Software Requirements Analysis for Real-Time Process-Control Systems

Matthew S. Jaffe
Nancy G. Leveson
Mats Heimdahl
Bonnie Melhart

Technical Report 90-04

Submitted for journal publication.

Software Requirements Analysis for Real-Time Process-Control Systems[1]

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

Nancy G. Leveson
Mats Heimdahl
Bonnie Melhart

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

1 Introduction

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]. Errors introduced during the requirements phase can cost up to 200 times more to correct than errors introduced later in the life cycle [Boe81] and can have a major impact on safety. Therefore, techniques to provide adequate requirements specifications and to find errors early are of great importance.

One application area for which requirements specification is particularly critical is process-control, i.e., software that controls arbitrarily large or energetic physical phenomena. Such software is 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 the software is used to assist in the formulation and implementation of decisions made by the computer or by humans for the purpose of controlling the larger system. In such process-control systems, minor behavioral distinctions often have significant consequences. It is therefore particularly important that the requirements specifications distinguish the behavior of the desired software from that of any other, undesired program that might be designed, i.e, they must be both precise (unambiguous) and correct with respect to the encompassing system requirements. The requirements analysis techniques

[1]This work was partially supported by NASA grant NAG-1-668, NSF Grant CCR-8718001, and by NSF CER Grant DCR-8521398.
A shorter and quite different version of this paper was presented at the 11th International Conference on Software Engineering, Pittsburgh, May 1989. The content has changed significantly.

discussed in this paper are most relevant for these types of systems.

Our goal is to provide analysis procedures to help find flaws in the software requirements specifications for process-control systems early in the software development. The approach is an engineering approach where a model (the requirements specification) is built and then analyzed to ensure that the properties of the model match the desired behavior. Some of the analysis procedures to be described in this paper involve the checking for consistency with criteria that must be satisfied by all such systems; these criteria often arise from the basic properties inherent in any process-control system. Other procedures rely on heuristics that can be used to improve the specification by examining, within the context of the particular process being controlled, properties that are often present in such systems.

This approach complements other approaches to the same problem that have been proposed. For example, prototyping is very useful for ensuring that the human/machine interface (HMI) has been properly described. It is less powerful when attempting to ensure that physical processes are correctly controlled by the software, especially when, as is usual, the rest of the system has not yet been built or even completely designed when the software development must begin. Furthermore, prototyping has the same limitations as testing when the goal is to ensure with very high confidence that certain constraints, such as safety, are always satisfied. It is possible to make guarantees about the behavior of the software only for the particular inputs applied and not for all inputs. Prototyping may involve a lot of work just to find out that the wrong software was built and the work must start again. It is helpful to find and eliminate as many problems as possible before the prototyping begins.

Executable specifications (such as [BCF83, Zav82, Rea89]) are related to prototypes in that such specifications may actually act as a prototype, but they may involve less coding than prototyping. Like a prototype, such specifications may be inefficient since they may require specifying more than would normally be necessary and again are limited in power of analysis and assurance to that possible with testing. Some executable specifications are also not very readable and thus are less useful as specifications for the rest of the software development process. But like prototypes, executable specifications are better than the approach presented in this paper at finding problems in the specification of the HMI — few models of human behavior exist and very little is known about the desirable and necessary properties of HMI design.

Because our goal is to present general analysis procedures that can be applied to black-box requirements specified in any language, we use a notation and analysis model in this paper that can be easily mapped to many of the current real-time requirements specification languages (such as [Hen80, Har87, Alf77]), i.e., the first-order predicate calculus and a simple state machine model. Our goal is not to provide another language for specification of requirements; the formal notation is for the purpose of providing rigor in defining the analysis procedures and criteria while requiring only a small number of primitives that are easily mapped to existing specification languages.

The next section defines what we mean by semantic analysis of software requirements. In the rest of the paper we introduce the analysis model and provide a description of the analysis criteria we have developed. In this paper, we consider only analysis procedures that examine the behavioral description of the computer. However, our long range goal is to define procedures that include detailed consideration of the behavior of the entire system working together, i.e., a melding of system engineering and software engineering in order to model and to analyze the interface between the process and the computer. Also, only black-box requirements are considered, i.e., the behavior of the software is described only in terms of observable phenomena external to the software. Although requirements and design often become intertwined, 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.

2 Semantic Analysis of Requirements

A system is a set of components working together within a given environment to achieve some common purpose or objective. For the most part, the systems of interest in this paper are physical systems or processes. Besides the basic objective or function implemented by the process, these types of systems also may have constraints on their operating conditions. Constraints may be regarded as boundaries that define the range of conditions within which the system may operate. Constraints may arise from several sources including:

- quality considerations,
- physical limitations and equipment capacities (e.g, avoiding overload of equipment in order to reduce maintenance),
- process characteristics (e.g., limiting process variables to minimize production of byproducts), and
- safety (e.g., avoiding hazardous states).

Early in the development process, tradeoffs between goals in the functional description and constraints must be identified and resolved according to priorities assigned to each. Identifying these conflicts and resolving them is a major component of both the system and software requirements analysis process. The other component is ensuring that the behavior of the process-control system implements the function to be achieved by the system. Both of these together are necessary to determine whether the software is "correct" for process-control systems. Semantic analysis of the software requirements addresses both of these elements as it examines the correctness of the specification.

Software correctness can be separated into two aspects: subsystem correctness and system correctness. Subsystem correctness implies that the implementation or constructed

version of the subsystem satisfies the subsystem requirements. The most important property of the requirements specification with respect to achieving this is precision or lack of ambiguity. The behavior of the software must have been specified in sufficient detail 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 for the designers to distinguish between observably distinct behavioral patterns, then the specification is ambiguous. If the differences between two programs that satisfy the same set of requirements is not significant for a given application (i.e., will not affect the achievement of the system goals and the satisfaction of the constraints), the ambiguity 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. Ambiguity or imprecision in the software requirements specification can have a major impact on testing, formal verification, and reuse of the software. It can also affect the ability to determine system correctness.

System correctness implies that the subsystems working together, if they satisfy their requirements (i.e., the subsystems are correct), will implement the required system function while satisfying the constraints on how that function may be achieved. For most process-control systems, this must be accomplished under all possible conditions, i.e., the system must be *robust*. Note that the constraints may include acceptable failure behavior if it becomes impossible to continue to achieve the goals of the system. More specifically, the behavior of the control subsystem (in our case, the computer) is defined with respect to assumptions about the behavior of the other parts of the system, i.e., the conditions in the environment within which it operates. A robust system will detect and respond appropriately to violations of these assumptions. Therefore, the robustness of the software built from the specification depends upon the completeness of the specification of the environmental assumptions; there must be no observable events that leave the program's behavior indeterminate.

Documenting all environmental assumptions and checking them at run-time may seem expensive and unnecessary. Many assumptions 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 use. 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 degrees 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 software to check assumptions at run-time when the improper performance of the software may cause serious consequences. Examples abound of accidents resulting from incomplete requirements and non-robust software [Lev86, Neu85]. In one case, an accident occurred when a flight-control system was not programmed to handle a particular attitude of the aircraft [Neu85]. 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 software requirements specification must contain the information necessary to make possible the determination of both of these types of correctness — system and subsystem. The problem then becomes to determine the type and amount of information sufficient to ensure correctness, and the type of analysis that can be performed to ensure that this information is present and to detect missing or incorrect information. That is the topic of this paper, with special emphasis on procedures for ensuring robustness and lack of ambiguity.

3 Basic Model and Terminology

Traditionally process-control has been defined as:

> An arrangement of elements (such as amplifiers, converters, and human operators) interconnected in such a way as to maintain, or to affect in a prescribed manner, some physical quantity or condition of the process which forms a part of the system [Low71].

The scope of modern process-control is, however, not limited to maintaining and regulating the variables in a system but also includes supervision and planning by means of scheduling and sequencing events and operations in the system.

The operation of a system can be expressed as a function F relating system inputs ($\mathcal{I}_s$), outputs ($\mathcal{O}_s$), and time (t). This system function consists of the functional description and the set of constraints on the system. At any moment, there is a unique set of relationships between inputs and outputs whereby each output value will be related to the past and present values of the inputs and time. These relationships will involve fundamental chemical, thermal, mechanical, aerodynamic, or other laws as embodied within the nature and construction of the system. The system is constructed from components whose interaction implements F including, usually, a control component or components whose function is to ensure that F is correctly achieved.

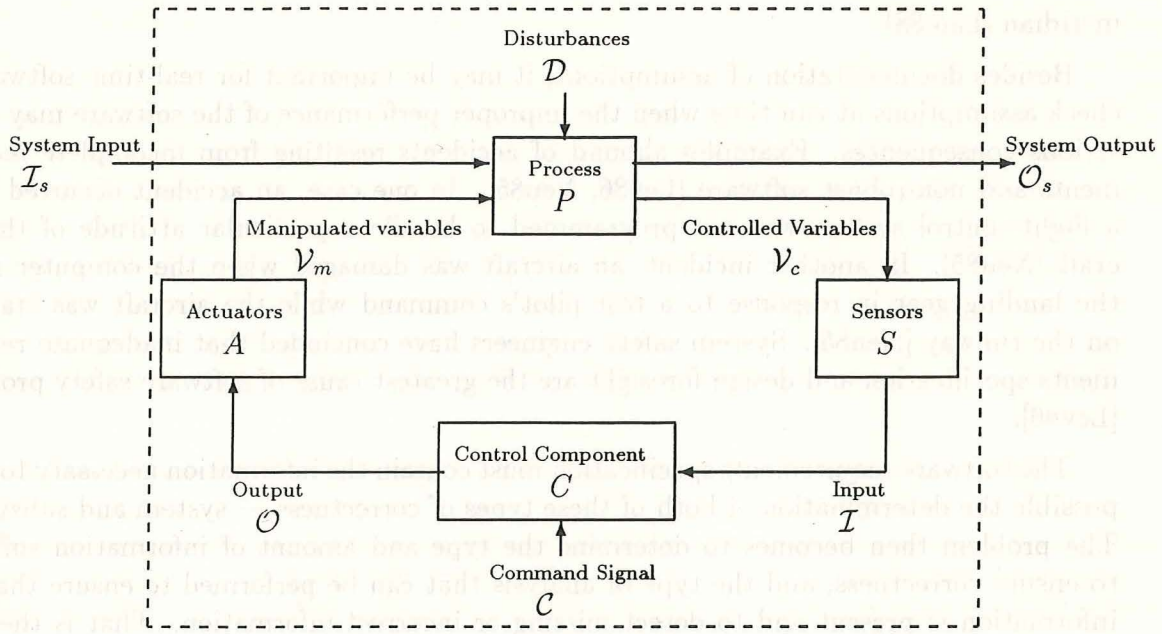


Figure 1: The control loop

A closed-loop process-control system can be modeled as four basic components within a *control loop* (figure 1): the process, the control component, actuators, and sensors.

