Constraint Checking During Error Recovery

Robyn R. Lutz *
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Johnny S. K. Wong
Department of Computer Science
Iowa State University
Ames, IA 50011

July 26, 1996

Abstract

The system-level software onboard a spacecraft is responsible for recovery from communication, power, thermal, and computer-health anomalies that may occur. The recovery must occur without disrupting any critical scientific or engineering activity that is executing at the time of the error. Thus, the error-recovery software may have to execute concurrently with the ongoing acquisition of scientific data or with spacecraft maneuvers.

This work provides a technique by which the rules that constrain the concurrent execution of these processes can be modeled in a graph. An algorithm is described that uses this model to validate that the constraints hold for all concurrent executions of the error-recovery software with the software that controls the science and engineering activities of the spacecraft. The results are applicable to a variety of control systems with critical constraints on the timing and ordering of the events they control.

INTRODUCTION

This paper presents a technique for checking constraints on the asynchronous software processes involved in error recovery aboard spacecraft. These autonomous processes are constrained at the event level by timing, precedence, and data-dependency rules. Violations of these constraints can jeopardize the spacecraft.

Analyzing all the potential process interactions during spacecraft error recovery is difficult and tedious. A single failure on the spacecraft may at times trigger several different processes whose actions must then be compatible. More than one failure may also occur at a time, causing several error-recovery processes to be invoked. In addition, there is at any time a unique sequence of uplinked commands (instructions to subsystems) executing on the spacecraft. These commands must also be compatible with the actions of the error-recovery software.

The spacecraft software is highly interactive in terms of the degree of message-passing among system components, the need to respond in real-time to monitoring of the hardware and environment, and the complex timing issues among parts of the system. Opportunities for unanticipated process interactions will grow in future missions with advances in hardware, distributed architectures, and "smart" science instruments.

As the opportunity for process concurrency increases, the ability to perform constraint checking on these concurrent processes also must increase. In order to detect hazardous error-recovery scenarios, an improved capability to model and analyze precedence, timing, and data-dependency constraints is needed.

Timing constraints usually have been defined in terms of periodic actions or deadline requirements. This definition is inadequate to model the timing constraints on spacecraft commands. The work described here extends the definition of timing constraints to represent allowable intervals between commands and the execution times of activities initiated by commands. These extensions of recent research results allow more accurate modeling of the required ordering and timing relationships among commands.

The model represents timing, precedence, and data-dependency constraints on spacecraft commands by means of a labeled graph in which the nodes represent commands and the edges represent constraints

* *Proceedings of the NASA Technology 2002 Conference*, December, 1992.

on those commands. This constraints graph, together with a number of potentially concurrent software processes (including the error-recovery processes and the current sequence of uplinked commands), are input to an algorithm, called the Constraints Checker (see Fig. 1). The algorithm tests each edge (representing a constraint) in the constraints graph to determine whether any interleaving of the commands in those processes can fail to satisfy the constraint.

Any documented constraint which is not satisfied in every interleaving of the concurrent processes is recorded in a file for later analysis. The analysis may lead to a change in the documented constraint or to a change in the software. The Constraints Checker is then run again on the corrected input to verify that the constraint can no longer be violated during error recovery by the asynchronous execution of these processes.

This work was performed in the context of the Galileo spacecraft, an interplanetary probe currently journeying to Jupiter. Preliminary results were reported in [8] and additional results in [9]. Ongoing research indicates that the results are applicable to a variety of asynchronous systems with precedence and critical timing constraints.

ERROR RECOVERY

System-level software onboard the spacecraft monitors and responds to failures. The standard definitions are used here of a *failure* as an event in which the behavior of the system deviates from its specification and of an *error* as an incorrect state of the system which must be remedied [12]. The error-recovery processes include those that respond to a loss of uplink or downlink communication, to thermal, power, or pressure anomalies, and to indicators regarding the health of the computers.

Instructions called *commands* instruct the different subsystems of the spacecraft to take specific actions at specific times. Error-recovery processes, composed of commands, are invoked in response to a detected failure. Individual commands are also assembled into groups of time-tagged commands, called *command sequences*, which are periodically sent to the spacecraft from the ground. Command sequences are stored temporarily in the spacecraft's memory until the time comes for each command to execute.

Some command sequences are so critical to the success or failure of the spacecraft's mission that they are labeled *critical sequences*. The command sequences used at launch or to direct Galileo's science and engineering activities at Jupiter are examples of critical sequences.

Should a failure occur during a non-critical sequence, a computer may cancel its activity. A critical sequence, however, must continue to execute even during error recovery. This requirement for concurrent execution of a critical sequence and error recovery drove much of the work presented here.

