Requirements-based Monitors for Real-Time Systems

Dennis K. Peters, *Member, IEEE* and
David Lorge Parnas, *Senior Member, IEEE*

***Abstract*—Before designing safety- or mission-critical real-time systems, a specification of the required behavior of the system should be produced and reviewed by domain experts. After the system has been implemented, it should be thoroughly tested to ensure that it behaves correctly. This is best done using a monitor, a system that observes the behavior of a target system and reports if that behavior is consistent with the requirements. Such a monitor can be used both as an oracle during testing and as a supervisor during operation. Monitors should be based on the documented requirements of the system.**

If the target system is required to monitor or control real-valued quantities, then the requirements, which are expressed in terms of the monitored and controlled quantities, will allow a range of behaviors to account for errors and imprecision in observation and control of these quantities. Even if the controlled variables are discrete valued, the requirements must specify the timing tolerance. Because of the limitations of the devices used by the monitor to observe the environmental quantities, there is unavoidable potential for false reports, both negative and positive.

This paper discusses design of monitors for real-time systems, and examines the conditions under which a monitor will produce false reports. We describe the conclusions that can be drawn when using a monitor to observe system behavior.

***Index Terms*—Automated testing, Test oracle, Real-time system, Supervisor**

I. Introduction

COMPUTER systems are increasingly being used in situations where their correct behavior is essential for the safety of people, equipment, the environment and businesses. In many cases there are *real-time* requirements on the behavior of these systems — failure to satisfy timing constraints is as costly as responding incorrectly.

When designing such safety- or mission-critical real-time systems, good engineering practice dictates that a clear, precise and unambiguous specification of the required behavior of the system be produced and reviewed for correctness by experts in the domain of application of the system. Research suggests that such reviews are more effective if the system behavioral requirements documentation:

- expresses the required behavior in terms of the quantities from the environment in which the system operates (i.e., external to the system),
- uses terminology and notation that is familiar to, or can be learned by, the domain experts, and
- is structured to permit independent review and application of small parts of the document.[13]

After the system has been implemented, it should be tested to ensure that its behavior satisfies the requirements. In safety-critical applications the system should be monitored by an independent safety system to ensure continued correct behavior. To achieve these goals there must be a means of quickly determining if the observed behavior is acceptable or not; this can be quite difficult for complex real-time systems. Several authors (e.g., [40]) have suggested that a practical approach to analyzing the behavior of a real-time system is to use a *monitor*: a system that observes and analyzes the behavior of another system (the *target system*). Such a monitor could be used either as an 'oracle'[43] during system testing, or, for a limited class of systems, as a 'supervisor'[36] to detect and report system failure during operation.

This paper examines the relationship between the target system and the monitor, in particular with respect to the means by which the monitor observes the system behavior, and the impact of this on the usefulness of the monitor. It gives some necessary conditions for monitor feasibility. In related work [27] we have developed techniques for automatically generating software to implement a monitor from a System Requirements Document (SRD) written in a notation that is based on the method presented in [41], which developed from the A-7E project at the US Naval Research Laboratory.[14]

This work focuses on monitors for computer-based systems that are intended to observe and/or control some quantities external to the computer. Such quantities are often best represented by continuous, rather than discrete, valued functions. In particular, the requirements for any real-time systems will include time, which we model as a continuous variable.

The remainder of this section defines the notation and terminology used in this paper. Section II introduces a system that is used as a running example throughout the remainder of the paper. Section III briefly presents the "Four Variable Model", which relates system and software requirements in terms of the behavior of the input and output devices. Section IV formally defines a monitor and its accuracy, and discusses possible monitor configurations. Section V discusses the impact of realistic monitor input devices on the conclusions that can be drawn

Dennis Peters is with Electrical and Computer Engineering, Faculty of Engineering and Applied Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1B 3X5, E-mail: dpeters@engr.mun.ca

David Parnas is with Department of Computing and Software, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada L8S 4K1, E-mail: parnas@mcmaster.ca

Manuscript received Mar. 2001; revised Sept. 2001; accepted Sept. 2001. Recommended for acceptance by M.J. Harrold and A. Bertolino. For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 115154.

from using the monitor, and gives some necessary conditions that must be satisfied in order for a monitor to be useful. The final section draws some conclusions and suggests future work.

A. Notation and Terminology

There are at least two systems of interest in any application of this work:

- The *target system* is the system to be monitored. Its required behavior is specified in the System Requirements Document (SRD).
- The *monitor system* is the system that observes the behavior of the target system and reports whether or not it conforms to the SRD.

As discussed further in Section IV-B, in some configurations these two systems may share components.

The following notational conventions are used in this paper. The symbol "$\stackrel{\text{df}}{=}$" is used to represent "is defined as", so, for example "$\mathrm{f}(x) \stackrel{\text{df}}{=} x + 5$" defines the function f. The common bracketing notation for describing an open or closed range of real numbers is used, e.g., $(x, y] = \{z \in \mathbf{Real} \mid x < z \leq y\}$, and this is extended to ranges of fixed-precision numbers by replacing "," with "...", so, for example, $[0.0 \ldots 0.7] = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$.

For a set, $\mathbf{R}$, the *characteristic predicate*, R, is the predicate such that $\mathrm{R}(x)$ is $\underline{true}$ if and only if $x \in \mathbf{R}$, i.e., $\mathrm{R}(x) \Leftrightarrow x \in \mathbf{R}$. We say that the predicate R *characterizes* the set $\mathbf{R}$. By convention sets are denoted in bold font, e.g., $\mathbf{R}$, and the characteristic predicate of the set is denoted with the same name but not bolded, e.g., $\mathrm{R}(x)$.

II. Example: Maze-tracing Robot

The Maze-tracing robot is based on a course project for a software design and documentation course taught to undergraduate computer engineering students at McMaster University. Although it is a contrived example, it is "real" in that the software is expected to control real hardware using vendor supplied interface libraries, and the safety requirements are strictly enforceable and have potentially costly consequences if violated (i.e., damage to laboratory equipment).

The environment relevant to the target system consists of

- a *draw-bot*, which is a three-jointed (two horizontal, one vertical) robot arm on a fixed base that is capable of moving a pen to mark on paper,
- a two-dimensional maze placed within reach of the draw-bot,
- three momentary contact buttons labeled "stop", "home" and "back", and
- a human operator.

The target system consists of

- a computer (with software) capable of controlling the robot, interpreting commands from the operator and displaying messages, and
- interface hardware to interface with the robot and the buttons.

The system is required to control the draw-bot such that it traces a path on the maze from the beginning to the end (if such a path exists) without coming too close to any maze wall. The goal is to complete the path as quickly as possible. Since the maximum speed of the draw-bot is fixed, a shortest path through the maze should be chosen by the system.

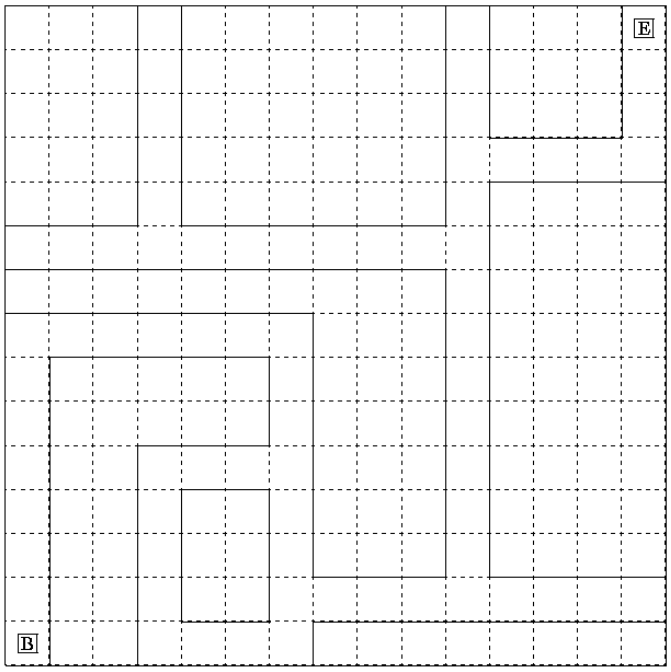


Fig. 1. Sample Maze

We represent the location of the draw-bot pen tip using a Boolean, $^{mc}\mathsf{penDown}$, to indicate if the pen is touching the maze surface or not, and a pair, $(^{mc}\mathsf{penPos.x}, ^{mc}\mathsf{penPos.y})$ of reals, representing the location in the horizontal plane where the pen tip is touching the maze (if $^{mc}\mathsf{penDown}$ is $\underline{true}$) or would touch the maze if lowered (if $^{mc}\mathsf{penDown}$ is $\underline{false}$). The location is specified by the distance, in millimeters, from the respective axis, which are parallel (x = 0) and perpendicular (y = 0) to the front edge of the robot arm base. The extent of the region of interest is defined by the constants MIN_X, MAX_X, MIN_Y and MAX_Y. The origin is the center of the robot base post. The home location of the pen-tip, to which it is returned on initialization of the draw-bot, is $(\mathsf{HOME\_X}, \mathsf{HOME\_Y})$.