The Process is the actual process to be controlled. The behavior of the process is controlled through the *manipulated* variables ($\mathcal{V}_m$) and the actual behavior is monitored through the *controlled* ($\mathcal{V}_c$) variables. The process can be described by the process function F_P, a mapping from $\mathcal{V}_m \times \mathcal{I}_s \times \mathcal{D} \times t \rightarrow \mathcal{O}_s \times \mathcal{V}_c$. Unfortunately, it is usually difficult to derive a mathematical model of the process due to the fact that most processes are highly nonlinear (i.e., the process characteristics depend on the level of operation), and, even at a constant operating level, the process characteristics change with time (i.e., the process is nonstationary). However, some of the process characteristics can be described or assumptions can be made about the characteristics, and these can be used to derive the control function. It is important to remember that any attempt to provide a mathematical expression describing the process involves simplifying assumptions and therefore will be imperfect although useful.

The Control Component is the implementation of the control function. The implementation can be either analog or digital. The functional behavior of this component is described by a control function (F_C) mapping $\mathcal{I} \times \mathcal{C} \times t \rightarrow \mathcal{O}$. The process may change state not only through internal conditions and through the manipulated variables, but also by disturbances ($\mathcal{D}$) that are not subject to adjustment and control by the control component. The general control problem is to adjust the manipulated variables so as to achieve the system goals despite disturbances. *Feedback* is

meridian [Lam88].

Besides documentation of assumptions, it may be important for real-time software to check assumptions at run-time when the improper performance of the software may cause serious consequences. Examples abound of accidents resulting from incomplete requirements and non-robust software [Lev86, Neu85]. In one case, an accident occurred when a flight-control system was not programmed to handle a particular attitude of the aircraft [Neu85]. 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 software requirements specification must contain the information necessary to make possible the determination of both of these types of correctness — system and subsystem. The problem then becomes to determine the type and amount of information sufficient to ensure correctness, and the type of analysis that can be performed to ensure that this information is present and to detect missing or incorrect information. That is the topic of this paper, with special emphasis on procedures for ensuring robustness and lack of ambiguity.

3 Basic Model and Terminology

Traditionally process-control has been defined as:

> An arrangement of elements (such as amplifiers, converters, and human operators) interconnected in such a way as to maintain, or to affect in a prescribed manner, some physical quantity or condition of the process which forms a part of the system [Low71].

The scope of modern process-control is, however, not limited to maintaining and regulating the variables in a system but also includes supervision and planning by means of scheduling and sequencing events and operations in the system.

The operation of a system can be expressed as a function F relating system inputs ($\mathcal{I}_s$), outputs ($\mathcal{O}_s$), and time (t). This system function consists of the functional description and the set of constraints on the system. At any moment, there is a unique set of relationships between inputs and outputs whereby each output value will be related to the past and present values of the inputs and time. These relationships will involve fundamental chemical, thermal, mechanical, aerodynamic, or other laws as embodied within the nature and construction of the system. The system is constructed from components whose interaction implements F including, usually, a control component or components whose function is to ensure that F is correctly achieved.

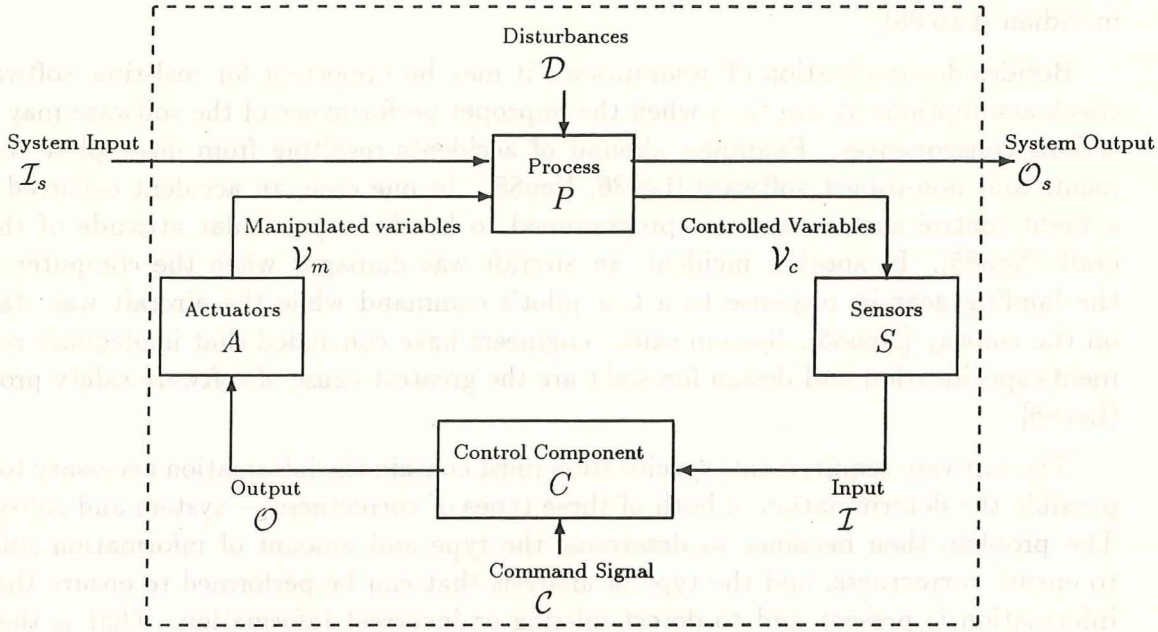


Figure 1: The control loop

A closed-loop process-control system can be modeled as four basic components within a *control loop* (figure 1): the process, the control component, actuators, and sensors.

The Process is the actual process to be controlled. The behavior of the process is controlled through the *manipulated* variables ($\mathcal{V}_m$) and the actual behavior is monitored through the *controlled* ($\mathcal{V}_c$) variables. The process can be described by the process function F_P, a mapping from $\mathcal{V}_m \times \mathcal{I}_s \times \mathcal{D} \times t \rightarrow \mathcal{O}_s \times \mathcal{V}_c$. Unfortunately, it is usually difficult to derive a mathematical model of the process due to the fact that most processes are highly nonlinear (i.e., the process characteristics depend on the level of operation), and, even at a constant operating level, the process characteristics change with time (i.e., the process is nonstationary). However, some of the process characteristics can be described or assumptions can be made about the characteristics, and these can be used to derive the control function. It is important to remember that any attempt to provide a mathematical expression describing the process involves simplifying assumptions and therefore will be imperfect although useful.

The Control Component is the implementation of the control function. The implementation can be either analog or digital. The functional behavior of this component is described by a control function (F_C) mapping $\mathcal{I} \times \mathcal{C} \times t \rightarrow \mathcal{O}$. The process may change state not only through internal conditions and through the manipulated variables, but also by disturbances ($\mathcal{D}$) that are not subject to adjustment and control by the control component. The general control problem is to adjust the manipulated variables so as to achieve the system goals despite disturbances. *Feedback* is

meridian [Lam88].

Besides documentation of assumptions, it may be important for real-time software to check assumptions at run-time when the improper performance of the software may cause serious consequences. Examples abound of accidents resulting from incomplete requirements and non-robust software [Lev86, Neu85]. In one case, an accident occurred when a flight-control system was not programmed to handle a particular attitude of the aircraft [Neu85]. 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 software requirements specification must contain the information necessary to make possible the determination of both of these types of correctness — system and subsystem. The problem then becomes to determine the type and amount of information sufficient to ensure correctness, and the type of analysis that can be performed to ensure that this information is present and to detect missing or incorrect information. That is the topic of this paper, with special emphasis on procedures for ensuring robustness and lack of ambiguity.

3 Basic Model and Terminology

Traditionally process-control has been defined as:

> An arrangement of elements (such as amplifiers, converters, and human operators) interconnected in such a way as to maintain, or to affect in a prescribed manner, some physical quantity or condition of the process which forms a part of the system [Low71].

The scope of modern process-control is, however, not limited to maintaining and regulating the variables in a system but also includes supervision and planning by means of scheduling and sequencing events and operations in the system.

The operation of a system can be expressed as a function F relating system inputs ($\mathcal{I}_s$), outputs ($\mathcal{O}_s$), and time (t). This system function consists of the functional description and the set of constraints on the system. At any moment, there is a unique set of relationships between inputs and outputs whereby each output value will be related to the past and present values of the inputs and time. These relationships will involve fundamental chemical, thermal, mechanical, aerodynamic, or other laws as embodied within the nature and construction of the system. The system is constructed from components whose interaction implements F including, usually, a control component or components whose function is to ensure that F is correctly achieved.

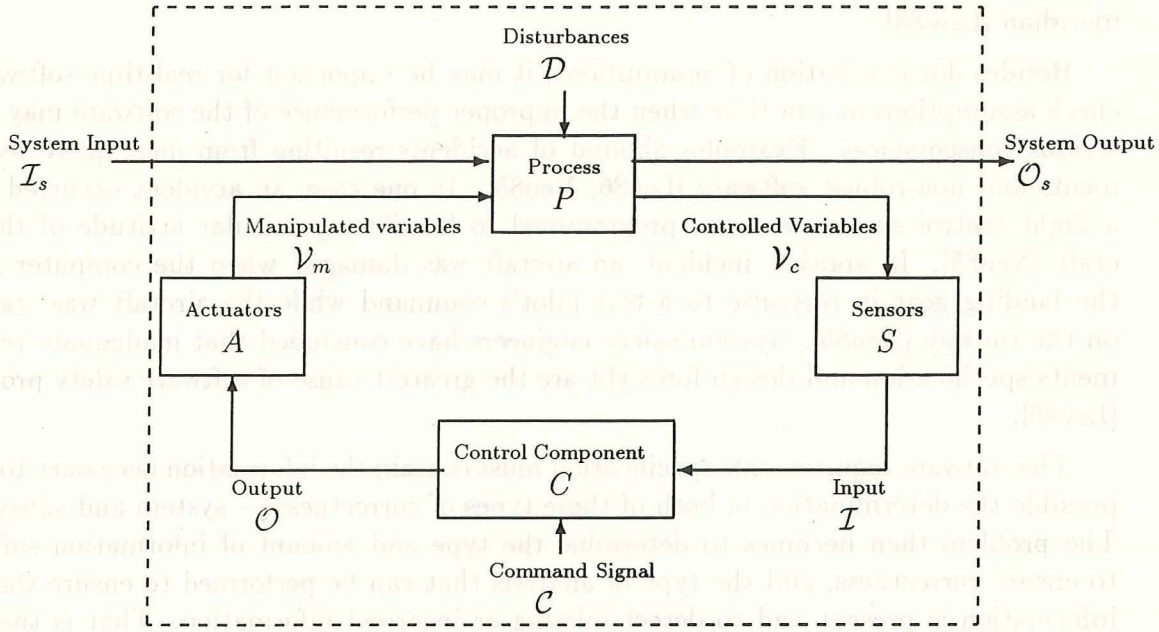


Figure 1: The control loop

A closed-loop process-control system can be modeled as four basic components within a *control loop* (figure 1): the process, the control component, actuators, and sensors.