Constraints are imposed on the commands in an effort to preclude conflicting interactions among the possibly concurrent processes. Some commands can interfere with the effect of other commands if they are executed too closely or too far apart in time. Certain commands must precede or follow other commands to accomplish the desired action. Some commands change the values of parameters used by other commands. Commands relating to power or propellant usage, to temperature or attitude control, to spacecraft or data modes, can endanger the collection of scientific data, a subsystem, or the spacecraft if intervening commands issued by another process leave the spacecraft in an unexpected state.

To restrict the interactions that are allowed, constraints are placed on the interleavings of commands. The flight rules as well as other documented system-performance or operational specifications forbid certain interactions as unsafe. The constraints also describe ordering (precedence) and timing relationships between commands that must be maintained even when processes containing those commands execute concurrently.

Detecting undesirable interactions involving error-recovery processes is especially important since error-recovery software usually executes only when a failure has already been detected onboard the spacecraft. If a critical sequence is executing, the software must quickly (thus, autonomously) reconfigure the spacecraft to the state that is the precondition for the next activity in the sequence.

In addition, because error-recovery software is responsive, it is asynchronous [3]. It may begin execution at any time. In fact, because the spacecraft often is most taxed during the most critical science activity, error-recovery software is most likely to execute when the spacecraft is active. Error-recovery capabilities not only increase the number of concurrently executing processes, but also tend to be executed at the busiest (in terms of process interactions) times. It is also the case that hardware failures due to physical damage tend to be clustered in time [10].

APPROACH

The problem addressed here is how to check that the concurrent execution of the asynchronous processes that cooperate during error recovery satisfy the precedence constraints, maintain data-dependency (read/write) constraints, and satisfy the timing constraints.

The model which was developed allows the command's duration (the length of time required to execute the activity initiated by a command) to be attached to a command. Thus, a constraint of the form, "If command c_j occurs, then the activity initiated by the occurrence of commmand c_i must first have completed," can be represented. This type of constraint is common on both spacecraft and other real-time systems. If the duration of the commanded activity is variable, it is limited by a worst-case time which is represented in the model.

The model takes into account both precedence constraints and timing constraints. Precedence and timing are fundamentally different in that precedence does not require a notion of duration [11]. Most methods that currently exist to model precedence constraints do not incorporate timing requirements and so are inadequate for modeling the timing constraints on spacecraft. Similarly, many techniques that are currently available to model timing constraints tend to ignore precedence constraints.

A wide variety of powerful formalisms exists to model the specifications and behavior of real-time systems. See, e.g., [1, 4, 5, 6, 7, 13, 14, 15]. However, none of the available methods readily translates to the domain of validating error recovery on spacecraft. Some techniques consider both timing and precedence constraints but define timing constraints only in terms of periodic events (e.g., sampling rates), fixed execution times for events, and deadline or timeliness requirements. This provides too limited a model for the aperiodic and interval timing constraints on spacecraft commands.

Requirements are often modeled in terms of a lower time bound (after which an event may occur) and an upper time bound (by which the event must occur). In contrast, on the spacecraft there is often a need to model an operational constraint such as an interval within which an event is permitted to occur (but perhaps won't) and outside of which interval the event must never occur. This distinction between a "hard" time constraint (an interval within which the action should be taken) and a "soft" time constraint (an interval within which the action may occur) [2] is often absent in formal models.

The work described here brings together the study of real-time constraints with the study of precedence and data-consistency constraints.

MODELING THE CONSTRAINTS

The constraints that exist at the command level on the asynchronous execution of the spacecraft error recovery are modeled via a constraints graph. A constraints graph is a directed graph G=(V,E) in which each node c_i, $c_j \in$ V is labeled by a command and each edge e $\in$ E is labeled by a constraint. The edge (c_i, c_j) is also written $c_i \longrightarrow c_j$ and is drawn as in Fig. 2.

Intercommand constraints are rules that govern the ordering or timing relationships between commands. There are three main types of intercommand constraints: timing constraints, precedence constraints, and data-dependency constraints. Fig. 3 presents an example of each of these three constraint types.

Intercommand timing constraints are safety properties. They impose a quantitative temporal relationship between the commands by asserting that "every c_i can only occur with timing relationship τ to c_j."

Precedence constraints enforce an ordering on the commands and so involve functional correctness, a concern of safety properties. Precedence constraints also involve liveness properties since they assert that if one command occurs, then another command must precede it: "If c_j occurs, then a c_i must precede it."

If a timing constraint exists between commands c_i and c_j, either command can legally occur alone. However, if a precedence constraint exists requiring command c_i to precede command c_j, then c_j cannot occur in isolation from c_i.

