Safety Analysis of Timing Properties in Real-Time Systems

FARNAHM JAHANIAN AND ALOYSIUS KA-LAU MOK, MEMBER, IEEE

Abstract—Two important characteristics of time-critical systems are: the requirement to satisfy stringent timing constraints, and the need to guard against an imperfect execution environment. In this paper, we formalize the safety analysis of timing properties in real-time systems. Our analysis is based on a formal logic: RTL (Real-Time Logic) which is especially suitable for reasoning about the timing behavior of systems. Given the formal specification of a system and a safety assertion to be analyzed, our goal is to relate the safety assertion to the systems specification. There are three distinct cases: 1) the safety assertion is a theorem derivable from the systems specification, 2) the safety assertion is unsatisfiable with respect to the systems specification, or 3) the negation of the safety assertion is satisfiable under certain conditions. A systematic method for performing safety analysis will be presented.

Index Terms—Real time, real-time logic, safety analysis, systems specification, time-critical system, verification.

I. INTRODUCTION

WITH the availability of inexpensive and powerful microprocessors, there has been a rapid increase in the use of computers in real-time applications, e.g., industrial plant control, digital fly-by-wire avionic systems. For these applications, the issues of safety and reliability are particularly important because a failure may result in catastrophic destruction of life and property. As is well known, the software systems for these applications are difficult to design and analyze owing to their time-critical nature, and their reliability has been of much concern to software designers and users. Two major areas of research addressing software reliability have been: formal verification methods which attempt to prove the correctness of programs with respect to systems specifications, and software engineering which promotes adherence to programming principles to restrict the complexity of large software systems.

Although prevention and elimination of errors are essential in any trustworthy software system, past research efforts have not fully addressed issues peculiar to real-time systems. Specifically, they do not take into account two important characteristics of the real-time environ-

ment, namely, the need for the computation to continually satisfy stringent timing constraints, and the need to guard against an imperfect execution environment which may violate design assumptions. (Most time-critical systems interact with physical processes in the external world and design assumptions, e.g., model of the external world are often implicitly built into the software.) The objective of this paper is to formalize the safety analysis of timing properties in real-time systems.

A. The Approach

Our analysis is based on a formal logic RTL (Real-Time Logic) which is especially amenable to reasoning about the timing behavior of systems. Given the timing specification of a system and a safety assertion to be analyzed, the goal is to relate the safety assertion to the systems specification. There are three cases: 1) The safety assertion is a theorem derivable from the systems specification, 2) The safety assertion is unsatisfiable with respect to the systems specification. 3) The negation of the safety assertion is satisfiable under certain conditions. In the first case, the system is safe with respect to the behavior denoted by the safety assertion, as long as an implementation satisfies the requirements specification. In the second case, the system is inherently unsafe because the requirements specification will cause the safety assertion to be violated. In the third case, additional constraints must be imposed on the system to ensure its safety.

While we emphasize formal analysis, our goal is to create practical tools for software engineers to use. (The work reported in this paper is part of a larger effort to automate the analysis and synthesis of real-time systems. An overview of this project, called SARTOR, can be found in [9].) To capture the semantics of practical real-time systems, we introduce a computation model: the event-action model which enables the system designer to specify the timing behavior of a system without having to worry about the formal representation and mechanical aspects of the reasoning process (analysis). Our choice of this event-action model is guided closely by the work of Parnas et al. [5] which completely documented the software requirements of an operational aircraft; we want our model to be sufficiently realistic to describe a typical time-critical application. For formal analysis, instances of the event-action model will be translated into RTL formulas by a mechanical procedure to be described later.

Manuscript received August 30, 1985; revised February 28, 1986. F. Jahanian was supported by a gift from the Lockheed Missiles and Space Company, Inc., to the University of Texas at Austin. A. K. Mok was supported by a research grant from the Office of Naval Research under Contract N00014-85-K-0117.

The authors are with the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712.
IEEE Log Number 860740.
B. Relation to Other Work

The notion of safety or reliability analysis is well known to reliability engineers, and there are established techniques such as fault trees and event trees [10] for analyzing the failure modes and failure probability of mechanical systems. Recently, Leveson has applied fault tree analysis to software systems [8]. She proposed the use of a software fault-tree to identify failure scenarios and relate them to run-time (external) conditions which could cause a system to fail. The result of the analysis was used to trace possible design errors leading to system failures, and to determine the specific conditions which must be satisfied to avoid potential failures. However, there was little discussion on how to mechanize the generation of a fault tree from a program and the analysis presented by Leveson relied heavily on human inspection. More importantly, there was no attempt to formally capture the timing behavior of a system. The reader is referred to a survey paper by Leveson [7] for additional references on software fault tree analysis. (Reference [7] presents a survey of issues and problems in software safety along with some of the existing techniques for solving these problems.)

Our work differs from others in two important aspects. Firstly, our overall goal is to synthesize software which has certain prescribed timing properties as dictated by an analysis of the systems specification. Thus our logic RTL is mainly for analyzing systems specifications instead of detailed program designs. This is important for keeping the analysis manageable for mechanization, since automated analysis of large finished designs is not likely to be practical. Secondly, we are primarily interested in the timing behavior of systems. Our logic is specifically invented to describe systems for which the absolute timing of events as well as their relative ordering is important.

In Section II, we present the event-action model together with a design example. Section III introduces our Real Time Logic (RTL). Section IV describes how an event-action specification is mechanically transformed into an RTL specification. An inference mechanism for reasoning about the timing behavior of real-time systems is presented in Section V. Section VI contains the concluding remarks and future direction of our work.

II. The Event-Action Model

The purpose of our event-action model is to capture the data-dependency and temporal ordering of the computational actions that must be taken in response to events in a real-time application. There are four basic concepts in this model:

ACTION: Actions are schedulable units of work. They are either primitive or composite. A primitive action is an operation which consumes a bounded amount of system resources (cpu or communication bandwidth). The implementation of primitive actions (programming-in-the-small) is assumed to be given and their effects on the system are captured by causality relations. A causality relation relates the events that occur before and after an action is executed. In this paper, we shall be concerned primarily with assertions that express timing constraints (see TIMING CONSTRAINT below). A composite action is a partial ordering of other actions (multiple occurrence of an action in a composite action is allowed). If the partial ordering of a composite action, say X, contains the action Y, then Y is said to be a subaction of X. Composite actions are not permitted to be circularly defined, i.e., we do not allow any set of composite actions to form a circular chain such that each action in the chain is a subaction of its predecessor in the chain.

The syntax for an action is its unique name. A composite action is expressed by means of notations which constrain the precedence ordering of its subactions. In this paper, we shall use the notation "| X; Y |" and "| X || Y |" to denote, respectively, the sequential execution of X followed by Y and the parallel execution of X and Y. In the case that the partial ordering of a composite action is not a series/parallel graph, the notation "| ! N |" can be used to denote a synchronization point labeled N; the notations "| X | ! N |" and "| ! N Y |" together denote the precedence constraint that the execution of action X must be completed before the execution of action Y can start. Examples of primitive actions are: TOP (Turn On Pump) and LCD (Lock Chamber Door). An example of a composite action is: PC (Pressurize Chamber) whose partial ordering is given by LCD; TOP.

STATE PREDICATE: A state predicate is an assertion about the physical state of the system in which the embedded computer(s) functions. The value of a state predicate may change as a result of the execution of an action or of an external event and is in general a function of time. State predicates have unique names. In this paper, we shall use acronyms of the corresponding physical states, e.g., CP (Chamber is Pressurized).

EVENT: An event serves as a temporal marker, i.e., the occurrence of an event marks a point in time which is of significance in describing the behavior of the system. We distinguish between four classes of events: 1) External event which cannot be caused to happen by actions taken by the system but can cause other events to happen, e.g., pilot pushes the red button. 2) Start event which marks the initiation of an action. 3) Stop event which marks the completion of an action. 4) Transition event which marks a change in certain attribute of the system state, e.g., airplane becomes airborne. Events have unique names, e.g., PDBSL (Pressure Drops Below Safety Level).