The Process is the actual process to be controlled. The behavior of the process is controlled through the *manipulated* variables ($\mathcal{V}_m$) and the actual behavior is monitored through the *controlled* ($\mathcal{V}_c$) variables. The process can be described by the process function F_P, a mapping from $\mathcal{V}_m \times \mathcal{I}_s \times \mathcal{D} \times t \rightarrow \mathcal{O}_s \times \mathcal{V}_c$. Unfortunately, it is usually difficult to derive a mathematical model of the process due to the fact that most processes are highly nonlinear (i.e., the process characteristics depend on the level of operation), and, even at a constant operating level, the process characteristics change with time (i.e., the process is nonstationary). However, some of the process characteristics can be described or assumptions can be made about the characteristics, and these can be used to derive the control function. It is important to remember that any attempt to provide a mathematical expression describing the process involves simplifying assumptions and therefore will be imperfect although useful.

The Control Component is the implementation of the control function. The implementation can be either analog or digital. The functional behavior of this component is described by a control function (F_C) mapping $\mathcal{I} \times \mathcal{C} \times t \rightarrow \mathcal{O}$. The process may change state not only through internal conditions and through the manipulated variables, but also by disturbances ($\mathcal{D}$) that are not subject to adjustment and control by the control component. The general control problem is to adjust the manipulated variables so as to achieve the system goals despite disturbances. *Feedback* is

provided via the controlled variables in order to monitor the behavior of the process. This feedback information (along with external command signals $\mathcal{C}$) can be used as a foundation for future control decisions as well as an indicator of whether the changes in the process initiated by the control component have been achieved. Note that the control component may have only partial control over the process — state changes in the process may occur due to internal conditions in the process or because of external disturbances.

Actuators are devices designed to manipulate the behavior of the process, e.g. valves controlling the flow of a fluid. The actuators physically execute commands issued by the control component in order to change the manipulated variables. The functionality of the actuators is described by the actuator function F_A mapping $\mathcal{O} \times t \rightarrow \mathcal{V}_m$.

Sensors constantly monitor the actual behavior of the process. For example, a thermometer may measure the temperature of a solvent in a chemical process. Sensors provide the control component with measurements of the controlled variables. The sensor function F_S maps $\mathcal{V}_c \times t \rightarrow \mathcal{I}$.

This model describes a classic *feedback control system* where the control component monitors the process through controlled variables and makes adjustments accordingly. There are disadvantages with such a system. It does not take corrective action until the controlled variables already deviate from their desired values. Furthermore, any corrective action is not realized until the changing conditions have propagated around the control loop. Delays, called *time lags*, caused by each component may cause serious problems in the system.

A complimentary approach, *feedforward control*, measures and estimates disturbances and tries to predict their effect on the process. Corrective actions are taken before the controlled variables deviate from their desired values. Feedforward control cannot replace a feedback control, however. The effectiveness of a feedforward system is limited by the mathematical model describing the system and the accuracy of the components involved. Inaccuracies may cause the feedforward system to have a steady-state offset taking the system out of balance. Also, the feedforward control component cannot compensate for unexpected disturbances. Process-control systems often combine feedforward and feedback strategies in order to offset the disadvantages of each.

Process-control is a time-based concept. The objective is to maintain some property at some specified value in time or to effect some sequence of state changes in time. The sequence of steps to be taken may be dependent on both satisfaction of prerequisite conditions in the process or environment and on time.

Control actions will lag in their effects on the process because of delays in signal propagation around the control loop. An actuator may not respond immediately to an external command signal. Also the process may have delays in responding to manipulated variables. Sensors usually obtain values only at certain "sampling" intervals. Time lags also restrict

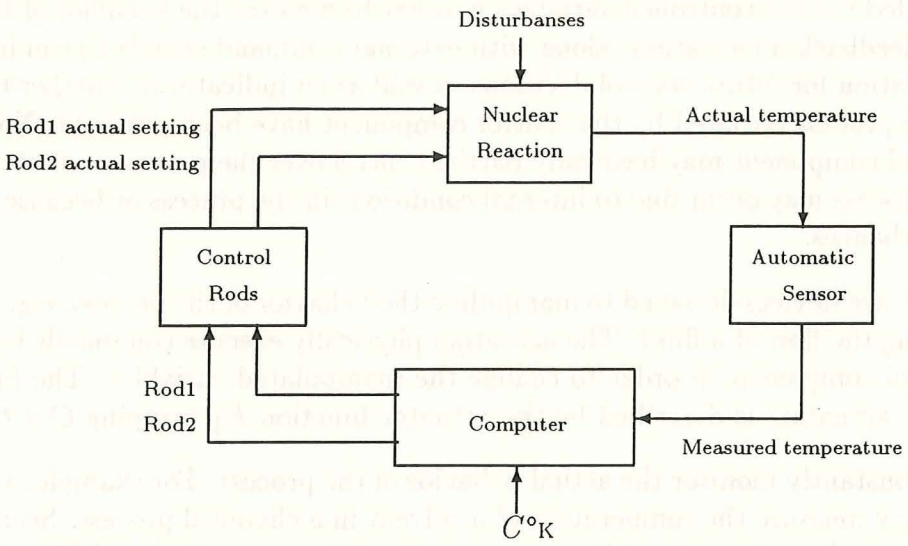


Figure 2: Block diagram of the temperature control system.

the speed and extent with which the effects of disturbances can be reduced. On the other hand, the time lags restrict the rapidity with which disturbances can upset the process, and there are instances where the presence of time lags makes some types of closed-loop control both possible and effective. Furthermore, time lags in some components may be so small in comparison with other components that their effects may not be crucial. In fact, if the sampling period of the sensors becomes small in relation to the time lags of the rest of the system, the sampled control system becomes, in effect, continuous. This is also true for actuators that operate in a discrete manner. Because many of the problems associated with computerized process-control stem from timing, a large amount of the analysis described in this paper involves timing.

3.1 An Example System

Figure 2 shows a process-control system that will be used throughout the paper to illustrate the concepts introduced. (The complete example is included in the appendix.) It is based on a set of requirements introduced in [JM87] as an example of safety-assertion analysis. The system controls the coolant temperature in a reactor tank by moving two independent control rods. The goal or basic required functionality is maintenance of the coolant at an externally defined temperature of C°K. The coolant temperature is measured by an automatic sensor and is provided to the control component when the temperature changes by c_r°K. If the temperature of the coolant cannot be changed, a complete shutdown is required. A safety constraint on the control function of the cooling system is that a rod cannot be moved within 30sec of its previous movement.

The following assumptions are made:

1. The rods have mechanical stops so that an attempt to move a rod past its limit will have no effect on the system.

2. Every rod movement will perturb a stable system enough to report a temperature change.

3. The goal will always be attainable; i.e. the possible movement of the rods is always enough to stabilize the temperature at the right value.

4. The process is dynamically stable; i.e., no runaway temperature changes will take place. There will never be a need to move the rods with shorter than 30sec intervals.

3.2 The Requirements State Machine

The goal of this paper is to describe semantic analysis criteria that can be applied to any behavioral requirements specification. In order to make these criteria independent of any specific, existing requirements language, we introduce a general behavioral model of the control function, called a requirements state machine (RSM), which is an abstraction of most state-based specification languages. Either requirements written in a state-based language could be translated into an RSM model and analyzed, or the criteria can be mapped onto a specific language and applied directly to the specification. In this paper, the RSM is used to model the control function F_C; properties of the process, sensor, and actuator functions, F_P, F_S, and F_A, respectively, are used to derive the semantic analysis criteria.

The RSM model we use is a Mealy machine ([HU79]) with outputs on the transitions between states. Transitions are labeled with logical expressions of the form *Input predicate*||*Output predicate*, and a transition is taken if the *Input predicate* on that transition evaluates to **true**. If an output is to be produced, the constraints on that output are expressed in the *Output predicate* associated with the transition.

The RSM is denoted as a seven-tuple $(\Sigma, Q, q_0, P_T, P_O, \delta, \gamma)$ where:

- Σ is the set of input/output variables, $\mathcal{I}$ and $\mathcal{O}$, used by the control component (software).
- Q is the finite set of states of the control component C.
- $q_0 \in Q$ is the initial state of the control component C; the software is in this state before startup.
- P_T is the set of boolean functions over Σ, which represent predicates on the values and timing of the inputs ($\mathcal{I}$) from the sensors. These predicates are called *trigger* predicates since they trigger a state change in the RSM.

- P_O is the set of boolean functions over Σ, which represent predicates on the outputs ($\mathcal{O}$) of the control component.

- δ is the state transition function mapping $Q \times P_T$ to Q. That is, $\delta(q, p)$ where $q \in Q$ and $p \in P_T$ defines the next state when the software is in state q and takes the transition having p as the input predicate.

- γ is the trigger-to-output relationship mapping from $Q \times P_T$ to P_O. That is, $\gamma(q, p)$ gives the predicate describing the output $\mathcal{O}$ to the actuators to be generated when the transition with input predicate p is taken out of state q.

In addition the RSM has the following properties:

- Predicates in P_T and P_O are expressed using the standard boolean operators and ordinary arithmetic operators. The expression $X\uparrow$ represents an input or output occurrence of X. This expression evaluates to **true** the moment input X arrives at the black-box boundary or output X is produced and presented at the black-box boundary.

- When an input I arrives at the software boundary, i.e. an $I\uparrow$ event has occurred, it is denoted as I_j or simply I. The previous occurrence of the same input will be denoted I_{j-1} and so forth. The ordering of outputs will be expressed in the same manner. Note that the first variable X arriving at the black-box boundary will be referred to as X_1. The ordering of inputs and outputs is externally visible.

- The predicates in P_T are either equivalent to the form *Input event*$\wedge$*Input Constraints* or they represent a timeout. Every input predicate that is not a timeout will contain only one *Input event*. For a timeout, the mere passage of time with no triggering input event will eventually result in a **true** predicate.

- A clock and a function giving the absolute time of an event are needed to express timing. The expression $t(I\uparrow)$ denotes the time when I arrives at the black-box boundary. The clock is started when the system receives the signal to startup, i.e. $t(Su\uparrow) = 0$.

- In certain instances the value of a variable is interesting . The value of a variable X is denoted $v(X)$.

To illustrate, consider the reactor coolant example. If no rods have been moved recently (within 30sec) and an input I indicating a temperature change occurs, the control tries to move Rod1. The rod will be moved up or down, depending on the value of I. After moving Rod1, the RSM will wait 30sec in state *Rod1Moved*. If no additional input arrives

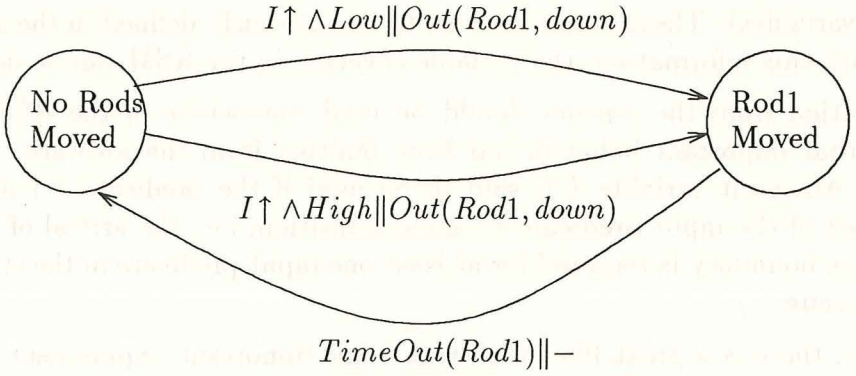


Figure 3: A fragment of an RSM

during this time, the RSM changes to *NoRodsMoved*. In figure 3 the predicates used will be abbreviated:

$$\begin{aligned} Low &= v(I) < C°K \\ High &= v(I) \geq C°K \\ TimeOut(X) &= t > t(X\uparrow) + 30 \\ Out(O, X) &= O\uparrow \wedge (v(O) = X) \end{aligned}$$

The transition predicate on the uppermost transition can be expressed informally as: If an input I arrives and the value of I is less than $C°$K then produce an output *Rod1* with the value *up*.

