Detecting Unsafe Error Recovery Schedules

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 31, 1996

Abstract

This paper presents a mechanism for modeling timing, precedence, and data-consistency constraints on concurrently executing processes. The model allows durations and intervals between events to be specified. An algorithm is provided to detect schedules which may be unsafe with respect to the constraints. This work, motivated by the design and validation of autonomous error-recovery strategies on the Galileo spacecraft, appears to be applicable to a variety of asynchronous real-time systems.

I. INTRODUCTION

This paper presents a technique for detecting unsafe schedules involving the asynchronous software processes responsible for error recovery onboard spacecraft. These autonomous processes are constrained at the event level by timing, precedence, and data-consistency rules. A schedule (ordering of events) that violates these constraints can jeopardize the spacecraft and is labeled unsafe. Safety involves those correctness properties required by the static portions of the specifications [3].

The motivation for this work comes from the difficulty of analyzing the potential process interactions during spacecraft error recovery. 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.

During system development any possibility of interaction between a sequence of commands and the error-recovery software is carefully scrutinized. Often, with the limited software tools currently available, this is a tedious and difficult task. A tool such as the

**IEEE Transactions on Software Engineering*,18:8, Aug, 1992. 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

one described here, which provides a graphical representation of the relevant command constraints and an algorithm to aid in detecting unsafe interleavings of the software processes, can help in this analysis.

The need for tools able to detect unsafe interleavings of processes will increase in future spacecraft missions [14, 18, 25]. Spacecraft will become more complicated as hardware advances and distributed architectures allow both more science and more autonomy. The opportunity for unplanned interactions will grow as concurrency increases. The additional computer capability embedded in future science instruments will bring with it the risk of more complicated interfaces with other subsystems.

However, as spacecraft become more complex and employ more concurrent processors, software simulation becomes more costly and time-consuming to develop. Hardware testbeds usually depend on scarce spare components and are too slow to test many paths. The result is that as the opportunity for spacecraft concurrency increases, the ability to detect unsafe schedules involving those concurrent processes is not increasing commensurately. The complexity of the spacecraft requires the capability to model and analyze precedence, timing, and data-consistency constraints in order to detect unsafe error-recovery schedules.

Traditionally, real-time constraints 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 timing relationships among commands.

The model represents timing, precedence, and data-consistency constraints on spacecraft commands by means of a labeled graph in which the nodes represent commands and the edges represent constraints on those commands. An algorithm is described which accepts as input this constraints graph, a number of potentially concurrent software processes, and user-provided intervals during which the processes can execute. The algorithm tests each edge in the graph to determine whether a schedule comprised of interleaved commands from these processes can fail to satisfy that constraint.

This work was performed in the context of the Galileo spacecraft, an interplanetary probe currently journeying to Jupiter. Preliminary results were reported in [22]. Ongoing research indicates that the results are equally applicable to other spacecraft as well as to other asynchronous systems with precedence and critical timing constraints.

The remainder of the paper is organized as follows. Section II describes the problem in more detail and in the spacecraft setting. Section III discusses the current research in this area. Section IV describes the modeling of the timing, precedence, and data-consistency constraints by means of a constraints graph. Section V presents the constraints checker algorithm and its time complexity. It also provides an example to illustrate how the constraints checker is applied to detect unsafe error recovery schedules. Section VI describes the work that remains to be done and other application domains appropriate for the constraints checker. The final section offers concluding remarks.

II. ERROR-RECOVERY SCHEDULES

The Galileo spacecraft, enroute to a 1995 encounter with the planet Jupiter, is a complex system with many interactions among its computers, science instruments, and hardware components. The spacecraft is a hard real-time system in that failure to satisfy the timing constraints jeopardizes the correctness of the system's behavior as well as the spacecraft [36].

The spacecraft is controlled by 28 processors, six of which form the shared-memory central Command and Data Subsystem. This subsystem interfaces with the other spacecraft subsystems through two buses and through hard-line connections. The central subsystem's architecture is based on five virtual machines which hide the details of the system scheduling algorithms, interrupt handling, and multilevel message queues. The virtual machines operate on a 2/3-second granularity, producing relatively coarse-grained timing constraints on the spacecraft instructions.

Instructions, called *commands*, to the various subsystems, direct the spacecraft to take specific actions at specific times. Individual commands are 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.

A *process* is a sequence of commands, executed sequentially on a single processor, which may be initiated by ground command or in response to an error condition detected onboard. Command sequences are straight-line, while error-recovery processes may include branches. On spacecraft, however, the amount of branching is minimal.

The detection of unsafe schedules is aided by the fact that, unlike the situation in many other asynchronous systems, one process does not wait on another. The time at which a command is issued depends on the starting time of the process and on the internal structure of the process, not on the actions of other processes.

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 [31]. 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.

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 system 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.

This work addresses the task of trying to detect the conflicting interactions–those interleavings of commands from asynchronous processes that may be unsafe. An unsafe schedule is one that defies the intercommand constraints.

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 [7]. 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 interactions among 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 [29].

The following assumptions hold:

1. Each process has a correct initial state, runs to completion, and produces correct results if executed alone. These assumptions exclude from consideration those cases in which the execution of a single process in isolation is unsafe. Such cases are properly detected by other methods prior to the use of the constraints checker presented here. The assumptions also exclude those cases in which a command constraint is not satisfied because one command was never issued due to a process abortion.

2. Simplifying assumptions about the Galileo architecture have been made with respect to the existence of backup hardware components and the possibility of rollbacks during critical sequences.

III. BACKGROUND

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 consistency (read/write) constraints, and satisfy the timing constraints.

The problem differs in four significant ways from the focus of most of the research in this area:

1. The effort here is to validate that all possible error-recovery schedules allowed by the spacecraft system are safe. Most research on scheduling concentrates on developing schedulers capable of choosing a single safe schedule.

