Completeness, Robustness, and Safety in Real-Time Software Requirements Specification

Matthew S. Jaffe
Nancy G. Leveson

Technical Report 89-01

Completeness, Robustness, and Safety in Real-Time Software Requirements Specification *

Matthew S. Jaffe
Hughes Aircraft Company
Ground Systems Group
Fullerton, California

Nancy G. Leveson
Info. & Computer Science Dept.
University of California, Irvine
Irvine, California

February 15, 1989

Abstract

This paper presents an approach to providing a rigorous basis for ascertaining whether or not a given set of software requirements is internally complete, i.e., closed with respect to questions and inferences that can be made on the basis of information included in the specification. Emphasis is placed on aspects of software requirements specifications that previously have not been adequately handled, including timing abstractions, safety, and robustness.

1 Introduction

Completeness of requirements specifications is of major significance in modern software engineering. Software requirements errors have been found to account for a majority of production software failures [BMU75, End75] and have been implicated in a large number of accidents [Lev86]. Many (if not most) of the failures associated with requirements turn out to involve incompleteness.

In defining the nature of the software to be designed and produced, the requirements must be sufficient to distinguish the behavior of the desired software from that of any other, undesired program that might be designed. If a requirements document contains insufficient information to allow designers to distinguish between observably distinct behavioral patterns, the requirements document is incomplete. If the differences between two

*This work has been partially supported by NSF Grant CCR-8718001, NSF VPW Grant RII-8800505, NSF CER Grant DCR-8521398, and NASA Grant NAG-1-668.

programs that satisfy the same set of requirements are not significant for a given application, the incompleteness may not matter, but languages or specification procedures that do not permit the expression of subtle distinctions or do not include requirements to cover all possible circumstances will be inadequate for some applications. Incompleteness in the software requirements specification can have a major impact on testing, formal verification, reuse, robustness, and safety of software.

One application for which completeness is particularly critical is process control, i.e., systems that control arbitrarily large or energetic physical phenomena. Such systems are usually real-time and often embedded within some larger system such as a ship, aircraft, missile, spacecraft, manufacturing or processing plant, or transportation system where computers are used to assist in the formulation and implementation of decisions made either by the computer and/or humans for the purpose of controlling the larger system. In such process control systems, minor behavioral distinctions often have significant real world consequences. The completeness issues discussed in this paper are most relevant for these types of systems. Current requirements specification techniques are not sufficiently powerful to differentiate among observably distinct behaviors that are potentially important in many such process-control systems, nor do they force elucidation of, and specification of responses to, "unexpected events."

Despite its importance, there is no consensus as to precisely what constitutes completeness in requirements specifications nor how to go about achieving it. Many discussions essentially assert that "a requirements specification is complete if some relevant aspect has not been left out ..."[Rom85]. Ramamoorthy and So [RS78] state that the most difficult part of the requirements statement is the definition of all relevant performance parameters, but then they continue to conclude that "there is essentially no notion of completeness as far as performance requirements are concerned." Parnas and Clements [PC86] provide perhaps the best practical definition of completeness to date, but their discussion, focusing on the beginning of the requirements engineering cycle and on output characteristics, necessarily omits much of the genuine black-box behavior that is induced by consideration of black-box input phenomena. Loading factor, for instance, identified by Ramamoorthy and So (*op. cit.*) as "one of the most important performance requirements," is not discussed. More rigorous criteria need to be developed to help ascertain whether or not a given set of requirements is, in fact, complete.

When attempting to define completeness, it is important to distinguish between completeness and sufficiency. A set of requirements may be sufficient without being complete. Absolute completeness in requirements specification may be unnecessary and uneconomical for some applications. It is necessary, however, to determine the limits of specification before deciding what can be left out. As will be seen, a complete specification of a real-time system may be quite large and require a great deal of time and effort to produce. For many applications, this effort is not justified by the consequences of having some incompleteness in the specification. In others, the consequences of any incompleteness may justify the

extra effort.

The goal of the work described in this paper is to provide a theoretical foundation for the concept of completeness in real-time software requirements specification and to provide a rigorous basis for ascertaining whether or not a given set of software requirements is internally complete, where internally complete is defined as minimally closed with respect to a key set of questions and inferences that are based on information already specified. Although current real-time requirements specification techniques collectively contain many of the features described in this paper, none contain them all and some features are not contained in any of them. The goal here is not to propose yet another language for specification of requirements, but instead to describe formally what has been scattered throughout the literature, to delineate what features real-time specification languages should have, and to suggest completeness analysis techniques that are applicable to specifications written in such languages.

2 SCOPE

There are limits to what currently can be dealt with analytically. For one thing, completeness is relative to point of view. What is required by one observer may be a matter of indifference to another. For example, a development engineering team may want the specification to include requirements for instrumentation (such as data recording and checkpointing). To the engineering team, a specification without such requirements would be incomplete, while to the customer or end user, the omission of such requirements is not significant and would not necessarily be considered an incompleteness.

Completeness is also relative to the life cycle phase, even beyond the conventional notion of refinement in "level" of detail. For example, a requirements specification must often be prepared before a host processor can be selected, and yet after the processor is selected, there may be more requirements on the software (such as responses to the full set of processor hardware signals observable to the software) induced or derived from the characteristics of the selected hardware. If, for example, the chosen processor includes a power-out-of-tolerance interrupt that guarantees the existence of at least 30 milliseconds further processing time before power down, *some* response to that interrupt should be specified. Such a response is still a black-box, behavioral requirement: the trigger, i.e., power out of tolerance, and the response are both visible from outside the box. After processing hardware characteristics are known, software requirements must be specified for the full set of hardware signals observable to the software. Before a processor is selected, however, the lack of such requirements could not be said to constitute an incompleteness in the software requirements specification. Thus completeness criteria can be seen to vary as a function of the stage of the development cycle. Requirements engineering is a complex activity with different stages, different possible paths among those stages, and different

completeness criteria needs at each distinct stage.

With these considerations in mind, this paper considers what might be called "internal" completeness. In any application at any given point in time, there is a set of "kernel" requirements that derive from the current knowledge of the needs and environment of the application itself. They are analytically independent of one another in that the need for the existence of any one of them could not be derived from the existence of the others. For example, an autopilot program may or may not control the throttle along with the aerodynamic surfaces. Without knowledge of the intent of the application, there can be no means of ascertaining whether a particular requirements specification is complete with respect to the kernel requirements. This problem of external completeness must be attacked from a system engineering viewpoint, e.g., using modeling and analysis of the application [LS87, LH83] with respect to various desired properties. This paper deals strictly with internal completeness.

Internal completeness has two aspects. First, there are a great many additional details of behavioral description that must be specified in addition to an initial or kernel set of intended and/or required functionality. Many of these details cannot be obtained merely by the addition of standard "performance" requirements to the kernel functions. Kernel requirements are often viewed as being *allocated* to programs as part of an earlier, systems engineering design activity while the remaining information needed to completely characterize the software behavioral description must be derived from the kernel requirements in a later stage of the development cycle. The additional information is still black-box behavior; the place to document it is in the requirements document. Note that the requirements specification, then, may itself appear in different guises in successive stages of the development life cycle.

In addition to derived information, there may also be genuine functionality that has been inadvertently omitted and that can be uncovered by rigorous examination and analysis of the specified behavior. For example, a specification whose original, principle functionality includes a requirement to generate an alert condition to tell an air traffic controller that an aircraft is too low is probably incomplete unless it also includes another requirement to inform the controller that an aircraft previously noted to be too low is now back at a safe altitude. Safety and robustness considerations may be exploited to develop such application-independent closure criteria.

A practical complication involves the inclusion of design information in the requirements specification. Often requirements and design become intertwined. In order to limit the size of the problem, the focus in this paper will be solely on the software behavior, i.e., the "what" and not the "how." The product of the requirements phase may be a stand-alone document containing only a black-box behavioral description of the software, as recommended by Parnas and Clements [PC86], or it may be a document that intersperses the requirements with design information, as Swartout and Balzer [SB82] assert is often unavoidable. However, even a specification that includes design information needs to

include a complete set of black-box behavioral requirements as well, and these behavioral requirements are the subject of this paper.

Notational simplicity and sparsity are stressed in this paper: No new notation or models are introduced: The first-order predicate calculus and a simple state machine model are used. These are adequate to express and analyze the types of black box requirements needed for real-time software. Note again that the goal is not to provide another language for specification of requirements (and certainly the predicate calculus is too awkward to be a specification language of choice); the formal notation is for the purpose of providing maximum rigor while requiring a small number of primitives in the discussion of completeness of requirements in general. The next section introduces the basic notation and definitions.

3 Definitions and Notation

A black-box statement of behavior permits statements and observations to be made only in terms of outputs and the externally observable conditions or events that stimulate or trigger those outputs. Conceptually, the behavioral requirements may be viewed as a set of assertions of the form: *trigger* $\Leftrightarrow$ *output* where the trigger specifies the conditions under which an output or set of outputs is to be produced. Note that the implication is bi-directional ($\Leftrightarrow$): Not only must the output exist under certain specified circumstances, (i.e., *trigger* $\Rightarrow$ *output*), it also must *not* exist unless it is supposed to (i.e., *output* $\Rightarrow$ *trigger*). This second direction of the implication has significant impact on the derivation of completeness criteria: Because the existence of the output allows the existence of the trigger conditions to be inferred, a complete trigger specification must include the full set of conditions that may be inferred from the existence of the specified output. Such conditions represent assumptions about the environment in which the program or system is to execute.