As illustrated in Figure 2, the maze is contained within a $\mathsf{M\_WIDTH}\,\mathrm{mm} \times \mathsf{M\_HEIGHT}\,\mathrm{mm}$ region of the horizontal plane bounded by the lines $x = -\mathsf{M\_X\_OFFSET}$, $y = \mathsf{M\_Y\_OFFSET}$, $x = -\mathsf{M\_X\_OFFSET} + \mathsf{M\_WIDTH}$ and $y = \mathsf{M\_Y\_OFFSET} + \mathsf{M\_HEIGHT}$, which are the external walls of the maze. The internal walls of the maze are segments of the lines $x = -\mathsf{M\_X\_OFFSET} + n \times \mathsf{M\_CELL\_SIZE}$ mm and $y = \mathsf{M\_Y\_OFFSET} + n \times \mathsf{M\_CELL\_SIZE}$ mm, where n is an integer (i.e., a square grid with line spacing M_CELL_SIZE mm). The endpoints of the walls lie at intersections of these grid lines. Figure 1 is a sample maze with dashed lines indicating the possible wall locations.

III. The Four Variable Requirements Model

As pointed out, e.g. in [41], [42], [44], it is important when specifying system and software requirements to distinguish

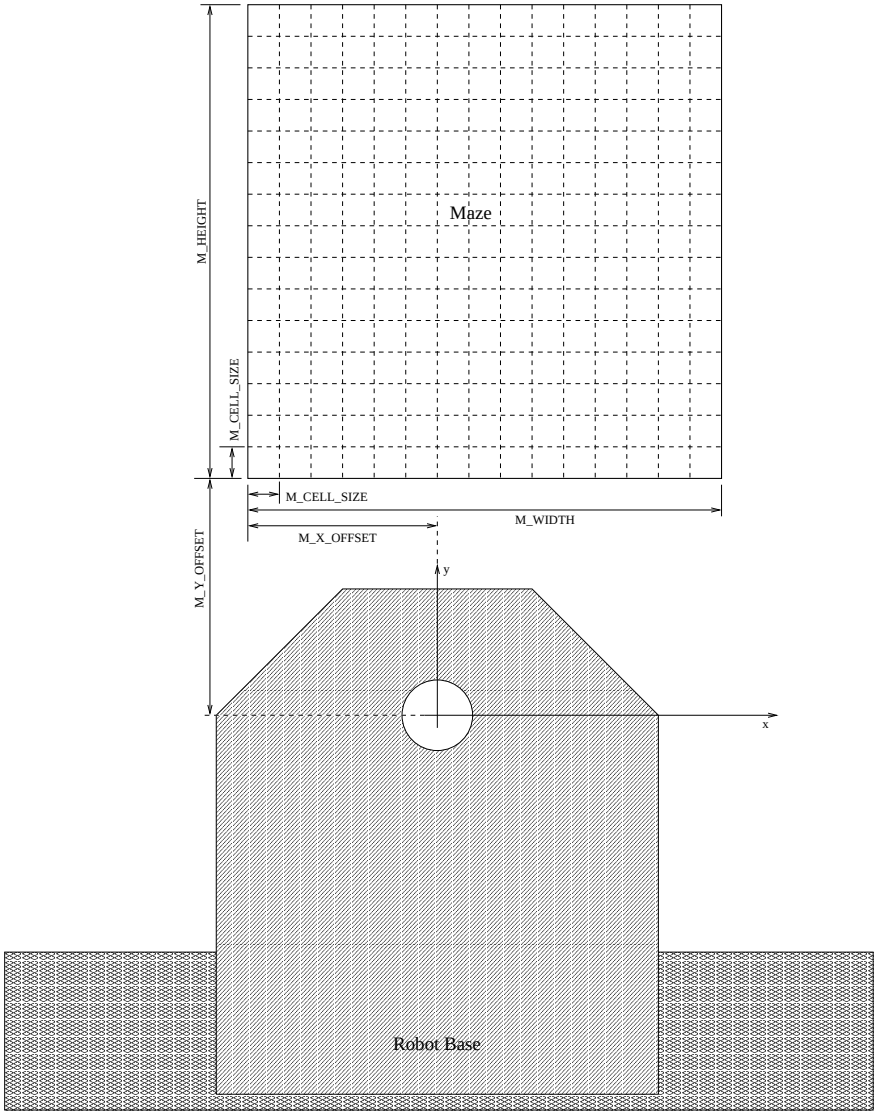


Fig. 2. Robot and Maze Parameters

quantities that are in the environment, i.e., external to the system, from those that are internal to the system or artifacts of the description of either the requirements or the design. Environmental quantities are those that are independent of the chosen solution and are apparent to the "customer"; they are the best quantities to use when describing the requirements for the system since they are the quantities that the customer will be concerned with and knowledgeable about. These quantities will include such things as physical properties (e.g., temperature, pressure, location of objects), values or images displayed on output display devices, settings of input switches, dials etc., and states of controlled devices. Variables internal to the system are inappropriate for system requirements specification since they are artifacts of the solution, not the problem, and will be unfamiliar to the customer. The requirements for the *software* alone can be expressed in terms of variables internal to the system, see Section III-D. The "Four Variable Model", introduced in [24], [41], [42], gives a model for system requirements and design and is adopted here as a framework for describing requirements.

It is widely accepted (e.g., see [14], [24], [30], [42]) that environmental quantities can be modeled by functions of time. Given the environmental quantities relevant to a particular system, $q_1, q_2, \ldots, q_n$, of types $\mathbf{Q_1}, \mathbf{Q_2}, \ldots \mathbf{Q_n}$, respectively, we can represent the behavior of the system in its environment by an *environmental state function*, $\mathrm{S} : \mathbf{Real} \rightarrow \mathbf{Q_1} \times \mathbf{Q_2} \times \ldots \times \mathbf{Q_n}$, defined on all intervals of system operation. For convenience we define $\mathbf{St} \stackrel{\mathrm{df}}{=} \mathbf{Q_1} \times \mathbf{Q_2} \times \ldots \times \mathbf{Q_n}$ (i.e., $\mathbf{St}$ is the set of possible environmental states).

The environmental quantities of interest can be classified into two (not necessarily disjoint) sets: the *controlled* quantities — those that the system may be required to change the value of, and the *monitored* quantities — those that should affect the behavior of the system.[1] Assuming that the monitored quantities are $q_1, q_2, \ldots, q_i$, the *monitored state function*, $\underline{m}^t : \mathbf{Real} \rightarrow \mathbf{Q_1} \times \mathbf{Q_2} \times \ldots \times \mathbf{Q_i}$, is derived from the environmental state function by including only the monitored quantities. Similarly,

[1]There may also be environmental quantities that are neither monitored nor controlled but are relevant to the design or analysis of the system. These quantities are not relevant to the the four-variable model and so are not considered in this presentation.

if the controlled quantities are $q_j, q_{j+1}, \ldots, q_n$, the *controlled state function*, $\underline{c}^t : \mathbf{Real} \rightarrow \mathbf{Q_j} \times \mathbf{Q_{j+1}} \times \ldots \times \mathbf{Q_n}$, is derived. In this paper, a pair of functions $(\underline{m}^t, \underline{c}^t)$ will denote an environmental state function. With respect to a particular target system, $\mathbf{M}$ denotes the set of functions of type $\mathbf{Real} \rightarrow \mathbf{Q_1} \times \mathbf{Q_2} \times \ldots \times \mathbf{Q_i}$, (type correct for a monitored state function) and $\mathbf{C}$ denotes the set of functions of type $\mathbf{Real} \rightarrow \mathbf{Q_j} \times \mathbf{Q_{j+1}} \times \ldots \times \mathbf{Q_n}$ (type correct for a controlled state function). The environmental quantities relevant to the Maze Tracing Robot are given in Figure 3.