2. The issue here is how to manage concurrency, not how to maximize it. The work addresses the question of whether the spacecraft already allows too much concurrency to guarantee that the constraints are always met. The spacecraft's scheduling software here is assumed to be already in place. Most research in concurrency instead addresses the question of how concurrency can be maximized in a future system within some given constraints.

3. Solutions to error recovery usually involve the notion of atomic actions, but a spacecraft command cannot be adequately modeled as an atomic action [20, 37]. A command is defined to be an action. An issuance of a command is defined to be an event, the instantiation of an action at a specific point t in time. The issuance of a command is then assumed to occur instantaneously at this point in time. Following convention, this is stated as "command c occurs (or is issued) at time t."

However, the activity or operation initiated by command c is usually not instantaneous. It is the duration of the activity to which the constraints often refer. For example, if c is a command to slew the scan platform to a particular position, the mechanical activity of moving the platform may take m minutes to complete. This is stated as "command c takes m minutes to execute."

The model described below 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.

4. The solution here must take into account both precedence constraints and timing constraints. Precedence and timing are fundamentally different in that precedence does not require a notion of duration [30]. 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 [15, 24, 28].

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 aperiodic and interval timing constraints on spacecraft commands [8, 35, 36, 40]. The work described here brings together the study of real-time constraints with the study of precedence and data-consistency constraints.

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 [10, 21]. 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 [5, 13, 23]. Alternatively, processes can be repesented as timed Büchi automata (with upper time bounds on events) and the languages of timed traces that they accept can be investigated [2]. Real Time Logic models the timing aspects of a system specification to establish timing properties (periodic events and deadlines) [19]. 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 [1, 9, 12, 17, 30]. 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.

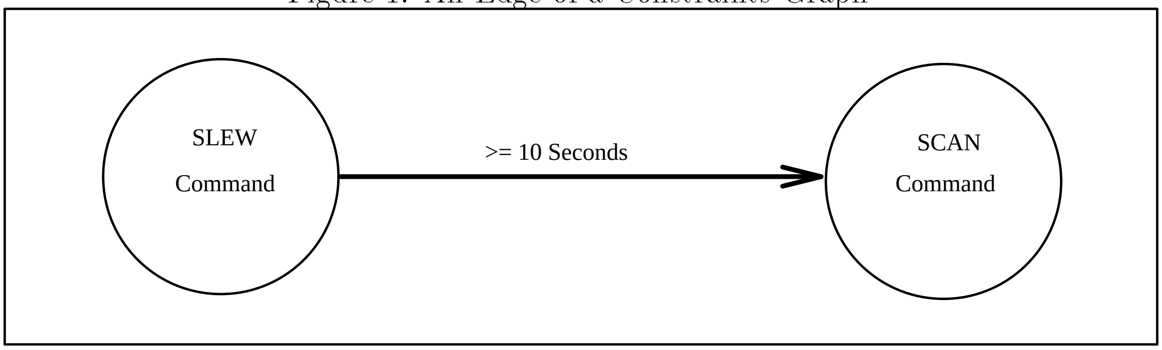
The work decribed here discusses many of the same timing issues addressed by recent work in interval temporal logic [11, 32, 34, 38]. However, the emphasis there is on specifying and verifying system requirements (what the spacecraft must do) while the emphasis here is on verifying operational constraints (what the spacecraft should not be allowed to do). For example, a system requirement often is stated in terms of a lower time bound (after which an event may occur) and an upper time bound (by which the event must have occurred). In contrast, an operational constraint often is stated in terms of an interval during 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) [5] is often absent in formal models.

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 quantitative timing constraints within the context of spacecraft error recovery.

IV. THE MODEL

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 $c_i \longrightarrow c_j$. For example, the timing constraint "A SCAN command shall not be sent within 10 seconds of a SLEW command" is modeled as a labeled edge from a node labeled SLEW to a node labeled SCAN (Fig. 1).

Figure 1: An Edge of a Constraints Graph



The precedence subgraph $G' \subseteq (V', E')$, $V' \subseteq$ V, $E' \subseteq$ E, $E' \subseteq V' \times V'$ is the subgraph composed only of precedence edges. It is required to be acyclic since a cycle of precedence constraints cannot be satisfied simultaneously.

Each node (representing a command) has 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 (the longest time for which the designers must plan). For example, the tape-rewind command on Galileo has a worst-case time field of seven minutes.

Each edge has associated with it an edge type (described below), two optional time fields, t_1 and t_2, a code indicating where the constraint is documented, and the variable, if the edge represents a data-consistency constraint.

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-consistency 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" [39].

A. 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 from it." An example is, "The time separation between powering on the S-Band transmitter and powering on the X-Band transmitter shall be either less than one-half minute or greater than 6 minutes." A minimum-interval constraint is a special case of an outside-interval constraint with $t_1 = 0$, but is called out separately here for clarity of presentation.

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. This constraint is discussed further below.

4. Maximum-interval constraint. "If c_i occurs, then c_j can only occur within time t_1 of it." An example is, "A maximum of 6 minutes delay between spectrometer power off and replacement-heater power on can be tolerated."

5. Inside-interval constraint. "If c_i occurs, then c_j can only occur within a range t_1 to t_2 from it." An example is, "Each Low-Gain Antenna Motor power off command follows the on command no sooner than 9 seconds and no later than 30 seconds." A maximum-interval

constraint is a special case of an inside-interval constraint with $t_1 = 0$ or of an outside-interval constraint with $t_2 = +\infty$.

A constraint of the form, "If c_i occurs, then c_j can only occur after it" can be expressed either as an outside-interval edge with t_1 = -∞ and $t_2 = 0$, or as an inside-interval edge with $t_1 = 0$ and $t_2 = +\infty$.