TIMING CONSTRAINT: A timing constraint is an assertion about the absolute timing of certain events. Timing constraints are often specified as performance requirements of a system. Two types of timing constraints are of particular interest to us: periodic and sporadic timing constraints. A periodic constraint requires some action to be executed at fixed intervals while some state predicates are true. A sporadic timing constraint requires some action to be executed before a specified deadline elapses after the occurrence of a certain event. The syntax of a periodic
timing constraint is:

\[ \text{While } \langle \text{state predicate} \rangle, \text{execute } \langle \text{action} \rangle \text{ with} \]
\[ \text{period } = \langle \text{time} \rangle, \text{deadline } = \langle \text{time} \rangle \]

The syntax of a sporadic timing constraint is:

\[ \text{When } \langle \text{event} \rangle, \text{execute } \langle \text{action} \rangle \text{ with} \]
\[ \text{deadline } = \langle \text{time} \rangle, \text{separation } = \langle \text{time} \rangle \]

The separation parameter in a sporadic timing constraint specifies a lower bound on the length of an interval separating two successive occurrences of the triggering event. The purpose of the separation parameter is to prevent any source of sporadic requests for computation to hog the system. However, since \( E \) may be an external event, the constraint represented by the second assertion above may appear to be unreasonable. In practice, a hardware buffer can be used to queue up a sporadic request if it occurs too close to a previous occurrence, in which case the specification only guarantees that sporadic request will be serviced at a maximum rate indicated by the separation parameter. We now give an example of an event-action specification for a simple real-time system.

Example 1: This specification describes the landing system for a Martian Lander. The landing system operates in one of two modes: normal or emergency. When the system is turned on (an external event), it operates in the normal mode.

While in the normal landing mode, the pilot can control acceleration, velocity, and position by adjusting the downward thrust generated by the rocket motor, thus bringing the space vehicle to a safe landing. This task is continually performed in two phases. In phase 1, an I/O device is started to read the acceleration set by the pilot, and a hardware timer is started which generates a timer interrupt (i.e., external event TMRINT) after 100 ms. In phase 2, when the I/O operation is completed (i.e., external event IOINT), the motor thrust is adjusted appropriately. However, if for some reason the I/O operation is not done within 100 ms, the timer interrupt initiates the emergency landing mode.

While in the emergency mode, the altitude and velocity is periodically sampled and a retro-rocket is automatically fired to bring the vehicle to a safe landing.

There are four timing constraints, two periodic and two sporadic, mentioned in the English description above. We give below the event-action model specification of the landing system for our Martian Lander. The computation times of each primitive action is shown in Table I. We note that the computation time of an action is processor dependent.

<table>
<thead>
<tr>
<th>Primitive Actions</th>
<th>Action</th>
<th>Time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>RACC</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>STMR</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>ADJM</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>TDP</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>IEM</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>ETC</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>IVEL</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>IALT</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>CKDT</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>RRM</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Composite Actions</th>
<th>Action</th>
<th>Time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP1</td>
<td>Phase 1 of Normal Operation of Landing System: RACC, STMR</td>
<td></td>
</tr>
<tr>
<td>NOP2</td>
<td>Phase 2 of Normal Operation of Landing System: ADJM</td>
<td></td>
</tr>
<tr>
<td>TIH</td>
<td>Timer Interrupt Handler: IEM; ETC</td>
<td></td>
</tr>
<tr>
<td>EOP</td>
<td>Emergency Operation of Landing System: (IVEL</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State Predicates</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELSM</td>
<td>Emergency landing system mode is initially false</td>
</tr>
<tr>
<td>DONE</td>
<td>I/O status flag denotes no I/O in progress, initially true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>External Events</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>Systems start up</td>
</tr>
<tr>
<td>IOINT</td>
<td>I/O device signals completion, DONE to true</td>
</tr>
<tr>
<td>TMRINT</td>
<td>Timer Interrupt, occurs at least 100 after start of STMR</td>
</tr>
</tbody>
</table>

Timing Constraints

1. While ELSM is off, execute NOP1 with period = 200 ms, deadline = 40
2. When external events IOINT occurs, execute NOP2 with deadline = 60 ms, separation = 100
3. When external event TMINT occurs, execute TIH with deadline = 60 ms, separation = 100

4. While ELSM is on, execute EOP with period = 200 ms, deadline = 100

III. Real-Time Logic (RTL)

While the event-action model captures the timing requirements of a real-time system, we need a representation which is more amenable to mechanical manipulation by a computer in a formal analysis. In this section, we introduce a formal language: RTL (Real-Time Logic) for this purpose. Although the analysis of real-time systems necessarily involves reasoning about system behavior with respect to time, we do not think that temporal logic is appropriate for the task at hand. However, since a variety of temporal logics have been proposed by a number of researchers for reasoning about the temporal properties of real-time programs (e.g., [1], [11], [12]), a discussion to justify our choice is in order.

In general, temporal logic is more concerned with the relative order in which actions are executed rather than the absolute timing of events. For example, the action (in our event-action model terminology)

\[
A : (B \parallel C)
\]

is characterized in temporal logic by two execution sequences: ABC and ACB, but nothing is said about the completion time of C in the first sequence or the completion time of B in the second sequence. For our purposes, these absolute time values may appear in a safety assertion and are therefore important. Even for the same sequence ABC, we might want to distinguish between the execution that schedules A at time = 0, B at time = 1, and C at time = 2 from the execution that schedules A at time = 0, B at time = 2, and C at time = 3. For temporal logic, this distinction is unimportant. Furthermore, it is at the least awkward in some cases to reason about system behavior in terms of execution sequences alone. For example, if all three actions A, B, and C each takes 1 time unit to execute and the composite action above must be completed by time = 2, then B and C must be executed in parallel on two processors. In this case, neither of the sequences ABC nor ACB captures the desired behavior.

It has been suggested by some researchers (e.g., [6]) that real time can be modeled in temporal logic simply as another global variable: the clock, and then assertions involving real time will simply be temporal logic formulas involving the clock variable. The main problem with this approach is when to increment the clock variable in relation to the other activities in the system. One possible way to achieve this is to insert the assignment statement:

\[
\text{clock} := \text{clock} + c
\]

at the end of every action where \( c \) is the time required to execute the corresponding action. If all actions are executed in some sequential order, such as on a single processor, then the clock variable will indeed keep track of real time. (For this to work, we should treat each action and the clock update statement at its end to be one inseparable atomic action.) Unfortunately, this is not true when two or more actions can be executed in parallel.

Another way to model real time in temporal logic is to create a process whose body is an infinite loop which increments the clock variable ad infinitum. This approach, however, only begs the question since we must then find appropriate scheduling restrictions so as to meter the progress of the clock process in relation to the other processes in the system. Since most temporal logics dictate only minimal restrictions (usually a fairness criterion) on process scheduling, imposing additional scheduling restrictions seems to move against the original spirit of using temporal logic for program verification. More importantly, these additional scheduling restrictions will depend on the execution environment, e.g., the rate of scheduling the clock process when there is one processor will differ when two processors are available to execute all the processes. Proof rules which are sound under one execution environment may not apply under a different execution environment.

The previous discussion brings out a fundamental issue in proving real-time properties of time-critical systems, namely, the validity of the assertions that we want to prove about these systems often cannot be established without knowing more details about the run-time scheduler. This is unlike the usual safety and liveness properties of non-time-critical systems which we want to hold true in spite of the scheduler as long as we are assured that some fairness criterion in scheduling is met. Our Real-Time Logic promises to provide a uniform way to incorporate different scheduling disciplines in the inference mechanism. RTL reasons about occurrences of events: execution of actions are represented by occurrences of start and stop events. Time is captured by a function (the ‘@’ function to be described in a later section) which assigns time values to event occurrences. Timing constraints and scheduling disciplines are restrictions on the ‘@’ function. A safety property is established if there is no mapping of event occurrences to time values which is consistent with the negation of the safety property and all the other specified restrictions on the ‘@’ function. RTL is a first order theory and has no modal operators. We now give the details of RTL. A summary of the syntax of RTL is given in Appendix A.

A. Action Constants

There are three types of constants in RTL: action constants, event constants, and integers. Action constants are introduced in this subsection and event constants in the following subsections.

Actions have unique names. We shall use capital letters in naming action constants so as to distinguish them from variable names. For example, the constant RACC can be used to denote the primitive action associated with reading the acceleration setting commanded by the pilot.
Actions are either primitive or composite. A composite action is a partial ordering of other actions. A syntactic definition of a composite action is given by a text string representing the corresponding partial ordering, using the symbols "\cdot", "\|", and "\|N" to denote precedence constraints as in the event-action model. An action may appear in more than one composite action, and an action may appear more than once in a single composite action. Composite actions are not permitted to be circularly defined. If an action \( A \) appears exactly once in a composite action \( B \), then we create an action constant that denotes the unique occurrence of \( A \) in \( B \). The syntax for this constant is:

\[ B.A \]

If action \( A \) appears more than once within composite action \( B \), the notation \( B.A \) denotes the \( i \)th appearance of \( A \) in the syntactic definition of \( B \). The following example illustrates the notion of primitive and composite actions.