Semantic analysis of requirements involves procedures for examining each of the parts of the RSM within the context of the system function F in order to find flaws in the software requirements specification and to ensure that it (and the software built from it) has certain desirable properties. This paper describes some of the semantic analysis that might be performed. Unfortunately, it is usually not possible to specify a complete formal model of the system F. However, it is possible to define criteria that must be satisfied for all such systems and also heuristics that will aid in finding missing requirements, incorrect requirements, and ambiguous requirements when considered within the context of the particular goals and constraints of the system being developed. We define these criteria and heuristics in terms of the parts of the RSM, i.e., input/output variables (Σ), states (Q), trigger predicates (P_T), output predicates (P_O), the trigger-to-output relationship (γ), and transitions (δ), looking at each in turn.

4 Input/Output Variables

The input and output alphabet Σ represents all the types of information the sensors can provide to the software and the commands that the software can provide to the actuators

(manipulated variables). These variables need to be rigorously defined in the specification document. With this information, the variable coverage in the RSM can be determined.

All information from the sensors should be used somewhere in the RSM. If not, it is very likely that important behavior has been omitted from the software requirements specification. An input variable I is said to be used if the predicate $I\uparrow$ appears as a conjunctive part of the input predicate on some transition, i.e., the arrival of the variable at the black-box boundary is required for at least one input predicate in the state machine to evaluate to **true**.

Like inputs, there is a great likelihood that some important requirement for software behavior has been forgotten if there is some legal value for an output that is never produced. An output variable O is used if $O\uparrow$ appears in the predicate, i.e., the production of this variable is included in an output predicate on at least one transition. For example, if $v(O) \in \{up, down\}$ and the RSM specifies when to generate $v(O) = up$ but not when to generate $v(O) = down$, the specification is almost certainly incomplete.

5 States

The possible states of the control software requirements are elements of Q. These will of course be specific to the application. However, there are states, those involved in software startup and shutdown, that must be treated in the requirements specification of all systems. There are also some general characteristics of Q that may be examined.

5.1 Startup and Shutdown

There are two different startup situations: initial startup after complete process shutdown and startup after the software has been temporarily off-line. These present special problems that need attention in the requirements specification; many accidents and run-time failures stem from inadequacies in the software dealing with the transitions between normal processing and various types of partial or total shutdown.

In the initial startup or after temporary process shutdown, the clock (as well as other system and local variables) will need to be initialized, i.e., $t = 0$. In the second case where only the software has been shutdown, the variable and status of the process, including time, will probably have changed since the computer was last operational. Unlike other types of software, such as data processing systems, an important consideration when developing process-control is that they usually continue to change state even when the computer-controlled is not executing. There needs to be a specification of the responses to input that arrives before software startup and after software shutdown, or at least an assurance that this information can safely be ignored. Serious accidents have occurred 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.

There are a variety of potential solutions to this re-synchronization problem. Message serialization, for example, is a commonly used technique that will detect "lost" information and indicate potential discontinuities in software operations. Another technique often used involves checking elapsed time between apparently successive inputs by means of a self-contained time-stamp in each input (requiring clock synchronization) or via reference to a time-of-day clock upon the receipt of each input. Shutdown will be different depending on the application, but all shutdown states require that consideration be given to how to restart. Each method and each application will lead to unique requirements, but the need to ensure synchronization of data and status between the software and the external system elements is invariant.

If the hardware can retain a signal indicating the existence of an input I prior to software startup, the program has two startup states with respect to the given input (i.e., the input was present or not) and (at least) two separate requirements must be specified: one to deal with startup when there is indication of a prior input signal, one when there is not. Note that in the case of an input event I occurring before program startup, $t(I\uparrow)$ is unobservable by the software. Systems where the software time-stamps incoming data must include special requirements to handle this situation correctly at startup or errors can result. Furthermore, careful consideration must be given to the use of the value of I, i.e. $v(I)$, (for the pre-startup input event I), as it is hardware dependent which $v(I)$ is retained in the case (unobservable by the software) that there were multiple events I prior to program startup: Some hardware may retain the value from the first such event, some the most recent, etc.

Any specification for a real-time system should also include requirements to detect a possible disconnect between the computer and the environment that occurred prior to any program startup. 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. This is very similar to a maximum-time-between events condition (discussed below) but applies to the time after startup in the absence of even the first input of a given type. 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.

5.2 Modes

Some languages for requirements specification, e.g. [Hen80, Har87], have features that provide for specifying modes of operation (e.g., an aircraft is taking-off, in-transit, landing). These modes partition Q into subsets with common characteristics. Modes may be defined in terms of previous states of the process, current process characteristics, or possible future states.

The use of modes helps to simplify the description of required operation of the software. It also affects the analysis in that particular criteria or procedures may be applied differently when in different modes or may apply only to particular modes.

Modes generally allow a higher-level view of control flow in the system. It is possible to consider modes in the RSM, but it is not organized to do this efficiently; RSM is a general model suitable for use with languages with and without modes. However, it is useful to describe some of the analysis procedures with respect to particular general characteristics.

Besides general operational characteristics, Q may be partitioned according to certain desired, or undesired, properties of the system. One partitioning that is used in this paper groups states according to risk of leading to an accident or unacceptable event as defined by system engineers or clients. Risk is related to possible events or paths that could result from a particular state. In particular, when in a state q, there are a set of certain states, Q_q, that it is possible to reach. These may be partitioned into safe and hazardous subsets respectively: Q_{q_s} and Q_{q_h}. Q_{q_h} could even be ordered so that a further subset of hazardous states with minimum risk is defined, $Q_{MinRisk}$.

Various properties or modes may have complex interactions. For example, the appropriate action to be taken in the event that an unsafe state is reached may differ depending upon the current mode of operation (e.g., the aircraft is landing or is in level flight).

6 Trigger Predicates

There must always be a way for the RSM to leave every state if the system it describes is to be robust [2]. Formally, let P_{T_I} be the subset of the input predicates in P_T that have $I\uparrow$ as a conjunctive component. A first condition for robustness may then be stated:

$$\forall I, q \exists q_1, p : (\delta(q,p) = q_1) \wedge (p \in P_{T_I})$$

where $I \in \Sigma$ and q, $q_1 \in Q$. This is, however, not enough. In addition, the logical OR ($\vee$) of the input predicates on the transitions out of any state must form a tautology, i.e.:

$$\models \bigvee_i p_i$$

[2] This is not a sufficient condition for robustness — other conditions are discussed elsewhere in this paper — but it is a necessary condition.

where the p_is are the input predicates leading out of the state. That is, if there is a trigger condition for a state to handle inputs within a range, there needs to be some transition defined to handle data that is out-of-range. There must also be a requirement that specifies what to do if no input occurs at all, i.e., a timeout.

Another restriction on the input predicates is necessary to ensure deterministic behavior in the RSM. Consider the case of two transitions with the input predicates $I\uparrow \wedge v(I) > 0$ and $I\uparrow \wedge v(I) < 2$ respectively. If $v(I) = 1$, *both* predicates will evaluate to **true**, and *both* transitions should be taken. This leads to undesirable nondeterministic behavior of the RSM. The problem is eliminated by forcing all transitions out of any state to be disjoint. Formally, let p_i represent the input predicate on the ith transition out of a state. Then deterministic behavior is guaranteed by the requirement

$$\forall i \forall j (i \neq j) \Rightarrow \neg(p_i \wedge p_j).$$

Ensuring that the RSM satisfies the tautology requirements along with a transition involving a timeout ensures that exactly one predicate will always evaluate to **true** and that, therefore, there is always a transition out of every state. It does not ensure, however, that all assumptions about the environment have been adequately specified, i.e., that there is a defined response for all possible input conditions the environment can produce. This is dependent upon the amount and type of information (restrictions and assumptions) that is included in the triggers. In the rest of this section, we define the type of information needed in the trigger predicates for process-control systems.

At the black-box boundary to the computer, only time and value are observable to the software. Therefore, the predicates in P_T and P_O defining triggers and outputs must be defined only in terms of constants and the time and value of observable events or conditions.

6.1 Essential Value Assumptions

The existence of an input at the black-box boundary does not in itself require a value assumption. For example, a hard-wired hardware interrupt has no value; it may still trigger an output. For each input I, a value assumption is *essential* only if $v(I)$ is used in defining the value or time of some output O. 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 $p \in P_O$, 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. The completeness of value specification has been adequately covered elsewhere, e.g., [Cri84].

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. Since

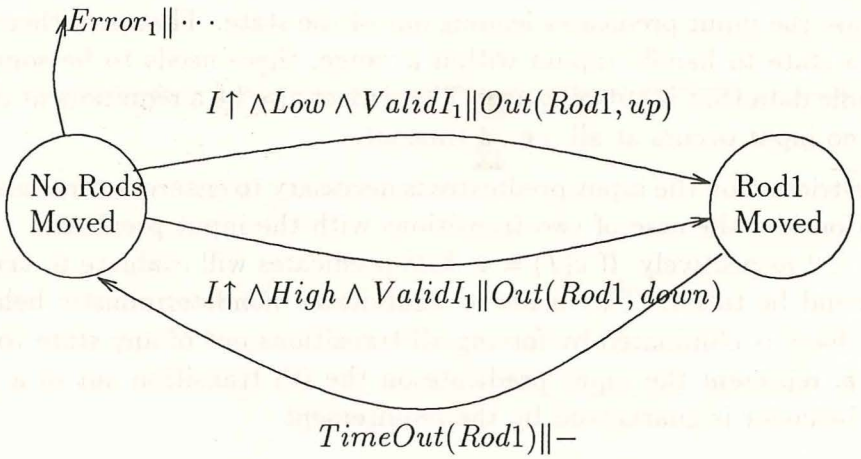


Figure 4: Value assumptions added.

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.

For example, consider the coolant in the reactor. Assume that physical limitations in the process ensure that the temperature of the coolant will never drop below 273°K or rise above 500°K. It is also clear from the example that a temperature reading I_i should differ from a previous reading I_{i-1} by c_r. i.e. $I_i = I_{i-1} \pm c_r$. Violations of these assumptions is an indication of a sensor failure. By adding the predicates

$$\begin{aligned} ValidI_1 &= (273 \leq v(I) \leq 500) \wedge (v(I_i) = v(I_{i-1}) \pm c_r) \\ Error_1 &= I\uparrow ((v(I) < 273) \vee (v(I) > 500) \vee (v(I_i) \neq v(I_{i-1}) \pm c_r)) \end{aligned}$$

the RSM is modified to check the required value assumptions (figure 4).