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 is an additional constraint on the time at which command c_j can occur. In the case of a minimum-interval constraint with time parameter $= t_1$, the worst-case execution time is 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 is added to the timing constraint. The cases for the inside-interval and the outside-interval edge types follow accordingly.

B. Precedence Constraints

While intercommand timing constraints are clearly safety properties, intercommand precedence constraints contain aspects of both safety and liveness properties [23]. 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 τ 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, e.g., "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. This places a restriction on the schedules that will be considered legal. For example, if a command to turn off heater A occurs between the two "ON" commands, the schedule will be flagged as illegal by this method. Because the tool is used not for mechanical verification but as an analysis aid during development, these occasional false positives have been tolerated as easily resolvable by subsequent inspection of the code.

C. Data-Consistency Constraints

Data-consistency 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 [4]. 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-consistency constraint is used to specify the order in which the commands that read/write the variable must occur. Such a data-consistency constraint is expressed as a precedence constraint.

For example, on Galileo the Commanded-Maneuver-Status variable is updated by the command sequence before the thruster burn that places Galileo in orbit around Jupiter. If a failure occurs adjacent to the time of the burn, this variable is read by an error-recovery process to determine the appropriate response. The documented constraint requires that the update precede the read. This write/read constraint is modeled in the graph by adding a precedence edge from a node representing the update to a node representing the read.

The constraints checker does not model the state of the spacecraft. Instead, it accepts as input several sequences of commands (the processes) and a set of rules (the constraints graph) and looks for schedulings in which the concurrent execution of those processes violates a constraint. Consequently, if some other process that was not input to the constraints checker issues a command, the constraints checker does not 'see' the effect of the command.

V. THE CONSTRAINTS CHECKER ALGORITHM

Section IV described how each edge represents a constraint that must hold between the commands which are the edge's endpoints. Each constraint is translated in the constraints graph into an edge type (precedence, minimum interval, etc.).

The constraints graph and a set of processes (time-tagged lists of commands) are 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 processes' start times. 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 that case some scheduling of the processes can cause the constraint represented by that edge to be violated.

Since the constraints graph is sparse, an adjacency list representation is used to store the graph [27]. Inputting a process to the constraints checker consists of adding a two-way pointer from each command in the process to the node in the constraints graph that represents the command. Each command has an associated time tag. The command represented by a node may occur in any or all of the processes. Some processes may have multiple occurrences of the command.

The commands in the various possible paths through a process are interleaved. This is possible because only interactions between distinct processes are analyzed. Furthermore, the tree of possible paths is small since command sequences are straight-line processes and error-recovery processes display little branching. In practice, the control structure of the spacecraft processes is simple enough to allow the complete unraveling of the process constructs.

To fix the timeline, one process' start time (thus, timeline) is fixed and the other processes' timelines are moved relative to that fixed process. If one of the processes is a command sequence, it is the natural choice for the fixed timeline since sequences have absolute start times. Otherwise, the process with the lowest processor identification (pid) is chosen.

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 (see Appendix for a detailed example).

Define S_i = Actual start-time, if fixed, for the process issuing c_i (the start-time is known for command sequences), E_i = Earliest possible start time for the process issuing c_i, and L_i = Latest possible start time for the process issuing c_i. E_i and L_i are provided by the user to define the maximum range of start times to be analyzed for each process. Define Δ_i = Time interval from start-time of P_i until c_i is issued (fixed in the process). Let Gc_i be the earliest time by which an instance of c_i can be assumed to have occurred (used for precedence edges only). It is defined below.

The variables S_j, E_j, L_j, and Δ_j are defined for P_j analogously. S_v, E_v, and L_v refer to the actual, earliest, and latest start times, respectively, for P_v, the process whose timeline is variable.

For the sake of clarity, the description of the algorithm in Fig. 2 assumes that E_i, E_j, L_i, and L_j have been provided with initial values by the user. If the user has not provided these values then the constraints checker cannot detect violations of edges and will not issue warning flags to the user. However, useful information regarding the range of allowable start times for processes can still be accumulated and output, as is described below.

A. The Algorithm

The constraints checker (Fig. 2) 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 command c_i in a process that may or may not execute cannot with certainty satisfy a precedence edge.

The variable Gc_i is the earliest time by which a process that is guaranteed to execute can guarantee that c_i will occur. If P_i is a sequence, Gc_i is the timetag of the earliest instance of command c_i in the sequence. For example, in analyzing the precedence constraint on Galileo that c_{22} must precede c_{24}, the Gc_i for c_{22} is the fixed time at which the command sequence issues c_{22}, namely the value 2:00:38 (see Appendix). If P_i is not a sequence, then Gc_i is an optional, user-provided value (in practice, often equal to $E_i + \Delta_i$). If the user can make no guarantee regarding when an instance of c_i will occur, then Gc_i is set to an arbitrarily large integer. The constraints checker uses Gc_i to detect a possible violation of a precedence edge by requiring that c_j not occur before Gc_i.

Given that S_j is the actual start time of the process issuing c_j, Δ_j is the time interval from S_j until command c_j is issued, and E_j is the earliest time at which process P_j can start, the command c_j occurs at $S_j + \Delta_j$ when S_j is fixed. However, when S_i (the start time of the process issuing c_i) is fixed and the value of S_j is not fixed, the algorithm then considers the earliest time at which a c_j can occur—namely at $E_j + \Delta_j$. If the earliest time at which a c_i can occur precedes or equals Gc_i, then the edge is flagged. This means that the constraint that the edge represents is not always satisfied by the process interactions.

As noted in Section IV, data-consistency constraints are modeled as precedence edges. The constraints checker algorithm thus validates data-consistency constraints by testing those edges.

