced by extending the vocabulary of sequences to include more sophisticated control constructs. The simplest extensions are conditionals and loops. Adding these constructs would make a sequencing language look more or less like a traditional programming language or scripting language, and would come with all the difficulties associated with such a language. In particular, verifying the correctness of a sequence would be tantamount to verifying the correctness of a program, which is undecidable in general. We describe an extended vocabulary for non-linear sequencing based on the architectural notion of cognizant failure. A cognizant failure architecture is divided into components whose contract is to either achieve (or maintain) a certain condition, or report that they have failed to do so. Cognizant failure is an easier condition to verify than correctness, and it can provide high confidence in the safety of the spacecraft. Because cognizant failure inherently implies some kind of representation of the intent of an action, the system can respond to contingencies in more robust and general ways. We will describe an implemented non-linear sequencing system that is being flown on the NASA New Millennium Deep Space 1 Mission as part of the Remote Agent Experiment.

TABLE OF CONTENTS

1. INTRODUCTION

Spacecraft are traditionally controlled by linear sequences of open-loop commands. This command methodology has a number of drawbacks. Sequences are time-consuming and expensive to generate, they require the ability to predict the state of the spacecraft in detail, and they are brittle in the face of unexpected contingencies. These problems are magnified when faced with the requirement for a critical sequence like an orbit insertion, where unexpected contingencies cannot be allowed to simply send the spacecraft into a quiescent safe mode.

In this paper we describe a new approach to spacecraft sequencing which we call nonlinear or conditional sequencing. Non-linear sequences are composed of closed-loop commands whose contract is to either achieve a certain condition or report that they have failed to achieve it. This design constraint - that all commands report when they have failed to achieve their intended effects - is known as cognizant failure, and it results in new, more complex execution semantics for sequences. However, the complexity can be hidden behind a layer of abstraction (a language). The resulting interface is a rich vocabulary for giving a spacecraft instructions that produce robust, intentional behavior.

2. LINEAR SEQUENCES

Nominal Semantics

The execution semantics of traditional linear sequences are very simple. A command typically consists of an execution time, a command code, and a set of parameters. The command code is an index into a table of subroutines which are called by the spacecraft sequencer in response to sequence uplinks. The repertoire of subroutines in the command table defines the command dictionary for the spacecraft.

Sequence execution nominally proceeds according to a very simple algorithm:

1. Wait for the time of the next command
2. Call the subroutine corresponding to the command code
3. Go to step 1

Note that the semantics of a sequence are order-independent, since every sequence step is tagged with an execution time. A command sequence can therefore be modeled as an unordered set of (time, command} pairs. The "sequentiality" of a sequence does not arise from its lexical structure, but from its execution semantics and the monotonicity of time. Time is the "program counter" for a traditional spacecraft sequence.

The fact that traditional linear sequence execution is fundamentally driven by time has important consequences

for sequence analysis. For example, linear sequences are always guaranteed to terminate, which helps make sequence analysis tractable. The difficulty of sequence analysis derives from the complex effects and interactions of the individual commands, not from complications arising from the theory of computation.

Fault Semantics

Unfortunately, this simple picture is significantly disturbed when anomalous behavior (usually caused by hardware faults) is taken in to account. The spacecraft's normal response to anomalies is to abort the sequence, enter safe mode, and listen for commands from Earth. But turning around a ground command is time consuming, which could endanger the mission during critical sequences like orbit insertions. So for critical sequences the response to anomalies is to attempt to repair the problem (typically using a hard-coded fault response) and then "rewind" or roll back the sequence to an earlier point indicated by a mark.