**Example 2:** Consider the following composite actions, \( CA \) and \( CB \).

\[ CA: \text{SAMPLE; TEST; SAMPLE; TEST; CALCULATE; TRANSMIT} \]
\[ CB: (\text{INPUT1} \parallel \text{INPUT2}); \text{CALCULATE} \]

If we need to specify an assertion about the action which performs sampling, we use the action constant \( \text{SAMPLE} \) in the assertion. If the assertion specifies something about the second sampling action within composite action \( CA \), then the action constant \( CA.\text{SAMPLE} \) will be used. Similarly, the \( \text{CALCULATE} \) action in the composite action \( CA \) (denoted by \( CA.\text{CALCULATE} \)) will be distinguished from the \( \text{CALCULATE} \) action in \( CB \) (denoted by \( CB.\text{CALCULATE} \)).

**B. Event Classes and Event Constants**

As mentioned earlier, events serve as temporal markers when describing the timing behavior of real-time systems. Event constants are categorized into four classes: 1) start events marking the initiation of actions, 2) stop events marking the completion of actions, 3) transition events marking a change in certain attributes of the system state, and 4) external events.

For a given real-time system, each event class consists of a set of event constants. Each event constant corresponds to a specific event in the system. For instance, the class of external events may contain the events corresponding to "clock tick," "pressing button #8," etc. It should be noted that event constants are different from event occurrences which will be introduced in a later section in relation to the "@" function. Subsections III-C-E below present the RTL syntax for each class of event constants.

**C. Start and Stop Event Constants**

Start and stop event constants represent the initiation and completion of actions. We use the notation \( \uparrow A \) to represent the event marking the initiation of action \( A \). Similarly, \( \downarrow A \) denotes the event marking the completion of action \( A \). We emphasize that \( A \) can be any action constant as described in Section III-A. Thus \( \uparrow B.A \) denotes the event marking the start of subaction \( A \) within the composite action \( B \). Similarly, \( \downarrow B.A \) signifies the event associated with the start of the second appearance of subaction \( A \) in the composite action \( B \).

For a primitive action, the time of occurrence of a start or stop event is defined by the implementation of the action. A composite action is said to have started at time \( t \) if \( t \) is the minimum of the start-times of all of its subactions, and it is said to have stopped at time \( t \) if \( t \) is the maximum of the stop-times of all of its subactions.

**D. Transition Event Constants**

A transition event marks a change in a certain state attribute. A state attribute is an assertion about the state of the real-time system and its environment, e.g., the oil pressure of an engine being below safety level. We use the notation

\[ (S := T) \]

to represent the transition event that makes the state attribute \( S \) true. Similarly, we use

\[ (S := F) \]

to denote the transition event which makes the state attribute \( S \) false. It should be noted that state attributes appear only in the syntax of transition events. State attributes are not themselves constants of RTL although they define transition events. More importantly, they should not be confused with the predicates in RTL. The truth value of a state attribute over a time interval can be described by a special class of predicates, which will be discussed in Section III-G.

**E. External Event Constants**

External events are the ones that cannot be caused to happen by the computer system but can impact system behavior. We use the notation of any name in capital letters prefixed by the special character Ω (omega) to denote an external event. For instance, ΩCLOCK denotes the external event corresponding to a clock tick; ΩBUTTON1 denotes the external event associated with pressing button #1.

**F. The Occurrence Function**

The occurrence function, denoted by the character "@", is introduced to capture the notion of real time. The event constants introduced earlier represent the things that can happen in a real-time system. For a given mission, a particular event may happen periodically, sporadically or not at all. The occurrence function is a mapping from the space \((E, W)\) to \(W\) where \( E, W \) are, respectively, the set of event constants and nonnegative integers.
Definition 1: \( @ (e, i) \equiv \text{time of the } i\text{th occurrence of event } e \); where \( e \) is a start, stop, external, or transition event constant, and \( i \) is an integer constant/variable.

The notion of an occurrence function is central to RTL.
In particular, a timing property (an RTL assertion) of a system can be established by showing that there is no occurrence function which is consistent with the systems specification in conjunction with the negation of the property under investigation. The occurrence function is in general not a total function since some sporadic events may occur only a finite number of times or even not at all.

Occurrence Function Axioms: For each event constant \( E \) in the system specification,
\[
\forall i @ (E, i) = t \rightarrow i \geq 0
\]
\[
\forall i \forall j [ @ (E, i) = t \land @ (E, j) = t' \land i < j ] \rightarrow t < t'
\]

Notice that we do not write the first axiom as \( \forall i @ (E, i) \geq 0 \), since the \( i \)th instance of the event \( E \) may not occur at all. The second axiom expresses the requirement that two occurrences of the same event must be distinguishable.

Start/Stop Event Axioms: For each action \( A \) in the system specification,
\[
\forall i \forall t @(A, i) = t \rightarrow @(A, i) < t
\]
For each subaction \( A \) in the syntactical definition of a composite action \( C \),
\[
\forall i \forall t @(\uparrow A, i) = t \rightarrow 3j @ (\uparrow A, j) = t
\]
\[
\land [ @(\uparrow A, i) = t' \land @(\downarrow A, j) = t' ]
\]

The first axiom requires every occurrence of a stop event to be matched by a corresponding start event. The second axiom defines the meaning of executing a subaction of a composite action: if \( X \) is an action constant (a name) created to identify an action which is in the partial ordering of a composite action, then every execution of \( X \) must correspond to an execution of the action it identifies. Notice that RTL does not forbid a single execution of an action to match the executions of two distinct subactions both of which identify the same action. This is because we want to allow for situations where the timely execution of an action will satisfy two or more timing constraints simultaneously. For example, consider an alarm which must be triggered shortly after a fire or a flood occurs, but it is sufficient to trigger the alarm only once if both a fire and a flood happen to break out simultaneously.

Transition Event Axioms: A state attribute \( S \) is defined to be true initially iff the transition event \( (S := T) \) occurs at time 0, i.e., \( @ ((S := T), 1) = 0 \). Similarly, a system attribute \( S \) is defined to be false initially iff the transition event \( (S := F) \) occurs at time 0, i.e., \( @ ((S := F), 1) = 0 \).

For each state attribute \( S \) in the system specification, if \( S \) is initially true, the following axioms apply
\[
\forall i @ ((S := F), i) = t \rightarrow @(S := T), i < t
\]
\[
\forall i @ ((S := T), i + 1) = t \rightarrow @(S := F), i < t
\]
If \( S \) is initially false, the following axioms apply
\[
\forall i @ ((S := T), i) = t \rightarrow @(S := F), i < t
\]
\[
\forall i @ ((S := F), i + 1) = t \rightarrow @(S := T), i < t
\]
The preceding transition event axioms define the order in which two successive transition events of the opposite sign can occur depending on whether \( S \) is true or false initially.

With the @ function, we can express the real-time behavior of time-critical systems. In particular, the @ function can be used in algebraic relations to express stringent timing constraints.

Example 3:
\[
\forall i @ (@\text{BUTTON1}, i) < @ (@\text{SAMPLE}, i) \land @ (@\text{SAMPLE}, i) \leq @ (@\text{BUTTON1}, i + 20)
\]
This is interpreted as: action SAMPLE is executed only after button number 1 is pressed, and every execution of SAMPLE must complete within 20 time units of the corresponding press of button number 1.

G. State Predicates

Safety assertions about real-time systems often talk about the physical state of a system over time. One way to express these assertions is to introduce predicates which are time-dependent, as is done in some temporal logics. In RTL, these assertions are translated into algebraic relations involving the occurrences of appropriate transition events. For convenience, RTL uses a notational device, called a state predicate, for expressing the truth value of a state attribute (e.g., the autopilot switch of an airplane being in the ON position) during an interval. It should be emphasized that we purposely refrain from talking about the state of a system at a certain point in time because the system state may be undergoing a transition right at that point and the value of a state attribute is undefined when it is changing.

Suppose \( S \) is a state attribute whose truth value remains unchanged over an interval. RTL provides nine different forms of a state predicate to assert the value of \( S \) over an interval, depending on the boundary conditions: \( S[x, y] \), \( S(x, y) \), \( S < x, y > \), \( S[x, y] \), \( S(x, y) \), \( S(x, y) > \), \( S(x, y) > \), \( S < x, y > \), and \( S < x, y > \). Each state predicate qualifies the timing of two events, one marking the transition event that changes the value of a state attribute \( S \) to true and the other marking the transition event that changes the value of \( S \) to false. The two arguments, \( x \) and \( y \), in the state predicates are used in conjunction with the symbols ‘[’, ‘]’, ‘(‘, ‘)’, ‘<’, ‘>’, to denote an interval over which the state attribute remains true.

The convention we use for arriving at this syntax requires some explanation. Suppose \( E_r \) and \( E_l \) denote the transition events making \( S \) true and false, respectively. Informally,
\[ [x] \] denotes that \( E_i \) occurs at time \( x \).
\[ (x) \] denotes that \( E_i \) occurs before or at time \( x \).
\[ < x \] denotes that \( E_i \) occurs before time \( x \).
\[ y \] denotes that \( E_i \) occurs at time \( y \).
\[ \neq y \] denotes that \( E_i \) does not occur before time \( y \).
\[ y \] denotes that \( E_i \) does not occur before or at time \( y \).