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].

6.2 Essential Timing Assumptions

Timing problems are one of the common causes of run-time failures in process-control systems, and timing is often inadequately specified. The need for and importance of specifying timing assumptions in the software requirements stems from the basic nature and importance of timing in process-control systems as described previously. Several different timing

assumptions are essential in the requirements specification of triggers: ranges, capacity, and load.

Time Ranges

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 is never sufficient — at the least, inputs must be required to arrive after program startup or handled as described above. For systems described using RSM, this basic timing assumption is explicitly defined by the initial state where the transition labeled with the startup event must be the only exit point. Besides this, a trigger specification must include either (1) the occurrence of an observable signal (or signals) or (2) the specification of a duration of time without a specific signal.

The arrival of an input at the black-box boundary has to include a lower bound on time and will, in general, include further timing constraints, i.e., an upper bound on the time interval in which the input is to be accepted. Requirements dealing with input arriving outside the time interval and the non-existence of an input during a given time interval, i.e., a duration of time without an expected signal, also have to be defined to ensure robustness.

The accepting interval will always be bounded from below by the time of the event that brought the machine to the current state. Some other lower bound may be desirable, but the limit must always be expressed in terms of previous, observable events.

In the rod-example it is assumed that two consecutive temperature readings will be separated by at least t_{min} seconds. By modifying *ValidI* and *Error* this timing constraint is included in the RSM.

$$\begin{aligned} ValidI_2 &= ValidI_1 \wedge (t(I_i\uparrow) \geq t(I_{i-1}\uparrow) + t_{min}) \\ Error_2 &= Error_1 \vee (t(I_i\uparrow) < t(I_{i-1}\uparrow) + t_{min}) \end{aligned}$$

Note that requirements such as $t(I\uparrow)$ =11:00am are ambiguous. The value of $t(I\uparrow)$ is the value of the reference clock observed "simultaneously" with the occurrence of I. Conceptually, the clock is ticking at the rate of one tick per unit of temporal precision. 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. 10:59am< $t(I\uparrow)$ <11:01am (commonly written as $t(I\uparrow)$ =11:00am ±1min), which specifies an accuracy of plus or minus a minute on the timing.

For requirements of the second type, i.e., those that involve the *non*-existence of a signal

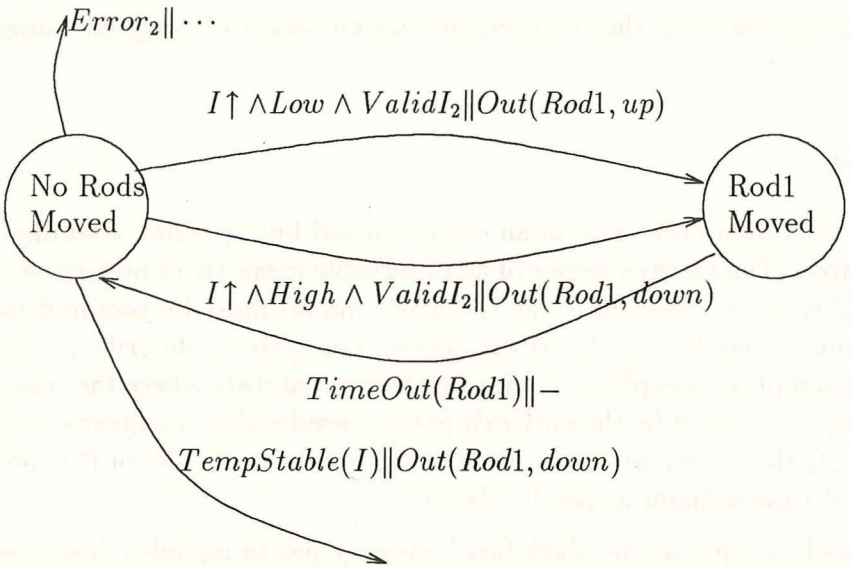


Figure 5: Complete requirement for state $NoRodsMoved$

during a given 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 be 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 ambiguous. 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 sec preceding or following event E. An example of a completely bounded interval is the requirement that when in state *NoRodsMoved*, an output *Rod1* is to be generated if t_{max} seconds elapse without the receipt of a temperature reading I. If

$$TempStable(X) = t > t(X\uparrow) + t_{max}$$

the complete requirement can be seen in figure 5 where $t(I\uparrow)$ provides the lower, observable bound of the interval and the duration of t_{max} seconds effectively sets the upper bound.

Capacity and Load

In an interrupt driven system, the count of unmasked input interrupts received over a given period of time partitions the software state space into at least two states: normal and overloaded. The required response to an input will differ in the two states; there must therefore be separate output assertions to deal with them. The term *capacity* will be used to refer to the count of inputs of a single homogeneous type, while *load* (to be discussed shortly) is the count of a set of heterogeneous input events.

Capacity. 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 be accommodated within a duration of time d. This is the requirement called capacity.

Specifications of capacity requirements are properly included in a black-box requirements specification since they stem from properties in the environment of the computer (i.e., properties of the process being controlled or of the sensors providing information about the process) and because the arrival rate of inputs is observable at the black-box boundary. In general, inputs to process-control 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 from the omission of behavior to deal with "excessive" load in an ATM system [Pur87]. When the central computer was unable to cope with the load, the ATMs dispensed cash whether there were adequate 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 damage as that which resulted from the lack of an explicit, black-box overload response behavior.

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 required; 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 in Section 8 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 $I\uparrow$ event in the production of an output O clearly requires

a capacity assumption, since a "lost" I (caused by interrupt lockout) is a violation of the requirement. If it is not the only input event in the trigger, its capacity may be dominated by some other event. Domination of I_1 by another input I_2 occurs when, for example, a transition predicate can be written as $\cdots \wedge (t(I_2\uparrow) < t(I_1\uparrow) < t(I_2\uparrow) + d) \| \cdots$. In this case, an interrupt for I_1 could potentially be disabled until the event $I_2\uparrow$ is detected, then enabled and left enabled until $I_1\uparrow$ 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.

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 the trigger predicates for all outputs that are capacity-dependent. Let n be the number of inputs of type I that have arrived at the black-box boundary, w_i be predefined weights allowing penalties if more than one input arrives during the time interval d, and $M_I(d)$ be the capacity for input I over the interval d. Then the trigger predicates may be expressed:

$$I\uparrow \wedge \left(\sum_{i=1}^{n} k_i < M_I(d)\right) \wedge \cdots \quad \text{where } k_i = \begin{cases} w_i & \text{if } t(I_i\uparrow) - t(I_{i-1}\uparrow) < d \\ 0 & \text{otherwise} \end{cases}$$

Load. Whereas capacity is defined in terms of one single type of input, load involves multiple input types. Assume that we have an ordering of all types of inputs from 1 to m, and let $I_i\uparrow$ symbolize the arrival of an input of type i. Then the jth arrival of this input can be written $I_{i_j}\uparrow$. The acceptable load when input I arrives at the black-box boundary can now formally be expressed as:

$$I\uparrow \wedge \left(\sum_{i=1}^{m} h_i \cdot \left(\sum_{j=1}^{n} k_j\right) < L(d)\right) \wedge \cdots \quad \text{where } k_j = \begin{cases} w_i & \text{if } t(I\uparrow) - t(I_{i_j}\uparrow) < d \\ 0 & \text{otherwise} \end{cases}$$

and where n is a count of inputs of type i that have arrived, w_i is defined as in the capacity definition, the h_is are weights that allow some inputs to be specified as more expensive than others, and $L(d)$ is the required load or capacity limit for a period of time with duration d.

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

The smallest time 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 $I\uparrow$ events within the interval of duration d preceding each such event — where n/d is the assumed minimum arrival rate — then no more than time d can elapse between any two occurrences of $I\uparrow$ or the minimum arrival rate assumption would be violated. For process-control 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 process is unlikely. Robust software should have the capability to query its environment with regard to inactivity over a given communication path.

Although inputs from human operators or other, slow, system components may be normally incapable of overloading a program, various malfunctions can cause excessive, spurious inputs to be generated. Robustness requires consideration of that case and specification of a capacity limit for such inputs as a means of detecting possible external malfunctions. In one serious accident, an aircraft went out of control and crashed 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 [FM84].

7 Output Predicates

As with inputs, the complete specification of the behavior of an output O requires both its time $t(O\uparrow)$ and its value $v(O)$. Again note that the time is required. There is no limit to the complexity of timing specifications for outputs, but, at the least, specification of bounds and minimum and maximum time between outputs is required as it is for inputs. Besides these, there are also some special requirements for the specification of predicates on the outputs: environment capacity, data age, and latency.

Environmental Capacity Considerations. The rate at which the sensors are producing data and sending it to the computer is the concern in input capacity. Output capacity, on the other hand, defines the rate the actuators can accept and react to data produced by the software. If the sensors can generate $M_I(d)$ inputs of type I during the time period of duration d, but the output environments can only "absorb" or process a lower number of outputs, an output overload might occur.

Output capacity limitations often stem from the constraints on behavior of the control system. They may be required because of physical limitations in the actuators (e.g., a valve can only perform a limited number of adjustments per second), constraints on process behavior (excessive wear on actuators might increase maintenance costs), or safety considerations (e.g., a restriction on how often a catalyst can be added safely to a chemical process).

Differences in input capacity and output capacity result in the need to handle three

cases:

1. The input and output rates are both within limits and the "normal" response can be generated.
2. The input rate is within limit but the output rate would be exceeded if a normally timed output were produced, in which case some sort of special actions are required.
3. The input rate is excessive, in which case some abnormal response is necessary (graceful degradation).

When input and output capacities differ, there must be multiple periods for which discrete capacity assumptions are specified. For the largest interval in which both input and output capacities are assumed and specified, the absorption rate of the output environment must equal or exceed the input arrival rate or the program might never catch up; but over short durations, the program can buffer or shield the output environment from excessive outputs. Contingency action must be specified for cases where these assumptions do not hold.

Data Age. Another important aspect of the specification of output 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, i.e., the process that the computer is controlling, is unlikely to be. Control decisions have to be based on data from the current state of the system, not on obsolete information.

Data obsolescence considerations require that all output events are properly bounded in time:

$$\cdots \| O\uparrow \wedge t(I\uparrow) + D_V > t(O\uparrow) \wedge v(O) = \cdots$$

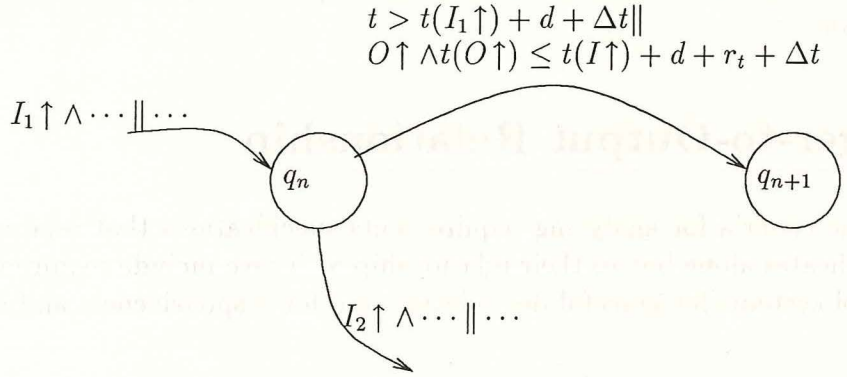
where D_V is the data validity factor or age limit for the input I triggering O. The input is only valid for the output O if it occurred within the preceding period of time of duration D_V. As an example of the possible implementation implications of such a requirement, MARS, a distributed fault-tolerant system for real-time applications, includes a validity time for every message in the system after which the information in the message is discarded [KM85, KD87].