A time-constrained edge 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 $Poss = [E_v, L_v]$ 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 T that satisfy the predicate for that edge type. 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: $Poss \subseteq Safe$ (Fig. 2).

Fig. 2. The Constraints Checker Algorithm

begin
  input $Poss$ {user-supplied interval};
  for each edge (c_i,c_j) in the constraints graph
  begin
    τ:= edge type of (c_i, c_j);
    if $\tau = 0$ {precedence edge} *then*
    for each instance c_j' of c_j
      begin
        fix P_i or P_j according to the rules;
        if P_j is fixed and $(S_j + \Delta_j) \leq Gc_i$
          then output warning flag;

if P_i is fixed and $(E_j + \Delta_j) \leq Gc_i$
then output warning flag
end
else {time-constrained edge}
for each pair (c'_i, c'_j) of instances of
c_i and c_j where $P_i \neq P_j$
begin
fix P_i or P_j according to the rules;
if P_i is fixed {see Fig. 3 for predicate Θ_τ}
then $Safe := \{T | \Theta_\tau(S_i, T, \Delta_i, \Delta_j, t_1, t_2)\}$
else $Safe := \{T | \Theta_\tau(T, S_j, \Delta_i, \Delta_j, t_1, t_2)\}$;
if $Poss \not\subseteq Safe$
then output warning flag
end
end
end

Define τ to be the edge type of the edge being checked. The edge type τ=0 if the edge is of type *precedence*; τ=1 if the edge is of type *minimum-interval* (i.e., "command c_j cannot occur within time t_1 of command c_i"); τ=2 if the edge is of type *outside-interval* ("command c_j cannot follow command c_i by more than time t_1 but less than time t_2"); τ=3 if the edge is of type *forbidden-combination* ("command c_j cannot occur if command c_i does"); τ=4 if the edge is of type *maximum-interval* ("command c_j cannot occur more than time t_1 after command c_i"); and τ=5 if the edge is of type *inside-interval* ("command c_j cannot occur within time t_1 of command c_i or more than time t_2 after c_i"). Thus, $\tau \in \{0,1,2,3,4,5\}$. For simplicity of exposition, both time parameters t_1 and t_2 have been given in the algorithm, although not all edge types use both these parameters.

Each of the five time-constrained edge types ($\tau \neq 0$) has associated with it a predicate Θ_τ that is used to determine the set *Safe* (Fig. 3).

Fig. 3 Predicates for Determining Safe Times: $\Theta_\tau(S_i, S_j, \Delta_i, \Delta_j, t_1, t_2), \tau \in \{1,2,3,4,5\}$

$\Theta_1 \equiv (S_j + \Delta_j < S_i + \Delta_i) \vee (S_j + \Delta_j > S_i + \Delta_i + t_1)$
$\Theta_2 \equiv (S_j + \Delta_j < S_i + \Delta_i + t_1) \vee (S_j + \Delta_j > S_i + \Delta_i + t_2)$
$\Theta_3 \equiv FALSE$
$\Theta_4 \equiv S_i + \Delta_i < S_j + \Delta_j < S_i + \Delta_i + t_1$
$\Theta_5 \equiv S_i + \Delta_i + t_1 < S_j + \Delta_j < S_i + \Delta_i + t_2$

B. 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 is 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.

The notion of precedence constraints presented here is restricted in two ways. First, precedence constraints have looked only backward in time: "if command c_j occurs, then there exists an instance of command c_i that precedes it." Secondly, precedence constraints have contained no timing information about the events they order.

If the constraints to be modeled make it appropriate to do so, both of these restrictions can be relaxed with only minor adjustments to the algorithm. To allow precedence constraints to look forward in time ("every instance of c_i is followed by some instance of c_j"), a new variable, Gc_il, is defined and used in place of Gc_i in the algorithm. The variable Gc_il is defined to be the latest time at which a c_i might occur. In addition, E_i (when P_j is fixed) and S_i (when P_i is fixed) replace S_j and E_i, respectively.

Adding timing information to precedence constraints (e.g., "if c_j occurs, then some c_i must precede it by at least time t"), blurs the useful distinction between ordering requirements and timing requirements. However, the distinction between existential constraints ("there exists a c_i that precedes each c_j") and universal constraints ("every c_j can only occur in a specific time interval relative to c_i") is preserved. The algorithm is thus readily extended to check these additional types of precedence constraints if the application domain requires them.

Similarly, the timing details of a specific system architecture can be incorporated into the predicates if the user so wishes. For example, on the Galileo spacecraft the commands issued by system-fault-protection processes to other subsystems in a given process cycle are then transferred to output queues in the following cycle. How long a command then spends in a queue depends upon which queue it is in, the type of command, the number of commands issued proximate to that time, which process issued the command, and the operating mode, among other factors. The worst-case time by which each command will be output from the queue can be calculated and this delay included in the algorithm. On Galileo, the granularity of the time units involved in command sequencing (2/3 seconds) is large enough that architectural details were not included in the constraints checker.

C. Time Complexity and Optimizations

The time complexity of the constraints-checker algorithm is
O($\mid E \mid \cdot n^2 \cdot \bar{k}^2$), where E is the set of edges, n is the number of processes, and $\bar{k}$ is defined as follows.
Let $P_1, P_2, \ldots, P_n$ be the processes under consideration,
(c_i, c_j) be an edge,
E_p = the set of precedence edges,
E_t = the set of timing and forbidden edges,
$c_i(x)$ = the number of instances of a fixed c_i in process P_x,
$\bar{k}_i = \frac{1}{n}\sum_{x=1}^{n} c_i(x)$ = the average number of instances of a command per process,
$\bar{k} = \max_i\{\bar{k}_i\}$ = the maximum over all commands of the average number of instances of that command per process.