The mark-and-rollback mechanism significantly complicates the generation and analysis of sequences because it introduces into the execution semantics an internal program counter that does not progress monotonically as does physical time. Sequence steps can no longer be tagged with absolute times. Instead, segments of the sequence that can potentially be rolled back must be tagged with relative times (otherwise rollback would have no effect). The semantics of a sequence are no longer independent of lexical order. The resulting sequence execution algorithm is significantly more complicated:

```
START:
  Set SEQUENCE-POINTER to the start of the sequence
  Set TIME-BASE to the current time
  Set MARK-POINTER to NIL

MAIN-LOOP:
  IF the command at SEQUENCE-POINTER is a MARK command
  THEN
    set MARK-POINTER equal to SEQUENCE-POINTER
  ELSE
    IF the command at SEQUENCE-POINTER is an absolute time command whose
time is in the past
    THEN
      increment SEQUENCE-POINTER
      go to MAIN-LOOP
    ELSE
      Wait for the time T or TIME-BASE plus DELTA-T as appropriate
      Call the subroutine corresponding to the command code
      IF a fault occurred THEN
        IF this is not a critical sequence
        THEN
          enter safe mode
        ELSE IF MARK-POINTER is not NIL
        THEN
          set SEQUENCE-POINTER to MARK-POINTER
          go to MAIN-LOOP
        ELSE
          enter safe mode
```

This algorithm is an order of magnitude more complicated than the simple non-fault case, and it is actually a simplified version of reality. The actual implementation of this algorithm on the Galilleo spacecraft consists of multiple parallel state machines interacting through global variables. There are also additional features that have been left out of this pseudo-code, like limit counters on rollbacks to prevent infinite loops. The development and analysis of critical sequences is similarly complicated, and comensurately expensive. Developing a critical sequence can cost orders of magnitude more than a non-critical sequence.

Local recoveries

One way to address the problems of traditional linear sequences is to extend the envelope of behavior that is considered nominal for a sequence command. For example, the sequence command repertoire for a spacecraft with redundant devices typically includes a separate command to turn on and initialize each device. A hardware failure in a particular device would cause the corresponding initialization command to fail. The probability of failure can be reduced by adding a command whose effect is to initialize some device of a particular class without specifying the particular device to be used. Such a command would be free to cycle through all the instances of the class until it found one that was working properly. It might also try various strategies to recover from failures, like resetting a device. This strategy is known as *local recovery*, and it is used extensively on the Cassini spacecraft [4].

Local recoveries, however, do not change the fundamental semantics of sequence execution. Even commands with local recoveries can fail, and when they do the spacecraft has to fall back on traditional safing mechanisms.

The traditional approach to fault management can be defended on the grounds that faults are rare. Unfortunately, the cost of fault responses is associated largely with *anticipating* faults, not actually responding to them. So while local recoveries can make individual commands more reliable and thus reduce the frequency of safing, it is not clear that it alleviates the high costs associated with critical sequence development.

3. NONLINEAR SEQUENCING

Non-linear or conditional sequencing is an extension to traditional sequencing designed to reduce the cost of sequence development by eliminating the distinction between critical and non-critical sequences. Non-linear sequencing is based on an extended sequence semantics that combines fault and nominal command execution into a unified framework. The result is a system that greatly simplifies the design and analysis of critical sequences while retaining the capabilities of traditional sequences.

The central problem of linear sequences is that the knowledge employed in constructing them has all been "compiled away" in the final sequence. The overall impact of an anomaly during sequence execution could range from a minor glitch to a major disaster, but there is usually no way for the system to tell. For example, consider the following two-step sequence:

> At time X do command A
> At time Y do command B

If something goes wrong during command A, there is no way for the system to know whether the problem has any material impact on command B or not. It is entirely

possible that the problem with A was minor enough that command B could still be executed and have its originally intended effect. It is likewise possible that the failure of A has left the system in a state where executing B would destroy the spacecraft. Because there is no information about the intentions and interdependencies of commands in a sequence, any deviation from nominal behavior requires safing and human intervention.

The solution to this problem is to extend the vocabulary of sequences to include annotations about the interdependencies among steps. For example, one possible extension is to include meta-commands that say something like, "The successful completion of step 1 is required for step 2." Another possibility is to add a conditional control construct, an IF statement, so that one could write something like, "If step 1 completed successfully, do command B," for step 2.

In designing extensions we must be careful to keep in mind the original motivation for keeping the vocabulary of traditional sequences impoverished: it is to make the analysis of sequences tractable. It does not take many extensions to turn sequences into a full-blown Turing-complete programming language, which would make sequences subject to the halting problem and thus impossible to analyze in general. The trick is to make sequences more expressive while at the same time retaining the ability to predict their behavior.