For black-box requirements, the requirements must involve only observable phenomena external to the program whose behavior is being specified. Parnas and Clements [PC86] point out the advantages of allowing *any* external, observable phenomena to be used to express trigger conditions, even phenomena not "observable" at the black box boundary by the program itself. This is useful in the early life cycle phases, but at some point the external observables must get translated into input/output terms: More completeness criteria can be applied after this translation. The implication for this paper is that some of the requirements completeness criteria presented will be applicable to specifications in terms of general observable phenomena, but many will apply only after the specific inputs have been determined.

Completeness requires that both the characteristics of the outputs and the assumptions about their triggering events be specified. Formally, requirement assertions are statements

of the form

$$\exists E_1, \ldots, E_n \ni P_E \Leftrightarrow \exists! O \ni P_O,$$

where P_E is a predicate describing the set of stimuli or triggering events E_i for an output or set of outputs O, and P_O is a predicate describing the required characteristics of O. Note that the existential quantification for the output must be unique ($\exists!$): In response to a single occurrence of the given stimulus or trigger, the program must produce only a single output set O satisfying the predicate.

At the black-box boundary, only time and value are observable to the software. Therefore, the two predicates, P_E and P_O, must be defined only in terms of constants and the time and value — $t(E)$ and $v(E)$, respectively — of existentially quantified, observable events or conditions. Events are limited to program inputs, prior program outputs, program startup (a unique observable event for each execution of a given black-box program), and hardware-dependent events such as power-out-of-tolerance interrupts. In addition, conditions may also be expressed in terms of the value of hardware-dependent attributes accessible by the software such as time-of-day clocks, sense switches, etc.

For reasons to be discussed shortly, every existentially quantified event (other than a constant, such as 12:00 noon, for example) must be fully bounded in time — $t_l < t(E) < t_u$ — except for one: The latest or proximate trigger E_{pt} is bounded only from below and defines a lower bound (not necessarily the greatest lower bound) for the time of the output O. The proximate trigger, then, is the event (including the passage of a specific instant in time) that will actually cause the output O to be produced.

Other events in the trigger clause represent the necessary state history or conditions that must hold upon the observation of E_{pt} for O to be required. For example,

$$\exists! S_u, I_1, I_2 \neg \exists I_3 \ni t(S_u) < t(I_2) < t(I_3) < t(I_1) \Leftrightarrow \ldots,$$

where S_u represents program startup, establishes I_1 as the proximate trigger when there has been a previous I_2 and no intervening I_3.

In general, a given output O may be triggered by any one of several such trigger clauses; thus a trigger may be composed of a disjunction of several trigger clauses. A term or set of terms that appears in every disjunctive phrase of the trigger clause for a given output represents a *prerequisite* for the output O in the sense that the output cannot be triggered (i.e., required) unless the prerequisite conditions are true.

Using these definitions, section 4 presents criteria for the complete specification of trigger events. Section 5 discusses the problems involved in completely specifying outputs.

4 Trigger Event Completeness

Robustness and safety are intimately connected to completeness in the specification of trigger conditions. Trigger conditions specify assumptions about the conditions in the en-

vironment within which the software is to execute. A *robust* system will detect and respond appropriately to violations of these assumptions. Therefore, the robustness of the resulting software system depends upon the completeness of the specification of the environmental assumptions, i.e., there must be no observable events that leave the program's behavior indeterminate.

Documenting all environmental assumptions and checking them at run-time often seems expensive and unnecessary. Many assumptions, for example, are made on the basis of the physical characteristics of input devices and cannot be falsified even under unreasonable physical conditions and failures. For example, an input line connected to a 1200 baud modem cannot fail in such a fashion as to cause the data rate to exceed 1200 baud. The interrupt signal may stick high (i.e., on), but for most modern hardware that will stop data transfer, not accelerate it. However, if the environment in which the program executes ever changes, the assumption may no longer remain valid; e.g., the 1200 baud modem may be upgraded to 9600 baud. Similarly, if the software is ever reused, the environment for the new program may differ from that of the earlier one. A striking example of this type of problem involved the reuse of air traffic control software in Great Britain that was originally written and designed for air traffic control centers in the U.S. It was not discovered until after the software was installed that the American designers had not taken zero longitude into account which caused the computer to fold its map of Britain in two at the Greenwich meridian [Lam88].

Besides documentation of assumptions, it may be important for real-time systems to check assumptions at run-time when the improper performance of the software may cause serious consequences. Examples abound of serious accidents resulting from incomplete requirements and non-robust software [Lev86, Neu85]. For example, an accident occurred when a mechanical malfunction in a fly-by-wire flight control system set up an accelerated environment for which the flight control computer was not programmed; the aircraft went out of control and crashed [FM84]. In another incident, an aircraft was damaged when the computer raised the landing gear in response to a test pilot's command while the aircraft was standing on the runway [Neu85]. System safety engineers have concluded that inadequate requirements specification and design foresight are the greatest cause of software safety problems [Lev86].

The formal mechanism for ensuring complete specification of assumptions is logical completeness. The specification of the trigger conditions is logically complete if the logical 'OR' of all trigger conditions in the set of requirements is a tautology. For example, if there is a requirement on a trigger that the value of an input be greater than 5 (i.e., $\exists I \ni v(I) > 5$), then a tautologically complete specification would also include a requirement with a trigger condition where the input is less than or equal to 5, (i.e., $\exists I \ni v(I) \leq 5$). That is, if there is a trigger condition for a requirement assertion to handle data within a range, there needs to be some other requirement assertion to handle data that is out-of-range. There would also need to be a requirement that specified what to do if no input occurred at all.

Formally, if $\{C_i\}$ is the set of trigger clauses for the specification and $\bigvee_i C_i = \mathcal{T}$, the specification may be said to be logically complete to the point of $\mathcal{T}$. If $\mathcal{T}$ is a tautology, the specification is tautologically complete; in other words, any possible condition expressible in terms of the inputs and outputs that appear in the set of trigger conditions $\{C_i\}$ makes at least one of the C_i true and hence there is a requirement to deal with that condition. If there is a trigger of the form $\exists E \ni P_E$, logical completeness will require dealing with the case where $\neg(\exists E \ni P_E)$, which is equivalent to $\neg\exists E \bigvee \exists E \ni \neg P_E$.

Tautological completeness is obviously not in itself enough for a practical requirements specification, since a set of as few as two requirements could be tautologically complete. The degree to which tautological completeness forces the inclusion of additional requirements and thus influences completeness and robustness is dependent on the extent of the restrictions or assumptions (such as "legal range") specified within the trigger conditions. The more information in the trigger conditions (i.e., the more assumptions about the environment of the software that are specified), the more that tautological completeness will ensure that the requirements include responses to the set of "undesired events," i.e., circumstances where the assumptions are violated. The problem then reduces to determining what constitutes a complete specification of assumptions.

Many assumptions and conditions are, of course, application-dependent. There are, however, some types of assumptions that are essential and should always be specified for inputs to safety-critical real-time systems. In the context of real-time systems, the times of the inputs and outputs are as important as the values. Therefore, both value and time are required in the characterization of the environmental assumptions (triggers) and, as will be seen in the next section, the outputs.

Generally assumptions about value and time can be specified in separable phrases although inseparable assumptions are occasionally used. An example of an inseparable assumption is a requirement to check the currency of an input containing a time stamp placed on it by an external system: $v(I) = t(I) \pm c$. In contrast, a separable, individual timing assumption for I would be of the type

$$t(S_u) < 11:59\text{am} < t(I) < 12:01\text{pm},$$

if the input were only to be valid if it came at noon. Practical real-time specification languages must allow for the general case and permit the specification of assumptions in which time, value, and history are inseparable.

The rest of this section discusses various types of assumptions that should be included in trigger event specifications. Some of the assumptions relate to what can and should be specified for the proximate triggers. There are also criteria relating to the states in which an event occurs: The required response to a given event may depend not only on the characteristics of the event itself, but on the history of past events and the temporal relationship between them. The completeness criteria discussed below include essential value assumptions, essential timing assumptions, assumptions to deal with "unexpected states"

(which are needed for logical completeness), assumptions about startup and shutdown states (which are also needed for logical completeness), and responsiveness assumptions.

Essential Value Assumptions

The mere existence of an input does not in itself require a value assumption. Consider, for example, a hard-wired hardware interrupt that has no value; it may nevertheless trigger an output. For each input I, a value assumption is *essential* only if $v(I)$ (or some subset of the bits of $v(I)$) is used in defining the value or time of some output O. If $v(I)$ does not appear in any output predicate P_O, no assumptions concerning $v(I)$ need to be specified. In other words, the *existence* of I helps trigger O, but $v(I)$ is not referred to further in the definition of $v(O)$ or $t(O)$. When $v(I)$ *is* used in the definition of P_O, appropriate assumptions on the acceptable characteristics of $v(I)$ must be specified, e.g., range of acceptable values, set of acceptable values, parity of acceptable values, etc.

As noted earlier, even where an assumption is not essential, it should be specified whenever possible, i.e., whenever it is known: The receipt of an input with an "unexpected" value is a sign that something in the environment is not behaving as the designer anticipated. Checking simple value assumptions on inputs is comparatively inexpensive, and since failure of such assumptions is one indication of various, reasonably common hardware malfunctions or of misunderstanding about software requirements, it is difficult to envision an application where the specification should not require robustness in this regard, i.e., incoming values should have their values checked *and* there should be a specified response in the event of an out-of-range condition. If legal values are specified, tautological completeness will ensure that specifications contain the necessary information to provide this form of robustness.

Even when real-time response is not required, it is important that violations of assumptions be logged for off-line analysis. A hole in the ozone layer at the South Pole was not detected for six years because the depletion of the ozone was so severe that a computer analyzing the data had been suppressing it, having been programmed to assume that deviations so extreme must be errors [NYT86].