For example, the state predicate \( S[x, y] \) indicates that a state attribute \( S \) is true exactly between the interval \( x \) and \( y \). That is, a transition event making \( S \) true occurs at time \( x \); \( S \) remains true between time \( x \) and \( y \); and the transition event making \( S \) false occurs at time \( y \). Consider another example, the state predicate \( S(x, y) \) denotes that a state attribute \( S \) is true during the interval between \( x \) and \( y \), but it says nothing about the value of \( S \) before time \( x \) or after time \( y \). Observe that we use this notational device only when \( x < y \) and that we do not require the second event to occur in all cases. The formal definitions of these state predicates follow.

**Definition 2:**

\[
S[x, y] = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) = x \land \\
y = \forall ((S := F), j) \land x < \forall ((S := F), k) \land \forall ((S := F), k) < y
\]

\[
S[x, y] = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) = x \land \\
x < \forall ((S := F), k) \land \forall ((S := F), k) < y
\]

\[
S[x, y] = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) = x \land \\
x < \forall ((S := F), k) \land \forall ((S := F), k) \leq y
\]

\[
S(x, y) = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) \leq x \land x < \forall ((S := F), k) \land \forall ((S := F), k) < y
\]

\[
S(x, y) = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) \leq x \land \\
y = \forall ((S := F), j) \land x < \forall ((S := F), k) \land \forall ((S := F), k) < y
\]

\[
S(x, y) = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) \leq x \land x < \forall ((S := F), k) \land \forall ((S := F), k) \leq y
\]

\[
S[x, y] = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) < x \land x < \forall ((S := F), k) \land \forall ((S := F), k) < y
\]

\[
S < x, y > = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) < x \land \forall ((S := F), k) \land \forall ((S := F), k) < y
\]

\[
S < x, y > = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) < x \land \forall ((S := F), k) \land \forall ((S := F), k) < y
\]

\[
S < x, y > = \text{true iff } \exists i \exists j \exists k \ ((S := T), i) < x \land \forall ((S := F), k) \land \forall ((S := F), k) < y
\]

Similar state predicates can be defined for the case when a state attribute \( S \) is false during an interval: \( \overline{S}[x, y], \overline{S}(x, y), \overline{S} < x, y >, \) etc. The formal definitions for these predicates are straightforward: replace each \((S := T)\) with \((S := F)\) and vice versa in the above definitions.

The preceding definitions for state predicates are a bit cumbersome to use in a formal analysis. They can be considerably simplified if the initial value of the state attribute involved is known. The following lemmas provide alternative definitions for state predicates which assert the truth of a state attribute \( S \) during an interval. Similar lemmas hold for the state predicates defined for the case when a state attribute \( S \) is false.

**Lemma 1:** If a state attribute \( S \) is initially true,

\[
S[x, y] = \exists i ( (S := T), i) = x \land y = \forall ((S := F), i)
\]

\[
S(x, y) = \exists i ( (S := T), i) = x \land \forall ((S := F), i) = t \to y \leq t
\]

\[
S[x, y] = \exists i ( (S := T), i) = x \land \forall ((S := F), i) = t \to y < t
\]

\[
S(x, y) = \exists i ( (S := T), i) \leq x \land \forall ((S := F), i) = t \to y \leq t
\]

\[
S(x, y) = \exists i ( (S := T), i) \leq x \land \forall ((S := F), i) = t \to y < t
\]

\[
S[x, y] = \exists i ( (S := T), i) < x \land \forall ((S := F), i) = t \to y < t
\]

\[
S < x, y > = \exists i ( (S := T), i) < x \land \forall ((S := F), i) = t \to y < t
\]

\[
S < x, y > = \exists i ( (S := T), i) < x \land \forall ((S := F), i) = t \to y \leq t
\]
Lemma 2: If a state attribute \( S \) is initially false,
\[
S[x, y] = \exists i @((S := T), i) = x \land y = @((S := F), i + 1)
\]
\[
S[x, y] = \exists i @((S := T), i) = x \land [\forall t @((S := F), i + 1) = t \to y \leq t]
\]
\[
S(x, y) = \exists i @((S := T), i) \leq x \land [\forall t @((S := F), i + 1) = t \to y \leq t]
\]
\[
S(x, y) = \exists i @((S := T), i) \leq x \land y = @(S := F), i + 1)
\]
\[
S(x, y) = \exists i @((S := T), i) \leq x \land [\forall t @((S := F), i + 1) = t \to y \leq t]
\]
\[
S<x, y] = \exists i @((S := T), i) < x \land [\forall t @((S := F), i + 1) = t \to y \leq t]
\]
\[
S<x, y] = \exists i @((S := T), i) < x \land y = @(S := F), i + 1)
\]
\[
S<x, y] = \exists i @((S := T), i) < x \land [\forall t @((S := F), i + 1) = t \to y \leq t]
\]

When both arguments of a state predicate are the same, two very useful predicates are defined. Specifically, the state predicate \( S(x, x) \) says that a state attribute \( S \) is true at an interval around time \( x \). Similarly, \( S < x, x \) denotes that \( S \) is true prior to time \( x \), i.e., \( S \) becomes true sometimes before time \( x \) and it remains true at least up to time \( x \).

IV. Transformation from the Event-Action Model to RTL

In this section we will show how the specification of a system in the event-action model can be mechanically transformed into a set of formulas in Real Time Logic. The resulting RTL formulas are axioms about the real-time system being analyzed.

A. Relationship between an Action and Its Start/Stop Events

As discussed earlier, start and stop events mark the initiation and completion of actions. An action is said to be executing during the internal delimited by the occurrences of the corresponding start and stop events. The following axiom applies to all actions.

\[
\forall i, \forall j, \forall i @((A, i) = t_1 \land \leq @((A, i) = t_2) \to t_1 + c(A) \leq t_2
\]

The preceding assertion is interpreted as: each occurrence of the event denoting the completion of action \( A \) happens at least \( c(A) \) time units after the corresponding occurrence of the start event, where \( c(A) \) denotes the specified execution time of action \( A \).

In performing safety analysis, we often replace the preceding assertion by a simpler form.

\[
\forall i @((A, i) + c(A) \leq @((A, i)
\]

The justification for allowing this replacement follows from the procedure used by Bledsoe and Hines [3] which is used to perform safety analysis in this paper. (The key ideas behind the procedure are described in Section V). An important step in the procedure eliminates a universally quantified variable from a clause if the variable is not an argument of interpreted functions within that clause. Applying the variable elimination step to the original assertion above removes variables \( t_1 \) and \( t_2 \), yielding the simpler form. In the remainder of this paper, assertions are presented in the simpler form.

B. Constraints Imposed by Composite Actions

A composite action places certain precedence constraints on the execution of its subactions. To mechanize timing analysis, it is necessary to express them in terms of assertions in RTL. In general, if subaction \( B \) of a composite action \( A \) precedes subaction \( C \), then we add the assertion which says that the time at which \( C \) starts must not be earlier than the time at which \( B \) is completed in the same execution of \( A \). We give the translations for the precedence relations, \(''\ldots', '','\ldots'',' and '','\ldots'' below.

Sequencial Constraint: Suppose \( B \) and \( C \) are subactions of a composite action \( A \). If ‘‘\( B \ll C \)’’ and ‘‘\( B \ll C \)’’ appear in the syntactic definition of \( A \) in the event-action model, then we add the following assertion in RTL.

\[
\forall i @((A, B), i) \leq @((A, C), i)
\]

The same applies for the constraint ‘‘\( B \ll C \)’’.

Parallel Constraint: Suppose \( B \), \( C \), and \( D \) are subactions of a composite action \( A \). If ‘‘\( B \parallel C \)’’ and ‘‘\( B \parallel C \)’’ appears in the syntactic definition of \( A \), then we add:

\[
\forall i @((A, B), i) \leq @((A, C), i)
\]

\[
\forall i @((A, B), i) \leq @((A, D), i)
\]

Similarly, if ‘‘\( \ldots, B \parallel C \ldots \),\( D \)’’ appears in \( A \), then we add:

\[
\forall i @((A, B), i) \leq @((A, D), i)
\]

\[
\forall i @((A, C), i) \leq @((A, D), i)
\]

Constraints Imposed by the Start/Stop Events: The time of occurrence of a start/stop event of a composite action is defined by the initiation/completion of its subactions. Specifically, the start-time of an instance of a composite action is defined to be the minimum of the start-times of all its subactions which have no predecessors. Similarly, the stop-time of an instance of a composite action is the maximum of the stop-times of all its subactions which have no successors.
The constraints imposed by the start event of a composite action can be expressed in RTL as follows. If $X$ is an action constant denoting a subaction of a composite action $A$ and $X$ has no predecessors in $A$, then
\[ \forall i \overline{\exists} (\overline{\forall} A, i) \leq \overline{\forall} (\overline{\exists} X, i) \]
\[ \forall i \overline{\exists} (\overline{\forall} A, i) = \overline{\forall} (\overline{\exists} X_1, i) \lor \overline{\forall} (\overline{\exists} X, i) \]
where $X_1, X_2, \cdots X_4$ are all the action constants that denote subactions of $A$ which have no predecessors.