However, to keep things in perspective we must also keep in mind current limitations on sequence analysis. Sequence behavior is predictable with relative ease only in the case of non-critical sequences. Even in that case, the only true guarantee that can be made is that the sequence will *either* produce a particular desired result *or* that the spacecraft will enter safe mode, and even that is not an absolute guarantee, as demonstrated by Mars Observer. It is never possible to make absolute guarantees about sequence behavior, even non-critical sequences, because of the constant potential for non-deterministic hardware faults.

Constructs

Our conditional sequencing infrastructure is based on a few simple foundational constructs. From these we can construct a library of more advanced constructs, some of which are briefly described in section 5. The design is more powerful than traditional linear sequences, but it is not Turing-complete since it does not include arbitrary loops. Analyzing non-linear sequences does not imply solving the halting problem.

Primitives and cognizant failure: Ultimately, any sequencing system must issue primitive commands whose operation is outside the scope of the sequencing system itself. Ideally, the design of the sequencing system should place as few constraints on the structure of primitives as possible. In traditional linear sequences, the only assumption about primitives is that they are subroutines that either achieve an intended effect, or invoke fault protection (which can mean either going into safe mode or rolling back the sequence to the most recent mark).

For a nonlinear sequencing system we must modify this structure somewhat. Because the goal of nonlinear sequencing is to enable more robust responses to contingencies we can no longer allow failed primitives to invoke fault protection directly. Instead, primitives should simply report their status to the sequencer, which will now assume responsibility for taking the appropriate actions. We will assume that primitives are designed so that they exhibit *cognizant failure*, that is, they never fail to report when they have failed. We further assume that there is a library procedure or macro called FAIL that is called by a primitive in order to signal a cognizant failure. FAIL causes the primitive to terminate, either by returning or by throwing an exception (an implementation-dependent design decision). FAIL is also allowed to accept an optional argument containing implementation-dependent information about the failure.

Cognizant failure is the central feature of our design. It commits us to a strong dichotomy between a single nominal course of action and multiple off-nominal courses of action. This design can be contrasted with similar systems which support multiple courses of action without distinguishing between nominal and off-nominal cases [3,5]. Our formulation better reflects the structure of spacecraft operations, where deviations from the nominal are problems far more often than they are opportunities.

Recovery procedures: Cognizant failure is only useful if the system can do something about failures when they happen. A sequence segment designed to recover from a cognizant failure is called a recovery procedure. There is a wide range of alternative designs for recovery procedures. In our implementation, recovery procedures are dynamically scoped, and have a limited number of invocations before they "expire" and propagate a failure out to their enclosing dynamic scope (if any). The outermost recovery procedure, the recovery of last resort, is to enter a traditional safe mode.

The semantics of a primitive invocation in the presence of recovery procedures are:

```
CALL the primitive
IF the primitive terminated by calling FAIL then
  IF there is a recovery procedure for this failure
     AND the recovery procedure's invocation count is > 0
    THEN
      Decrement the recovery procedure's invocation count
      CALL the recovery procedure
    ELSE
      FAIL
```

There are three significant features to note about these semantics:

1. The execution of a primitive in the nominal case is precisely the same in the linear and non-linear case. In both cases, the system simply calls the primitive.

2. Primitive invocation is guaranteed to terminate if its constituents (primitive and recovery procedure) do.

3. In the presence of recovery procedures a primitive invocation will fail if and only if both the primitive and the recovery procedure fail. Thus, the probability of primitive failure in the presence of recovery procedures, P(FPR), is:

$$P(FPR) = P(FP) * P(FR) <= P(FP) \quad (1)$$

where P(FP) is the probability of the primitive failing and P(FR) is the probability of the recovery procedure failing. Thus, the presence of a recovery procedure is guaranteed to only decrease the probability of failure, never to increase it.