Essential Timing Assumptions

For trigger conditions, while the specification of the value of an event is usual but optional, a timing specification is *always* required: The mere existence of an observable event (with no timing specification) in and of itself cannot be a complete trigger — with the exception of program startup. One way to demonstrate this is to examine the formal definition. The specification that an output O is triggered by the existence of an input I — i.e., $\exists I \Leftrightarrow \exists! O$ — implies that it must also be the case that the non-existence of the output implies that the input does not exist — i.e., $\neg\exists O \Rightarrow \neg\exists I$. However, this is not necessarily

true. The simplest example of the implication not being valid is that before the program starts running (which is a primitive, observable event), the input I may exist but the program will obviously not produce the output O. This is not an unrealistic case: Serious accidents have occurred precisely because designers did not consider the problem of how to handle information about the state of the world that arrived while the system was in a manual mode and the computer was temporarily off-line. As an example, an accident occurred in a batch chemical reactor when a computer was taken off-line to modify the software [Kle88]. At the time the computer was shut-down, it was counting the revolutions on a metering pump that was feeding the reactor. When the computer came back on-line, the software continued counting where it had left off with the result that the reactor was overcharged.

As a result, all triggers must include at least one event, i.e., program startup S_u, and for all events other than startup, at least one timing assumption is essential, i.e., that the event occurs after startup. Many other timing assumptions may be essential, including bounds, capacity and load, maximum time between events, and graceful degradation, depending on the utilization of the event in the specification. Each of these, and the circumstances under which they are essential, is discussed below. It is interesting to note that many specification languages for embedded systems require the specification of a value condition, while at most merely allowing the specification of a timing condition, whereas they should actually do the opposite.

Specifying Bounds on Timing. A valid trigger specification must include either: (1) an observable signal (appearing as $\exists E$ terms in the notation for triggers); or (2) a specification involving the non-existence of events, i.e., a duration of time without a specific signal.

To be specified completely, a trigger of the first type must include at least a lower bound on time and will, in general, include further timing constraints. In fact, specifying a trigger event whose only lower bound on time is program startup gives rise to the need to specify extra requirements, often called capacity or load requirements; these extra requirements are discussed below. Table 1 summarizes the various possibilities for timing constraints on observable signals. In this table, x stands for any timing expression.

Note that even a requirement such as $t(I) = 11 : 00\text{am}$ is incomplete. The value of $t(I)$ is the value of some reference clock observed "simultaneously" with the occurrence of I. Conceptually, the clock is a counter that is ticking at the rate of one tick per unit of temporal precision. There is a problem with this definition of $t(I)$ in that I will not, in general, occur simultaneously with a tick. In fact, the simultaneity of observed events is not physically well-defined. In general, I will occur between two ticks of *any* clock, no matter how frequent the ticks. To say that it must occur *exactly* at 11:00am is meaningless unless the specification also specifies what clock is to be used, and, even then, the time cannot be known more precisely than the granularity of the clock. Concrete discussion

of specific clocks should be avoided in a software requirements specification; all that it is really necessary to know is the required precision of the clock. Translating the clock's precision into an attribute of the input results in a requirement with bounding inequalities rather than an equality, e.g. $10 : 59\text{am} < t(I) < 11 : 01\text{am}$ (commonly written as $t(I) = 11 : 00\ \text{am} \pm 1\text{min}$) which specifies an accuracy of plus or minus a minute on the timing.

For triggers of the second type, i.e., those that involve the *non*-existence of an observable signal during an interval, both ends of the interval must be either bound by or calculable from observable events. Informally, there must be an upper bound on the time the program "waits" before producing the output O. There must also be a specific time to start timing the lack of inputs or an infinite number of intervals (and thus outputs) would specified. For example, a requirement of the type "if there is no input I for 10 seconds, then produce output O" is not bound at the lower end of the interval and therefore is incomplete. Should the non-existence interval start at time t, at $t+\epsilon$, $t+2\epsilon$, etc.? The observable event need not occur at either end of the interval, the ends need only be calculable from that event, e.g., there is no input for 5 seconds preceding or following event E. An example of a completely bounded interval is the requirement that an output O be generated if ten seconds elapses without the receipt of a specified input message, i.e.,

$$\exists!\, S_u, I_1 \neg \exists I_2 \ni S_u < t(I_1) < t(I_2) < t(I_1)+10\ \text{sec} \Leftrightarrow \ldots$$

where $t(I_1)$ provides the lower, observable bound of the interval and the duration of 10 seconds effectively sets the upper bound. The complete rules for timing bounds on non-existence events are shown in Table 2. Again, in this table x stands for any timing expression.

Capacity, Load and Maximum Time Between Events. In an interrupt driven system, the count of unmasked input interrupts received over a given period of time partitions the system state space into at least two states: normal and overloaded. The required response to an input must differ in the two states; there must therefore be separate output assertions to deal with them. The term "capacity" seems to be used to refer to the count of inputs of a single type, while load — to be discussed shortly — is the count of a set of diverse input events.

The treatment of capacity depends upon whether interrupts are allowed to be disabled or not. Assuming for the moment that interrupts are not locked out on a given port, there is always some arrival rate for an interrupt signaling an input that will overload the physical machine. Either it will run out of CPU resources as it spends execution cycles responding to the interrupt or it will run out of memory as it stores the data for future processing. Thus, both the hardware selection and the software design require that an assumption be made about the maximum number of inputs N signaled by a given interrupt that must

Table 1: Timing Constraints for Observable Trigger Events

$\exists E$	Not by itself a valid trigger except for the event S_u.
$\exists E \ni t(E) = x$	Not a valid trigger condition since the simultaneity (equality) of two observation times is not physically well-defined.
$\exists E \ni t(E) < x$	The validity of this phrase is dependent on the relationship between x and $t(S_u)$. $\boxed{x < t(S_u)}$ Not a valid trigger condition: a program cannot differentiate between $\exists E \ni t(E) < x < t(S_u)$ and $\exists E \ni x < t(E) < t(S_u)$. $\boxed{x = t(S_u)}$ The validity of this phrase depends on the characteristics of the underlying hardware. $\boxed{x > t(S_u)}$ Depending again on the underlying hardware, this form might represent a valid trigger condition, but safety considerations dictate careful examination. Even when the hardware supports the necessary observability of an event E prior to time $t(S_u)$, such a condition should usually lead to two separate requirements, $\exists E \ni t(E) < t(S_u)$ and $\exists E \ni t(S_u) \leq t(E) < x$, since, in the former case, the time $t(E)$ is not well defined and hence could be used either not at all or only with cumbersome formalism in the output predicate for the requirement.
$\exists E \ni t(E) > x$	The validity of this phrase is dependent on the relationship between x and $t(S_u)$: $\boxed{x < t(S_u)}$ Not a valid trigger condition since the program cannot distinguish between $\exists E \ni t(E) < x < t(S_u)$ and $\exists E \ni x < t(E) < t(S_u)$. $\boxed{x \geq t(S_u)}$ A valid trigger phrase.
$\exists E \ni x < t(E) < y$	In general (as per above), this phrase is a legal trigger condition iff $x \geq t(S_u)$.

Table 2: Timing Constraints for Simple Non-Existence Event Triggers

$\neg\exists E$	Never a valid trigger condition by itself.
$\neg\exists E \ni t(E) < x$	The validity of this phrase depends on the characteristics of the underlying hardware and the relationship between x and $t(S_u)$: $\boxed{t(S_u) > x}$ Not a valid trigger phrase, as the program cannot differentiate between the case where $\left[\neg\exists E \ni t(E) < x\right] \wedge \left[\exists E \ni x < t(E) < t(S_u)\right]$ and the case $\exists E \ni t(E) < x$. $\boxed{t(S_u) = x}$ The validity of this phrase depends on the characteristics of the underlying hardware. $\boxed{t(S_u) < x}$ This phrase usually should be part of a fully bounded trigger: $\neg\exists E \ni t(S_u) < t(E) < x$. When the requirement is to pertain to the absence of E even prior to program startup, the validity is dependent on the characteristics of the underlying hardware.
$\neg\exists E \ni t(E) > x$	Never a valid trigger condition.
$\neg\exists E \ni t(S_u) < x < t(E) < y$	The normal trigger condition for "non-event" events. Note that y is a lower bound on $t(O)$ for the output O triggered by this condition.

be accommodated within a duration of time d. This is the requirement called "capacity". Multiple capacity assumptions are meaningful, although not necessarily required in any given case. For example, the capacities could be 4 per second but not more than 7 in any two seconds nor more than 13 in four seconds, etc. *One* capacity assumption is necessary for completeness; multiple assumptions may derive from application-specific considerations. There can also be multiple capacities assumed for a given input based on additional data characteristics, such as: not more than 4 inputs per second when $v(I) > 8$ but not more than 3 per second when $v(I) > 20$. Finally, note that a capacity assumption with $N = 1$ is the same as an assumption on the minimum time between successive inputs — another common "performance constraint."

Even if a particular statistical distribution of arrivals over time is assumed and specified, a capacity limit assumption is still required: Assuming the arrival distribution to be Poisson or Erlang does not preclude the possibility, no matter how improbable, of an "overflow" of any given capacity. If capacity is exceeded, there must be some specification of the ways that the system can acceptably fail soft or fail safe. This is discussed below with respect to specifying graceful degradation.