Frola and Miller[FM84] report on an accident related to and perhaps caused by lack of specification of a data age factor. 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 door 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.

Latency. 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 latency interval, Δt, 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.

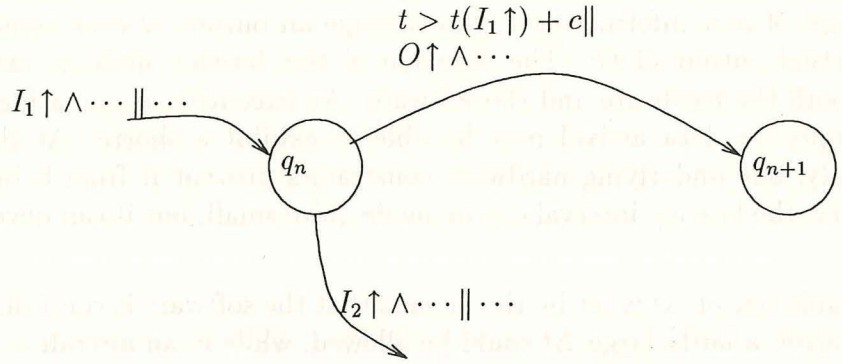
The acceptable size of Δt is set by the process that the software is controlling. In chemical process-control, a fairly large Δt could be allowed, while in an aircraft a much smaller value of Δt may be required. The choice of operating system, interrupt logic, scheduling priority and/or system design parameters will be influenced by the latency value. Also, behavioral analysis of the requirements to determine consistency with process functional requirements and constraints may not be correct unless the value of this behavioral parameter is known and specified for the software. Therefore, the requirements must include the allowable latency factor.

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_2 has been received within the previous time period of duration d. This could be specified as



The use of an interval of time without some event $I\uparrow$ to trigger an output always requires the specification of a Δt factor between the end of the interval and the occurrence of the output. Where the upper bound on the interval is a simple event, i.e., the 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 is application-dependent. For example:



If the intent is 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. Because software may be re-used in environments where the current intentions may differ from what was actually implemented, safety and other considerations dictate that the latency factor always be included in the specification when the non-existence interval's upper bound is not a simple observable event.

There may need to be additional transitions in the RSM 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.

8 Trigger-to-Output Relationship

There are some criteria for analyzing requirements specifications that relate not to input or output predicates alone but to their relationship γ. These include requirements in most process-control systems for graceful degradation and for responsiveness and spontaneity.

8.1 Graceful Degradation.

The requirements needed 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 of the software or, in extreme cases, to shutdown the computer and/or the process.

The first three cases are handled in an obvious way. The fourth case, commonly called performance degradation, should be graceful, i.e., predictable and non-abrupt. Graceful degradation may be specified by including the load in the timing or accuracy factors for the output. Assume the observed load, L, (during the interval of duration d immediately preceding the input I) is defined as:

$$L \equiv \sum_{i=1}^{m} h_i \cdot \left(\sum_{j=1}^{n} k_j \right) \quad \text{where } k_j = \begin{cases} w_i & \text{if } t(I\uparrow) - t(I_{i_j}\uparrow) < d \\ 0 & \text{otherwise} \end{cases}$$

and where n, w_i, and h_i are defined as in the definition of load given previously. Then a gracefully degrading predicate for O can be written as:

$$\begin{aligned} O\uparrow \quad & \wedge[v(O) = x \pm f_a(L)] \\ & \wedge[(t(I\uparrow) + f_{low}(L)) < t(O\uparrow) < (t(I\uparrow) + c + f_{high}())] \end{aligned} \tag{1}$$

where f_a defines the accuracy of O and f_{low} and f_{high} defines the limits on the response time. f_{high} and f_{low} are continuous, monotonically-increasing functions of load, $f_{low}(x) \leq f_{high}(x)$ for all x, and c is a constant. If f_{high} is a faster growing function than f_{low}, the time window in which an output can legally be produced will grow and the predictability of the response time will decrease (figure 6a). Two inputs occurring quite close to one another in time could then legally trigger outputs having widely different response times, potentially even appearing in the reverse order from the order in which their respective triggers arrived.

For safety-critical systems, abrupt degradation (figure 6c) 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 6b).

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 to a "degraded performance" state wherein some observable action should usually be taken such as alerting a human operator, disabling or requesting resets of busy interfaces, recording critical parameters for subsequent analysis, 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; the exact same set of circumstances that caused it to leave may still exist. For example, let the event that caused the state

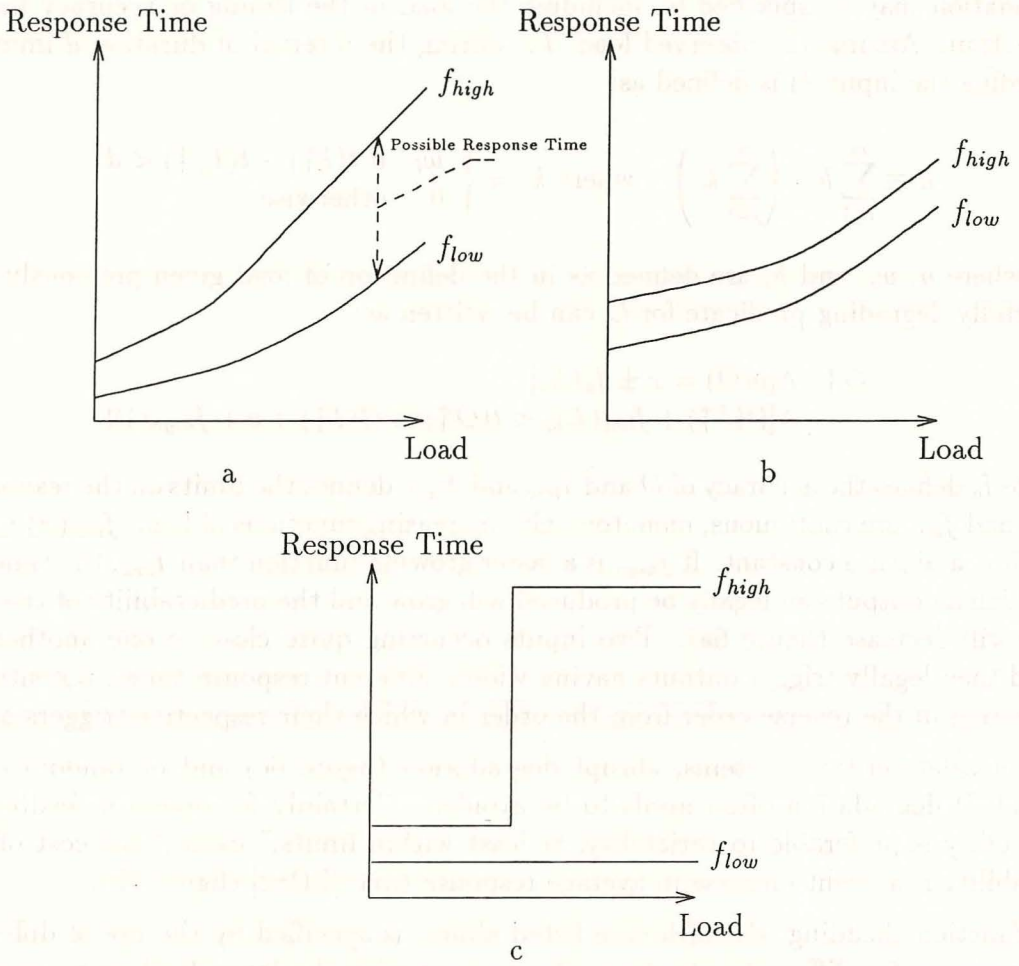


Figure 6: **a.** f_{high} faster growing than f_{low}. **b.** $f_{high} = f_{low}$, i.e. predictability is maintained. **c.** Abrupt degradation of f_{high}.

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 $x \ll d$, 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 transition to return to normal operation is triggered by an input I_{rn}, the transition taking the RSM from an overloaded state to normal operation should include a hysteresis factor assuring that the return signal is not too close in time to the input I_{ol} that caused the overload. The input predicate on the transition could be written as:

$$I_{rn}\uparrow \wedge (t(I_{rn}\uparrow) - t(I_{ol}\uparrow) > h_d) \wedge \cdots$$

where h_d is the hysteresis delay required before the transition can be taken.

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 these need to be considered when developing the requirements.

8.2 Responsiveness and Spontaneity

Responsiveness and spontaneity deal with the actual behavior of the controlled process and how it reacts (or does not react) to output produced by the controller. In particular, does a given output O cause the process to change, and, if so, is that change detectable by means of some input I. Basic process-control models include feedback to provide information to the control component about changes in state caused by disturbances or about expected responses caused by changes to the manipulated variables. This is a basic property of almost all process-control systems: If feedback information is not used by the software, the requirements specification is probably deficient. That is, basic feedback loops need to be included in the software requirements and missing feedback loops provide clues as to deficiencies in the requirements specification.

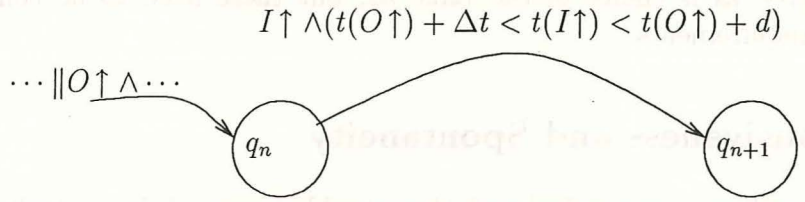
As an example, an accident occurred when a steel plant furnace was returned to production after being shut down for repair [Bah88]. A power supply had burned out in a digital thermometer during power-up so that the thermometer continually registered a low constant temperature. The controller, knowing 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 been. After four hours, the furnace had burned itself out, and major repairs were required.

A situation like this one could easily have been avoided if the information about the characteristics of the process was used as a *predictor* to forecast its expected behavior of the system. In this case the only knowledge needed to avoid the accident was that the temperature should increase if the burners are on.

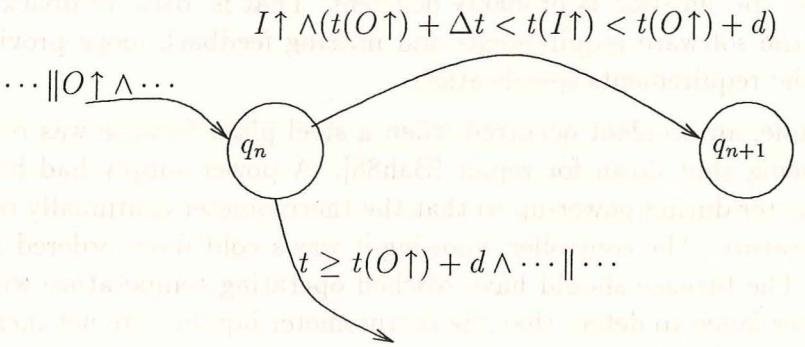
The information about the expected response of the process is to be found in the process function F_P. If the process does not respond to an output as expected and within some expected period of time, there is presumably something wrong and the software should be required to act accordingly — perhaps by trying a different output, by alerting a human operator, or at the least, by logging the abnormality for future, off-line analysis.