We usually are only interested in the environmental state function during the periods when the system is operating (i.e., it is turned on). An environmental state function defined on the (possibly infinite) time interval of a single execution of the system is known as a *behavior* of the system. A behavior is *acceptable* if it describes a situation in which the system is operating in a manner that is consistent with the requirements.

A. System Requirements

The *system behavioral requirements* (or, where the meaning is clear from context, *system requirements*) characterize the set of acceptable behaviors. Since the system is expected to observe the monitored quantities and control the values of the controlled quantities accordingly, it is natural to express this as a relation, $\mathbf{REQ} \subseteq \mathbf{M} \times \mathbf{C}$. A behavior is acceptable if and only if $\mathrm{REQ}(\underline{m}^t, \underline{c}^t)$ is $\underline{true}$. Note that, since implementations will invariably introduce some amount of unpredictable delay, or inaccuracy in the measurement, calculation, or output of values, for real systems, $\mathbf{REQ}$ will be a relation that is not a function, i.e., there will be more than one acceptable $\underline{c}^t$ for a given $\underline{m}^t$.

In many cases $\mathbf{REQ}$ will be independent of the actual date and time, and will depend only on the time elapsed since some event (e.g., the system being turned on). In these cases, equivalence classes of behaviors can be represented by the behavior formed by translation along the time axis such that time = 0 corresponds to that event. In cases where aspects of the date or time (e.g., day of month, hour of day) are significant, time = 0 will need to be chosen to correspond to an appropriate clock time and appropriate functions defined to determine the needed quantities.

The following is an informal description of the system behavioral requirements for the Maze-tracing Robot. A formal requirements document is given in [27].

1) Safety Requirements: If at any time the stop button is pressed (${}^m\mathsf{stopButton} = {}^s\mathsf{Down}$) the robot must stop moving within RESPONSE_TIME seconds and must remain stationary until the stop button is released (${}^m\mathsf{stopButton} = {}^s\mathsf{Up}$).

When the pen is down (${}^{mc}\mathsf{penDown} = \underline{true}$) the pen tip must never come within WALL_SPACE mm of a wall point.

2) Messages: Whenever a significant event occurs (i.e., a button is pressed or released, the pen reaches the start or end point of the maze or the home position, or an error is detected) the system must display a diagnostic message describing the event and the system's response to it.

3) Performance: The performance goal for the system is to minimize the time between the pen first touching the paper and it being returned to its home position.

4) Initialization: When the system is started it must attempt to find a legal path through the maze. If an error occurs (e.g., maze read failure) or if there is no path through the maze, then an appropriate diagnostic message must be output and the system must halt without turning on the robot power (${}^c\mathsf{powerOn} = \underline{false}$).

5) Starting: After it has been determined that there is a path through the maze, the robot power must be turned on (${}^c\mathsf{powerOn} = \underline{true}$), which initializes the pen to the home position (${}^{mc}\mathsf{penPos} = (\mathsf{HOME\_X}, \mathsf{HOME\_Y})$) with the pen up (${}^{mc}\mathsf{penDown} = \underline{false}$). The pen must then be moved to the start position of the maze.

6) Forward: Once the starting position has been reached the pen must be lowered (${}^{mc}\mathsf{penDown} = \underline{true}$) and a path traced through the maze to the end. When the pen reaches the end of the maze it must be raised (${}^{mc}\mathsf{penDown} = \underline{false}$) and returned to the home position.

7) Reverse: If at any time while the path is being traced the "back" button is pressed (${}^m\mathsf{backButton} = {}^s\mathsf{Down}$) the Draw-bot is required to reverse the direction of its tracing within RESPONSE_TIME seconds and begin to re-trace its path back to the beginning. It should continue to re-trace its path only as long as the "back" button is held down — when it is released the Draw-bot should continue in the forward direction. If, while reversing, it reaches the start position it should stop there until either the "back" button is released or the "home" button is pressed.

8) Home: If at any time while the path is being traced (in either direction) the "home" button is pressed (${}^m\mathsf{homeButton} = {}^s\mathsf{Down}$) the Draw-bot is required to stop tracing within RESPONSE_TIME seconds, raise the pen (${}^{mc}\mathsf{penDown} = \underline{false}$) and return to the home position, without making any further marks.

9) Done: When the pen has been returned to the home position, the power must be turned off (${}^c\mathsf{powerOn} = \underline{false}$) and the system must halt.

B. Environmental Constraints

The possible values of the environmental state function are constrained by physical laws independent of the system to be built. For example,

- the rate of change (or higher order derivatives) of a quantity may be constrained by some natural laws,
- some quantities may be related to each other (e.g., pressure and temperature in a closed container),
- values may be only able to change in certain ways (e.g., positions of selector switches), or
- certain events may not be able to occur simultaneously.

These laws are described by the relation $\mathbf{NAT} \subseteq \mathbf{M} \times \mathbf{C}$, which contains all values of $(\underline{m}^t, \underline{c}^t)$ that are possible in the environment.

In the case of the Maze-tracing Robot, some environmental constraints are as follows:

- Since the home position is guaranteed to be outside the maze and ${}^m\mathsf{mazeStart}$ and ${}^m\mathsf{mazeEnd}$ are inside the maze and more than POS_TOL mm from the maze wall, the pen cannot be within POS_TOL mm of both the home position and either of ${}^m\mathsf{mazeStart}$ or ${}^m\mathsf{mazeEnd}$.

Variable	Description	Monitored	Controlled	Value Set	Notes
^{m}t	Current time	•		**Real**	1
^{m}mazeWalls	The set of points that make up the walls of the maze. Note that the exterior walls (i.e., the perimeter) are included.	•		set of **Posn**	2, 4
^{m}mazeStart	Start position for the maze.	•		**Posn**	2, 4
^{m}mazeEnd	Finish position for the maze.	•		**Posn**	2, 4
^{m}stopButton	Status of the "stop" button.	•		**buttonT**	
^{m}homeButton	Status of the "home" button.	•		**buttonT**	
^{m}backButton	Status of the "back" button.	•		**buttonT**	
^{mc}penPos	The position of the pen relative to the origin $(0,0)$, which is the centre of the robot base post.	•	•	**Posn**	2
^{mc}penDown	*true* iff the pen is touching the plane containing the maze. Assumed to be initially *false*.	•	•	**Boolean**	
^{c}powerOn	*true* iff the robot power is on. Assumed to be initially *false*.		•	**Boolean**	3
^{c}message	The message displayed on the operator console.		•	**string**	

Notes

1) Time is represented as the amount of time elapsed since some fixed arbitrary time before the system is started (only time differences are relevant to the requirements). The required resolution is 0.1 seconds.
2) Positions are represented using (x, y) coordinates as described in Section II. The required resolution is 0.5 millimetres.
3) This is the power for the robot motors only. It is independent of the power for the controlling computer system.
4) The maze is constant throughout a particular execution of the system, but may be different for different executions. The maze is described in a file on the computer system disk that is to be read during initialization.

Fig. 3. Maze-tracing Robot Environmental Quantities

- The pen tip can move at a maximum of 2.0 mm/s.

C. System Design

The environmental quantities cannot usually be directly observed or manipulated by the system software, but must be measured or controlled by some devices (e.g., sensors, actuators, relays, buttons), which communicate with the software through the computer's input or output registers, represented by program variables. The *input* quantities are those program variables that are available to the software and provide information about the monitored quantities. For input quantities, $i_1, i_2, \ldots, i_n$, of types $\mathbf{I_1}, \mathbf{I_2}, \ldots, \mathbf{I_n}$, respectively, an *input state function* is a function, $\underline{i}^t : \mathbf{Real} \rightarrow \mathbf{I_1} \times \mathbf{I_2} \times \ldots \times \mathbf{I_n}$, representing the values of the input quantities during system operation. Similarly, the *output* quantities are those program variables through which the software can change the value of the controlled quantities. For output quantities, $o_1, o_2, \ldots, o_m$, of types $\mathbf{O_1}, \mathbf{O_2}, \ldots, \mathbf{O_m}$, respectively, an *output state function* is a function, $\underline{o}^t : \mathbf{Real} \rightarrow \mathbf{O_1} \times \mathbf{O_2} \times \ldots \times \mathbf{O_m}$, representing the values of the output quantities during system operation. For convenience, with respect to a particular system being specified, the set of functions of type $\mathbf{Real} \rightarrow \mathbf{I_1} \times \mathbf{I_2} \times \ldots \times \mathbf{I_n}$ is denoted $\mathbf{I}$, and the set of functions of type $\mathbf{Real} \rightarrow \mathbf{O_1} \times \mathbf{O_2} \times \ldots \times \mathbf{O_m}$ is denoted $\mathbf{O}$. The behavior of the interface between the environment and the software is described by the input relation, $\mathbf{IN} \subseteq \mathbf{M} \times \mathbf{I}$, which characterizes the possible values of $\underline{i}^t$ for any instance of $\underline{m}^t$, and the output relation, $\mathbf{OUT} \subseteq \mathbf{O} \times \mathbf{C}$, which characterizes the possible values of $\underline{c}^t$ for any instance of $\underline{o}^t$. Figure 4 illustrates the data flow between the components in the system design, and the relations that describe the behavior of these component.