Similarly, the stop event of a composite action $A$ satisfies the assertions:
\[ \forall i \overline{\exists} (\overline{\forall} A, i) \geq \overline{\forall} (\overline{\exists} X, i) \]
\[ \forall i \overline{\exists} (\overline{\forall} A, i) = \overline{\forall} (\overline{\exists} X_1, i) \lor \overline{\forall} (\overline{\exists} A, i) \]
where $X_1, X_2, \cdots X_4$ are all the action constants that denote subactions of $A$ which have no successors.

C. Sporadic Timing Constraint
Recall that a sporadic timing constraint requires some action to be executed within a specified time interval after the occurrence of a particular event. We also associate a separation parameter with each sporadic timing constraint which specifies a lower bound on the length of the time interval between two consecutive occurrences of the triggering event. The syntax of a sporadic timing constraint in the event-action model is:

when $E$ occurs, execute $A$ with deadline = $d$,
separation = $p$

where $A$ is an action and $E$ is the triggering event. The above sporadic timing constraint can be expressed by the following assertions in RTL:
\[ \forall i \exists j \overline{\forall} (E, i) \leq \overline{\forall} (\overline{\exists} A, j) \land \overline{\forall} (\overline{\exists} A, j) \leq \overline{\forall} (E, i) + d \]
\[ \forall i \overline{\forall} (E, i) + p \leq \overline{\forall} (E, i + 1) \]

We note that if $E$ is an external event, then $\overline{\forall} (E, i)$ is the time of the $i$th occurrence of $E$ as recognized by the computer: there is always a finite delay between the occurrence of a physical event in the external world and its recognition by the computer system. This delay is determined by the speed of the interrupt hardware and whether interrupts are queued, etc.

D. Periodic Timing Constraint
Recall that a periodic timing constraint requires some action to be executed periodically as long as a certain state predicate is true (or false). A periodic timing constraint is expressed in the event-action model by:

while $S$ is true, execute $A$ with period = $p$, deadline = $d$

where $S$ is a state attribute and $A$ is an action. The following assertion in RTL captures the temporal relationship expressed by the preceding periodic timing constraint. Notice the use of the state predicate $S(x, y)$ which says that the state attribute $S$ becomes true at time $x$, remains true in the interval $[x, y)$, and may become false only at or after time $y$.

\[ \forall x \forall y \exists i S(x, y) \land y - x > n^* p \rightarrow x + n^* p \leq \overline{\forall} (\overline{\exists} A, i) \land \overline{\forall} (\overline{\exists} A, i) \leq x + n^* p + d \]

E. Causal Assertions
In a real-time system, a state attribute may change value and a transition may occur because of: 1) the occurrence of an external event, or 2) the execution of an action by

\[ \overline{\forall} (\overline{\exists} X_2, i) \lor \cdots \lor \overline{\exists} (\overline{\forall} A, i) = \overline{\forall} (X_4, i) \]

the system. RTL allows a uniform treatment of both cases. Specifically, both cases can be viewed as an event causing another event. In the first case, we have the occurrence of an external event implying the occurrence of a transition event. In the second case, the occurrence of a stop event of an action implies the occurrence of a transition event.

For each external event $E$ that changes the value of a state attribute $S$ to true, the following causal assertions can be made:
\[ \forall i \exists j \overline{\forall} (E, i) \leq \overline{\exists} S < t, i \rightarrow \overline{\forall} (S := T, j) = i \]

If the result of executing an action $A$ is a transition event, then causal assertions of the following form may be specified:
\[ \forall i \exists j \overline{\forall} (A, i) \leq \overline{\exists} S < t, i \rightarrow \overline{\forall} (S := T, j) = i \]

Similar assertions are needed for events that change the value of $S$ to false.

F. Accountability Assertions
In the verification of safety properties, we often require assertions which say that certain events must have happened no later than the occurrence of a particular event. These assertions are in general not directly deducible from the systems specification. The intuitive reason for this is that even though the timing constraints and causal assertions do specify what must happen, they do not automatically forbid a run-time scheduler from executing actions which are consistent with the timing constraints but are not required by them, i.e., these actions can be skipped without violating any of the timing constraints. A malicious scheduler may be able to execute superfluous actions which can cause a safety assertion to be violated; there are more than sufficient resources for satisfying the timing constraints. To prevent a run-time scheduler from maliciously violating a safety assertion, we introduce restrictions on the scheduler and call them accountability assertions.

The general form for this second type of causal assertions is:
\[ \forall i \exists j (p \land \overline{\exists} (A, i) = i) \rightarrow \overline{\forall} (S := T, j) = i \]

where $p$ is a predicate which can be interpreted as a precondition for transition event to occur as a result of the action $A$. The event-action triple described in this paper does not provide for the specification of this type of causal assertions. Causal assertions allowing different forms of the $p$ condition will be pursued in future work.
assertions. Accountability assertions are somewhat analogous to the weakest precondition of Dijkstra [4].

The idea is to permit the scheduler to execute an action \( A \) only if that instance of \( A \) can be counted towards meeting a timing constraint which requires the execution of \( A \). The following assertion, which we refer to as the accountability assertion for timing constraints, must be added for each action \( A \) appearing in the syntactical definition of a timing constraint:

\[
\forall i \, r_1 \lor r_2 \lor r_3 \lor \cdots \lor r_m
\]

where \( m \) is the number of timing constraints that involve action \( A \) and \( r_i \) is defined as follows. If the \( k \)th timing constraint is sporadic (defined in Section IV-C), then \( r_k \) is

\[
\exists j \, @((E, j) \leq @((\uparrow A, i) \land @((\downarrow A, i) \leq @((E, j) + d)
\]

If the \( k \)th timing constraint is periodic (syntax described in Section IV-D), then \( r_k \) is

\[
\exists x \exists y \forall n \, n \cdot p \leq @((\uparrow A, i) \land @((\downarrow A, i)
\]

Finally, if \( A \) is involved in only one sporadic timing constraint and \( A \) is not involved in any periodic timing constraint, then the accountability assertion is reduced to the following:

\[
\forall i \, @((E, i) \leq @((\uparrow A, i) \land @((\downarrow A, i) \leq @((E, i) + d)
\]

Observe that if \( A \) is a composite action, similar accountability assertions are not necessary for the subactions within \( A \) because these constraints are already captured by the assertions described in Sections IV-A and B.

Although the accountability assertions are artificial constraints, they seem to be reasonable. These assertions are in fact needed except when utilization of resources is 100 percent which prevents the scheduler from scheduling actions not required by the timing constraints.

A related notion to the accountability assertions for timing constraints concerns the causal relationship between a transition event and an external/stop event (see Section IV-E). Again we cannot directly deduce from the event-action model that an external/stop event must have happened before the occurrence of a particular transition event. For each state attribute \( S \), the following assertion must be added to capture this notion:

\[
\forall i \, r_1 \lor r_2 \lor r_3 \lor \cdots \lor r_m
\]

where \( m \) is the number of events changing the value of \( S \) to true, and \( r_i \) is defined as:

\[
\exists j \, @((E := T), i) = @((E, j)
\]

where \( E \) is the \( k \)th event causing the change in value of \( S \). Similar assertions must be added for the events changing the value of \( S \) to false.

G. An Example

It is appropriate at this point to present an example illustrating how a system specification described in the event-action model can be transformed into a set of RTL formulas. For this purpose, recall the event-action specification of our Martian Lander presented in Example 1. Appendix B contains the corresponding set of formulas expressed in RTL. The mechanization of the translation procedure should be obvious from the discussion in this section.

V. A Deductive Reasoning System for Safety Analysis

The major components of a safety analysis can be divided into three parts: systems specification, safety assertions, and a decision procedure. A systems specification is a behavioral description of the real-time system (temporal properties expressible in RTL). A safety assertion describes a property (a RTL formula to be analyzed) that must be guaranteed to avoid a system failure. Finally, a decision procedure describes how to manipulate the known facts about the system to determine if the safety assertion is consistent with the specification.

In the following, we shall first review how systems specifications and safety assertions are expressed in RTL.

\[
x + n \cdot p + d \land S(x, y) \land n \cdot p < y - x
\]

Finally, we shall discuss the design of an analysis procedure.