It is desirable to design a process-control system such that the effects of every output O, affecting a manipulated variable in the system, can be detected by some input I provided by the feedback loop. The situation is, however, not always that simple. Disturbances interfering with the process can cause changes that are not initiated by the software or can inhibit desired changes that the software has commanded.

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.,



and the requirement, 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.,



where Δt represents the latency period, i.e., the time between the receipt of an input and its processing (as defined previously). If the environment responds *too* quickly, one suspects coincidence rather than appropriate stimulus-response behavior. Most processes do not react instantaneously but only after a delay (time lag). A value-based handshake protocol can be used to eliminate the need for the latency 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.

9 Transitions

Requirements may involve looking not only at the triggers and outputs associated with each state, but also at paths between the states. In particular, some analyses rely on the guarantee that certain states will be reachable.

If a path exists between states q_1 and q_2, it can be uniquely defined by a sequence of input predicates. Formally, the transition function δ is extended to apply to sequences of input predicates. Define $\hat{\delta}$ as a mapping $Q \times P_T^* \mapsto Q$.

$$\begin{aligned}&\text{Let } q \in Q,\ p \in P_T, \text{ and } s \in P_T^* \text{ then}\\ &\hat{\delta}(q, \lambda) \ = q\\ &\hat{\delta}(q, sp) = \delta(\hat{\delta}(q, s), p)\end{aligned}$$

where λ is the null predicate. The existence of an input predicate sequence s from q_1 to q_2 (*path*) is formally expressed as

$$\exists s : \hat{\delta}(q_1, s) = q_2$$

The analysis in this section is based on examination of the paths between states of interest and the predicate sequences describing these paths.

9.1 Basic Reachability

A state q_m is said to be *reachable* from state q_n if there exists a path from q_n to q_m and the logical AND ($\wedge$) of the predicates in the instantiated predicate sequence s_i corresponding to that path does not result in a contradiction. Formally, let $p \in P_T$ and $s \in P_T^*$. Define ϕ as:

$$\begin{aligned}&\phi(\lambda) = \textbf{true}\\ &\phi(sp) = \phi(s) \wedge p\end{aligned}$$

State q_m is reachable from state q_n iff

$$\exists s : (\hat{\delta}(q_n, s) = q_m) \wedge (\phi(s_i))$$

In the RSM, all states have to be reachable from the initial state q_0. This requirement may be stated:

$$\forall q \exists s : (\hat{\delta}(q_0, s) = q) \wedge (\phi(s_i)).$$

Most state-based models include techniques for reachability analysis.

If a state is unreachable there are two possibilities: Either the state has no function and can be eliminated from the specification, or the state should be reachable and the requirements document must be modified accordingly.

9.2 Recurrent Behavior

Most process-control software is cyclic in nature, i.e., it is not designed to terminate under normal operation. The purpose of the software is to control and monitor a physical environment and the nature of the problem usually calls for software to perform continuously one single task, to alternate between a finite set of distinct tasks, or to perform continuously a sequence of tasks while in a given mode. Most systems, however, do include some states with non-cyclic behavior such as temporary or permanent shutdown states or states where the software changes to a different operating mode.

It is important to analyze the RSM to assure that desired behavior is repeatable. For instance, in the control rod example it is essential to be able to issue a sequence of *Rod1* commands. Since the number of states in the RSM is finite, at least one state issuing a *Rod1* command has to be included in every cycle or the RSM will eventually reach a state where Rod1 cannot be moved again, and the temperature cannot be controlled as required. Formally, state q is part of a cycle iff

$$\exists s : (\hat{\delta}(q, s) = q) \wedge (\phi(s_i)) \wedge (s \neq \lambda)$$

Assuring that a state is included in cyclic behavior will in many cases be insufficient; the nature of the path describing the cycle may also need to be examined. Consider an output to start a piece of equipment. It may be necessary to start the equipment more than once, i.e. the state q_{start} where the *start* command is issued has to be in a cycle. It may, however, be harmful to the equipment if two *start* commands are issued without an intermediary *stop* command. Therefore, every cycle including q_{start} also has to include a state q_{stop}. Formally

$$\begin{aligned}\forall s : ((\hat{\delta}(q_{start}, s) = q_{start}) \wedge (\phi(s_i)) \wedge (s \neq \lambda)) \Rightarrow \\ \exists s_1, s_2 : (((\hat{\delta}(q_{start}, s_1) = q_{stop}) \wedge (\phi(s_{1_i})) \wedge \\ (\hat{\delta}(q_{stop}, s_2) = q_{start}) \wedge (\phi(s_{2_i})) \wedge (s = s_1 s_2)))\end{aligned}$$

The analysis of the recurrent behavior in the RSM can easily be generalized to cover arbitrarily complex sequences of events.

9.3 Reversibility.

In a process-control system, there are frequently cases where a command issued to an actuator can be canceled or reversed by some other command or combination of commands. This capability is referred to as reversibility. Outputs should be reviewed and classified as to their reversibility. If an *on* command is to be reversible, then a state where the canceling *off* command is issued must be reachable from the state in which the *on* command was issued. Otherwise, the command cannot be reversed.

As another example, an alert condition to an operator (such as a below-minimum-safe-altitude warning to an air traffic controller) should be reversible when the condition no longer holds (e.g., the aircraft is now at a safe altitude). There will usually be several different classes of the reversing outputs. For example, the appropriate reversing output may depend 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].

9.4 Reachability of Safe States

We define an unsafe state as one with an unacceptable level of risk of leading to an accident. In the early conceptual stages of the development of a safety critical system, system safety engineers usually perform a preliminary hazard analysis to identify hazards (unsafe states) and categorize them as to risk level. Further analysis can be performed to translate system hazards into software hazards, i.e., software outputs that would put the system into a hazardous state given certain environmental conditions.

It is not always possible to enforce a requirement that the system can reach no hazardous states — in some systems, temporarily being in an elevated risk state is unavoidable. However, the system should never be in an *undesired* hazardous state. Safe states may have limited or no functionality, but they all have the property that they have acceptable risk of leading to an accident.

One simple definition of a safety policy, for which the specification could be checked, is the following:

- The computer never initiates a control action (command) that will move the process from a safe to an unsafe state. Let Q_s and Q_h represent the set of all safe states and the set of all hazardous states respectively. Then

$$\forall q_s, q_h \neg \exists s : (\hat{\delta}(q_s, s) = q_h) \wedge (\phi(s_i))$$

where $q_s \in Q_s$ and $q_h \in Q_h$.

- Given that the system gets into an unsafe state (by a failure of a component, including a computer error, or by a transformation that is not initiated by the computer, e.g. human error or environmental disturbances or stress), then the computer-controller will transform the hazardous state into a safe state, i.e., every path from a hazardous state leads to a safe state. That is,

$$\forall q_h, s\ [(\hat{\delta}(q_h, s) = q) \wedge (\phi(s_i)) \Rightarrow (q \in Q_s)].$$

  There may be several possible safe states, depending upon the type of hazard or on conditions in the environment. For example, the action to be taken if there is a failure in a flight-control system may depend on whether the aircraft is in level flight or if it is landing.

It may not be possible to build a safe system, i.e., it may not be possible to get from every hazardous state to a safe state. In that event, the system must be redesigned, abandoned, or a level of risk accepted. This risk can be minimized by providing procedures to minimize the probability of the hazardous state leading to an accident or to minimize the effects of an accident. Then a third criteria for safety is:

- If a system gets into a hazardous state and there is no possible path to a safe state, then the computer will transform the state into one with the minimum risk possible given the hazard and the environmental conditions. Formally,

$$\forall q_h\{[\neg\exists s, q_s(\hat{\delta}(q_h, s) = q_s) \wedge (\phi(s_i))] \Rightarrow [\forall s, q(\hat{\delta}(q_h, s) = q) \wedge (\phi(s_i)) \rightarrow q \in Q_{MinRisk}]\}.$$

9.5 Path Robustness

For most safety-critical, process-control software, there are concerns in addition to pure reachability. Even if a state fulfills all reachability requirements, there is still the question of the *robustness* of the path, or paths, affecting this particular state.

Consider an output O such that $v(O) \in \{up, down\}$. Suppose that every possible path from a state q_{up} with $v(O) = up$ to any state q_{down} that sets $v(O) = down$ includes the arrival of input I in at least one input predicate. Then if the software's ability to receive I is ever lost (e.g., through sensor failure), there are circumstances under which it will not be able to set $v(O) = down$. Thus, the loss of the ability to receive I can be said to be a **soft-failure mode** since it *could* inhibit the software from setting $v(O) = down$. Formally, the inability to receive $I\uparrow$ is a soft-failure mode iff

$$\exists q_{up} \forall q_{down}, s[(\hat{\delta}(q_{up}, s) = q_{down}) \Rightarrow (\neg\phi(s_i) \vee I\uparrow)]$$

If the predicate $I\uparrow$ occurs in every path expression from *all* states in which $v(O) = up$ to *all* states that set $v(O) = down$, the loss of the ability to receive I is now said to be a **hard-failure mode** since it *will* inhibit the software from producing a *down* command. Lacking ability to receive $I\uparrow$ is a hard-failure mode iff

$$\forall q_{up} \forall q_{down}, s[(\hat{\delta}(q_{up}, s) = q_{down}) \Rightarrow (\neg\phi(s_i) \vee I\uparrow)]$$

The more failure modes the RSM has, whether soft or hard, the less robust with respect to external disturbances will be the software that is correctly built to that specification.

9.6 Constraint Analysis

In addition to ensuring that the basic goals of F_C are implemented correctly in the software, an evaluation of the consistency of the requirements with the system constraints is necessary. Conflicts may exist — these should be detected and tradeoffs to resolve them evaluated.

As an example, path robustness requirements may conflict with safety constraints. Consider the following conflicting requirements. 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. This conflicts with the requirement that 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 state-machine analysis procedures to provide the information necessary to detect and resolve some of these types of conflicts.

The type of analysis required to guarantee consistency with the system constraints will depend upon the type of constraints involved. The presence of constraints can potentially affect most of the criteria and analysis methods mentioned in this paper. Some constraints can be ensured using the criteria already described. Others require additional analysis. For example, procedures for ensuring that only safe states are reachable have been described. In addition, basic reachability analysis must be extended. Even though a path predicate does not form a contradiction, it might be infeasible or undesirable when viewed in conjunction with the behavior of the rest of the system and with the constraints. Therefore, reachability analysis needs to be performed considering not only the basic path predicates, but also constraints on the sequence of events.