Where interrupts can be masked or disabled, the situation is more complex. If disabling the interrupt could result in a "lost" event (depending on the hardware, the duration of the lockout, and the characteristics of the device at the other end of the channel), the need for a capacity assumption will then depend on the usage of the input in the specification. An input I appearing as the only $\exists I$ event in a trigger clearly requires a capacity assumption since a "lost" I (caused by interrupt lockout) is a violation of the requirement. If I is not by itself a disjunctive trigger but a conjunctive part of one, its capacity may be dominated by some other event. Conjunctive domination of I by another event E occurs when, for example, a disjunctive trigger clause can be written as: $\exists! E, I \ni t(E) < t(I) < t(E) + d$. In this case, an interrupt for I could potentially be disabled until the event E is detected, then enabled and left enabled until I occurs or a period of time d elapses (whichever occurs first) and then disabled again. Thus the interrupt could not be overly disruptive of the computation, in that it could occur at most once in the specified interval. Note that a trigger clause such as $\exists! I_1, I_2 \ni |t(I_1) - t(I_2)| < d$, which appears to contain two undominated events, is actually an abbreviation for a trigger of two disjunctive clauses, each of which has exactly one undominated event:

$$\exists! I_1, I_2 \ni \Big[t(S_u) < t(I_1) < t(I_2) < t(I_1) + d\Big] \bigvee \Big[t(S_u) < t(I_2) < t(I_1) < t(I_2) + d\Big]$$

An interrupt-signaled event that is at any time undominated in the requirements specification requires a capacity assumption. The capacity of a totally dominated event is inferable from its dominators' capacities.

Formally, capacity is not some separate, special type of requirement (i.e., "performance"). Instead it is specifiable as a conjunctive phrase in all disjunctive clauses in the

triggers for all outputs that are capacity-dependent:

$$\exists! I \ni \left[\left|\left\{I' \mid t(I)-t(I') < d\right\}\right| < M_I(d)\right] \bigwedge \cdots \Leftrightarrow \cdots ,$$

where $M_I(d)$ is the capacity limit for I during some period of time d.

Whereas capacity involves a single type of input, load is defined in terms of multiple inputs:

$$\exists I \ni \left[\sum_i k_i \cdot \left|\left\{E_i \mid t(I)-t(E_i) < d\right\}\right| < D_L(d)\right] \bigwedge \cdots \Leftrightarrow \cdots . \qquad (A)$$

where the $k_i \geq 1$ are weights that allow some inputs to be specified as more "expensive" than others and $D_L(d)$ is the design load limit for a period of time of duration d. Capacities are assumptions on homogeneous sets of events whereas load is an assumption on a heterogeneous set.

Load is more general than capacity, in that a load condition such as that above will suffice to implicitly define $M_{E_i}(d)$, even if no explicit definition is given. In that case, for any undefined $M_{E_i}(d)$, the maximum number of E_i possible within a duration of time d will be $D_L(d)$ (or $min\{D_{L_i}(d)\}$ if there are multiple load assumptions, D_{L_i}), since all the other event terms could conceivably be zero unless there are *minimum* arrival rate assumptions $m_{E_i}(d)$ specified as well. If there are minimum arrival rate assumptions specified, the maximum capacity for any undefined M_{E_i} would be $\min\{D_{L_k}\} - \sum_{j \neq i} m_{E_j}$.

The smallest period d for which a minimum arrival rate assumption is explicitly assumed and specified is the maximum possible time between successive events. If there must be at least n events of a given type E within the interval of duration d preceding each event of that type — where n/d is the assumed minimum arrival rate — then no more than time d can elapse between any two occurrences of events of type E or the minimum arrival rate assumption would be false. For embedded systems, robustness dictates the specification of a minimum arrival rate assumption for most, if not all, possible inputs: Indefinite, total inactivity on the part of any real-world system is unlikely. Robust system design should provide a capability for the program to query the environment with regard to inactivity over a given communication path.

In general, inputs to embedded systems should have both minimum and maximum capacity assumptions and will often be part of one or more load assumptions as well. A bank in Australia reportedly lost a great deal of money by the lack of a requirement to deal with "excessive" load [Pur87]. When the central computer was unable to cope with the load, the ATMs dispensed funds whether there were funds in the account to cover the withdrawal or not. The inability to handle the true load, although irksome, would not by itself have caused as much economic damage as that which resulted from the lack of an explicit, black-box overload response requirement.

Although inputs from human operators or other, slow, external systems may be normally incapable of overloading a program, various malfunctions could cause excessive, spurious inputs to be generated. Robust system design should consider that case and specify a capacity limit for such inputs as a means of detecting possible external malfunctions.

Note that more complicated expressions than the ones discussed here could be specified and still provide the necessary restrictions on $M_{E_i}(d)$ and $m_{E_i}(d)$ and yet not have names as standard "performance" requirements. For example, in the specification of capacity, $D_L(d)$ (i.e., the design load or capacity limit for a period of time of duration d) may not be a constant but instead may be equal to some function $f(E_i)$ for some set of E_i. Such expressions and others bearing no obvious relationship to standard performance requirements may well be valid and required for a given application. Such a possibility is one of the reasons that well defined notions of completeness for performance requirements have proven to be so elusive. The distinction between performance requirements, functional requirements, and other, non-standard but none-the-less black-box behavioral requirements is largely a matter of convenience in the interpretation of syntactic forms rather than any intrinsic feature or semantic characteristic of the requirements themselves.

Absorbtivity There may need to be a distinction drawn between input capacity and output capacity. If the input environment can generate up to $M_I(d)$ inputs of type I during a period of duration d, but the output environment can only "absorb" a lower number of outputs, the program will need to handle three cases:

1. The recent input and output rates are both within limits, thus the "normal" response can be generated.

2. The input rate is within limits but the output rate would be exceeded if a normally timed output were produced, in which case some sort of delayed response will be required.

3. The input rate is excessive, in which case some abnormal response is necessary (see below).

In the case where input and output capacities differ, there must be multiple periods for which discrete capacity assumptions are specified. For the largest interval for which both input and output capacities are assumed and specified, the absorbtion rate of the output environment must equal or exceed the input arrival rate or the program might never catch up; but over shorter durations, the program can buffer or shield the output environment from excessive inputs. Input rate assumptions may be determined to be essential based solely on specification usage criteria, as discussed earlier; output capacity restrictions always stem from application-dependent considerations of the output environment's absorbtivity.

Graceful Degradation. The requirements needed for logical completeness to deal with overload will generally fall into one of five classes:

1. Requirements to generate warning messages.
2. Requirements to generate outputs to reduce the load — i.e., messages to external systems to "slow down".
3. Requirements to lock out interrupt signals for the overloaded channels.
4. Requirements to produce outputs (up to some higher load limit) that have reduced accuracy and/or response time requirements and/or some other characteristic that will allow the CPU to continue to cope with the higher load.
5. Requirements to reduce the functionality provided by the system (i.e., to cease providing certain outputs), or, in extreme cases, to shutdown, perhaps only temporarily.

The first three cases are handled in an obvious way. The fourth case, commonly called performance degradation is, as described, somewhat abrupt (i.e., not graceful). Graceful degradation may be specified by including the load in the timing or accuracy factors for the output, i.e., if the actual or observed load (during the interval of duration d immediately preceding the proximate trigger I_{pt}) is defined as

$$A_L(d) \equiv \sum_i k_i \cdot \left| \left\{ E_i \mid t(I_{pt}) - d < t(E_i) < t(I_{pt}) \right\} \right|,$$

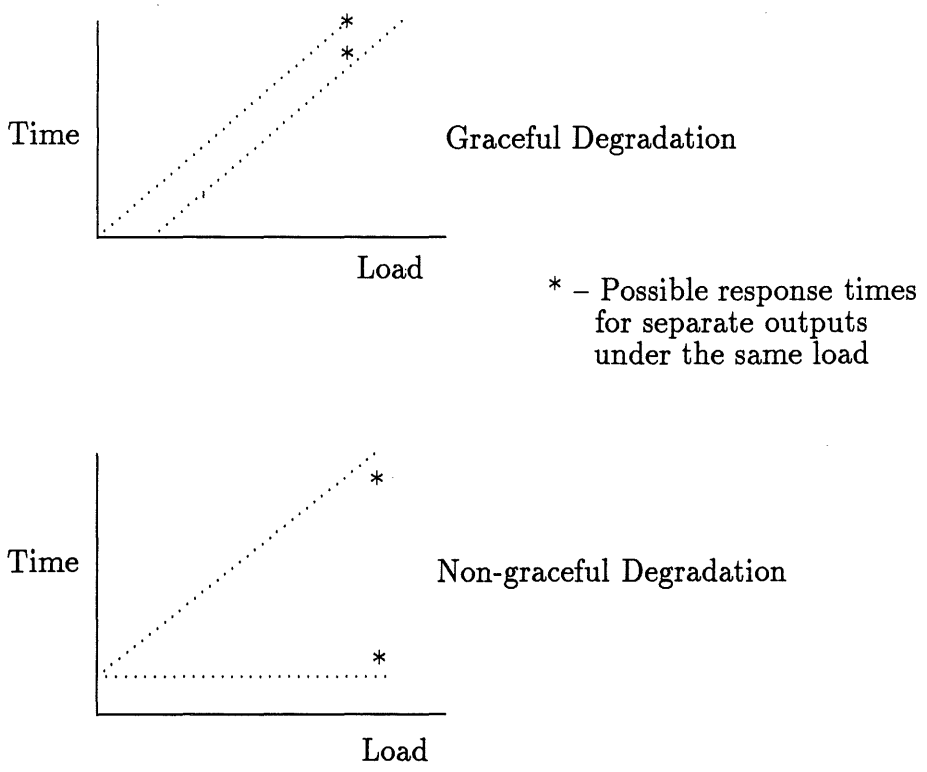
then

$$\begin{aligned} \exists I_{pt} \ni \cdots \Leftrightarrow \exists! O \quad \ni \quad & \left[v(O) = x \pm a \cdot A_L(d) \right] \\ \wedge \quad & \left[[t(I) + c_1 \cdot A_L(d)] < t(O) < [t(I) + c_2 + c_3 \cdot A_L(d)] \right] \end{aligned}$$

where $0 < c_i$ and $c_3 \geq c_1$. Note that c_3 must not be zero or the response time is not being specified as gracefully degrading: If c_1 is equal to zero, the upper limit is increasing, but not the lower limit, or delay. Two inputs occurring quite close to one another in time could then legally trigger outputs having widely different response times, potentially even appearing in inverse order from the order in which their respective triggers arrived; see Figure 1.