A. Representing Systems Specification

A specification is a set of constraints imposed on the behavior of the system in question. We have shown in Section IV how to mechanically translate the specification of a real-time system in the event-action model into a set of assertions (axioms) in RTL. Consequently, a systems specification can be viewed as a conjunction of these assertions.

B. Representing Safety Assertions

A safety assertion describes a property that must be ensured to avoid a system failure. It is crucial to point out that a safety assertion can be viewed as just another constraint added to the specification of the system. Thus we expect that the same model employed to define the specification can also be used to express the safety assertions. In other words, safety assertions ought to be able to be expressed in terms of RTL formulas. The following two examples illustrate how safety assertions can be represented in the RTL notation. Both examples refer to the Martian Lander described earlier.

Example 4:

Safety Assertion: If the I/O operation to read acceleration is always performed within 100 ms, then the emergency landing mode will never be invoked.

Representation in RTL:

\[
\forall i \, @((DONE := T), i + 1) \land @((DONE := F), i) \land \cdot < 100 \rightarrow \{ \exists x \exists y \, \text{ELSM}(x, y) \}
\]

Example 5:

Safety Assertion: The retro-rockets should not be fired during the first 80 time units. This safety assertion is required by the system designer, because it takes 80 time units, for example, to initialize some of the critical system parameters relating to the retro-rockets.

Representation in RTL:

\[
\neg \exists i \, (1RRM, i) \leq 80
\]
C. An Analysis Procedure

We have thus far shown how to express systems specifications and safety assertions in RTL. We now turn to the issue of reasoning about the timing properties of real-time systems. Even though there are more than one type of constants, a RTL formula currently consists of only algebraic relations and state predicates connected by first-order logic operators. The state predicates in turn can be expanded into simple algebraic relations (see Section III-G). Hence the formulas that we have to analyze contain only integer terms. This raises the possibility of using a procedure similar to those used for deciding Presburger arithmetic. (Informally, Presburger formulas are those that can be constructed from integer constants, integer variables, the addition function, the predicates (\(<\), \(\leq\), \(\geq\), \(=\)), and the first-order logical connectives.) However, RTL formulas may contain constants other than integers (action constants and event constants), and these are not allowed in Presburger arithmetic. We shall shortly show how RTL formulas can be mechanically transformed into the equivalent formulas in Presburger arithmetic with uninterpreted functions.

Much research has been devoted to finding decision procedures for Presburger arithmetic and determining the complexity of such algorithms. Unfortunately, it has been shown that a decision procedure for the full Presburger arithmetic is inherently computationally expensive and these algorithms are therefore not practical for nontrivial problems (see [14, Section 2] for a brief historical perspective). Other work (e.g., [2], [13]) have concentrated on the subclass of quantifier-free Presburger formulas. Shostak [14] further extended the work on quantifier-free Presburger logic to include uninterpreted functions. This allows us to have unrestricted universal and existential quantifiers in the formulas. The idea is to replace all the existentially quantified variables by the corresponding Skolem constants or functions. In [3], Bledsoe and Hines describe a resolution-based procedure for proving theorems about general linear inequalities. These last two procedures provide the necessary tools for analysis of RTL formulas. But we must first transform RTL formulas into Presburger formulas with uninterpreted functions.

Eliminating State Predicates and Noninteger Constants from RTL Formulas: Given a system specification in terms of RTL formulas, the following transformation procedure describes how state predicates and noninteger constants can be eliminated.

1) In each formula, replace each state predicate with its definition as described in Section III-G. This step in effect substitutes, for each state predicate, a subformula containing only algebraic relations.

2) Recall that event constants only appear within occurrence functions of the form \(@e, i\) where \(e\) is an event constant and \(i\) is an integer variable/constant. For each event constant \(e\), define an uninterpreted function \(f_e\) mapping from \(W\) to \(W\) where \(W\) is the set of nonnegative integers. Then replace each appearance of \(@e, i\) in RTL assertions by the corresponding uninterpreted integer function \(f_e(i)\).

Overview of Bledsoe and Hines’ Approach: As suggested earlier, applying the above transformation to a specification of a system expressed in RTL allows us to use the procedure described by Bledsoe and Hines, show that a safety assertion \(SA\) is consistent with a system specification \(SP\). It must be proved that the formula \(SP\rightarrow\neg SA\) is valid. This is equivalent to showing that \(\neg F\) is unsatisfiable.

Bledsoe and Hines based their approach on a modif resolution procedure to show that \(\neg F\) is unsatisfiable special clause called \(TY\) is defined which is essential conjunction of ground inequality literals. Due to a splitting procedure, ground literals are guaranteed to occur only in unit clauses. The \(TY\) clause can be checked for contradiction by using a ground inequality prover, such the one described in [14].

During each resolution cycle, a chaining procedure applied to generate a new resolvent \(R\). If \(R\) is false, proof is successfully terminated. If \(R\) is a ground inequality clause, it is added to \(TY\), and then checked for contradiction. Otherwise, after removing eligible variables, \(R\) is simplified, split into independent clauses possible, and added to the set of clauses for subsequent resolution cycles. This procedure is continued until a contradiction is detected, thus proving the unsatisfiability of \(\neg F\).

Appendix C illustrates how the safety assertion of ample 4 can be shown to be consistent with the specification of our Martian Lander. (The approach Bledsoe and Hines is used.)

D. Efficiency Considerations

In complex real-time systems, verifying an assertion against the full specification of the system may be inefficient. For some systems, it may be the case that a subset of the specification can have a bearing on validity of a particular safety assertion. In this case, effort can be saved if we can “filter out” the irrelevance assertions from the systems specification before we invoke the primary analysis procedure. For example, safety assertion can be expressed in the form of an predicate about a state attribute \(S\), then we may be able to conclude its validity just from the set of assertions we can directly or indirectly cause a transition of \(S\) to \(\infty\). If we can determine such a set of supporting assertions an efficient procedure, e.g., by computing a trans closure using the accountability and causal assertions, then the primary procedure may only have to contend a much smaller set of axioms.

Another possible source of efficiency gain is the containing characteristics of the real-time system itself.

3The chaining procedure for producing new resolvents is essentially "limited application" of the transitivity axiom. Specifically, \((a < b)\) is inferred from \(a < b\) and \(b < c\). If \(b\) or \(b\) is an uninterpreted function containing at least one variable, and \(a\) and \(b\) are unifiable with the general unifier \(\theta\).
operation of many process control systems is often structured according to different modes. Intuitively, a mode corresponds to some assertion(s) about the values of a set of state attributes, e.g., an airplane is in emergency landing mode if the hydraulic pressure in the landing gear subsystem is below a critical value or if the fuel gauge indicates an empty tank. The key here is that the assertions in a system specification are often qualified by a collection of modes, i.e., they are required to be true only under those modes. Under other modes, these assertions need not be true. If there is an easy way for us to deduce that a given safety assertion is true in certain modes and these modes include all the ones that qualify a certain assertion \( p \) in the system specification, then we ought to be able to prove the validity of the safety assertion without making use of \( p \) at all.

It is interesting to point out that techniques to improve the efficiency of the inference mechanism such as the ones mentioned above often enable us to prove the validity of a safety assertion independent of whether a feasible schedule (one that meets all the performance requirements of the system) exists.

Although existing procedures for Presburger Arithmetic with uninterpreted functions is suitable for our purposes, greater efficiency may be achieved by inventing a procedure which is tailored specifically for our domain. For example, consider the observation that the RTL assertions described in this paper do not contain algebraic relations involving functions that take an instance of the same function as an argument. An analysis procedure that takes this fact into consideration may indeed provide a more efficient prover for checking ground inequality unit clauses.

For large systems, general techniques for improving the efficiency of the inference mechanism alone are probably inadequate to render the verification of a complete design practical. In such cases, good software engineering methods must be developed to ease the job of design verification by restricting design complexity. A good design method should encourage designers to structure their systems so that each safety property can be established from a readily determinable subset of the systems specification and modules.

VI. CONCLUDING REMARKS

The focus of this paper has been to formalize the safety analysis of timing properties in real-time systems. We presented a formal logic RTL (Real-Time Logic) which is especially amenable to reasoning about the timing behavior of systems. We described a mechanical procedure for translating instances of the event-action model, specified by a system designer into RTL formulas. After these RTL formulas are transformed into predicates of Presburger Arithmetic with uninterpreted integer functions, existing procedures can be used to determine if a given safety assertion is a theorem derivable from the systems specification.