Recovery procedures themselves can terminate in one of three ways. 1) They can return, in which case their contract is to have fulfilled the intention of their associated primitive. 2) They can restore the state of the spacecraft to one where the associated primitive can be retried. In the latter case, the recovery procedure must be able to effect a non-local transfer of control to its associated primitive. This is called a RETRY, and it is the non-linear equivalent of a rollback. The difference is that in the linear case, rollback is automatic whenever a failure occurs in a critical sequence. In the non-linear case, a rollback happens only when the recovery procedure specifically requests it. No distinction is made between critical and non-critical cases. RETRY is the only looping mechanism in our formulation.

The third way a recovery procedure can terminate is by failing. In this case, the failure is handled in exactly the same way as a failure in the associated primitive. If a recovery procedure fails with the same failure as the associated primitive and the recovery procedure's retry count is greater than zero then the result of the failure will be to run the same recovery procedure again. The retry count is the mechanism that guards against infinite loops.

(In our implementation [1] there is actually a fourth way that a recovery can terminate: it can effect a non-local transfer of control to the continuation of its dynamic scope, that is, it can cause the block in which it is contained to return normally. This is called an ABORT. The presence of ABORT does not adversely impact the analyzability properties of non-linear sequences, but its utility is questionable so we do not currently include it in our formalism.)

Notice that we have not made any commitment to how recovery procedures are created, how their dynamic scope is established (we only require that a recovery procedure *have* a dynamic scope), or how failures and recovery procedures are associated with one another. These are implementation-dependent design decisions.

Conditions: Conditions are predicates on the state of the system. Conditions can be queried to determined whether they are true or false, and they can be checked for compatibility to see if it is possible for two conditions to be simultaneously true. Conditions are used for two purposes: to construct goal-directed commands out of primitives that may not be goal-directed, and to construct higher-level synchronization constructs that prevent mutually conflicting processes from running simultaneously.

Processes: Spacecraft control requires multiple parallel computational processes. In the past these processes have been written so that time-sharing among processes is hard-wired into the process code itself. We expect future spacecraft to have real operating systems, making manual time-slicing unnecessary. We will assume that any computational process, including primitives, that can be called as a subroutine can also be spawned as a process. A process has at least three states: running, successfully completed, and failed. (We will shortly add two more states: waiting for events, and waiting for time.) We will assume that any primitive command spawned as a process can be asynchronously aborted. We will specifically *not* assume that aborting a primitive necessarily leaves the spacecraft in any kind of reasonable state. For example, aborting an attitude control primitive in the middle could leave a thruster turned on, which if left uncorrected would result in the spacecraft rapidly spinning out of control.

Events: Whenever parallel processes are used, synchronization and interprocess communications (IPC) mechanisms are required. We could choose a standard repertoire of process synchronization and IPC mechanisms (semaphore and message queues, for example), but we choose instead to use a unified model called an *event*. An event is a construct that combines synchronization and IPC functions into a single object. There are two operations on events: a process can wait for an event, or a process can signal an event. When a process waits for an event that process blocks until the event is signaled by some other process. The SIGNAL method accepts an argument, which is returned from the WAIT call to any process waiting for the event. Thus, events can be used both for synchronization and for message passing.

Events are non-queuing. If a producer process signals events faster than a consumer can wait for them then some events will be lost. This places an upper bound on the amount of memory an event can consume. (The memory consumed by an event is not necessarily constant, since it must maintain a list of pending tasks. However, the number of tasks in the system is bounded, so the size of this list is bounded.)

Unwind-protect: Unwind-protect [7] (also called stack unwinding or dynamic-wind) is a standard mechanism for insuring that certain "cleanup" procedures are executed when a dynamic context is exited, even if that exit is caused by a non-local transfer of control. We assume the reader's familiarity with this concept.

Conditionals: The final foundational construct in our formulation is a standard conditional (i.e. an IF statement). Again, we assume the reader's familiarity with the concept.

Specifically excluded from our formulation is a WHILE loop, or any kind of branching that would be equivalent to a while loop. This is the constraint that prevents non-linear sequencing from being a general-purpose programming model, and thus subject to the halting problem.

4. EXAMPLES

There are a host of derived constructs that can be built out of the foundational constructs described in the previous section. In this section we briefly describe a few of the more common ones.