For safety-critical systems, abrupt degradation and/or random (although bounded) degradation often needs to be avoided. Certainly for operator feedback, "predictability is preferable to variability, at least within limits," even if the cost of the predictability is a slight increase in average response time [FD82].

Figure 1: Graceful Degradation



Function shedding, the fifth case listed above, is specified by the use of different load prerequisites for different outputs — the outputs with the lower load prerequisites being "shed" first. When the load is exceeded, then the program changes state, the "normal-mode-only" outputs cease being required, and possibly new outputs become required such as error messages, interface resets, etc. Once a state with a degraded performance has been entered, there needs to be a specification of the conditions required to return to a normal processing mode.

Informally, what is needed is a hysteresis delay. After detecting a capacity or load violation, the system must not attempt to return to the normal state too quickly or the exact same set of circumstances that caused it to leave may still exist. For example, let the event that caused the state to change be the receipt of the nth input of I within a period d, where the capacity is specified as limited to $n-1$. Then if the system attempts to return to normal within a period of $x << d$, the very next occurrence of an I might cause the state to change again to the overload state. The system could thus ping-pong back and forth in an unacceptable fashion.

If the trigger for returning to normal is a discrete input event $I \ni P_{rn}$, it is straightforward to expand it to become

$$\exists I \ni P_{rn} \bigwedge \left[\exists E_{en} \ni t(E_{en}) < t(I) \rightarrow t(I) - t(E_{en}) > h_d\right] \qquad (B)$$

where h_d is the hysteresis delay and E_{en} is an event that caused the system to exit the normal state. In other words, the most recent exit from normal was prior to time h_d ago. E_{en} should be a trigger in its own right, not just a definition: The system should take some observable action upon E_{en}, such as alerting a human operator, disabling or requesting resets of busy interfaces, recording critical parameters for subsequent analysis, etc.

Discrete events such as operator actions or reset messages from external (temporarily overloaded) interfaces are not the only way a system can return to normal processing. It may be desired to attempt to change state purely on the basis of time elapsed since last state change. System robustness considerations suggest the specification of a complex series of checks on the temporal history of mode exit/resumption activities to avoid constant ping-ponging at a cyclic rate h_d. Choice of responses and checking logic is an application-dependent activity, as is choice of the value h_d, but many real-time system requirements specifications will be incomplete without inclusion of such behaviors.

State Completeness Assumptions

As noted earlier, a trigger of the form $\exists I \ni P_I$ will, in a logically complete specification, require treatment of the case $\neg(\exists I \ni P_I)$, which is logically equivalent to $\neg\exists I \bigvee \exists I \ni \neg P_I$. For obvious reasons, the two alternatives generally lead to two very different requirements:

the $\exists I \ni \neg P_I$ case pertains to unexpected states while the $\neg\exists I$ case has implications for startup and shutdown.

Unexpected States. Informally, when a given state variable is part of an input trigger for an input I, a logically complete set of requirements must contain a specified response for the receipt of I in conjunction with all possible values for that state variable including, where appropriate, the common case of the undefined state. For example, suppose ON is an abbreviation for the fact that the last input received indicated that an interface device was '*on*' and OFF indicates that the last input received indicated the device was 'off.' If $I \wedge ON$ is a trigger for some output, the specification needs to address the case for $I \wedge OFF$ as well. Furthermore, $\neg ON$ is *not* equivalent to OFF, since the "state" of the device may be indeterminate if no input has been received. Therefore, there needs to be a requirement to deal with the case when $\neg(ON \vee OFF)$. Tautological completeness will ensure that such requirements are included.

Many software problems arise from incomplete specification of state assumptions. For example, Melliar-Smith [Neu85] reports a problem detected during an operational simulation of the shuttle software that occurred when the astronauts attempted to abort the mission during a particular orbit, changed their minds and cancelled the abort attempt, and then decided to abort the mission after all on the next orbit. The program got into an infinite loop which appears to have occurred because the designers of the simulation program had not anticipated that anyone would ever want to abort twice on the same flight. Another example involves an aircraft weapons management system that attempts to keep the load even and the plane flying level by dispersing weapons and empty fuel tanks in a balanced fashion [Neu85]. One of the early problems was that even though the plane was flipped over, the computer would still drop a bomb or a fuel tank which then dented the wing and rolled off.

In some cases, there really is no requirement to respond to a given input except in a subset of the states defined by the state variables to which it *is* sensitive; but, often, an input arriving in an "unexpected" state is a possible indication of a "disconnect" between the system and its environment that should not be ignored. For example, a target detection report from a radar that has been commanded to the 'off' state is probably an indication that either the radar did not shut off or that its detection logic may be malfunctioning. If, in fact, the input is of no significance, it is still important, as discussed earlier, for documentation and communication purposes that the requirements synthesist explicitly record the fact that all the cases have been considered and that the input may truly be ignored: An explicit "do nothing" requirement is a practical approach in that case.

Startup and Shutdown. If there is a requirement $\exists I... \Leftrightarrow ...$, then the requirement $\neg\exists I... \Leftrightarrow ...$ is needed for logical completeness. However, this is not an acceptable requirement, for the reasons presented earlier: The trigger condition must state bounds on

the interval during which no input I is received. To achieve logical completeness, program behavior must be specified for three bounded intervals: the interval prior to program startup, the interval of normal program execution, and the interval after program shutdown (specified by S_d). The disjunction of the separate triggers for these bounds:

$$\begin{aligned} \neg\exists I \ni \quad & t(I) < t(S_u) \\ \neg\exists I \ni \quad & t(S_u) < t(I) < t(S_d) \\ \neg\exists I \ni \quad & t(I) \geq t(S_d) \end{aligned}$$

is logically equivalent to the necessary $\neg\exists\ I$ case. In addition to considering the absence of input in these intervals, there may need to be a specification of responses to input that arrives before startup and after shutdown.

For the first interval, i.e., before the program starts up, completeness considerations require assertions with the triggers:

$$\begin{aligned} \neg\exists I \ni t(I) \leq t(S_u) \\ \exists I \ni t(I) \leq t(S_u) \end{aligned}$$

If the hardware cannot retain or indicate the receipt of an input prior to the event S_u, the program cannot differentiate between the two cases and both conditions should lead to a "do-nothing" response. If the (abstract) machine can in fact "observe" the existence of an input I prior to the event S_u, the program has two startup states with respect to the given input (i.e., the input was present or not) and distinct behavior can be required as appropriate:

$$\begin{aligned} \exists I \ni t(I) < t(S_u) \Leftrightarrow \cdots \\ \neg\exists I \ni t(I) < t(S_u) \Leftrightarrow \cdots \end{aligned}$$

Note that in the case of events occurring before program startup, $t(E)$ is undefined (although bounded from above) and careful consideration must be given to the use of $v(E)$ in the requirements, as it is hardware dependent which $v(E)$ is retained in the (unobservable) case that there were multiple events E prior to program startup: Some hardware may retain the first such event, some the most recent, etc.

After program startup, there should be some finite bound on the time the program waits without receiving a given input before it tries various alternative strategies such as alerting an operator or shifting to an open-loop control mechanism that does not utilize the absent input. Note that this is very similar to the previously discussed maximum-time-between events condition but applies to the time after startup in the absence of even the first input of a given type. This type of quiescence after startup yields one of the necessary $\neg\exists I$ intervals noted above, i.e.,

$$\neg\exists I \ni t(S_u) < t(I) < t(S_u)+d_0 \Leftrightarrow \exists! O \ni \cdots$$

There may (and in general, will) be a series of intervals d_1, d_2, etc. during which the program is required to attempt various means of dealing with the lack of input from the environment. Eventually, however, there must be some period after which, in the absence of input, the conclusion must be that there is some malfunction and that all future lack of that input is of no further significance.

Finally, logical completeness requires an assertion that states that after program shutdown, S_d, no outputs are required, i.e.,

$$\neg\exists I \ni t(S_d) \leq t(I) \Rightarrow \neg\exists O$$

and an assertion that considers what to do with inputs that arrive after shutdown, i.e.,

$$\exists I \ni t(S_d) \leq t(I) \Rightarrow \neg\exists O$$

Responsiveness and Spontaneity

Responsiveness deals with the classification of outputs as to their effect on the environment. In particular, does a given output O cause the environment to change, and, if so, is that change detectable at the program's black-box boundary by means of some input I? If the environment does not respond to an output within some expected period of time, there is presumably some abnormality somewhere and the program should be required to act accordingly — perhaps by trying a different output, by alerting a human operator, or, at the least, logging the abnormality for future, off-line analysis. Bahn [Bah88] reports an accident involving a steel plant furnace that was returned to production after being shutdown for repairs. A power supply burned out in a digital thermometer during power up so that the thermometer continually registered zero degrees. The controller, knowing it was a cold start, ordered 100% power to the gas tubes. The furnace should have reached operating temperature within one hour, but the computer failed to detect that the thermometer inputs were not increasing as they should have. After four hours, the furnace had burned itself out, and major repairs were required.