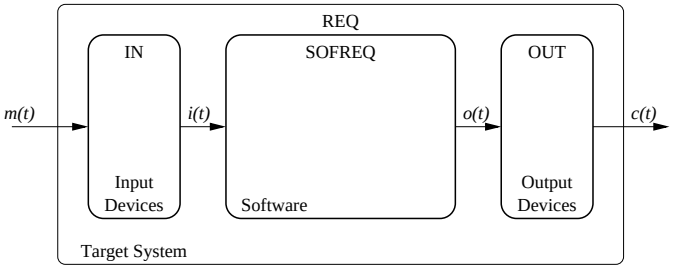


Fig. 4. System Design

For the Maze-tracing Robot system, the interface with the draw-bot is through a vendor-supplied software library, which allows a program to query the state of the input buttons and to

set the angle of the joints in the draw-bot arm, and thus control the position of the pen tip. The maze information is represented in a data file listing the locations of the maze walls and the start and stop positions. The input and output state functions thus represent the values of these variables as functions of time (e.g., the angle that each joint has most recently been set to at any time) and the **IN** and **OUT** relations relate these values to the actual maze, draw-bot position, and button status.

D. Software Requirements

In [24] the actual software behavior is described by the *software behavior relation*, **SOF**, and an expression is given for software acceptability. In this work, as in [12], we are interested in characterizing all acceptable software, so we use the *software requirements relation*, **SOFREQ**, which characterizes the set of acceptable behaviors of the software — those pairs, $\left(\underline{i}^t, \underline{o}^t\right)$, such that any possible environmental state function, with respect to the given input and output devices and environmental constraints, is acceptable. This is fully determined by **REQ**, **IN**, **OUT** and **NAT**, as follows

$$\begin{aligned}\mathbf{SOFREQ} \stackrel{\text{df}}{=} \\ \Big\{\left(\underline{i}^t, \underline{o}^t\right) \mid \Big(\forall \underline{m}^t, \underline{c}^t, \big(\text{IN}\left(\underline{m}^t, \underline{i}^t\right) \wedge \text{OUT}\left(\underline{o}^t, \underline{c}^t\right) \wedge \\ \text{NAT}\left(\underline{m}^t, \underline{c}^t\right)\big) \\ \Rightarrow \text{REQ}\left(\underline{m}^t, \underline{c}^t\right)\Big)\Big\} \quad (1)\end{aligned}$$

To understand the difference between **SOFREQ** and **SOF**, consider the simple requirements given in [11], where $\underline{m}^t$, $\underline{c}^t$, $\underline{i}^t$, and $\underline{o}^t$ are all taken to be **Real** valued (which is not realistic for $\underline{i}^t$ and $\underline{o}^t$) and the environment, requirements and system design are as follows:

$$\begin{aligned}\mathbf{NAT} &\stackrel{\text{df}}{=} \left(\forall t, \underline{c}^t(t) > 0 \wedge \underline{m}^t(t) < 0\right) \\ \mathbf{REQ} &\stackrel{\text{df}}{=} \left(\forall t, \underline{c}^t(t+3) = -\underline{m}^t(t)\right) \\ \mathbf{IN} &\stackrel{\text{df}}{=} \left(\forall t, \underline{i}^t(t+1) = \underline{m}^t(t)\right) \\ \mathbf{OUT} &\stackrel{\text{df}}{=} \left(\forall t, \underline{c}^t(t+1) = \underline{o}^t(t)\right)\end{aligned}$$

SOF describes the behavior of a particular software implementation, whereas **SOFREQ** characterizes all acceptable software behaviors. For this example, since **REQ**, **IN** and **OUT** are all functions, **SOFREQ** contains only relations that are functions on all possible (i.e., negative) inputs: $\mathbf{SOFREQ} = \left\{\left(\underline{i}^t, \underline{o}^t\right) \mid \forall t, (\underline{i}^t(t) < 0) \Rightarrow (\underline{o}^t(t+1) = -\underline{i}^t(t))\right\}$. In particular, the **SOF** given in [11] is not in **SOFREQ**, since, as the authors correctly point out, it does not represent acceptable software behavior.

Note that in many cases $\text{REQ}\left(\underline{m}^t, \underline{c}^t\right) \Rightarrow \text{NAT}\left(\underline{m}^t, \underline{c}^t\right)$. Further, any observed behavior must be in **NAT**, since, by definition, behaviors not in **NAT** are not possible.

IV. Monitors for Real Time Systems

Testing a real-time system typically involves running the target system in a test environment, observing its behavior and comparing it to that required by its specification. Making this comparison can be quite difficult since the requirements may be complex. A *monitor* is a system that observes the behavior of a system and determines if it is consistent with a given specification. That is, we consider a monitor to be ideal if it accurately reports the value of the predicate $\text{REQ}\left(\underline{m}^t, \underline{c}^t\right)$ for each observed behavior.

A. Using Monitors

A monitor can be used to check the behavior of a target system either while the target system is executing or post-facto, using a recording of the behavior. In either case, the monitor should report if *all* behaviors exhibited by the target system are acceptable (i.e., in **REQ**). For a given behavior $\left(\underline{m}^t, \underline{c}^t\right)$ on some interval $[t_i, t_f]$ and any $t_0 \in (t_i, t_f]$, the prefix behavior, $\left(\underline{\hat{m}}^t, \underline{\hat{c}}^t\right)$, formed by considering $\left(\underline{m}^t, \underline{c}^t\right)$ on $[t_i, t_0]$ only, i.e.,

$$\left(\underline{\hat{m}}^t, \underline{\hat{c}}^t\right)(t) \stackrel{\text{df}}{=} \begin{cases} \left(\underline{m}^t, \underline{c}^t\right)(t) & \text{for } t \in [t_i, t_0] \\ \text{undefined} & \text{otherwise} \end{cases}$$

is also a behavior of the system. Thus, if $\left(\underline{\hat{m}}^t, \underline{\hat{c}}^t\right) \notin \mathbf{REQ}$, the system has behaved unacceptably and the monitor should report a failure. That is, since a behavior captures the sequence of actions by the target system from some initial time up to the present, once a behavior has failed, no continuation of that behavior will be considered acceptable since it too will not be in **REQ**. For example, any behavior of the Maze-tracing Robot is unacceptable if the pen has touched a maze wall at some time, regardless of how long it operates without touching the wall after the collision.

This interpretation restricts these techniques to what [1] calls *safety properties* — if a behavior is unacceptable then no extension of that behavior is acceptable. Once a monitor has detected a failure, no further analysis of that behavior will give a different result. In applications such as supervision[36], where continued analysis of the behavior may be needed following detection of a failure, some intervention, either automatic or manual, will be required before the monitor will report acceptable behavior again. For example, if corrective action is taken in response to the failure, and the target system is restored to some known state, then the monitor will need to be re-initialized to correspond to that state. If, on the other hand, the target system is such that it can be assumed to return to some known state following an error, then the monitor can be designed to re-initialize itself correspondingly. Both of these cases can be viewed as a new behavior beginning when the system returns to a known state and hence the behavior would be reported as acceptable until a new failure occurs.