Linear sequencing: Traditional linear sequencing is subsumed by non-linear sequencing as follows. First, any invocation of normal fault protection is replaced by a

cognizant failure (i.e. an invocation of FAIL). For non-critical sequences, the sequence is executed with a recovery procedure whose dynamic scope is the entire sequence. The recovery procedure invokes local fault recovery followed by safing. For critical sequences, every MARK is replaced by a recovery procedure whose dynamic scope is the part of the sequence between the current mark and the next one. The recovery procedure invokes local fault recovery and does a RETRY.

Intentional constructs: ACHIEVE is a construct that combines a condition and the concept of cognizant failure to produce a command whose purpose is manifest in the command. ACHIEVE takes a condition as an argument. Its semantics are:

```
IF the condition is true THEN
  return
ELSE
  perform an action to try to make the condition true
  IF the condition is true THEN
    return
  ELSE
    FAIL
```

The net effect is to guarantee that upon termination either the condition is true or the construct will fail. This is an example of an *intentional* construct because the intent of the command, to make the condition true, is manifest in the command itself.

Task nets: A task-net is a set of parallel processes, each of which has an associated event, and each running in a lexical scope that allows access to those events. The task net also has a "master process" which monitors the progress of the other tasks. Task nets can be used to build constructs like AND-PARALLEL, which runs a number of processes in parallel until either they all finish successfully or one fails, OR-PARALLEL, which runs a number of processes in parallel until either one terminates successfully or they all fail, and WITH-GUARDIAN, which runs a pair of asymmetric processes, one of which performs a task while the other monitors a condition that the task depends on.

Property locks: A property lock is a mechanism for synchronizing tasks so that they do not attempt to achieve mutually exclusive conditions. Property locks have been previously described in detail in [2].

All of these constructs are constructed from the foundational constructs described in section 4. For more examples see [1].

5. SUMMARY

We have described a computational framework for nonlinear sequencing, a methodology for commanding spacecraft that extends the traditional linear sequencing paradigm with additional control constructs. The repertoire of control constructs is designed to make complex sequences easier to write while retaining the ability to analyze sequences without introducing the halting problem. Nonlinear sequencing can be particularly useful in the case of critical sequences in the presence of faults, but it can also make non-critical sequences more reliable and easier to write.

An implementation and application of non-linear sequencing on an actual spacecraft is described in [6].

ACKNOWLEDGEMENTS

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[1] Gat, E., “ESL: A language for supporting robust plan execution in embedded autonomous agents,” in Proc. of IEEE Aeronautics (AERO-98), Aspen, CO, IEEE Press, 1997.

[2] Gat, E. and Pell, B., “Abstract Resource Management in an Unconstrained Plan Execution System,” in Proc. of IEEE Aeronautics (AERO-98), Aspen, CO, IEEE Press, 1998.

[3] Firby, R. J., "Adaptive Execution in Dynamic Domains," Ph.D. thesis, Yale University Department of Computer Science, 1989.

[4] Hackney, J., et al., "The Cassini Spacecraft: Object-Oriented Flight Control Software." Proceedings of the 16th Annual AAS Guidance and Control Conference, Keystone, Colorado, 1993.

[5] Lyons, D. "Representing and Analyzing action plans as networks of concurrent processes," *IEEE Transactions on Robotics and Automation*, 9(3), June 1993.

[6] Pell, B. et al., “A Remote Agent Prototype for Spacecraft Autonomy,” SPIE Proceedings Volume 2810, Denver, CO, 1996.

[7] Steele, G. L. Jr., *Common Lisp, The Language*, second edition. Digital Press, 1990.

Dr. Erann Gat is a senior member of the technical staff at the Jet Propulsion Laboratory, California Institute of Technology, where he has been working on autonomous control architectures since 1988. In 1991 Dr. Gat developed the ATLANTIS control architecture, one of the first integrations of deliberative and reactive components to be demonstrated on a real robot. ATLANTIS was used as the basis for a robot called Alfred which won the 1993 AAAI mobile robot contest. Dr. Gat was also the principal architect of the control software for Rocky III and Rocky IV, the direct predecessors of the Pathfinder Sojourner rover. Dr. Gat escapes the dangers of everyday life in Los Angeles by pursuing safe hobbies like skiing, scuba diving, and flying small single-engine airplanes.