# Validating System-Level Error Recovery for Spacecraft

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

## Abstract

The system-level software onboard a spacecraft is responsible for recovery from communication, thermal, power, 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 paper 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 events on the spacecraft.

## 1 Introduction

Spacecraft software processes are composed of commands. Commands are instructions to the spacecraft to take specific actions at specific times. The processes are constrained at the command level by documented rules. Intercommand constraints are rules that govern the ordering of the commands, the timing relationships that must exist between certain commands, and the commands' access to shared variables. Every possible interleaving of commands from the asynchronous processes that cooperate during error recovery must satisfy these intercommand constraints. A failure to do so can jeopardize the collection of scientific data, a spacecraft subsystem, or even the spacecraft itself.

The research described here provides a technique for validating that the intercommand constraints are satisfied during error recovery onboard the spacecraft. The work was carried out in the context of the Galileo spacecraft. The examples are drawn from Galileo's error-recovery processes and from the preliminary sequence for Galileo's planetary encounter with Jupiter. Ongoing research is evaluating the applicability of these results to other spacecraft as well as to other asynchronous systems with critical precedence and timing constraints.

### 1.1 The Problem

Validating the system-level error recovery on the spacecraft requires the capability to analyze the possible interactions among concurrent processes. This is a difficult endeavor. A single fault on the spacecraft may at times trigger several different processes whose commands must not interfere with each other. More than one fault 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 executing on the spacecraft. Each sequence is a set of time-tagged commands relating to the upcoming mission activities. A sequence is periodically sent to the spacecraft from the ground and stored in the spacecraft's temporary memory until the time comes for each command in it to execute.

Some stored sequences of commands are so critical to the success or failure of the mission that they are labeled "critical sequences." The sequences of commands used at launch or to direct Galileo's probe relay and orbital insertion activites at Jupiter are examples

*Proc of AIAA Computers in Aerospace VII Conference*, Oct, 1991. This work was supported in part by NASA Grant NGT-50269. Author's mailing address is Department of Computer Science, Iowa State University, Ames, IA 50011. 

of critical sequences. A critical sequence, unlike a non-critical sequence, must continue to execute even during system-level error recovery.

The constraints imposed on the commands are 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.

### 1.2 Background

The solution to the problem of validating system-level error recovery must take into account both precedence constraints (the order in which commands occur) and timing constraints (the time between commands). These two types of constraints are fundamentally different in that precedence does not require a notion of duration [16]. Consequently, the tools that currently exist to model precedence constraints tend to ignore timing requirements and to be inadequate for modeling the timing constraints on spacecraft [6, 12, 15].

On the other hand, many techniques that are currently available to model timing constraints tend to ignore precedence constraints. Some techniques consider both timing and precedence constraints, but their definition of timing constraints only in terms of periodic events (e.g., sampling rates), fixed execution times for events, and deadline or timeliness requirements provides too limited a model for the timing issues constraining spacecraft commands [2, 19, 20, 22].

Discrete event simulation or emulation models the states of the spacecraft and the events that cause a transition from one state to another [5, 7, 13]. A simulation is run iteratively with a series of different combinations of process start times or fault-injection times. Each simulation produces a timeline which can then be analyzed (automatically or by hand) for intercommand constraint violations. Simulation is a large-scale technique, costly in terms of space (since there are many states that must be modeled and stored) and in terms of time (since typically many iterations must be made to test a scenario).

A wide variety of powerful formalisms exists to model the specifications and behavior of real-time concurrent systems. Many of these formalisms address to some degree the problem of checking timing constraints. However, none of the available methods readily translates to the domain of validating error recovery on spacecraft.

Petri net extensions model periodic events and deadlines [4, 9]. Automata-based methods model processes as a machine and try to prove a predicate (which may involve upper and lower time bounds on events) true for the reachable states in the machine [11]. Real Time Logic models the timing aspects of a system specification to establish timing properties (periodic events and deadlines) [8].

Various extensions to temporal logic and temporal logic model checking have been developed to formally describe timing requirements and to verify automatically that the system satisfies them [16, 3]. These methods provide a good basis for specifying timing requirements but are either more ambitious (in that they model states) or less expressive (in that they only model a subset of timing constraints) than is needed for the spacecraft.

The work described here discusses many of the same timing issues addressed by recent work in interval temporal logic [17, 18]. However, the emphasis there is on specifying and verifying system requirements (what the spacecraft can do) while the emphasis here is on verifying operational constraints (what the spacecraft may do). Autonomous error recovery onboard the spacecraft requires the spacecraft to have capabilities that it is only permitted to exercise subject to certain constraints. This paper offers a partial solution to the problem of modeling distance in time within the context of spacecraft error recovery.