One nice property of Real-Time Logic is that it provides a uniform way to express different types of constraints on a system, by means of algebraic relations on event occurrences. In place of finding invariants, the validity of a property in RTL is proved by showing the nonexistence of a function (the occurrence function) which can satisfy the negation of the property under investigation in conjunction with the assertions in the systems specification. If the validity of a desired property cannot be established from the systems specification alone, there is then the question of what practical constraints can be imposed on the system to establish the desired result. (The interaction between safety assertions and efficiency-related constraints imposed on the scheduler is a major issue in the synthesis component of the SARTOR [9] design system.) If it is impossible to guarantee the validity of some assertions in the systems specification owing to an imperfect execution environment, e.g., an interface violation may occur that cannot be controlled by the system, what good mechanisms can we use to detect violations before it is too late? For a class of unexpected events, can we determine a minimal core of essential assertions whose truth we must maintain so that we can recover from those unexpected events in acceptable time? What is a good design method (a theory of decomposition?) that will facilitate the verification of large real-time systems? These and other questions must be formalized and settled before we can have a sound theory for designing time-critical systems.

Appendix A

Summary of Real-Time Logic

Constants:
- Integer Constants
- Set of Action Constants
  - Action
  - Subaction
    - A
    - B.A
  - Set of Event Constants
  - Transition Event
  - External Event

Variables:
- range over integer, action or event constants denoted by names in lower case letters

Functions:
- Addition and Subtraction
- Multiplication by Constants
- Uninterpreted Functions (Range \( \subseteq \) Integers)
- Occurrence Function \( \@ (E, i) \)

Predicates:
- Equality/Inequality Predicates (\( =, <, \leq, >, \geq \))
State Predicates denoting the truth of a state attribute during an interval

Formulas:
RTL formulas are constructed using the above predicates, universal and existential quantifiers, and first order logic connectives.

Appendix B
Example of Transforming Event-Action into RTL Specification

Relationship between start/stop events:
\[ \forall i \in \{\text{RACC}, \text{STMR}\}, i + 10 \leq \langle i \rangle \text{RACC}, i \]
\[ \forall i \in \{\text{STMR}\}, i + 10 \leq \langle i \rangle \text{STMR}, i \]
\[ \forall i \in \{\text{ADJM}, \text{IEM}, \text{ETC}, \text{VEL}, \text{ALT}, \text{CKDT}, \text{RRM}\}, i + 10 \leq \langle i \rangle \text{ADJM}, i \]
Constraints imposed by NOP1 composite action:
\[ \forall i \in \{\text{NOP1}\}, i \leq \langle i \rangle \text{NOP1}.\text{RACC}, i \]
\[ \forall i \in \{\text{NOP1}\}.RACC, i \leq \langle i \rangle \text{NOP1}.\text{STMR}, i \]
\[ \forall i \in \{\text{NOP1}\}, i \leq \langle i \rangle \text{NOP1}.\text{STMR}, i \]
\[ \forall i \in \{\text{STMR}\}, i + 100 \leq \langle i \rangle \text{STMR}, i \]
Constraints imposed by NOP2 composite action:
\[ \forall i \in \{\text{NOP2}\}, i \leq \langle i \rangle \text{NOP2}.\text{ADJM}, i \]
\[ \forall i \in \{\text{NOP2}\}, i \leq \langle i \rangle \text{NOP2}.\text{TDP}, i \]
\[ \forall i \in \{\text{NOP2}\}, i \leq \langle i \rangle \text{NOP2}.\text{ADJM}, i \] \(\lor\)
\[ \forall i \in \{\text{NOP2}\}, i \leq \langle i \rangle \text{NOP2}.\text{TDP}, i \]
\[ \forall i \in \{\text{NOP2}\}, i \leq \langle i \rangle \text{NOP2}.\text{ADJM}, i \]
\[ \forall i \in \{\text{NOP2}\}, i \leq \langle i \rangle \text{NOP2}.\text{TDP}, i \]
Constraints imposed by TIH composite action:
\[ \forall i \in \{\text{TIH}\}, i \leq \langle i \rangle \text{TIH}.\text{IEM}, i \]
\[ \forall i \in \{\text{TIH}.\text{IEM}\}, i \leq \langle i \rangle \text{TIH}.\text{ETC}, i \]
\[ \forall i \in \{\text{TIH}\}, i \leq \langle i \rangle \text{TIH}.\text{ETC}, i \]
Constraints imposed by EOP composite action:
\[ \forall i \in \{\text{EOP}\}, i \leq \langle i \rangle \text{EOP}.\text{VEL}, i \]
\[ \forall i \in \{\text{EOP}\}, i \leq \langle i \rangle \text{EOP}.\text{ALT}, i \]
\[ \forall i \in \{\text{EOP}\}, i \leq \langle i \rangle \text{EOP}.\text{VEL}, i \] \(\lor\)
\[ \forall i \in \{\text{EOP}\}, i \leq \langle i \rangle \text{EOP}.\text{ALT}, i \]
\[ \forall i \in \{\text{EOP}\}, i \leq \langle i \rangle \text{EOP}.\text{VEL}, i \]
\[ \forall i \in \{\text{EOP}\}.\text{VEL}, i \leq \langle i \rangle \text{EOP}.\text{CKDT}, i \]
\[ \forall i \in \{\text{EOP}\}.\text{ALT}, i \leq \langle i \rangle \text{EOP}.\text{CKDT}, i \]
\[ \forall i \in \{\text{EOP}\}.\text{CKDT}, i \leq \langle i \rangle \text{EOP}.\text{RRM}, i \]
\[ \forall i \in \{\text{EOP}\}, i \leq \langle i \rangle \text{EOP}.\text{RRM}, i \]
Causal assertions for state attributes ELSM and DONE:
\[ \forall i \in \{\text{RACC}\}, i \leq \langle i \rangle \text{DONE} = \text{F}, i \]
\[ \forall i \in \{\text{TIH}\}, i \leq \langle i \rangle \text{DONE} = \text{T}, i \]
\[ \forall i \in \{\text{EOP}\}.\text{IEM}, i = \text{T} \land \text{DONE} < t, i \land \text{ELSMM} < t, i \] \(\rightarrow\)
\[ \langle i \rangle \text{ELSMM} = \text{T}, i \] \(\land\)

After variable elimination and simplification, the last assertion can be rewritten as:
\[ \forall i \in \{\text{EOP}\}.\text{IEM}, i = \text{T} \land \text{DONE} < t, i \land \text{ELSMM} < t, i \] \(\rightarrow\)
\[ \langle i \rangle \text{ELSMM} = \text{T}, i \] \(\land\)

Timing constraint 1 (periodic):
\[ \forall x, y \in \{\text{ELSMM}\}, (x, y) \land 200 \cdot n \leq y - x \] \(\rightarrow\)
\[ x + 200 \cdot n \leq \langle i \rangle \text{NOP1}, i \land \langle i \rangle \text{NOP1}, i \leq x + 200 \cdot n + 40 \]

Timing constraint 2 (sporadic):
\[ \forall i \in \{\text{TIH}\}, i \leq \langle i \rangle \text{NOP2}, i \land \langle i \rangle \text{NOP2}, i \leq 60 \]
\[ \forall i \in \{\text{TIH}\}, i \leq \langle i \rangle \text{TIMRINT}, i + 1 \]

Timing constraint 3 (sporadic):
\[ \forall i \in \{\text{TIMRINT}\}, i \leq \langle i \rangle \text{TIH}, i \land \langle i \rangle \text{TIMRINT}, i + 100 \]
\[ \forall i \in \{\text{TIMRINT}\}, i \leq \langle i \rangle \text{TIMRINT}, i + 1 \]

Timing constraint 4 (periodic):
\[ \forall x, y \in \{\text{ELSMM}\}, (x, y) \land 200 \cdot n \leq y - x \] \(\rightarrow\)
\[ x + 200 \cdot n \leq \langle i \rangle \text{EOP}, i \land \langle i \rangle \text{EOP}, i \leq x + 200 \cdot n + 100 \]

Accountability assertions for timing constraints:

Accountability assertions for transition events:
\[ \forall i \in \{\text{ELSMM}\}, i \leq \langle i \rangle \text{IEM}, i \land \langle i \rangle \text{DONE} = \text{F}, i \leq \langle i \rangle \text{ELSMM} = \text{T}, i \] \(\land:\)
\[ \langle i \rangle \text{ELSMM} = \text{T}, i \leq \langle i \rangle \text{DONE} = \text{T}, k + \]

Occurrence function axioms:
For each event constant E in the systems specified the following two assertions are added:
\[ \forall i \leq \langle i \rangle \text{E}, i \]
\[ \forall i < j \rightarrow \langle i \rangle \text{E}, i < \langle i \rangle \text{E}, j \]