Every output O to which a detectable response I is expected within a period of time d induces at least two requirements: The "normal" response requirement, i.e.,

$$\exists! O, I \ni t(O)+\delta t < t(I) < t(O)+d \Leftrightarrow \ldots ,$$

and the requirement (which is needed for logical completeness), to deal with a failure of the environment to produce the expected response. The failure could involve either the response having an erroneous or unreasonable value or the expected response might be missing entirely, i.e.,

$$\exists! O \neg\exists I \ni t(O)+\delta t < t(I) < t(O)+d \Leftrightarrow \ldots .$$

where δt represents a latency period, i.e., the time between the receipt of an input and its processing. This is discussed more fully below, but note the need to specify a $\delta t > 0$ here. If the environment responds *too* quickly, one suspects coincidence rather than appropriate stimulus-response behavior. A value-based handshake protocol can be used to eliminate the need for the δt factor, i.e., some field of the input I identifies it as uniquely a response to some specific output O. Note that some inputs I are spontaneous, i.e., they may be triggered by environmental factors not necessarily caused by some prior output O. But an input I that is supposed to be non-spontaneous, i.e., one that is only supposed to arrive in response to some prior system output, induces yet another requirement to respond to a presumably erroneous (i.e., spontaneous) input I:

$$\exists! I \neg \exists O \ni t(I) - d < t(O) < t(I) \Leftrightarrow \ldots .$$

5 Output Specification Completeness

The previous section examined completeness with respect to triggers. There are also completeness criteria that can be applied to the right side of the requirements assertions, i.e., the specification of the outputs. Certainly, the criteria for complete specification of outputs will differ from those for triggers. In particular, the notion of logical completeness for outputs is quite different (and less powerful in its consequences) for output predicates than for triggers. While it is necessary to consider all possible environmental conditions and specify responses (or non-responses) to them, it is doubtful that a software system would need to generate all possible types of outputs. Identifying the subset of outputs that is required is not a software engineering problem, however; it is a system engineering issue. If a particular software behavior is not specified in the system requirements specification, it is not reasonable or appropriate to include it in the software requirements specification. Therefore, there is no way to guarantee complete detection of missing output requirements by looking only at the software requirements specification.

It is possible, however, to apply logical completeness to that part of the output predicate that deals with the conditional selection of attributes. It is also possible to derive application-independent rules and criteria to "close" the output specification with respect to various criteria important in the controlled system and thus suggest some outputs that may have been inadvertently overlooked that are logically related to the outputs already specified.

Completeness of Output Predicates

The terms used in the expression of the predicate P_O must be one of the following:

- Constants,

- Existentially quantified variables in the triggering events for O, or
- Existentially quantified variables in conditional selectors

The point is that in specifying the characteristics of the output O, references to the time and/or value of other events E are mathematically undefined unless the other events actually exist.

Selector clauses are the formal representations of conditional logic for output value definition and need to be tautologically complete. In general, if

$$v(O) = \begin{cases} x & \text{if } A \\ y & \text{if } B \end{cases}$$

then $A \bigvee B$ must be a tautology. Similarly, $t(O)$ can be defined using conditional logic, although there seem to be few practical cases where it is necessary. Note the difference between triggers and selectors: Triggers determine if an output is to be required at all while selectors specify its contingent characteristics.

The complete specification of the behavior of an output O requires delineating both its time $t(O)$ and its value $v(O)$. Again note that the time is required. There are three possible types of value specifications:

1. The requirement for $v(O)$ involves interpretation of $v(O)$ as a real number, in which case upper and lower bounds are required.
2. The bit pattern for $v(O)$ is specified exactly, in which case, equalities such as $v(O) =$ '*Enter another file name*' are appropriate.
3. The bit pattern of O is not a requirement, but some observable attribute of the bit pattern (e.g., parity) is, in which case the requirement would be specified as $v(O) \in \{x | \cdots\}$.

Note that case (1), above, interpretation as a real number, can be expressed in terms of case (2) for formal simplification: Using the notation that $v_i^j(O)$ will stand for bit positions i through j inclusive, it is obvious that if $l < v(O) < u$, $\exists i, j \ni l_i^j = u_i^j$ and the requirement could be expressed as $v_i^j(O) = u_i^j$. If case (2) is considered specification via a fully deterministic bit pattern and case (3) is considered specification via non-bit-specific attributes, then interpretation as a real number might be considered a case of partially deterministic bit specification (i.e., some bit positions determined, some not) which is reducible to the catenation of a fully deterministic field with a non-bit-specific field.

Ambiguity of reference in requirements specification is a common result of incompleteness in selector clauses. As an example, consider a requirement to output the sum of the last three inputs received:

$$v(O) = \begin{cases} \Sigma v(I_i) & \text{if } \exists! \, I_1, I_2, I_3 \ni P_I \\ -1 & \text{otherwise} \end{cases}$$

where $P_I ::= t(S_u) < t(I_1) < t(I_2) < t(I_3) \bigwedge \neg\exists I_4 \ni t(I_3) < t(I_4) \bigwedge V(I_i)$, with $V(I_i)$ including value and range restrictions for the I_i. Without the uniqueness of the existential quantification, the term $\Sigma v(I_i)$ would not be well defined. Contrast this with a requirement where the output is to be zero if there have been at least three inputs received:

$$v(O) = \begin{cases} 0 & \text{if } \exists\, I_1, I_2, I_3 \ni t(I_i) < t(O) \\ -1 & \text{otherwise} \end{cases}$$

In this case, the unique existential quantification would not be required.

Timing. There are several special issues with respect to specification of the timing of outputs: latency, data age, and the problems involved in specifying complex sets of outputs. Note that there is obviously no limit to the complexity of timing specifications for outputs. Issues such as minimum and maximum time between outputs are as potentially vital as the same concepts pertaining to inputs. The difference is that for inputs, such specification is an internal completeness issue; it is possible to determine whether the specification is required by looking only at the software requirements. For outputs, a determination of whether such a specification is required is dependent upon external completeness issues, i.e., particular characteristics of the controlled system.

Latency. One potential timing incompleteness involves the specification of latency, a problem discussed in a slightly different context by Kopetz and Damm[KD87]. Since a computer is not arbitrarily fast, there is an interval of time during which the receipt of new information cannot change an output O even though it arrives prior to the actual output of O. The duration of this interval, called δt by Kopetz and Damm[KD87], is a factor influenced by both the hardware and the software. An executive or operating system that permits interrupts for data arrival may be able to exhibit a shorter δt than one that polls periodically, but underlying hardware constraints prevent it from being eliminated completely. Thus the latency interval can be made quite small, but it can never be reduced to zero. The choice of operating system, interrupt logic, scheduling priority and/or system design parameters may be influenced by the value of δt. Also, behavioral analysis of the requirements (see, e.g., Jahanian and Mok[JM86]) may not be correct unless the value of this behavioral parameter is known and specified for a given program. Therefore, the requirements must include the allowable δt factor in order to be complete.

As an example, consider an output O that is to signal, within a response time r_t, the fact that no input of type I has been received within the previous period of time of duration d. This could be specified as

$$\begin{aligned} &\exists!\, S_u, I_1, \neg\exists I_2 \ni t(S_u) < t(I_1) < t(I_2) < t(I_1)+d \\ &\quad \Leftrightarrow \quad \exists! O \ni t(I_1)+d+\delta t < t(O) < t(I_1)+d+\delta t+r_t \bigwedge v(O) \cdots \end{aligned}$$

The use of an interval of time without some event E to trigger an output always requires the specification of a δt factor between the end of the interval and the occurrence of the output or the specification is incomplete. Where the upper bound on the interval is a simple event, i.e., the proximate trigger is not the non-existence interval but the terminating event itself, then latency is not an issue. However, where the intent is to signal the non-existence of an input after some other event, then a latency specification is required. This is true for both trigger and output predicates.

In some cases, the need for latency specification may appear to be application dependent. Consider a trigger of the form:

$$\exists! \, S_u, I_1, \neg\exists I_2 \ni t(S_u) < t(I_1) - c < t(I_2) < t(I_1) + c \cdots \; .$$

If the semantic intent is to be that there is no input I_2 prior to the output, the latency factor is missing. If the intent is to be that there was an I_1 with no I_2 within the interval around it, the latency factor is unnecessary. Since intent is not analytically tractable, and since software may be re-used in environments where the current intentions differ from those at the time of the requirements specification, safety considerations dictate that the latency factor always be included when the non-existence interval's upper bound is not a simple observable event.

There may need to be additional requirement assertions to handle the case where an event is observed within the latency period. For example, if an action is taken based on the assumption that some input never arrived and if it is subsequently discovered that the input actually did arrive but too late to affect the output, it may then be necessary to take corrective action.

Data Age. Another important aspect of the specification of timing involves data obsolescence. In practical terms, there are few, if any, input values that are valid forever. Even if nothing else happens and the entire program is idle, the mere passage of time renders much data of dubious validity eventually. Although the program is idle, the real world in which the computer is embedded is unlikely to be. Data obsolescence considerations require that existential quantification of input (or output) events in selector clauses must be properly bounded in time.

$$\ldots \Leftrightarrow \exists! O \ni \ldots \bigwedge v(O) = \begin{cases} x & \text{if } \exists! \; I \ni t(S_u) < t(O) - D_V < t(I) < t(O) \\ y & \text{otherwise} \end{cases}$$

where D_V is the age limit or data validity factor for the input I. Note that $t(S_u)$ by itself is rarely a proper lower bound. The input is only valid for the output O if it occurred within the preceding period of time of duration D_V. As an example of the possible implementation implications of such a requirement, MARS [KM85], a distributed fault tolerant system for

real time applications, includes a validity time for every message in the system after which the information in the message is discarded.