To illustrate, consider the constraint to be satisfied in our example, i.e., a rod is not allowed to move within 30 seconds of its previous movement. Formally, let

$$\begin{aligned} Constraint = &(t(Rod1_i\uparrow) \geq t(Rod1_{i-1}\uparrow) + 30sec) \wedge \\ &(t(Rod2_j\uparrow) \geq t(Rod2_{j-1}\uparrow) + 30sec) \end{aligned}$$

where i and j are integers. To guarantee this property, all paths taking the RSM from one occurrence of a rod command to another must be consistent with this constraint. If

Q_{Rod1} is defined as the set of all states wherein a *Rod1* command can be issued, and Q_{Rod2} similarly, then consistency with *Constraint* is expressed as:

$$\forall s : [(\hat{\delta}(q_1, s) = q_2) \Rightarrow (\neg\phi(s_i) \vee Constraint)] \wedge$$
$$\forall s : [(\hat{\delta}(q_3, s) = q_4) \Rightarrow (\neg\phi(s_i) \vee Constraint)]$$

where $q_1, q_2 \in Q_{Rod1}$ and $q_3, q_4 \in Q_{Rod2}$.

To analyze with respect to *Rod1*, the first step is to identify the transitions where a *Rod1* command can be issued. Path analysis can then be applied to find the sequences of events that will make the software issue two consecutive *Rod1* commands. By showing that all possible paths described by these sequences will take at least 30 seconds to traverse, it is guaranteed that the constraint on *Rod1* will be satisfied. In the example there are two possible shortest-time sequences separating two *Rod1* commands, $s_1 = \{TimeOut(Rod1), TempHigh\}$ and $s_2 = \{TempStable(Rod1)\}$. s_1 is guaranteed to take at least 30 seconds since it contains the predicate $t > t(Rod1 \uparrow) + 30$ as a conjunctive trigger. s_2 consists of the single predicate $t > t(Rod1 \uparrow) + t_{max}$ which will guarantee consistency with the constraint **if** $t_{max} \geq 30sec.$ Thus consistency with the system constraint on *Rod1* requires the assumption $t_{max} \geq 30sec.$ The consistency for *Rod2* is handled similarly.

10 Conclusions

The software requirements specification must contain sufficient information to ensure system and subsystem correctness. This paper has presented the type and amount of information necessary to ensure system and subsystem correctness in process-control systems along with the types of analysis that can be performed to ensure that this information is present and to detect missing or incorrect information. Emphasis has been placed on procedures for ensuring robustness and lack of ambiguity. Our goal is to prevent or detect errors early in the software development cycle.

In real systems, requirements are often not complete before software development begins. Furthermore, changes are often made as the design of the other parts of the system becomes more detailed and problems are found necessitating changes in the desired software behavior. It is therefore unlikely that the analysis will be completed before software design begins. To avoid costly redesign and recoding, the requirements specification and analysis should be as complete as possible as early as possible. But realistically, much of the analysis may need to be put off or redone as the software and system development proceeds. Our procedures can easily be repeated to account for changes or left until incompletely specified aspects of the requirements are completed.

The RSM, as defined in this paper, models only the behavior of the control component C. Much useful analysis can be performed on this model, but our long-term goal is to extend the RSM to model the interface between the control component and the process,

i.e, the sensors S and the actuators A along with some aspects of the process P. This will extend the amount of analysis that is possible including many of the traditional types of analysis performed by system engineers such as failure modes and effects analysis (FMEA), potentially bridging some of the gap between system engineering and software engineering.

References

[Alf77] M.W. Alford. A requirements engineering methodology for real-time processing requirements. *IEEE Transactions on Software Engineering*, SE-3(1):60–68, Jan 1977.

[Bah88] D. 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, March 9, 1988.

[BCF83] R.M. Balzer, D. Cohen, M.S. Feather, N.M. Goldman, W. Swartout, and D.S. Wile. Operational specification as the basis for specification validation. In D. Ferrari, M. Bolognani, and J. Goguen, editors, *Theory and Practice of Software Technology*, pages 21–49. North-Holland Publishing Company, 1983.

[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.

[Boe81] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[Cri84] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, SE-10(2):163–174, March 1984.

[End75] A. Endres. An analysis of errors and their causes in system programs. *IEEE Transaction on Software Engineering*, SE-1(6):140–149, June 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., January 1984.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hen80] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.

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

[JM87] F. Jahanian and A.K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36:961–975, August 1987.

[KD87] H. Kopetz and A. Damm. MARS: Concepts and Design of the Second Prototype. Technical Report Nr. 4/87, Technical University of Vienna, Gusshausstrasse 30, A-1040 Wien, Austria, January 1987.

[Kle88] T. Kletz. Wise after the event. *Control and Instrumentation*, 20(10), October 1988.

[KM85] H. Kopetz and W. Merker. The architecture of MARS. In *International Symposium on Fault Tolerant Computing Systems*, pages 274–279, June 1985.

[Lam88] J. Lamb. The everyday risks of playing it safe. *New Scientist*, September 8, 1988.

[Lev86] N.G. Leveson. Software safety: What, why, and how. *ACM Computing Surveys*, 18(2):125–164, June 1986.

[Low71] E. I. Lowe. *Computer Control in Process Industries.* Peter Peregrinus Ltd., London, 1971.

[LS87] N.G. Leveson and J.L. Stolzy. Safety analysis using petri nets. *IEEE Transaction on Software Engineering*, SE-13(3):386–397, March 1987.

[Neu85] P.G. Neumann. Some computer-related disasters and other egregious horrors. *ACM Software Engineering Notes*, 10(1):6–7, January 1985.

[NYT86] New York Times. Science Times Section, Page C1, July 29, 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, June 18, 1987.

[Rea89] Reasoning Systems Inc., Palo Alto, CA. *Refine User's Guide*, June 1989.

[Zav82] P. Zave. An operational approach to requirements specifications for embedded systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–269, May 1982.

A The Rod-Example

Variables :

Input : I =int 273..500. The temperature reading sent by the automatic sensor.

Output : $Rod1, Rod2 = \{up, down\}$. The control commands to the rods in the reactor.

Constants :

t_{min} = minimum time between temperature readings
t_{max} = maximum time between temperature readings
c_r = resolution of temperature reading
d = maximum response time

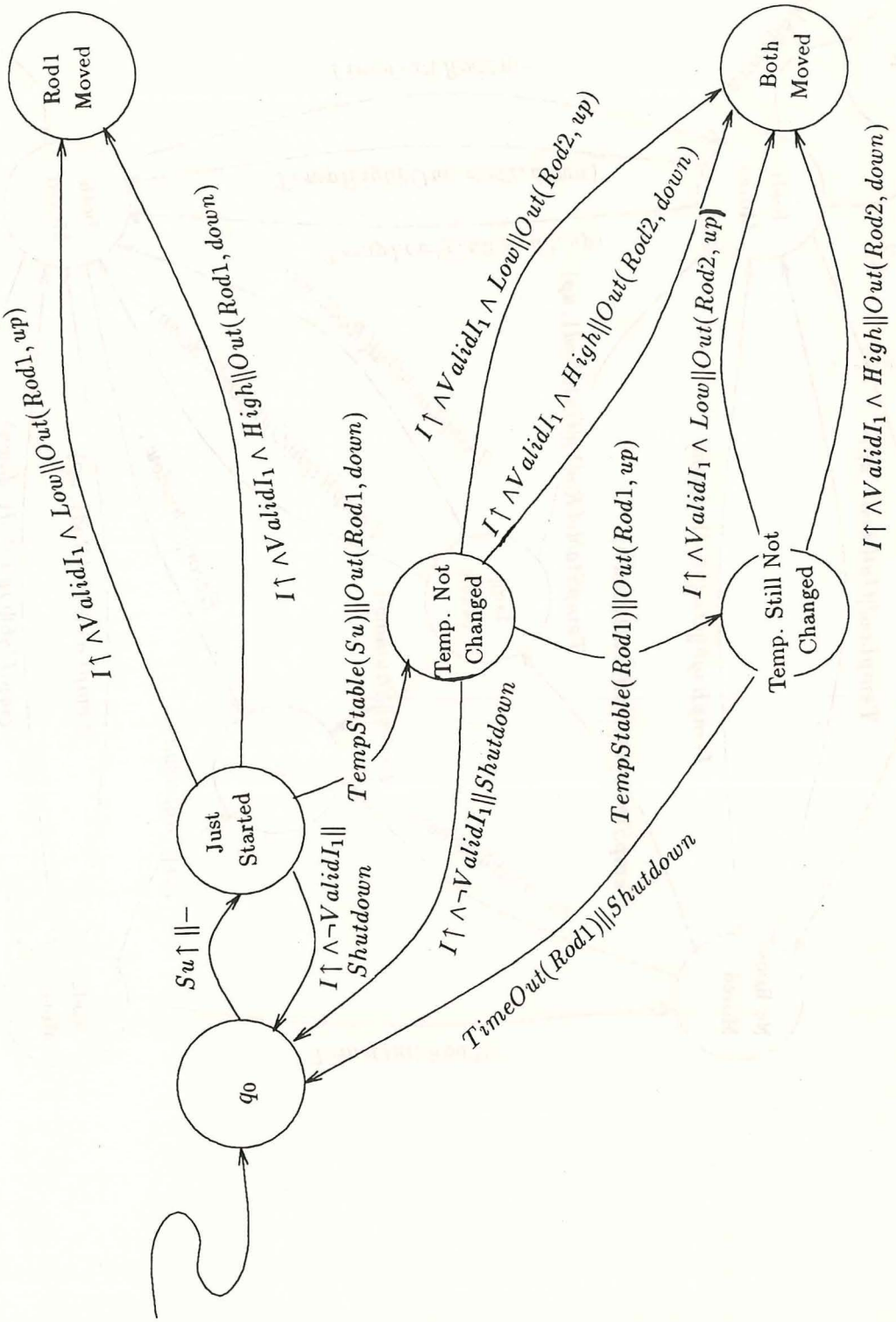
Input Predicate Abbreviations :

$ValidI_1 = 273 \leq v(I) \leq 500$
$ValidI = ValidI_1 \wedge (v(I_i) = v(I_{i-1}) \pm c_r) \wedge (t(I_i\uparrow) \geq t(_{i-1}\uparrow) + t_{min})$
$Error = I\uparrow (\neg ValidI_1 \vee (v(I_i) \neq v(I_{i-1}) \pm c_r) \vee (t(I_i) < t(I_{i-1}) + t_{min}))$
$Low = v(I) < C$
$High = v(I) \geq C$
$TempLow = I\uparrow \wedge ValidI \wedge Low$
$TempHigh = I\uparrow \wedge ValidI \wedge High$
$TimeOut(x) = t > t(x\uparrow) + 30$ where $x \in \Sigma$
$TempStable(x) = t > t(x\uparrow) + t_{max}$ where $x \in \{I, Su, Rod1\}$

Output Predicate Abbreviations :

$Out(O, x)O\uparrow \wedge (v(O) = x) \wedge (t(O\uparrow) \leq t(I\uparrow) + d)$
where $O \in \{Rod1, Rod2\}$ and $x \in \{up, down\}$

The Startup Sequence.



Normal Operation.