Transition event axioms:
\[ \langle i \rangle \text{ELSMM} = \text{F}, i \leq \langle i \rangle \text{DONE} = \text{T}, i \leq \langle i \rangle \text{ELSMM} = \text{T}, i \]
\[ \langle i \rangle \text{ELSMM} = \text{T}, i \leq \langle i \rangle \text{DONE} = \text{F}, i \leq \langle i \rangle \text{ELSMM} = \text{T}, i \]
\[ \langle i \rangle \text{ELSMM} = \text{T}, i \leq \langle i \rangle \text{DONE} = \text{F}, i \leq \langle i \rangle \text{ELSMM} = \text{T}, i \]
\[ \langle i \rangle \text{ELSMM} = \text{T}, i \leq \langle i \rangle \text{DONE} = \text{F}, i \leq \langle i \rangle \text{ELSMM} = \text{T}, i \]

39
Start/Stop event axioms:

\[ \forall i @((\text{STOP}, i)) = t \rightarrow \exists j @((\text{STOP}, j)) = t \]
\[ \land @((\text{STOP}, i)) = ((\text{STOP}, j)) \]

\[ \forall i @((\text{RTC}, i)) = t \rightarrow \exists j @((\text{RTC}, j)) = t \]
\[ \land @((\text{RTC}, i)) = ((\text{RTC}, j)) \]

\[ \forall i @((\text{ADJ}, i)) = t \rightarrow \exists j @((\text{ADJ}, j)) = t \]
\[ \land @((\text{ADJ}, i)) = ((\text{ADJ}, j)) \]

\[ \forall i @((\text{TCP}, i)) = t \rightarrow \exists j @((\text{TCP}, j)) = t \]
\[ \land @((\text{TCP}, i)) = ((\text{TCP}, j)) \]

Also for each action constant A.

\[ \forall i @((\text{A}, i)) = t \rightarrow \exists t' t' < t \land @((\text{A}, i)) = t' \]

APPENDIX C
EXAMPLE OF APPLYING BLEDSOE AND HINES’ PROCEDURE

Safety Assertion of Example 4:

\[ \forall i @((\text{DONE} = T), i + 1) - @((\text{DONE} = F), i) \]
\[ < 100 \rightarrow \neg \{\exists x y \text{ ELMS}(x, y)\} \]

Negation of the Safety Assertion:

\[ \forall i @((\text{DONE} = T), i + 1) - @((\text{DONE} = F), i) \]
\[ < 100 \land \exists x y z l @((\text{ELMS} = T), l) = \{x, y, z, l\} \]
\[ \land [\forall i @((\text{ELMS} = F), l + 1) = t \rightarrow y < t] \]

Negation of the SA after being skolemized:

1. @((\text{DONE} = T), i + 1) - @((\text{DONE} = F), i) < 100
2. @((\text{ELMS} = T), L) = X
3. @((\text{ELMS} = F), L + 1) = t \rightarrow Y \leq t

We will refer to these clauses as SA1, SA2, and SA3.

With a slight abuse of notation, we shall write @e, i for the uninterpreted integer function \( f_i \) whenever there is no confusion. (Recall that \( f_i \) is the function after eliminating the event constant from the occurrence function \( @e, i \).) The following steps illustrate how the above safety assertion is shown to be a theorem derivable from the systems specification. Although we may not show it in the explanation of each step, all assertions are supposed to be in clausal form before applying the algorithm.

Step 1:

\[ @((\text{ELMS} = T), L) = @((\text{IEM}, J(L))) \]  
(1.1)

\[ @((\text{DONE} = F), K(L)) < @((\text{ELMS} = T), L) \]
(1.2)

\[ @((\text{ELMS} = T), L) \leq @((\text{DONE} = T), K(L) + 1) \]
(1.3)

The above three clauses were generated by chaining SA2 separately with the following clauses from the specification:

\[ \forall i j k @((\text{ELMS} = T), i) = @((\text{IEM}, j)) \land @((\text{DONE} = F), k) < @((\text{ELMS} = T), i) \]
\[ \land @((\text{ELMS} = T), i) \leq @((\text{DONE} = T), k + 1) \]

Observe that \( J(L) \) and \( K(L) \) are skolem functions corresponding to variables \( j \) and \( k \).

Step 2:

\[ @((\text{IEM}, i) = @((\text{ELMS} = T), L) \land @((\text{ELMS} = T), L) < @((\text{IEM}, i) \lor @((\text{IEM}, i) \land @((\text{DONE} = F), L) \lor @((\text{DONE} = T), j + 1) < @((\text{IEM}, i) \lor @((\text{DONE} = F), j) \lor \]

The clause in Step 2 was generated by chaining SA2 with causal assertion for state attribute ELMS (see Appendix B). Observe that the above clause is a disjunction of five literals. Let us start with the first literal, @e, i = @((\text{ELMS} = T), L)

Step 3:

@((\text{IEM}, i) \leq @((\text{DONE} = T), K(L) + 1)

Generated by chaining the following two clauses:

@((\text{IEM}, i) = @((\text{ELMS} = T), L) (last literal in Step 2)
\[ @((\text{ELMS} = T), L) \leq @((\text{DONE} = T), K(L) + 1) \]  
(clause 1.3 in Step 1)

Step 4:

@((\text{IEM}, i) \leq @((\text{DONE} = F), K(L)) + 100

Generated by chaining the following two clauses:

@((\text{IEM}, i) \leq @((\text{DONE} = T), K(L) + 1)

(generated in Step 3)
\[ @((\text{DONE} = T), i + 1) < @((\text{DONE} = F), i) + 100 \]  
(clause SA1)

Step 5:

@((\text{IEM}, i) < @((\text{DONE} = F), K(L)) + 90

Generated by chaining the following two clauses:

@((\text{IEM}, i) < @((\text{DONE} = F), K(L) + 100

(generated in Step 4)

\[ \forall i @((\text{IEM}, i) + 10 \leq @((\text{IEM}, i) \text{ (from the spec.)}) \]
Step 6:
\[ @(\text{TTIH}, i) < @(\text{DONE} := F), K(L)) + 90 \]
Generated by chaining the following two clauses:
\[ @(\text{IEEM}, i) < @(\text{DONE} := F), K(L)) + 90 \]
\[ \text{Vi } @(\text{TTIH}, i) = @(\text{TTIH}.\text{IEEM}, i) \text{ (from the spec.)} \]
Step 7:
\[ @(\text{OTMRINT}, i) < @(\text{DONE} := F), K(L)) + 90 \]
Generated by chaining the following two clauses:
\[ @(\text{TTIH}, i) < @(\text{DONE} := F), K(L)) + 90 \]
\[ \text{Vi } @(\text{OTMRINT}, i) \leq @(\text{TTIH}, i) \text{ (from the spec.)} \]
Step 8:
\[ @(\text{STMR}, i) + 10 < @(\text{DONE} := F), K(L)) \]
Generated by chaining the following two clauses:
\[ @(\text{OTMRINT}, i) < @(\text{DONE} := F), K(L)) + 90 \]
\[ \text{Vi } @(\text{OTMRINT}, i) + 100 \leq @(\text{OTMRINT}, i) \text{ (from the spec.)} \]
Step 9:
\[ @(\text{NOP1.RACC}, i) + 10 < @(\text{DONE} := F), K(L)) \]
Generated by chaining the following two clauses:
\[ @(\text{STMR}, i) + 10 < @(\text{DONE} := F), K(L)) \]
\[ \text{Vi } @(\text{NOP1.RACC}, i) \leq @(\text{STMR}, i) \text{ (from the spec.)} \]
Step 10:
\[ @(\text{DONE} := F), i) + 10 < @(\text{DONE} := F), K(L)) \]
Generated by chaining the following two clauses:
\[ @(\text{NOP1.RACC}, i) + 10 < @(\text{DONE} := F), K(L)) \]
\[ \text{Vi } @(\text{RACC}, i) = @(\text{DONE} := F), i) \text{ (from the spec.)} \]

The clause generated in Step 10 is a contradiction. One way of showing its unsatisfiability is to chain the clause with itself. The remaining literals in the disjunction generated in Step 2 can be shown to be unsatisfiable using the same procedure. Observe that the free variable \(i\) and \(j\) have been instantiated with skolem function \(K(L)\).

Acknowledgment

We would like to thank Prof. A. Pnueli and Prof. J. Misra for helpful discussions.

References


Mr. Jahanian is a student member of the IEEE Computer Society.

Aloysius Ka-Lau Mok (M'84) received the S.B. degree in electrical engineering and the S.M. degree in computer science and electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1977. He received the Ph.D. degree in computer science, also from M.I.T., in May 1983, after completing a dissertation on the fundamental design problems of distributed systems in the hard-real-time environment. While a student, he worked at the disk engineering department of Digital Equipment Corporation, Maynard, MA, and was on the Research Staff of the Division of Sponsored Research at M.I.T. His current interests include the design of robust distributed real-time systems and software automation. He has been an Assistant Professor of Computer Science at the University of Texas at Austin since June 1983. Dr. Mok is a member of Eta Kappa Nu and Tau Beta Pi.