The outer for-construct of the constraints checker executes $\mid E \mid$ times. Subsequently, the algorithm branches, depending on whether the edge is a precedence or timing edge. In either case, testing the predicate and outputting the warning flag takes constant time. The algorithm thus requires time:

$$O(\sum_{(c_i,c_j)\in E_p}\sum_{x=1}^{n} c_j(x) + \sum_{(c_i,c_j)\in E_t}\sum_{x=1}^{n} c_i(x)\sum_{y=1}^{n} c_j(y))$$
$$= O(\sum_{(c_i,c_j)\in E_p} n \cdot \bar{k}_j + \sum_{(c_i,c_j)\in E_t} n^2 \cdot \bar{k}_i \cdot \bar{k}_j)$$
$$= O(\mid E_p \mid \cdot n \cdot \bar{k} + \mid E_t \mid \cdot n^2 \cdot \bar{k}^2)$$
$$= O(\mid E \mid \cdot n^2 \cdot \bar{k}^2).$$

Since the constraints graph is sparse (Galileo has over 1200 commands and only 284 flight rules), $\mid E \mid$ will be small. Since there are usually very few instances of each command per process, $\bar{k}$ will also be small. On the Galileo spacecraft $\bar{k}$ is slightly greater than 1. Although the analysis of the time complexity assumes that every command will be issued by every process, in actuality only those edges for which both nodes are issued by the input processes are checked. The user can further reduce the running time by pruning the constraints graph to include only those nodes (commands) present in the processes under consideration and only those edges (rules) which are of interest. The disadvantage of pruning the graph is that edges which should be checked may be accidentally deleted.

The number of processes, n, will usually be quite small. Since the constraints checker functions primarily as a visualization and validation tool for those interactions that cannot be easily grasped using current techniques, a typical run consists of a critical command sequence and a few error-recovery programs. Galileo, for example, has 26 system-level error-recovery programs, many of which cannot execute concurrently with each other.

D. An Example

The following example from the Galileo spacecraft illustrates how the constraints graph and the algorithm function. The scenario to be analyzed addresses the complicated timing issues involved when the Probe Relay/Jupiter Orbital Insertion (Relay/JOI) critical sequence (resident in the Command and Data Subsystem or CDS computer) issues commands to the AACS (Attitude and Articulation Control Subsystem) hardware that presuppose that the AACS is in a certain configuration. If the AACS computer experiences a Power-On-Reset failure during Jupiter Orbital Insertion, error-recovery responses in the CDS, jointly called AACS_INIT, must quickly reconfigure the AACS to the configuration required by the sequence.

This is accomplished by dividing the critical sequence into a number of segments, each consisting of a distinct AACS configuration and an activity (e.g., the 400-Newton burn) needed for the completion of the sequence [16, 33]. Each segment of the sequence has an AACS_INIT error-recovery response tailored to it, designed to bring the AACS back to the state required by the sequence at that point. The synchronization of the segments of the sequence with the AACS_INIT error-recovery responses is accomplished via a shared flag, updated by the sequence, that determines which AACS_INIT response will be invoked. Among the timing issues that must be taken into account is the fact that it is possible for commands to be issued from an AACS_INIT error-recovery response at a time earlier than they would have occurred in the sequence if an AACS_INIT response had not occurred.

There are ten synchronized error-recovery responses in AACS_INIT. This example examines six of these responses (AACS_INIT3 through AACS_INIT8) and the one-and-a-half-hour portion of the Relay/JOI sequence that most closely brackets JOI. (The entire sequence issues commands over a ten-day period.) The portion of the sequence analyzed here contains 70 commands, several of them issued redundantly. The six error-recovery responses together contain 237 lines of code, almost all of them commands. Many of these commands are issued in more than one response. No assumption was made here that each process satisfied the intercommand constraints if executed in isolation, so commands in the same process as well as in different processes were checked against the constraints.

The constraints graph input to the constraints checker (Fig. 4) consists of 40 nodes and 40 edges [6, 26]. Of these edges, 20 are precedence edges and 20 are timing edges. The commands represented by the nodes address four different spacecraft subsystems. Only intercommand constraints involving the commands found in the seven processes discussed in this example are shown in the figure. (The entire constraints graph for Galileo would contain on the order of 1200 nodes and under 1000 edges).

Each command, represented in the spacecraft code as a mnemonic (e.g., "7BIGZ" for the command to initiate the 400-Newton engine burn), is labeled in the example with a positive integer for ease of explanation. 7BIGZ, for example, is labeled c_{24}. Fig. 4 shows several precedence constraints (unlabeled) and several timing constraints (labeled "t") involving command 7BIGZ. The edge from node c_{37} to c_{24}, for example, represents the precedence constraint that if a 7BIGZ command occurs, then a 7ACCLON command (Accelerometers On) must have preceded it.

The results were as follows. Seven of the 40 edges in the constraints graph were flagged by the constraints checker (see the Appendix for details). Four of the flagged edges were