Data-dependency constraints involve restrictions placed on the order of commands when two or more processes access the same variable and at least one process changes the value of the variable In such cases a concurrent execution of the processes can lead to a result different from the sequential execution of the processes. To forestall the data inconsistency that could result from this, a data-dependency constraint is used to specify the order in which the commands that read/write the variable must occur. Such a data-dependency constraint is represented as a precedence constraint.

CONSTRAINT CHECKING

The constraints graph and a set of processes (time-tagged lists of commands) are input to the Constraints Checker. The Constraints Checker fixes one process' timeline and determines the range of start times that the fixed timeline and the constraint represented by each edge impose on the other processes. Each edge type is associated with an algebraic predicate which relates the time of issuance of the commands which are the edge's endpoints to the constraint represented by the edge. The Constraints Checker tests whether the appropriate predicate holds for each edge in the constraints graph. An edge which fails to satisfy the required predicate is flagged. In this case some scheduling of the processes can cause the constraint represented by that edge to be violated.

The Constraints Checker makes an assertion about the allowable start times of the process whose timeline is not fixed. It makes this assertion based on the edge type currently being surveyed, on the constraint represented by the edge, on the offset between the processes' start times and their issuances of c_i and c_j, and on the fixed start time of one process. If the Constraint Checker's assertion concerning when the other processes should start is inconsistent with the user-provided range of start times, then the edge is flagged.

If the edge was flagged due to erroneous information in one of the edge or node labels, the user can readily correct the input data and run the Constraints Checker again to verify the adequacy of the correction. If the edge is flagged due to a problem with the existing error-recovery response, the data output with the flagged edge helps the user identify the problem. The user responds by shifting the processes' timelines or by curtailing the concurrency that allowed the intercommand constraint to be violated. The goal is to adjust or limit the concurrent execution of the processes so that the edge will not be flagged in a subsequent run.

After all the edges have been surveyed, the Constraints Checker computes for every distinct pair of processes the range of safe start times of one process relative to the other. Within this time interval the intercommand constraints are satisfied.

RESULTS

The error-recovery scenarios chosen to evaluate the Constraints Checker involved failures during the execution of the critical sequence of commands that controls the Galileo spacecraft's arrival at Jupiter. The activities of the processes that must cooperate during error recovery are highly constrained due to the complexity and time criticality of the engineering and science activity during the planetary encounter. There are thus many opportunities for constraint violations during error recovery.

Experience with this method indicates that the Constraint Checker's major benefit is early detection of significant inconsistencies between the design and the constraints. The error-recovery scenario that was analyzed addressed the complicated timing issues involved when a critical sequence of commands had to continue execution during recovery from a hardware anomaly. The constraints graph input to the Constraints Checker consisted of 40 nodes and 40 edges (20 precedence and data-dependency edges and 20 timing edges).

The results were as follows. The Constraints Checker flagged seven intercommand constraint violations (four precedence constraints and three timing constraints). One precedence edge was flagged because a global variable could in some schedules be used before it was updated. Two other edges were flagged because a certain command that was required to precede another did not occur. (One of these violations was later traced to outdated documentation). Another flagged edge was, in fact, not violated but appeared to be, since the associated command appeared six hours earlier.

Two of the three timing edges that were flagged involved a timing discrepancy between the requirements and the code. The third timing edge that was flagged resulted from the unforeseen consequence of a data field taking on a value which is possible, but forbidden in operations.

Another eight errors, involving incorrect or inconsistent documentation, were identified during construction of the constraints graph. Seven of these eight errors were significant enough to have caused inaccuracies in the constraints graph and in the results of the Constraints Checker.

The intercommand constraint violations flagged by the Constraints Checker involved either discrepancies between the constraints and the code or unforeseen consequences of unlikely but possible error-recovery scenarios. The Constraints Checker appears to be useful in enhancing the developers' ability to visualize abnormal scenarios and in flagging constraint violations that occur only in some subset of the possible error-recovery responses.

The code analyzed here was a baseline version, rather than the most current flight version. This choice was made in order to provide code that had been well thought out but not tested. It is at this intermediate stage of the development process, when the intercommand constraints have been initially documented but the details of the design and the timing are still evolving, that the Constraints Checker may be most effective.

A version of the Constraints Checker underwent initial development for use on the later-canceled Comet Rendezvous/Asteroid Flyby (CRAF) spacecraft. The Constraints Checker's function was to serve as a software-development tool to analyze and validate scenarios involving the interactions among concurrent processes during error recovery.