Frola and Miller[FM84] report on an accident related to and perhaps caused by lack of specification of a data age factor where a computer issued a 'close weapons bay door' command on a B-1A aircraft at a time when a mechanical inhibit had been put in place on the door. The 'close' command was generated when someone in the cockpit punched the 'close switch' on the control panel during a test. Several hours later, when the maintenance was completed and the inhibit removed, the door unexpectedly closed. The situation had never been considered in the requirements definition phase; it was fixed by putting a time limit on all commands.

Specification of Complex Output Sets. Sometimes a given trigger is to require the production of multiple outputs, not just a single output. The outputs must have observably distinct characteristics in either time or value (or both). There is no limit to the complexity of such output set behavior and the details are always application-specific. A particular benefit of analyzing complex behavior in terms of the predicate calculus, however, is that it highlights the omission of information essential to the discrimination among observably distinct behavioral patterns. Such alternatives often have significant safety implications. Even so well an understood phenomenon as periodicity, for example, has several pitfalls that can be clearly revealed in this fashion. Phase-lock (i.e., the maintenance of a constant temporal relationship between two periodic signals), although only one of several potential problems that arises in specifying periodics, is perhaps the most obvious example.

Three free variables are required to be defined for even the simplest periodics: Let p = the required periodicity (expressed as a duration of time between successive outputs), a = the required timing accuracy, and r_0 = a reference time (to denote the start of the periodic output). Note that the reference time may be a more complex expression than just the time, $t(E)$, of some event. A program that starts a periodic output 10 seconds after receipt of some input is exhibiting different behavior than one that commences within 1 second of the receipt of that input, and this level of detail needs to be included in a complete specification. In this case, r_0 would need to be replaced by two variables: the event E and some initial delayed-response time d.

There are at least two distinct alternative expressions for periodic behavior. A phase-locked periodic requires expressing the time of an output as a multiple of the required periodic interval after the reference time, r_0:

$$\forall n \geq 1,\ \exists! O_n \ni [r_0+np-a] < t(O_n) < [r_0+np+a]$$

Alternatively, relating the required time for each output to the required interval from the preceding output (i.e., defining $t(O_0) \equiv r_0$), results in a specification of a free periodic of the form:

$$\forall n \geq 1,\ \exists! O_n \ni [t(O_{n-1})+p-a] < t(O_n) < [t(O_{n-1})+p+a]$$

These requirements both capture 'periodic' behavior, but they are quite different: Figure 2 illustrates the difference. Notice the "drift" of the second specification in comparison with the first; the phase-locked periodic in the figure results in 9 outputs in the same interval in which the free periodic (using relative time) has 10 outputs. In the first alternative, the maximum time that can elapse between successive outputs of O is $p + 2a$, whereas in the second alternative it is only $p + a$. Even if the first requirement is rewritten with an $a' = a/2$, which reduces the maximum time between successive outputs of O to $p + a$, the same as for the second alternative, the phase lock difference remains. Let $M_O(d)$ be the maximum possible number of occurrences of O in a period of time of duration d. For the first alternative, $\left|M_O(d) - d/p\right| \leq 1$, no matter how large d gets. For the second alternative, $\left|M_O(d) - d/p\right|$ is potentially unbounded as d grows arbitrarily large, regardless of the value of a.

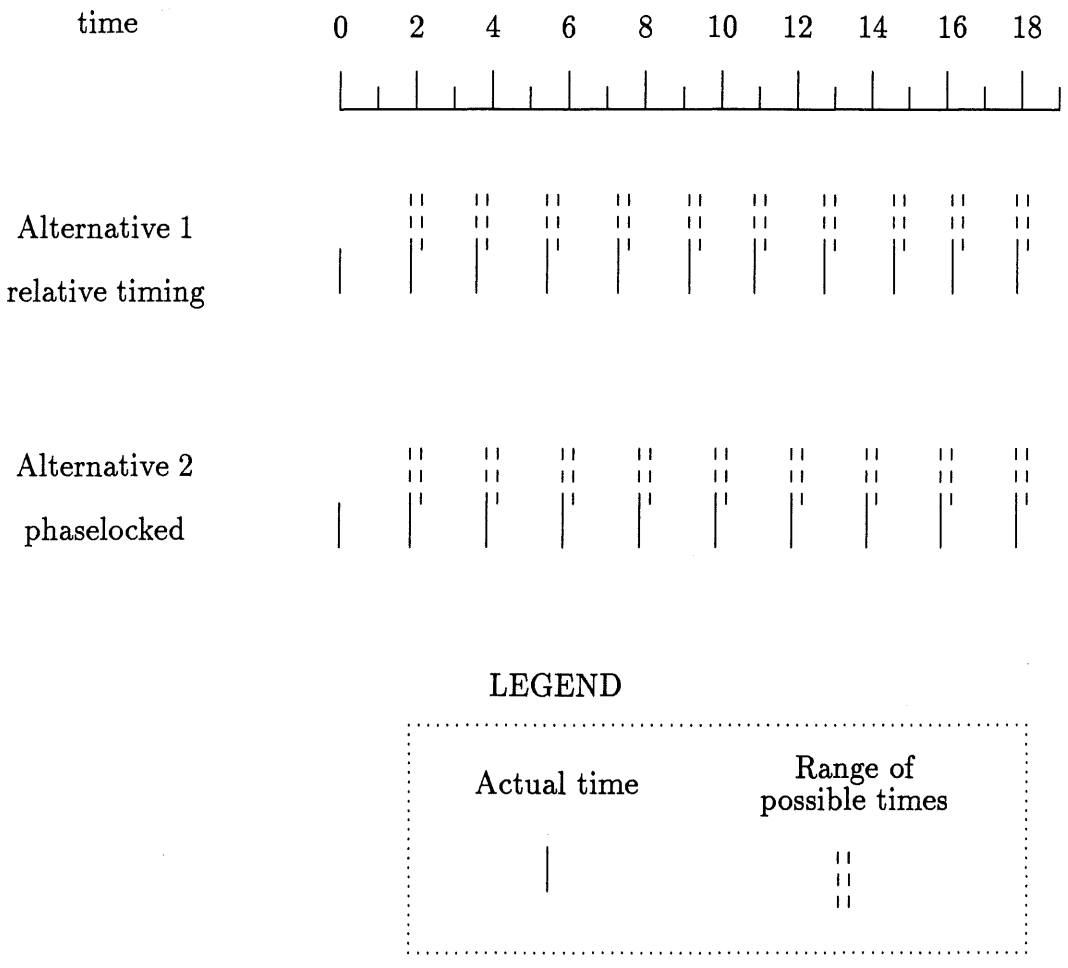
As stated in the introduction, the requirements specification document needs to contain enough information to allow designers to distinguish between observably distinct behavioral patterns. Languages or specification procedures that do not permit the expression of subtle distinctions will be inadequate for some applications. None of the existing, major requirements languages or techniques has syntax for periodics, for example, that can discriminate between the two cases described above. The existing abstractions omit potentially important details of observable behavior and therefore are inadequate to express completely the requirements for some systems. In fact, none of these languages include the ability to specify all the attributes of value and time described in this paper and considered by the authors to be essential. Some of these attributes are not specifiable in any of the languages.

Criteria for Detecting Missing Output Assertions

As stated earlier, logical completeness is not as powerful a concept when applied to outputs as it is for triggers; it is applicable only to that part of the output predicate that deals with the conditional selection of attributes. There are, however, some application-independent criteria in the form of rules and heuristics using state information that can be used to evaluate output event specifications and detect some missing types of functionality. Safety and robustness criteria, for example, may be exploited to develop these type of application-independent closure criteria. Several examples of such criteria are described below. There are additional useful criteria that can be developed other than those discussed below, but the criteria presented below are important for many real-time applications and are illustrative of the types of criteria that can be used.

Completeness Based on Output Values. Specification of possible values of outputs is important in completeness analysis as it was for triggers. There is a great likelihood that

Figure 2: Earliest possible outputs for the two periodics, one phase-locked, the other not. Both have $p = 2$ and $a = 0.2$ Note that after time 18, alternative 2 has produced one more output than has alternative 1; in general, after time 18n, that would be n more.



a specification is incomplete if there is some legal value for the output that never appears on the right side of an equality for $v(O)$. For example, if $v(O) \in \{\text{'}on\text{'}, \text{'}off\text{'}\}$ and the set of requirements specifies when to generate $v(O) = \text{'}on\text{'}$ but imposes no requirement to generate $v(O) = \text{'}off\text{'}$, the specification is almost certainly incomplete. Forcing the requirements to include explicitly the set of discrete values or range of values possible for a given output will highlight such incompleteness.

Reachability and Recurrent Behavior. A common incompleteness in a specification is failure to specify how the system can change states once it has "reached" a given state. Suppose, for example, that there is a state (a history of input and output events) abbreviated $MODEn$, and that $\neg MODEn$ is a prerequisite to the generation of some output X. Let the existence of an output X be abbreviated $STATEx$. In that case, when $MODEn$ prevents the generation of the output X, $STATEx$ is not "reachable" from system state $MODEn$: Generally, there are some circumstances where that unreachability is appropriate and correct, but it is often indicative of an incompleteness.