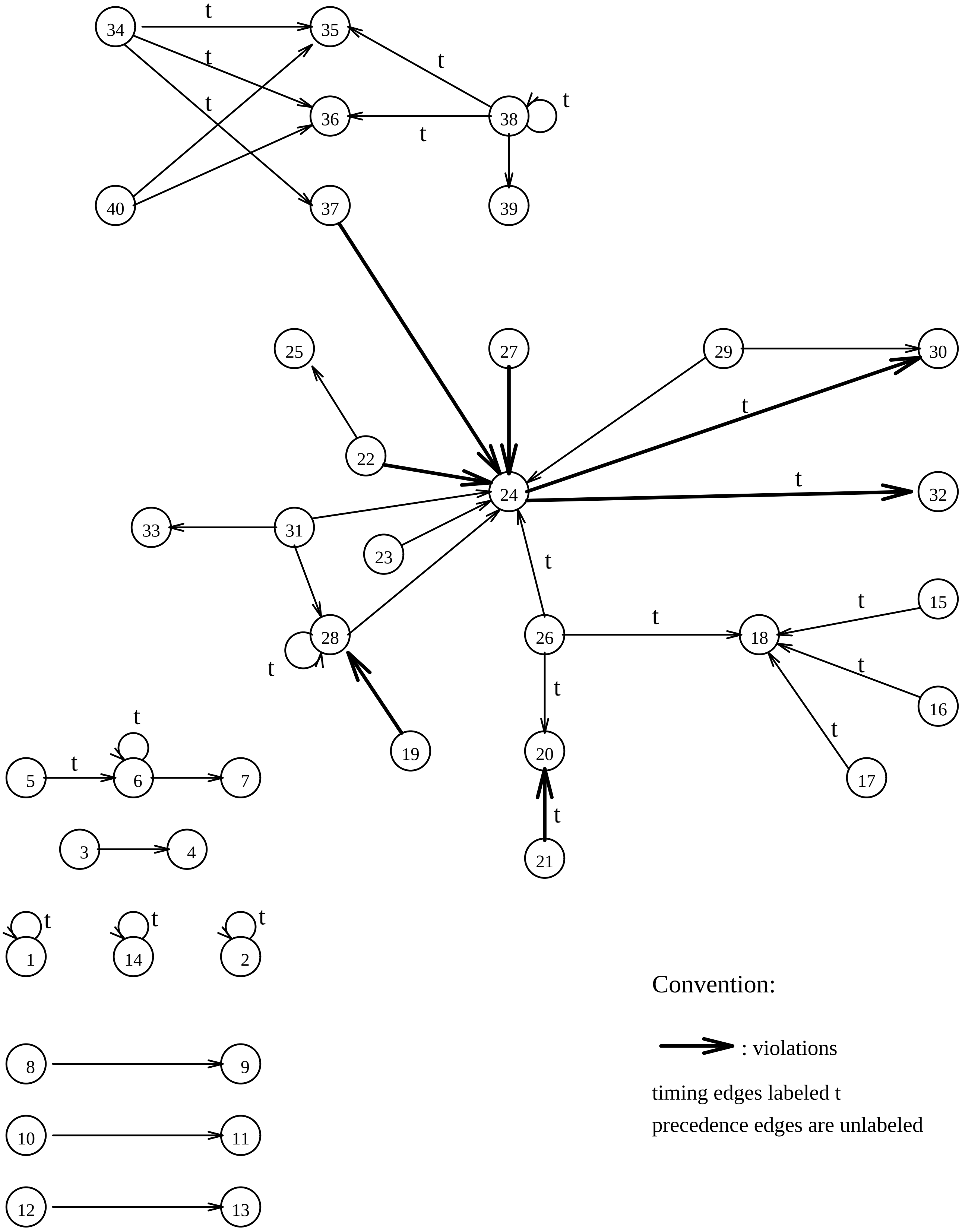


Figure 2: Example for the Constraint Checker

precedence edges and three were timing edges. 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 required tail-node command appeared six hours before the associated head-node command.

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. It is not surprising that the constraints checker is only as accurate as the constraints provided to it. What was unexpected was the number of errors discovered during the modeling process itself.

The intercommand constraint violations flagged by the constraints checker (in boldface in the graph) 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 timing or precedence constraint violations that occur only in some subset of the possible schedules.

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 yet thoroughly 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.

VI. FUTURE WORK AND OTHER APPLICATIONS

A version of the constraints checker is currently under development for use on the Comet Rendezvous/Asteriod Flyby (CRAF) spacecraft and the Cassini spacecraft. The initial use will be as a software development tool, analyzing and testing error-recovery processes and schedules involving their concurrent execution. Later, as command sequences are developed, the constraints checker will incorporate them into the error-recovery scenarios that it checks.

Experience on Galileo indicates that the early detection of significant inconsistencies between the design and the constraints will be the constraint checker's major benefit. It can evaluate the system's ability to satisfy precisely expressed timing and precedence properties early in the design of the software.

The constraints checker currently limits the kinds of intercommand constraints that can be modeled. Because its original focus was on solving a spacecraft problem, the existing core suffices to check most of the intercommand constraints on spacecraft. However, extension to other applications requires expansion of the constraints definition. Specifically, plans include extending it to include "follows" as well as "precedes," attaching times to precedence constraints, including Boolean operators in the constraints, and providing more flexibility in how similar commands with distinct parameters can be grouped. Currently, since only interactions among processes that execute correctly in isolation are analyzed, an edge is never flagged due to its nodes being issued from different paths of a single process. To drop the assumption that isolated processes execute correctly, a naming mechanism is being incorporated into the constraints checker. It will associate a path name with each command and, when an edge is flagged, test that both nodes are issued in the same path through the process. Finally, steps need to be taken towards automating the creation of the constraints graph.

Some issues that remain to be investigated are the modeling of an initial state in the form of a history of earlier events, the representation of an ordinal iteration of a command ("the second instance of c_j"), and the addition of variable times (where the time parameter is not a constant). Current research, especially in interval temporal logic, continues to identify additional constraint types. The challenge is to incorporate those features into the constraints checker without eroding its focus and usefulness in accurately detecting unsafe schedules in existing or developing systems.

The techniques outlined here are suitable for addressing related problems in domains other than spacecraft. The design of system-level error-recovery in event-driven systems often involves the analysis of how temporal constraints are affected by the concurrent execution of processes with unpredictable start times. Such issues are readily investigated with the constraints checker.