In [1], safety properties are distinguished from *liveness properties* — those requirements such that, for a given requirement and any finite duration behavior, the behavior can always be extended such that it satisfies the requirement. These include the common notions of liveness (the system must respond eventually) and fairness (if requested often enough eventually a given response will occur) as well as statistical properties on the behavior (e.g., the average response time must be less than T). No monitor can determine that a target system does not satisfy such a requirement, since that can only be determined using

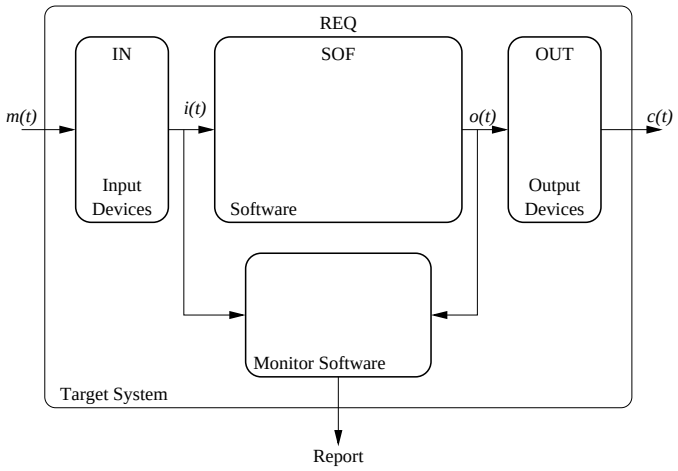


Fig. 5. Software Monitor

infinite behaviors (e.g., a pending request could be serviced in the future). For real systems, however, liveness requirements are rarely strong enough to specify the true requirements, and should be converted into requirements that can be checked for finite duration behaviors (e.g., the system must respond to requests within a fixed time limit), which can be checked by a monitor.

B. Monitor Configuration

In this work, the monitor is assumed to consist of some software running on a computer system. This software cannot, in general, observe the environmental state function, $(\underline{m}^t, \underline{c}^t)$, directly, but must do so through some input devices that communicate the values of the environmental quantities to input registers known as the *monitor software inputs*. For monitor software inputs, $s_1, s_2, \ldots, s_n$, of types $\mathbf{S_1}, \mathbf{S_2}, \ldots \mathbf{S_n}$, respectively, a *monitor input state function* is a function, $\underline{s}^t : \mathbf{Real} \rightarrow \mathbf{S_1} \times \mathbf{S_2} \times \ldots \mathbf{S_n}$, representing the value of the monitor software inputs for the periods of monitor operation. With respect to a particular monitor system, the set of all functions of type $\mathbf{Real} \rightarrow \mathbf{S_1} \times \mathbf{S_2} \times \ldots \mathbf{S_n}$ is denoted $\mathbf{S}$. The behavior of the monitor input devices is characterized by the monitor input relation, $\mathbf{IN_{mon}} \subseteq (\mathbf{M} \times \mathbf{C}) \times \mathbf{S}$. An environmental state function–input state function pair is in the monitor input relation, $((\underline{m}^t, \underline{c}^t), \underline{s}^t) \in \mathbf{IN_{mon}}$, if and only if $\underline{s}^t$ is a possible monitor input state function for the environmental state function represented by $(\underline{m}^t, \underline{c}^t)$. Since the monitor must observe all acceptable behaviors, it is required that $domain(\mathbf{IN_{mon}}) \supseteq \mathbf{REQ} \cap \mathbf{NAT}$.

The design of the monitor will determine, for each monitored or controlled quantity, whether it is observed independently of the target system (i.e., using different devices) or observed directly from the target system software. This results in two basic monitor configurations, in addition to the obvious mixtures of these approaches:

1) A *software monitor* is a monitor that directly observes the target system software input and output variables, i.e., $\underline{s}^t = (\underline{i}^t, \underline{o}^t)$, as illustrated in Figure 5. In this case $\mathbf{IN_{mon}}$ is related to $\mathbf{IN}$ and $\mathbf{OUT}$ as follows.

$$\mathbf{IN_{mon}} = \{((\underline{m}^t, \underline{c}^t), (\underline{i}^t, \underline{o}^t)) \,|\, \mathrm{IN}(\underline{m}^t, \underline{i}^t) \wedge \mathrm{OUT}(\underline{o}^t, \underline{c}^t)\} \quad (2)$$

Software monitors include all of the monitor "architectures" discussed in [39].

2) A *system monitor* is a monitor that observes $(\underline{m}^t, \underline{c}^t)$ using its own input devices as illustrated in Figure 6.

For the Maze-tracing Robot system, a software monitor observes the maze data file together with the sequence of calls, including argument and return values, to the draw-bot interface library. From these, the monitor can determine exactly the input and output state functions. A system monitor, on the other hand, requires some additional sensor hardware. For example the maze could be placed on a digitizing tablet so that a monitor could detect the position of the pen tip.

The software component of the monitor determines if the target system behavior is consistent with $\mathbf{REQ}$ under the assumption that the monitor system's input devices are functioning correctly, as described in $\mathbf{IN_{mon}}$. The software must take into account the fact that $\mathbf{IN_{mon}}$ is usually a relation that is not a function, which we characterize by the two extreme approaches of a pessimistic or an optimistic monitor. A pessimistic monitor requires that *all* behaviors that could have resulted in a particular observation of the target system behavior, $\underline{s}^t$, be in $\mathbf{REQ}$, so the monitor software determines if $\underline{s}^t$ is in the pessimistic monitor set, $\mathbf{MON_{pe}}$, which is defined as

$$\mathbf{MON_{pe}} \stackrel{\mathrm{df}}{=} \left\{\underline{s}^t \in range(\mathbf{IN_{mon}}) \,\middle|\, \left(\forall (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C}, \left(\mathrm{IN_{mon}}((\underline{m}^t, \underline{c}^t), \underline{s}^t) \wedge \mathrm{NAT}(\underline{m}^t, \underline{c}^t)\right) \Rightarrow \mathrm{REQ}(\underline{m}^t, \underline{c}^t)\right)\right\} \quad (3)$$

If $\mathrm{MON_{pe}}(\underline{s}^t)$ is $\underline{true}$ then the behavior is certainly acceptable, i.e., $(\mathrm{MON_{pe}}(\underline{s}^t) \wedge \mathrm{IN_{mon}}((\underline{m}^t, \underline{c}^t), \underline{s}^t)) \Rightarrow \mathrm{REQ}(\underline{m}^t, \underline{c}^t)$. A more optimistic view is to check if *any* behavior that could have resulted in $\underline{s}^t$ is in $\mathbf{REQ}$. The optimistic monitor set, $\mathbf{MON_{op}}$, is defined as

$$\mathbf{MON_{op}} \stackrel{\mathrm{df}}{=} \left\{\underline{s}^t \in range(\mathbf{IN_{mon}}) \,\middle|\, \left(\exists (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C}, \mathrm{IN_{mon}}((\underline{m}^t, \underline{c}^t), \underline{s}^t) \wedge \mathrm{NAT}(\underline{m}^t, \underline{c}^t) \wedge \mathrm{REQ}(\underline{m}^t, \underline{c}^t)\right)\right\} \quad (4)$$

and includes those observations that may, but do not necessarily, represent acceptable behavior. A monitor that evaluates $\mathbf{MON_{op}}$ will not give false negative results — reports that an acceptable behavior is unacceptable — but is not appropriate for safety-critical systems since it may give false positive results — unacceptable behavior reported as acceptable. The difference between $\mathbf{MON_{pe}}$ and $\mathbf{MON_{op}}$, or, more specifically their inverse image under $\mathbf{IN_{mon}}$, is indicative of the appropriateness of the monitor input devices as reflected in $\mathbf{IN_{mon}}$.