Although $STATEx$ may not be reachable from $MODEy$, it may still be reachable from $MODEz$ which is reachable from $MODEn$. Thus, one can distinguish between direct and indirect reachability. The nature of the reachability graph for the states involved in the trigger for a given requirement is central to one form of completeness. If every prerequisite state for a given output O is completely unreachable from every inhibiting state, then that output can never occur again if the system once reaches any state where it (the output) cannot be produced. Whether or not this is an incompleteness is dependent on the application. On the one hand, most embedded systems operate in an environment presumed to be cyclic (and hence, not irreversible) in nature; but on the other hand, most systems include special shutdown behavior and some may have other, application-dependent, non-recurring patterns as well. Startup and shutdown are two common examples. It is a straightforward (although non-trivial) task to determine whether or not a given output is reachable from a given state; where it is not, there is a potential incompleteness and the requirements synthesist must call into play knowledge from the application domain to resolve the issue.

Even where a given output's behavior is to be repeatable, there is the question of the nature of the prerequisites for repetition. An output to turn a piece of equipment 'on' may perhaps be inappropriate unless the last output turned the equipment 'off'. By including a check of the last output condition in the prerequisites for the next output condition, a specification that is to be logically complete would then be forced to deal both with the start-up problem (there has been no prior output) and the possible error condition that, for example, something seems to be trying repeatedly to turn a piece of equipment 'off' even though it is already 'off'. The point here is that repeatable outputs may have preconditions on their repetition that should be included in their trigger clauses.

Reversibility. The '*on*'/'*off*' behavior discussed above is significant not only from the standpoint of repeatability but also as an example of the interesting output characteristic of "reversibility". Outputs should be reviewed and classified as to their reversibility. In addition to obvious reversibility, as in the '*on*'/'*off*' case, many outputs are reversible by dissimilar outputs. For example, an alert condition to an operator — e.g., a minimum-safe-altitude-warning to an air traffic controller — should be reversible when the condition is no longer true. But there may need to be several different classes of reversing outputs, depending, for example, on whether the controller has acknowledged the receipt of the original alert, is in the process of reviewing the alert, or has taken positive action to ameliorate the alert condition. The human/machine interface, in particular, is full of complex classes of reversible phenomena [Jaf88]. Such "indirectly" reversible outputs require a complex set of preconditions, all of which should be specified in order to provide robustness in the form of explicit response to the detection of events that would "normally" trigger reversing outputs but which occur under "unexpected" circumstances.

Path Robustness and Safety. For safety-critical embedded systems, there are additional concerns. To move from one state to a directly reachable state requires an event; to move to an indirectly reachable state requires a series of events. Consider an output O such that $v(O) \in \{\text{'}on\text{'}, \text{'}off\text{'}\}$. Suppose that there is a state that generates $v(O) = \text{'}on\text{'}$ but does not generate $v(O) = \text{'}off\text{'}$. Even if a state that sets $v(O) = \text{'}off\text{'}$ can be reached from the state that set $v(O) = \text{'}on\text{'}$, there is still the question of the robustness of the behavioral path. Suppose that every possible path from a state that sets $v(O) = \text{'}on\text{'}$ to any state that sets $v(O) = \text{'}off\text{'}$ includes the event $\exists I$. Then if the system's ability to receive I is ever lost, there are circumstances under which it will not be able to set $v(O) = \text{'}off\text{'}$, depending on the state from which the system set $v(O) = \text{'}on\text{'}$. Thus, the loss of the ability to receive I may be said to be a **soft failure mode** for the event that sets $v(O) = \text{'}off\text{'}$, in that a failure that precludes receipt of I *could* inhibit setting $v(O) = \text{'}off\text{'}$. If there is no state from which both $v(O) = \text{'}off\text{'}$ and $v(O) = \text{'}on\text{'}$ can be generated, and the event $\exists I$ is in every path from a state that sets $v(O) = \text{'}on\text{'}$ to one that sets $v(O) = \text{'}off\text{'}$, the loss of the ability to receive I may be said to be a **hard failure mode** in that its loss *will* inhibit the event that sets $v(O) = \text{'}off\text{'}$. The more failure modes a set of requirements contains, whether soft or hard, the less robust will be the system that is correctly built to that specification.

Note that robustness will not, in general, be the only attribute of the total system situation that needs to be considered when specifying the requirements. It may not even be desirable at all! Consider the following safety criterion: an unsafe state, i.e., one from which an *a priori* "dangerous" output such as a command to launch a weapon can be produced, should have at least one, and possibly several, hard failure modes for the production of the output command: No input received from proper authority, no weapons launch! On the other hand, a fail-safe system should have no soft failure modes, much less

hard ones, on paths between dangerous states and safe states. Leveson and Stolzy [LS87] describe analysis procedures to provide this type of safety information.

6 Conclusions

This paper has described internal completeness criteria for triggers and output events in real-time black-box requirements specification. Emphasis has been placed on aspects of software requirements specifications that previously have not been adequately handled, including timing abstractions, safety, and robustness.

There are other types of completeness criteria that can be added to those described here: However, the rules included in this paper are certainly minimal. The theoretical foundation presented can be used as a basis for deriving and analyzing such extensions. One of the obvious omissions from this paper is consideration of the human-machine interface (HMI). At the individual requirement level, software requirements for the HMI are no different from any other requirements and may be expressed or analyzed via the observable formalism developed in this paper. There are additional closure criteria for a set of HMI requirements, however, and these are set forth in [Jaf88].

One of the conclusions of the work presented in this paper is that a complete requirements specification is large, tedious, and unwieldy. Fortunately, it may not be required. For example, it may be feasible to perform hazard analyses to determine what actions of the software are critical[Lev86, LH83, LS87] and to use these analyses to guide and limit the requirements specification. For example, outputs that are determined to be hazardous with respect to particular timing issues may require more careful and complete specification than those that can be shown to be non-hazardous. We are currently extending the analysis techniques to include system requirements and models in order to provide this type of information. We are also adding consistency analysis including consistency with safety criteria and studying how these completeness and consistency criteria may be applied to current formal specification languages. Our long-term goal is to design an environment for software requirements specification and analysis in safety-critical, real-time systems that includes languages and tools to assist the requirements analyst.

References

[Bah88] Bahn. "Reliance on Computers". FORUM ON RISKS TO THE PUBLIC IN COMPUTER SYSTEMS, ACM Committee on Computers and Public Policy, Peter G. Neumann, moderator; Volume 6 : Issue 40, 9 Mar 1988.

[BMU75] B.W. Boehm, R.L. McClean, and D.B. Urfig. "Some Experiences with Automated Aids to the Design of Large-Scale Reliable Software". *IEEE Transactions on Software Engineering*, SE-1(2), February 1975.

[End75] A.B. Endres. An Analysis of Errors and Their Causes in Software Systems. *IEEE Transactions on Software Enginerring*, SE-1(2), February 1975.

[FD82] J.D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* The System Programming Series. Addison-Wesley, Reading, Massachusetts, 1982.

[FM84] F.R. Frola and C.O. Miller. System Safety in Aircraft Management. Technical report, Logistics Mangement Institute, Washington, D.C., Jan, 1984.

[Jaf88] M.S. Jaffe. *Completeness, Robustness, and Safety in Real-Time Software Requirements Specifications.* PhD thesis, University of California, Irvine, 1988.

[JM86] F. Jahanian and A.K. Mok. "Safety Analysis of Timing Properties in Real-Time Systems". *IEEE Transactions on Software Engineering*, SE-12(9):890–904, Sep, 1986.

[Kle88] T. Kletz. "Wise After the Event". *Control and Instrumentation*, 20(10), Oct 1988.

[KD87] H. Kopetz and A. Damm. MARS: Concepts and Design of the Second Prototype. Technical Report 4/87, Technical University of Vienna, Jan 1987.

[KM85] H. Kopetz and W. Merker. "The Architecture of MARS". *FTCS-15*, 1985.

[Lam88] J. Lamb. "the everyday risks of playing safe". *New Scientist*, 8 Sept 1988.

[Lev86] N.G. Leveson. "Software Safety: What, Why, and How". *ACM Computing Surveys*, 18(2):125–164, June, 1986.

[LH83] N.G. Leveson and P.R. Harvey. "Analyzing Software Safety". *IEEE Transactions on Software Engineering*, SE-9(5):569–579, Sep, 1983.

[LS87] N.G. Leveson and J.L Stolzy. "Safety Analysis Using Petri Nets". *IEEE Transactions on Software Enfineering*, SE-13(3), Mar 1987.

[Neu85] P.G. Neumann. "Some Computer-Related Disasters and Other Egregious Horrors". *ACM Software Engineering Notes*, 11(5), Oct 1986.

[NYT86] *New York Times*, Science Times Section, July 29, 1986, p. C1.

[PC86] D.L. Parnas and P.C. Clements. "A Rational Design Process: How and Why to Fake It". *IEEE Transactions on Software Engineering*, SE-12(2), Feb, 1986.

[Pur87] D. Purdue. "Australian ATMs ...". FORUM ON RISKS TO THE PUBLIC IN COMPUTER SYSTEMS, ACM Committee on Computers and Public Policy, Peter G. Neumann, moderator; Volume 5 : Issue 3, 18 June 1987.

[Rom85] G.C. Roman. "A Taxonomy of Current Issues in Requirements Engineering". *IEEE Computer*, 18(4):14–23, Apr, 1985.

[RS78] C.V. Ramamoorthy and H.H. So. "Software Requirements and Specifications: Status and Perspectives". In C.V. Ramamoorthy and R. Yeh, editors, *Tutorial: Software Methodology.* IEEE, New York, New York, Nov, 1978.

[SB82] W. Swartout and R. Balzer. "On the Inevitable Intertwining of Specification and Implementation". *Communications of the ACM*, 25(7):438–440, July 1982.