Critical software applications such as many process-control, flight-control, command-and-control, or avionics systems involve aperiodic tasks and time-interval specifications. Frequently, however, only periodic and deadline specifications can be modeled and tested with the tools currently in use. The methods described above offer the capability to quickly and accurately model and check that even aperiodic and interval constraints among events will always hold in the system. For these reasons, interest has been expressed in the use of the constraints checker as an embeddable module in other simulation or executable-specification tools.

The constraints checker is also suited to operational situations where, as on the spacecraft, 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 need to be sent from the ground. The operational difficulties of quickly checking that new or temporary software will not conflict with the prior or "permanent" software can be eased by the availability of a constraints-graph model and constraints checker.

VII. CONCLUSIONS

This paper has shown how to construct a constraints graph to model the precedence, timing, and data-consistency constraints for which it has been historically difficult to design. The constraints graph provides a means of visualizing the command constraints that must be satisfied by every concurrent execution of processes. This paper has also presented an algorithm that, given a constraints graph and a set of processes, detects possibly unsafe schedulings of the processes.

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 authors thank the referees for their comments and suggestions.

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.

APPENDIX

This appendix provides the details of the example described in Section V.D. The sequence attached an absolute time tag to each command. The other six processes attached an offset to each command, measured in "minor frames" (2/3 second) from the process' start.

For an edge (c_i, c_j) in the graph, node c_i is called the tail and node c_j is called the head of the edge. Due to the fact that the Jupiter Orbital Insertion (JOI) sequence provides time tags relative to the probe entry rather than to the start of the command sequence, a time tag attached to a command in the sequence equals $(S_i + \Delta_i)$ when the command is a tail node and equals $(S_j + \Delta_j)$ when the command is a head node. For each command issued by the sequence, Gc_i for that command is the sequence's time tag for that command.

The user provided the algorithm with the earliest and latest start times of the AACS_INIT responses (Fig. 5). These are the earliest and latest times at which, if an AACS Power-On-Reset occurred at that point, the synchronization flag would cause an invocation of this response.

Each instance of a command in every process was linked to its node in the constraints graph, if such a node existed. Commands not involved in intercommand constraints did not appear in the graph and were not checked by the algorithm.

There were 20 precedence edges in the graph. Nine of these were discarded by the constraints checker because there was no occurrence of the commands represented by the edges' heads in the processes input to the checker. Each instance of the head in the other eleven precedence edges was checked by the constraints checker algorithm. There were a total of 46 such instances.

The precedence edge (c_{22},c_{24}) is one which the algorithm flagged as not satisfied. There were four instances of c_{24}: one in the sequence at 2:00:42 (hrs:min:sec), one in the sequence at 2:00:44, one in AACS_INIT6 (abbreviated P_6), and one in AACS_INIT7 (abbreviated P_7). There was only one instance of c_{22}, namely in the sequence. From Fig. 5, E_j = 2:00:13.3 for P_6 and E_j = 2:37:49.3 for P_7. The value of Gc_i = 2:00:38 (the time at which c_{22} was issued in the sequence). From the code, when $c_j = c_{24}$, Δ_j = 14.7 seconds for P_6 and Δ_j = 16 seconds for P_7. For the sequence, the value of $S_j + \Delta_j$ = 2:00:42 (the actual time that the command was issued).

In the cases in which the sequence issued c_{24}, the predicate required that $S_j + \Delta_j \geq Gc_i$. Since 2:00:44 > 2:00:42 $\geq$ 2:00:38, this was trivially satisfied.

In the case in which P_6 issued c_{24}, the predicate required that $E_j + \Delta_j \geq Gc_i$. But since $E_j + \Delta_j$ = 2:00:13.3 + 0:00:14.7 = 2:00:28 < Gc_i, the constraints checker flagged the edge with the information that if P_6 issued c_{24}, it could issue it before the sequence issued c_{22}, thus violating a precedence constraint.

Finally, when P_7 issued c_{24}, $E_j + \Delta_j$ = 2:37:49.3 + 0:00:16 = 2:38:05.3 $\geq Gc_i$, so precedence was preserved.

The other ten precedence edges were similarly analyzed by the constraints checker. Three other precedence edges were flagged because in each case, one or more instances of the command represented by the head node occurred but no instance of the command represented by the tail node preceded them, as required. In one of these cases, analysis showed that the tail did occur in the part of the sequence that was not provided to the example. In another case, the documented constraint was incorrect. In the other case, a global variable could be used before it was updated, contrary to the constraint.

There were 20 timing edges in the graph. Fifteen of these were dismissed by the constraints checker since there was no occurrence of both head and tail nodes in the processes. The algorithm computed whether the timing predicates were ever violated for the remaining five timing edges in which both nodes appeared in the code. For these edges the algorithm tested every pair of instances of commands represented by the nodes, a total of 83 pairs.

The inside-interval timing edge (c_{24}, c_{32}) is one which the constraints checker flagged as not satisfied. There were four instances of c_{24} and three instances of c_{32} in the processes, yielding twelve pairs of instances of (c_{24}, c_{32}) to check. Since the edge type was an inside-interval constraint, *Safe* = those values for which the predicate $S_i + \Delta_i + t_1 < S_j + \Delta_j < S_i + \Delta_i + t_2$ held. From the graph, t_1 = 0:00:22 and t_2 = 0:00:22.7. Since the constraint referred to when c_{24} completed (the maximum burn time), rather than to when it was issued, the execution time of 0:45:54 from the constraints graph was added to t_1 and t_2.