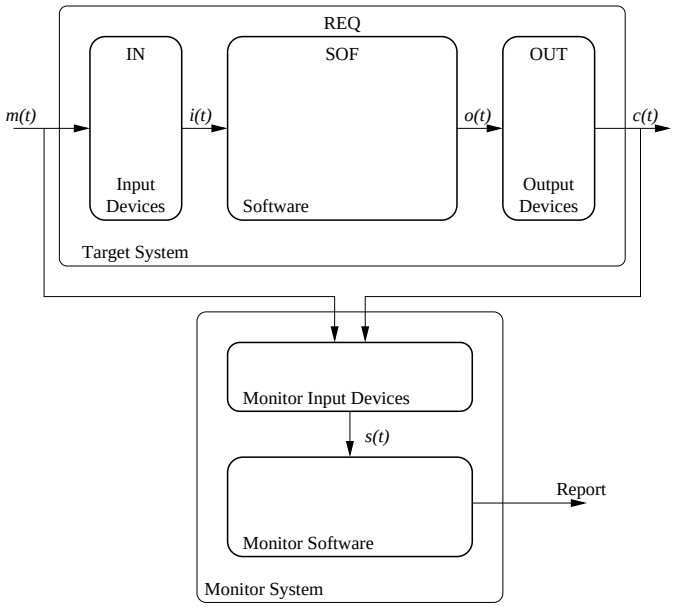


Fig. 6. System Monitor

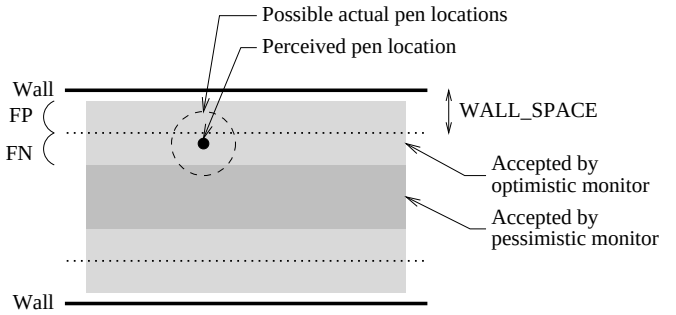


Fig. 7. Optimistic and Pessimistic Maze-tracing Robot Monitor

Consider, for example, a system monitor for the Maze-tracing Robot that uses a digitizing tablet to determine the pen position, and the state illustrated in Figure 7. If the tablet is such that the monitor can determine the pen position to within ± 2 mm, indicated by the dashed circle around the perceived pen location, then a pessimistic monitor will report a failure if it perceives the pen to be within WALL_SPACE $+ 2$ mm of a wall, as indicated by the darker shaded region. An optimistic monitor, on the other hand, would only report a failure if it perceives the pen to be within WALL_SPACE $- 2$ mm of a wall, as indicated by the lighter shaded region. Clearly if WALL_SPACE is less than 2 mm then this overly optimistic — the wall itself would be inside the region accepted by the optimistic monitor, and thus accurately reported collisions with the wall would still be accepted.

A realistic monitor may combine these approaches, for example being pessimistic with regard to some quantities, and optimistic with regard to some others.

In the case of the software monitor configuration, and neglecting impossible behaviors (i.e., $(\underline{m}^t, \underline{c}^t) \notin \mathbf{NAT}$), $\mathbf{MON_{pe}} = \mathbf{SOFREQ}$ — a software monitor determines if the target software is behaving in an acceptable manner.

From the above definitions we can see that $\mathrm{MON_{op}}(\underline{s}^t) \not\Rightarrow \mathrm{MON_{pe}}(\underline{s}^t)$ — the pessimistic monitor will reject some behaviors that are accepted by the optimistic one. Also, if we assume that the input devices are working, i.e., for any observed monitor input state function, $\underline{s}^t$, there is a possible corresponding environmental state function: $\exists (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C}, (\mathrm{IN_{mon}}((\underline{m}^t, \underline{c}^t), \underline{s}^t) \wedge \mathrm{NAT}(\underline{m}^t, \underline{c}^t))$, then the reverse implication holds, i.e., $\mathrm{MON_{pe}}(\underline{s}^t) \Rightarrow \mathrm{MON_{op}}(\underline{s}^t)$ — if a behavior is acceptable using the pessimistic approach, then it is acceptable using an optimistic approach. Both of these are consistent with our intuition.

In cases where $\mathbf{IN_{mon}}$ and $\mathbf{IN_{mon}}^{-1}$ are both functions (i.e., each $(\underline{m}^t, \underline{c}^t)$ maps to only one $\underline{s}^t$, which can be uniquely mapped back to $(\underline{m}^t, \underline{c}^t)$), and again assuming that any observed monitor input state function results from a possible environmental state function, $\mathbf{MON_{pe}} = \mathbf{MON_{op}}$. As discussed in Section V, for real input devices and discrete time systems, both $\mathbf{IN_{mon}}$ and $\mathbf{IN_{mon}}^{-1}$ are relations that are not functions.

C. Accuracy

The accuracy of a monitor is determined by the set of possible false negatives, denoted $\mathbf{FN}$, which is the intersection of $\mathbf{REQ}$ with the set of actual behaviors that the monitor may report as being unacceptable. The behaviors that may be reported as unacceptable are those in the image of $\overline{\mathbf{MON_{pe}}}$ under $\mathbf{IN_{mon}}^{-1}$, as follows.

$$\begin{aligned}\mathbf{NEG} &\stackrel{\mathrm{df}}{=} \left\{ (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C} \mid \left(\exists \underline{s}^t \in \mathbf{S}, \mathrm{IN_{mon}}\left((\underline{m}^t, \underline{c}^t), \underline{s}^t\right) \right.\right. \\ &\qquad \left.\left. \wedge \neg \mathrm{MON_{pe}}(\underline{s}^t)\right)\right\} \\ &= \left\{ (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C} \mid \left(\mathbf{IM}\left(\underline{m}^t, \underline{c}^t\right) \cap \overline{\mathbf{REQ}}\right) \neq \emptyset \right\} \end{aligned} \tag{5}$$

where $\mathbf{IM}(\underline{m}^t, \underline{c}^t)$ is the image of $(\underline{m}^t, \underline{c}^t)$ under $\mathbf{IN_{mon}} \circ \mathbf{IN_{mon}}^{-1}$:

$$\begin{aligned}\mathbf{IM}\left(\underline{m}^t, \underline{c}^t\right) \stackrel{\mathrm{df}}{=} \Big\{ &\left(\tilde{\underline{m}}^t, \tilde{\underline{c}}^t\right) \mid \\ &\left(\left(\underline{m}^t, \underline{c}^t\right), \left(\tilde{\underline{m}}^t, \tilde{\underline{c}}^t\right)\right) \in \left(\mathbf{IN_{mon}} \circ \mathbf{IN_{mon}}^{-1}\right) \Big\} \end{aligned} \tag{6}$$

$\mathbf{IM}(\underline{m}^t, \underline{c}^t)$ represents, for a given actual behavior, the set of behaviors that could be perceived the same by the software component of the monitor — it indicates the set of behaviors that the pessimistic monitor must consider to have possibly occurred. Thus, $\mathbf{FN}$ is

$$\begin{aligned}\mathbf{FN} &\stackrel{\mathrm{df}}{=} \mathbf{REQ} \cap \mathbf{NEG} \\ &= \left\{ (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C} \middle| \mathrm{REQ}\left(\underline{m}^t, \underline{c}^t\right) \wedge \right. \\ &\qquad \left. \left(\mathbf{IM}\left(\underline{m}^t, \underline{c}^t\right) \cap \overline{\mathbf{REQ}}\right) \neq \emptyset \right\} \end{aligned} \tag{7}$$

Referring to Figure 7, $\mathbf{FN}$ contains all actual pen positions in the region between the region accepted by the pessimistic monitor and the line WALL_SPACE from the wall.

Consider $\mathbf{FN}$ under the best and worst case scenarios with respect to $\mathbf{IN_{mon}}$. In the best case $\mathbf{IN_{mon}}$ is the identity relation, perfectly relaying the values of $(\underline{m}^t, \underline{c}^t)$ to the software component of the monitor. In this case, $\mathbf{IM}(\underline{m}^t, \underline{c}^t) =$

$\{(\underline{m}^t, \underline{c}^t)\}$, $\mathbf{NEG} = \overline{\mathbf{REQ}}$ and $\mathbf{FN} = \emptyset$ — the software of the monitor can detect exactly if the behavior is acceptable or not. In the worst case $\mathbf{IN_{mon}}$ is a constant function, mapping all values of $(\underline{m}^t, \underline{c}^t)$ to the same value. In this case $\mathbf{IM}\,(\underline{m}^t, \underline{c}^t) = domain(\mathbf{IN_{mon}})$, $\mathbf{NEG} = domain(\mathbf{IN_{mon}})$ and $\mathbf{FN} = \mathbf{REQ}$ — under no circumstances can the monitor be sure that the behavior is acceptable. In this case the monitor will be *infeasible*.

Definition 1: A monitor is said to be *infeasible* with respect to a monitor input relation, $\mathbf{IN_{mon}}$, system requirements relation, $\mathbf{REQ}$, and environmental constraints, $\mathbf{NAT}$, if and only if $\mathbf{MON_{pe}} = \emptyset$.

Note that infeasibility is an extreme case: it indicates that the monitor input devices are such that no behaviors will be accepted. The size of $\mathbf{FN}$ relative to the operational domain is a more precise measure of monitor usefulness.

Clearly if $\mathbf{IN_{mon}}$ is the identity relation, then $\mathbf{MON_{pe}} = \mathbf{REQ}$ and so, assuming a non-empty $\mathbf{REQ}$, the monitor is feasible.