Operationally, several stages of the software development process involve analysis of the interactions among spacecraft processes. The subsystem software designers analyze the interactions as the software is designed and written. During subsystem testing, system integration, and prelaunch testing, a limited set of the interactions is simulated on testbed hardware.

The process of developing sequences of commands also includes checks for constraint violations. Limited simulation of the sequence may occur. Critical sequences–those which must remain active even during a spacecraft error recovery–receive special attention during the sequence-development process. Similarly, changes and updates to the error-recovery software are checked and double-checked. Since this software is usually invoked only when a failure has already occurred on the spacecraft, possibly leaving the system in a vulnerable state, it is essential that the error recovery occur quickly, correctly, and predictably.

With the software tools currently available, analyzing the possible intercommand constraint violations is a tedious and complex task. The interactions among the error-recovery processes and the sequences of commands are both important to the spacecraft's safety and difficult to visualize fully. The method described here, which provides a graphical representation of the relevant command constraints and an algorithm to aid in detecting unsafe interleavings of the commands in the processes, can facilitate and improve the analysis of whether the intercommand constraints are always satisfied.

### 1.3 Proposed Solution

The model proposed here represents precedence, timing, and data-dependency constraints on spacecraft commands by means of a labeled graph in which the nodes represent commands and the edges represent documented constraints on those commands. A classification of the types of intercommand constraints is presented in Section 2.

Section 3 describes an algorithm, called the constraints checker, which accepts as input the constraints graph, a number of potentially concurrent processes, and user-provided intervals during which the processes can execute. The constraints checker associates each edge type in the graph with an algebraic predicate. The predicate 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 graph. An edge which fails to satisfy the required predicate is flagged as potentially unsafe. In that case there exists some interleaving of the commands in the processes which can cause the constraint represented by that edge to be violated. Section 4 offers concluding remarks in the context of the spacecraft.

## 2 Modeling the Intercommand Constraints in a Graph

Intercommand constraints are rules that govern the ordering or timing relationships between commands. There are two main classes of intercommand constraints: precedence constraints and timing constraints. Data-dependency constraints are typically represented as precedence constraints.

The classification into timing and precedence constraints corresponds loosely to the standard formal division of program correctness into safety properties and liveness properties. Safety properties can be stated informally as "nothing bad ever happens" and liveness properties as "something good eventually happens" [21].

### 2.1 Timing Constraints

Intercommand timing constraints are safety properties. They impose a quantitative temporal relationship between the commands. If $c_i$ and $c_j$ are distinct commands and $t_1$ and $t_2$ are time parameters, then the following five types of timing constraints can occur: (Examples are paraphased from Galileo documentation.)

1. Minimum-interval constraint. "If $c_i$ occurs then $c_j$ cannot occur within time $t_1$ of it." An example is, "A SCAN command shall not be sent within 10 seconds of a SLEW command."

2. Outside-interval constraint. "If $c_i$ occurs, then $c_j$ can only occur outside a range $t_1$ to $t_2$." An example is, "The time separation between powering on the S-Band transmitter and powering on the X-Band transmitter shall be either less then one-half minute or greater than 6 minutes." Note that a minimum-interval constraint is a special case of an outside-interval constraint with $t_1$ =0.

3. Forbidden combination constraint. "If $c_i$ occurs then $c_j$ cannot occur." This is a special case of an outside-interval constraint in which $t_1$ = $-\infty$ and $t_2$ = $+\infty$. An example is that if one of the two optics heaters is commanded on, then the other optics heater cannot be commanded on.

4. Maximum-interval constraint. "If $c_i$ occurs, then $c_j$ can only occur within time $t_1$ of it." An example is, "The initialization command shall be sent within 8 seconds after turn-on."

5. Inside-interval constraint. "If $c_i$ occurs then $c_j$ can only occur within a range $t_1$ to $t_2$." An example is, "Each Low-Gain Antenna Motor power on command shall be followed no sooner than 9 seconds and no later than 30 seconds by the Low-Gain Antenna motor power off command." Note that a maximum-interval constraint is a special case of an inside-interval constraint with $t_1$ =0.

In some cases a "nominal-to-worst-case" execution time may be documented for command $c_i$. This pair of values indicates the expected time that it takes command $c_i$ to complete as well as the longest completion time for which the developers must plan.