One of the twelve pairs checked involved the case in which P_6 issued c_{24} at Δ_i = 0:00:12.7 and the sequence issued c_{32} at 2:46:42. *Safe* = the process start times that satisfy the predicate $S_i + \Delta_i$ + execution time + $t_1 < S_j + \Delta_j < S_i + \Delta_i$ + execution time + t_2. Then *Safe* = the process start times that satisfy S_i + 00:46:28.7 < 2:46:42 < S_i + 00:46:29.4. Thus, *Safe* = [2:00:12.6, 2:00:13.3]. However, *Poss*, the interval of possible times at which

P_6, the process that issued c_{24}, could start was $[E_i, L_i]$ = [2:00:13.3, 2:37:49.2]. *Poss* $\not\subseteq$ *Safe*, so the edge was flagged as possible to violate.

Another pair checked involved the case in which the sequence issued c_{24} at time 2:00:42 and P_8 issued c_{32} at Δ_j = 00:00:16. The interval of possible times at which P_8 could start = Poss = $[E_j, L_j]$ = [2:46:36, 2:59:04.6]. However, *Safe* = the process start times that satisfy the predicate 2:00:42 + 00:46:16 < S_j + 00:00:16 < 2:00:42 + 00:46.16.7 = [2:46:42, 2:46:42.7]. Again, *Poss* $\not\subseteq$ *Safe*, so the edge was flagged and the additional information about cases that could violate the constraint was added to the output. The analysis of the other ten pairs followed similarly. In all, 129 instances of edges were checked by the algorithm, and another 24 edges discarded because the head and tail nodes were not found in the processes. From these calculations, the seven edges described above were flagged as representing intercommand constraints that were not always satisfied.

Fig. 5. Earliest and Latest Start Times for Processes

		E_i	L_i
AACS_INIT, segment	3	1:17:50.6	1:32:35.2
	4	1:32:35.3	1:58:21.9
	5	1:58:22	2:00:13.2
	6	2:00:13.3	2:37:49.2
	7	2:37:49.3	2:46:35.9
	8	2:46:36	2:59:04.6

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] R. Alur and D. Dill, "Automata for Modeling Real-Time Systems," in *17th ICALP* , Ed. M. S. Paterson. *Lecture Notes in Computer Science*, vol. 443. New York: Springer-Verlag, 1990, pp. 322-335.

[3] M. Ben-Ari, *Principles of Concurrent Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1982.

[4] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Transactions on Computers*, vol. EC-15,5, pp. 757-762, October 1966.

[5] 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.

[6] G. Beutelschies, *Galileo Space Flight Operations Plan, Command Dictionary.* 625-505, vol. 11, Rev. A. Pasadena, CA: Jet Propulsion Laboratory internal document, 1989.

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

[8] 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.

[9] E. M. Clarke, 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*, vol. 8, pp. 244-263, April 1986.

[10] 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.

[11] L. K. Dillon, et al., "An Interval Logic Based on Actions and Events," 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. 33-45.

[12] E. A. Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science, vol. B, Formal Models and Semantics.*, Ed. J. van Leeuwen. Cambridge/Amsterdam: MIT Press/Elsevier, 1990, pp. 995-1072.

[13] M. K. Franklin and A. Gabrelian, "A Transformational Method for Verifying Safety Properties in Real-Time Systems," in *Proceedings of the Real-Time Systems Symposium*. Los Alamitos, CA: IEEE Computer Society, 1989, pp. 112-123.

[14] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, pp. 31-53, Jan. 1990.

[15] H. García-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems*, vol. 8, pp. 186-213, June 1983.

[16] L. Grove, *Galileo Space Flight Operations Plan, Fault Protection System Design and Operations.* 625-505, vol. 8, Rev. A, Preliminary Release. Pasadena, CA: Jet Propulsion Laboratory internal document, April 1989.

[17] 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.

[18] J. C. Horvath and L. P. Perry, "Hypercubes for Critical Spacecraft Command Verification," *AIAA Second International Symposium on Space Information Systems*. Pasadena, CA: AIAA, 1990.

[19] 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.

[20] W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys*, vol. 13, pp. 149-182, June 1981.

[21] N. G. Leveson and J. L. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 386-397, March 1987.

[22] 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.

[23] N. Lynch and H. Attiya, "Using Mappings to Prove Timing Properties." MIT/LCS/TM-412.b. December, 1989.

[24] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft, *Operating Systems, Advanced Concepts.* Menlo Park, CA: Benjamin/Cummings, 1987.

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

[26] N. Medici, *Galileo Spacecraft Flight Rules and Constraints.* 625-270, Rev. C. Pasadena, CA: Jet Propulsion Laboratory internal document, July 1988.

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

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

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

[30] 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.

[31] 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.

[32] R. R. Razouk and M. M. Gorlick, "A Real-Time Interval Logic for Reasoning About Executions of Real-Time Programs," in *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification*, Ed. R. A. Kemmerer. In *Software Engineering Notes*, vol. 14, pp. 10-19, Dec. 1989.

[33] *Relay/JOI Sequence*, GLL3-120, Rev. D. Pasadena, CA: Jet Propulsion Laboratory internal document, pp. 108-177.

[34] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt, "An Interval-Based Temporal Logic," *Logics of Programs, Workshop*, Ed. E. Clarke and D. Kozen. *Lecture Notes in Computer Science*, vol. 164. Berlin: Springer-Verlag, 1984, pp. 443-457.

[35] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *SIGMOD RECORD*, vol. 17, pp. 82-98, March 1988.

[36] J. A. Stankovic and K. Ramamritham, *Tutorial: Hard Real-Time Systems.* Washington, D.C.: The Computer Society, 1988.

[37] D. J. Taylor, "How Big Can an Atomic Action Be?" in *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems.* Los Angeles: IEEE Computer Society, 1986, pp. 121-124.

[38] 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.

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

[40] 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.

Index of Terms

asynchronous systems, consistency constraints, error recovery, fault tolerance, hard-real-time systems, modeling, precedence, safety, scheduling, timing constraints.