For an alternative view of accuracy, consider the set of false positives that may be reported by a monitor using the optimistic approach defined in Eq. (4), as follows.

$$\begin{aligned}\mathbf{POS} &\stackrel{\text{df}}{=} \Big\{ (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C} \mid \big(\exists \underline{s}^t \in \mathbf{S}, \mathrm{IN}_{\mathrm{mon}}\left((\underline{m}^t, \underline{c}^t), \underline{s}^t\right) \\ &\qquad \wedge \mathrm{MON}_{\mathrm{op}}(\underline{s}^t)\big)\Big\} \\ &= \left\{ (\underline{m}^t, \underline{c}^t) \mid \left(\mathbf{IM}\,(\underline{m}^t, \underline{c}^t) \cap \mathbf{REQ}\right) \neq \emptyset \right\}\end{aligned} \tag{8}$$

where $\mathbf{IM}$ is as defined in Eq. (6). The false positive set, $\mathbf{FP}$, is thus

$$\begin{aligned}\mathbf{FP} &\stackrel{\text{df}}{=} \overline{\mathbf{REQ}} \cap \mathbf{POS} \\ &= \Big\{ (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C} \big| \neg \mathrm{REQ}\,(\underline{m}^t, \underline{c}^t) \wedge \\ &\qquad \left(\mathbf{IM}\,(\underline{m}^t, \underline{c}^t) \cap \mathbf{REQ}\right) \neq \emptyset \Big\}\end{aligned} \tag{9}$$

In Figure 7, $\mathbf{FP}$ contains all the actual locations in the region between the WALL_SPACE line and the edge of the region accepted by the optimistic monitor. Considering the $\mathbf{IN_{mon}}$ scenarios from above, in the best case $\mathbf{FP} = \emptyset$ and in the worst case $\mathbf{FP} = domain(\mathbf{IN_{mon}})$ — the monitor will report all observations as acceptable behavior.

For realistic cases $\mathbf{FN}$ and $\mathbf{FP}$ will be non-empty and should be used during monitor system design to determine if the monitor is accurate enough for the particular application. This is discussed further in the next section.

V. Practical Monitors

Practical monitors are likely to be implemented using either general- or special-purpose digital computers. This technology implies certain characteristics of the monitor input relation, and monitor behavior, which influence the conclusions that can be drawn from the monitor output. This section discusses these characteristics, and states some conditions which must hold in order for the monitor to produce meaningful results.

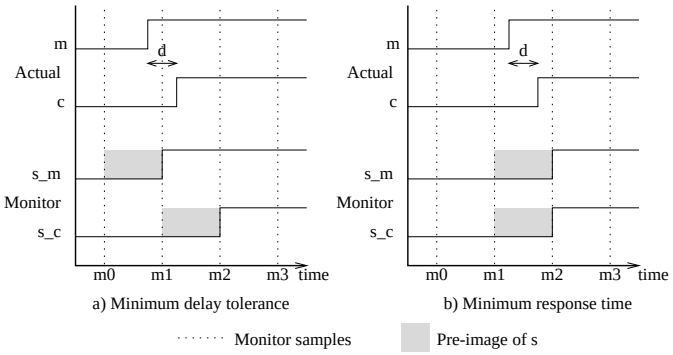


Fig. 8. Time Accuracy

A. Observation Errors

The choice of devices and/or software used by the monitor to observe the environmental quantities is a major design decision with respect to the monitor system. Design of a general mechanism for observing target system behavior in a non-intrusive manner is beyond the scope of this work — readers interested in that topic are referred to [35] for a survey of the relevant literature. The following are some factors that should be taken into consideration in choosing monitor input devices.

Assuming that the monitor is a discrete-time system, there are two basic approaches to observing behavior:

- Sample (i.e., observe the instantaneous value of) the relevant quantities at intervals.
- Modify the behavior of the target system, and/or the systems that interact with it, to have them notify the monitor system of the values of relevant quantities ($\underline{m}^t, \underline{c}^t, \underline{i}^t$ or $\underline{o}^t$) as they read or change them. Such notification is assumed to include a *timestamp* indicating the time at which the reported value was observed by the target system.

1) Discrete Time: Regardless of whether sampling or notification is used, time can only be measured at discrete points: if sampling is used then the sampling period determines the smallest relevant clock increment, whereas if notification is used it is determined by the precision of the notification timestamp. If we assume that the monitor receives notifications for all relevant changes, then the notification approach is not significantly different from a sampling approach where the sampling period is the precision of the notification timestamp and uninteresting samples discarded. Thus, the results from sampling theory (e.g., see [29]) can be applied here to show that, for infinite duration signals (behaviors), it is sufficient to sample at twice the maximum frequency of change in the environmental quantities. However, the monitor is typically concerned with what has happened between the most recent two samples, and so the discrete clock will introduce some error in the perceived time of events, which is referred to as the *time error*. For real-time systems, errors in measuring time are particularly important.

Consider the behaviors illustrated in Figure 8, in which the values of m and c represent that a condition of, respectively, a monitored and controlled quantity is either $\underline{false}$ (low) or $\underline{true}$ (high). Similarly, the values of s_m and s_c represent the values as they appear to the software component of the monitor, and the shaded regions represent the image of these changes under

$\mathbf{IN_{mon}}^{-1}$ — from the point of view of the software all that is known is that the changes occurred at some time in the shaded regions. Let δ_{mon} represent the monitor sampling interval (i.e., $m_i - m_{i-1}$) and d be the elapsed time between the change in m and c, as illustrated. Assuming that the change in c is a correct target system response to the change in m, consider the two cases illustrated.

a) The monitor sees distinct changes. The monitor can determine only that $0 < d < 2\delta_{mon}$. This behavior will be rejected (considered unacceptable) if the specified maximum delay for that change is less than $2\delta_{mon}$. This results in Condition 1, below.

Condition 1: The maximum time error introduced by the monitor input devices for a particular event must be less than $\frac{1}{2}min(\mathbf{Delay})$, where $\mathbf{Delay}$ is the set of maximum delay tolerances for the dependent[2] quantities given in the SRD.

b) The monitor sees simultaneous changes. Here the monitor can determine that $-\delta_{mon} < d < \delta_{mon}$ (i.e., c could change before m); hence this behavior will be rejected if c is only permitted to change following m. The implication is that δ_{mon} must be less than the minimum response time of the target system. This constraint can be weakened, however, by noting that, in order for the target system to have responded to the change in m, it must have observed its value between the changes in m and c, so this case can be avoided by ensuring that the monitor samples in that interval as well. Thus we have Condition 2, below, which can be satisfied by ensuring that sampling by the target and monitor systems is synchronized to within the minimum target system response time. If event notification from the target system is used, the monitor and target systems are assured to be synchronized.

Condition 2: The maximum difference between the time error in the target system and the time error in the monitor system for the same event must be less than the minimum time in which the target system might respond to that event.

With respect to a particular event and the systems response to it, a monitor system that does not satisfy Condition 2 may give false negative results for target systems responding too quickly. A monitor system that satisfies Condition 2, but does not satisfy Condition 1, will consider all behaviors containing that event and response to be unacceptable, so it will be practically infeasible with respect to that event.

2) Quantization and Measurement Error: As with time, other values observed by the software component of the monitor must be of finite precision, so $\mathbf{Real}$ valued environmental quantities must be quantized, such that, for example, discrete value v_i represents all continuous values, x, such that $l_i < x \leq h_i$. Whereas time is continuously increasing, so we know something about the error, other quantities do not necessarily have this property. As illustrated in Figure 9, if the quantization is perfect, i.e., $h_i = l_{i+1}$, the worst case error is half the quantization step size, $h_i - l_i$, and no non-determinism is introduced. Practical devices will exhibit some measurement error in addition to quantization, so the actual error will be larger, and $\mathbf{IN_{mon}}$ will be a relation that is not a function.

[2]A quantity c is dependent on m if the value of c may be required to change as a result of a change in the value of m.