POTENTIAL APPLICATIONS

The method outlined here is useful for addressing related problems in domains other than spacecraft. A primary concern in evaluating the safety of a complex, embedded system is whether error recovery can result in the violation of constraints on that system. More generally, the design of event-driven systems often involves analyzing whether the concurrent execution of processes with unpredictable start times can jeopardize the system. Such issues are readily investigated with this method. The Constraints Checker can also function as an extension to existing simulation or CASE tools to provide more accurate validation of complex timing constraints.

The Constraints Checker is also well suited to operational situations in which a portion of the control software is regularly replaced. On the spacecraft the sequences of commands are examples of this "temporary" software. On the space station, for example, procedures to sequence activities outside the astronauts' responsibilities will be regularly uplinked from the ground. The proposed method could ease the operational difficulties of quickly checking that new or temporary software is safe and will not conflict with the "permanent" software.

Commercial applications of this method could range from safety-critical process-control systems to event-driven flight control or command-and- control systems. Current tools often model and test only periodic or deadline timing constraints. The proposed method offers the capability to quickly and accurately model and check that even aperiodic and interval constraints among events will always hold in the system.

CONCLUSIONS

This paper has shown how to construct a constraints graph that models precedence, timing, and data-dependency constraints. The constraints graph provides a means of visualizing the intercommand constraints that must be satisfied by every concurrent execution of processes during error recovery. This paper has also described a constraint-checking tool that, given a constraints graph and a set of processes, detects violations of the modeled constraints.

The Constraints Checker is designed specifically to help answer the question of whether existing system-level error recovery is adequate. It offers a flexible, embeddable, and relatively simple alternative to simulation of error-recovery scenarios. In the context of the spacecraft, the algorithm identifies the unexpected effects resulting from the interleaving of error-recovery processes and mission-critical sequences of commands. In a broader context, the research presented here is part of an ongoing effort to investigate the behavior of concurrently executing processes subject to precedence and timing constraints.

Acknowledgments

The first author thanks Chris P. Jones of the Jet Propulsion Laboratory for many helpful insights.

The work of the first author was started at Iowa State University, supported by grant NGT-50269 from the National Aeronautics and Space Administration, and was completed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] R. Alur, C. Courcoubetis, and D. Dill, "Model-Checking for Real-Time Systems," in *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*. Los Alamitos, CA: IEEE Computer Science Press, 1990, pp. 414-425.

[2] A. A. Bestavros, J. J. Clark, and N. J. Ferrier, "Management of Sensori-Motor Activity in Mobile Robots," in *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*. Los Alamitos, CA: IEEE Computer Society, 1990, pp. 592-597.

[3] R. H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 811-826, August 1986.

[4] S. Cheng, J. A. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *IEEE Real-Time System Symposium*. New Orleans, LA: 1986, pp. 166-174.

[5] J. E. Coolahan, Jr., and N. Roussopoulos, "A Timed Petri Net Methodology for Specifying Real-Time System Timing Requirements," *International Workshop on Timed Petri Nets*. Silver Springs, MD: IEEE, 1985.

[6] T. A. Henziger, Z. Manna, and A. Pnueli, "Temporal Proof Methodologies for Real-Time Systems," in *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, 1991, pp. 353-366.

[7] F. Jahanian and A. K.-L. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering* vol. SE-12, pp. 890-904, Sept. 1986.

[8] R. R. Lutz and J. S. K. Wong, "Validating System-Level Error Recovery for Spacecraft," *AIAA Computing in Aerospace 8*, vol. 1. Washington: AIAA, 1991, pp. 69-76.

[9] R. R. Lutz and J. S. K. Wong, "Detecting Unsafe Error Recovery Schedules," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 749-760, August 1992.

[10] J. D. Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems, The Alpha Kernel.* Boston: Academic Press, 1987.

[11] A. Pnueli and E. Harel, "Applications of Temporal Logic to the Specifications of Real Time Systems," in *Formal Techniques in Real-Time and Fault-Tolerant Systems.*, Ed. M. Joseph. Berlin: Springer-Verlag, 1988, pp. 84-98.

[12] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *ACM Computing Surveys*, vol. 10, pp. 123-166, June 1978.

[13] F. H. Vogt and S. Leue, "The Paradigm of Real-Time Specification Based on Interval Logic," in *Proceedings of the Berkeley Workshop on Temporal and Real-Time Specification*, Eds. P. B. Ladkin and F. H. Vogt. Berkeley, CA: International Computer Science Institute TR-90-060, pp. 153-178.

[14] J. M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 23, pp. 8-26, Sept. 1990.

[15] J. Xu and D. L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, vol. 16, pp. 360-369, March 1990.