The worst-case execution time for $c_i$ can be considered to be an additional timing constraint on when command $c_j$ can occur. In the case of a minimum-interval timing constraint, call it $t_1$, the worst-case execution time can be added to $t_1$ in the graph. However, in the case of a maximum-interval timing constraint, adding the worst-case execution time to the initial timing constraint can mask a violation of the constraint by lessening the time interval between commands $c_i$ and $c_j$. Instead, the nominal execution time (or, to be conservative, 0) is added to the timing constraint. The cases for the inside-interval and the outside-interval edge types follow accordingly. In this way the time required to execute the command $c_i$ can be modeled.

## 2.2 Precedence Constraints

While intercommand timing constraints are clearly safety properties, intercommand precedence constraints contain aspects of both safety and liveness properties [11]. Precedence constraints enforce an ordering of 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." An example is, "Spin Detector B can only be powered on after Spin Detector A is turned off."

Thus, whereas timing constraints assert that "every $c_i$ can only occur with timing relationship $\tau$ to $c_j$", precedence constraints assert that "for every $c_j$, there must exist a $c_i$ that precedes 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 between commands $c_i$ and $c_j$, for example, "If $c_j$ occurs then $c_i$ must precede it," then $c_j$ cannot occur in isolation from $c_i$.

Many constraints of the form "State A is a precondition for issuing command $c_j$," where state A can be commanded, can be adequately though imperfectly modeled as precedence constraints. An example is the rule that "10-Newton thruster firings must be performed with the scan platform in a safe position." In the context of the spacecraft, it suffices to ensure that the command to place the scan platform in a safe position precedes the command for thruster firing. By representing the state (scan platform in the safe position) by a command (place the scan platform in the safe position), the constraint can be modeled in the graph. If the required state cannot be commanded (e.g., "low-radiation environment"), then it cannot be modeled in the graph.

A similar abstraction occurs with forbidden-combination constraints. For example, the constraints graph approximates a constraint forbidding the issuance of a command to turn on optics heater B if optics heater A is on by means of an edge that forbids the issuance of a command to turn on optics heater B following the issuance of a command to turn on optics heater A.

## 2.3 Data-dependency Constraints

Data-dependency constraints are 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 [1]. 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 expressed as a precedence constraint.

## 2.4 The Constraints Graph Model

The intercommand constraints are modeled in a directed graph, called a constraints graph. Each edge in the constraints graph represents a constraint on the relationship of the commands which form that edge's nodes. For example, the timing constraint "A SCAN command shall not be sent within 10 seconds of a SLEW command" would be modeled as a labeled edge from a node labeled SLEW to a node labeled SCAN.

The subgraph composed of just the precedence edges must be acyclic since otherwise every occurrence of a command in a cycle would have to be preceded by another occurrence of the same command.

A node (representing a command) has three labels associated with it: the command mnemonic that identifies the command, the nominal (predicted) execution time of the command, if any, and the worst-case execution time of the command, if any.

An edge (representing a constraint) has five labels associated with it: the type of edge, up to two time fields, a key to where the constraint is documented, and the variable associated with the edge, if it represents a data-dependency constraint.

The constraints graph is sparse, and can be stored in O($\mid V \mid + \mid E \mid$) space via an adjacency list representation [14].

# 3 Detecting Constraint Violations During Error Recovery

## 3.1 Overview of the Constraints Checker

The constraints graph and a set of processes (time-tagged lists of commands) are input to the constraints checker. Because there is little branching in spacecraft system-level error-recovery processes and command sequences, the commands in the alternative paths of a process can be interleaved for input to the constraints checker. The constraints checker algorithm 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 process' start time. 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 such a case some interleaving of the commands in the processes can cause the constraint represented by that edge to be violated. The algorithm and its variables are described briefly here, with details given in [10].

## 3.2 The Constraints Checker Algorithm

Let $(c_i, c_j)$ be an edge, with process $P_i$ issuing command $c_i$ and process $P_j$ issuing command $c_j$. If an edge violation occurs as the result of interleaving several processes, that same edge violation still occurs as the result of interleaving the two processes that issue the edge's nodes. Any interleaving of two processes that can occur via the concurrent execution of more than these two processes can occur with the concurrent execution of only these two processes. It thus suffices to check for each edge every ordered pair $(c_i', c_j')$ where $c_i'$ is an instance of $c_i$, $c_j'$ is an instance of $c_j$, and $c_i'$ and $c_j'$ are issued by distinct processes

The constraints checker distinguishes between precedence edges and time-constrained edges. A precedence edge requires that every $c_j$ be preceded by a $c_i$. The algorithm in effect examines all instances of $c_i$ for each instance of $c_j$. To capture the existential quantifier in the predicate for a precedence edge ("There exists a $c_i$ that precedes this $c_j$"), the constraints checker refers to information from the user. The user decides which of the processes can be considered to always execute in the current scenario. A $c_i$ in a process that may or may not execute cannot satisfy a precedence edge.

The variable Gci is the earliest time by which a process that is guaranteed to execute can guarantee that $c_i$ will occur. Gci is user-provided for $c_i$ on precedence edges. Gci=$\infty$ if no such guarantee can be made. The constraints checker uses Gci to detect a possible violation of a precedence edge. The precedence edge requires that $c_j$ not occur before Gci.

Let StartPj be the actual start time of the process issuing $c_j$, Deltaj be the time interval from StartPj until command $c_j$ is issued, and EInitPj be the earliest time at which process $P_j$ can start. When StartPj is fixed, the command $c_j$ occurs at StartPj + Deltaj. However, when the start time of the process issuing $c_i$ is fixed and the value of StartPj is not fixed, the algorithm then considers the earliest time at which a $c_j$ can occur—namely at EInitPj + Deltaj. If the earliest time at which a $c_i$ can occur precedes or equals Gci, then the edge is flagged. This means that the constraint that the edge represents is not always satisfied by the process interactions.

A time-constrained edge, on the other hand, requires that if command $c_i$ occurs, then command $c_j$ does not occur within some time interval. Whereas a precedence edge requires a $c_i$ for every $c_j$, in a time-constrained edge the presence of one command does not require the presence of the other. The predicates for the time-constrained edges are of the form "every $c_j$ that follows this $c_i$ must satisfy a certain timing constraint."

For a timing constraint, define the time interval Poss to be the range of possible times at which the process whose timeline is variable may start according to the user. Define Safe to be the set of safe times at which that process may start, that is, the set of all times that satisfy the predicate for that edge type. Then a time-constrained edge is satisfied if the set of possible times for the process whose timeline is variable is contained within the set of safe times for that process: Poss $\subseteq$ Safe.

Each of the five time-constrained edge types described in Section 2 has associated with it an algebraic predicate that is used to determine the values of the set Safe. For example, the predicate for a maximum-interval time-constrained edge ("command $c_j$ cannot occur more than time $t_1$ after command $c_i$") is: StartPi + Deltai < StartPj + Deltaj < StartPi + Deltai + $t_1$. The predicates for the other edge types follow accordingly [10].

The Constraints Checker Algorithm:

```
for each edge (c_i,c_j) in the constraints graph
  begin
    if edge type = precedence then
      begin
        for each instance c'_j of c_j
          fix P_i or P_j according to the rules;
          if P_j is fixed
          and (StartPj + Deltaj) ≤ Gci
            then output warning flag;
          if P_i is fixed
          and (EInitPj + Deltaj) ≤ Gci
            then output warning flag
      end
    else if edge type = time-constrained edge then
      begin
        for each pair (c'_i, c'_j) of instances of
        c_i and c_j where P_i ≠ P_j
          fix P_i or P_j according to the rules;
          if Poss ⊈ Safe
            then output warning flag
      end
  end
```

The time complexity of the algorithm is O($| E | \cdot n^2 \cdot \bar{k}^2$), where $E$ is the set of edges, $n$ is the number of processes, and $\bar{k}$ depends on the number of instances of a command per process [10]. The constraint checker's runtime is reduced by the facts that the constraints graph is sparse, that there are usually few instances of each command per process, that there is minimal branching in the processes, and that the number of concurrent error-recovery processes is small.

### 3.3 Output

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.

In order for the user to be able to reconstruct the concurrent execution which caused an edge to be flagged (i.e., the intercommand constraint that the edge represents to be violated), the constraints checker outputs the identity of any flagged edge and its nodes, as well as the identity of the two processes whose concurrent execution caused the constraint violation.

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 was flagged due to a problem with the existing error-recovery schedule, 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.

If an edge is not flagged either because the calculated start time is within the user-provided time range or because the user did not provide a range of possible start times, then the predicate is stored. Each pair of processes that forms the nodes of an edge yields a predicate relating the fixed start time of one process to the variable start time of the other process. As the edges are considered one-by-one in the constraints checker, the constraints that the edges impose on the scheduling of the processes accumulate. 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.

## 4 Conclusion

This paper has described a partial solution to the problem of validating that the concurrent execution of system-level error recovery processes with the critical command sequences satisfies intercommand constraints. The paper has presented a method by which the timing, precedence, and data-dependency constraints on the commands can be modeled in a constraints graph. A constraints checker algorithm is provided which uses the constraints graph to check for command interleavings that can violate the constraints.

The error-recovery scenarios chosen to test the algorithm involved failures during the execution of the critical command sequence that controls Galileo'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 during the planetary encounter. There are thus many opportunities for unsafe error-recovery schedules. The constraints checker offers a way to discover such process interactions early in the software development process.

The constraints checker algorithm 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 described in this paper 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] Bernstein, A. J. "Analysis of Programs for Parallel Processing." *IEEE Transactions on Computers* EC-15,5 (1966): 757-762.

[2] Cheng, Shengchang, John A. Stankovic, and Krithivasan Ramamritham. "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems." *IEEE Real-Time System Symposium* (1986): 166-174.

[3] Clarke, E. M., E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications." *ACM Transactions on Programming Languages and Systems* 8 (1986): 244-263.

[4] Coolahan, James E., Jr., and Nicholas Roussopoulos. "A Timed Petri Net Metholodology for Specifying Real-Time System Timing Requirements." *International Workshop on Timed Petri Nets, July 1–3, 1985*. Torino, Italy: Unione Industriale, 1985.

[5] Fujimoto, Richard M. "Parallel Discrete Event Simulation." *CACM* 33(1990): 31-53.

[6] García-Molina, Hector. "Using Semantic Knowledge for Transaction Processing in a Distributed Database." *ACM Transactions on Database Systems* 8 (1983): 186-213.

[7] Horvath, Joan C. and Louise P. Perry. "Hypercubes for Critical Spacecraft Command Verification." *AIAA Second International Symposium on Space Information Systems, Sept 17–19, 1990*. Pasadena, CA: AIAA, 1990.

[8] Jahanian, Farnam and Aloysius Ka-Lau Mok. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE Transactions on Software Engineering* SE-12 (1986): 890-904.

[9] Leveson, Nancy G., and Janice L. Stolzy. "Safety Analysis Using Petri Nets." *IEEE Transactions on Software Engineering* SE-13 (1987): 386-397.

[10] Lutz, Robyn. "Detecting Unsafe Error Recovery Schedules on Spacecraft." M.S. Thesis. Iowa State University, 1990.

[11] Lynch, Nancy and Hagit Attiya. "Using Mappings to Prove Timing Properties." MIT/LCS/TM-412.b. December, 1989.

[12] Maekawa, Mamoru, Arthur E. Oldehoeft, and Rodney R. Oldehoeft. *Operating Systems, Advanced Concepts.* Menlo Park, CA: Benjamin/Cummings, 1987.

[13] McDonald, Mark A. "Functional Emulation of Engineering Subsystem Interactions within the Galileo Spacecraft." M.S. Thesis. Univ. of Southern California, 1990.

[14] Mehlhorn, Kurt. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* Berlin: Springer-Verlag, 1984.

[15] Moss, J. Eliot B. *Nested Transactions, An Approach to Reliable Distributed Computing.* Cambridge, MA: MIT Press, 1981.

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

[17] Razouk, Rami R. and Michael M. Gorlick. "A Real-Time Interval Logic for Reasoning About Executions of Real-Time Programs." *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification, Key West, Florida, Dec. 13-15, 1989* Ed. Richard A. Kemmerer. In *Software Engineering Notes* 14(1989): 10-19.

[18] Schwartz, Richard L., P. M. Melliar-Smith, and Friedrich H. Vogt. "An Interval-Based Temporal Logic." *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, June 6–8, 1983.* Ed. Edmund Clarke and Dexter Kozen. Berlin: Springer-Verlag, 1984, pp. 443-457.

[19] Sha, Lui, Ragunathan Rajkumar, and John P. Lehoczky. "Concurrency Control for Distributed Real-Time Databases." *SIGMOD RECORD* 17(1988): 82-98.

[20] Stankovic, John A., and Krithi Ramamritham. *Tutorial: Hard Real-Time Systems.* Washington, D.C.: The Computer Society, 1988.

[21] Wing, Jeannette M. "A Specifier's Introduction to Formal Methods." *Computer* 23(1990): 8-26.

[22] Xu, Jia, and David Lorge Parnas. "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations." *IEEE Transactions on Software Engineering* 16(1990): 360-369.