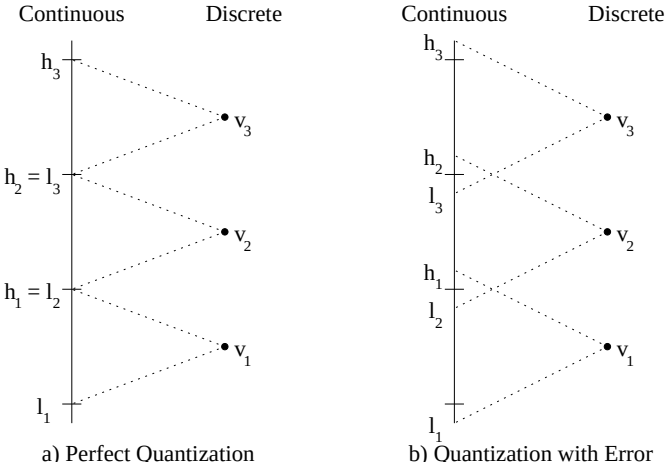


Fig. 9. Quantization and Error

For a monitor to be feasible, there must be some monitor input state functions, $\underline{s}^t$, for which all images under $\mathbf{IN_{mon}}^{-1}$ are acceptable. Because of the variety of ways that quantities may be used in the SRD, we cannot state generally applicable conditions on $\mathbf{IN_{mon}}$ that will ensure that a monitor is feasible. Condition 3 is a necessary, but not sufficient condition for feasibility.

Condition 3: The maximum error in observing a particular controlled quantity must be less than the difference between the maximum and minimum values of that quantity permitted by $\mathbf{REQ}$.

As an example, consider the digitizing tablet used by a Maze-tracing Robot system monitor, as mentioned above. If the tablet is such that the error in the perceived pen position is $\pm\epsilon$, then the monitor will be infeasible if $(\epsilon + \mathsf{WALL\_SPACE}) > \frac{1}{2}\mathsf{M\_CELL\_SIZE}$ since the pen could not touch the paper such that the monitor is sure it is not too close to a wall.

B. Non-determinism

As mentioned in Section III-A, practical requirements documents will specify relations that are not functions, so that they allow for unpredictable delays or errors in calculation or measurement. In particular, if the target system is to be implemented using a discrete-time system, then, for some small time, r, $\mathbf{REQ}$ must allow events that occur within r of each other to be treated as either a single event (i.e., simultaneous) or distinct events (i.e., non-simultaneous). The time r is known as the *time resolution* for the target system and must be stated as part of the requirements specification. The required time resolution will depend on the system being specified. For systems where simultaneous, or almost simultaneous, events may be treated individually in any order, r can be comparatively large, whereas systems that make a significant distinction between simultaneous and non-simultaneous events, or for which the order of events is important, will require a smaller value. The monitor system must take the non-determinism in $\mathbf{REQ}$ due to the time resolution into account when evaluating behavior.

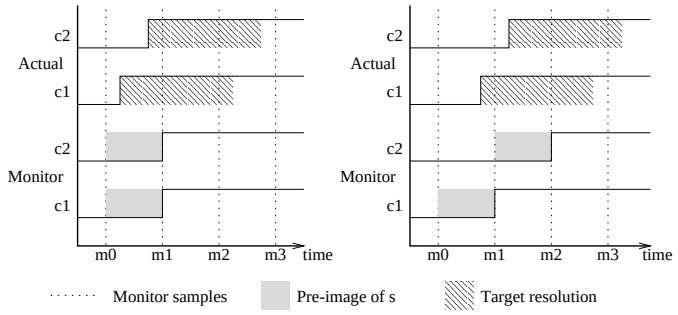


Fig. 10. Event Resolution

Consider the behavior illustrated in Figure 10, and the target time resolution as indicated — the hatched regions indicating the period following each event during which the requirements allow other events to be treated as simultaneous with it, but also allow them to be treated as non-simultaneous. Thus, the requirements allow the changes in C1 and C2 to be treated as either simultaneous or not in both cases illustrated. Assuming that the monitor system samples at the indicated times, it will observe the changes either simultaneously (left figure) or not (right figure), but can certainly tell that they occurred within $2\delta_{mon}$ of each other. If δ_{mon} is less than half the time resolution required for the target system, which is required to satisfy Condition 1, then in both cases all images of $\underline{s}^t$ under $\mathbf{IN_{mon}}^{-1}$ allow the changes to be interpreted as happening in either order or simultaneously, so $\mathbf{MON_{pe}}$ accepts a behavior in which the target system interprets them in either way. The monitor must take this non-determinism into account. In the case where the change in C2 is a result of the target system's response to the change in C1, then Condition 2 requires that a monitor sample occur between the changes in C1 and C2, i.e., the situation illustrated in right figure in Figure 10, so the events will be observed as non-simultaneous.

In the case of the software monitor configuration, as illustrated in Figure 5, the software components of the monitor and the target system are assured to see the same values (i.e., $\underline{s}^t = (\underline{i}^t, \underline{o}^t)$), so the monitor implementation can require deterministic behavior.

C. Response Time

Clearly the delay introduced by the monitor input devices will impose a lower limit on the monitor response time — the maximum time between a failure occurring and the monitor reporting it — since a monitor cannot report a failure before it is evident in $\underline{s}^t$. The choice of input devices can also affect the amount of processing required by the software component of the monitor, which will also affect response time, although less predictably so. For example, input devices may be available that can directly report the value of relevant conditions (e.g., sensors to detect if the Maze-tracing Robot pen has touched a wall) whereas a different choice of input devices would require that the software perform some, possibly expensive, calculations (e.g., search a list of wall locations to determine if the pen is touching any).

D. Computational Resources

Using any notation that is expressive enough to describe realistic target system requirements, it is certainly possible to express requirements such that $\mathrm{MON}_{\mathrm{pe}}(\underline{s}^t)$ is either not computable, or is computable only using an impractical amount of computational resources. Some possible causes of this are:

- $\mathrm{REQ}\,(\underline{m}^t, \underline{c}^t)$ or $\mathrm{NAT}\,(\underline{m}^t, \underline{c}^t)$ may not be practically computable. As in [26], this may result from specification errors such as infinite recursions in function or predicate definitions, or from computation of $\mathrm{MON}_{\mathrm{pe}}(\underline{s}^t)$ requiring quantification over large sets. Specification authors must take care to avoid these situations, if possible.
- $\mathrm{IN}_{\mathrm{mon}}$ may be such that the pre-image of $\underline{s}^t$ is not easily computed. Since real-valued monitored and controlled quantities are permitted, the pre-image of $\underline{s}^t$ will often be infinite, but, for most practical input devices, will be easily described by simple predicates, characterizing a range of possible values, for example. If this is not the case, however, it may be impractical to determine if all elements of the pre-image are acceptable.

Careful review of the SRD and judicious choice of monitor input devices may help to avoid these situations.

VI. Related Work

The design of a system to automatically generate software monitors from a formal system requirements document is discussed in [27].

This work does not address the challenges associated with gathering accurate and sufficiently precise information about the run-time behavior of a target system without changing its behavior. These issues are addressed in [10], [20], [21], [39], [35], among others.

In [5], Brockmeyer *et al.* discuss a tool for "monitoring and assertion-checking" as part of the Modechart toolset. The monitor in that work is an additional modechart state machine that is simulated concurrently with the target system specification to determine if the specification has certain critical properties. Since that monitor observes the behavior of the *specification* rather than the target system, it is not a monitor in the sense of this work, although it could possibly be used as a monitor if an appropriate interface with the target system were added.

Fickas and Feather [9] propose that *requirements monitors* be installed as components of systems. These monitors collect and report information about the run-time behavior of the system, which can be used to determine if it conforms with the requirements. They advocate this as a technique for gathering information about changing requirements or environmental conditions, and suggest that certain operating parameters could be automatically adjusted by the monitor. They propose that the monitor observe specific aspects of the behavior that are likely to indicate that assumptions about the environment are no longer valid.

Systems that address the oracle problem (see [3] for an excellent survey) can be classified by the subset of properties that they consider, most restricting their analysis to one of the following classes:

1) *Functional properties* — the values of the outputs for given inputs, which can be checked either by observing the start and stop states for a program, as in [28], [38], [19], [32]; by comparing the behavior of an abstract data type with that specified in a model-based specification notation, as in [37], [16], [22], [31]; or by comparing the run-time behavior of a reactive system with a finite state machine model, as in [36].
2) *Temporal properties* — the order of events, which can be checked by comparing the sequence of observed events with that specified by either a temporal logic [7], [8]; a finite state machine model [15], [6]; or a context-free grammar [2].
3) *Timing constraints* — the time elapsed between events, which can be expressed using a real-time logic [23], [17]; or as extensions on a finite state machine model [25], [34].

This paper contributes to this body of work in two ways: Firstly, it considers all three of these classes of properties, which we have only seen done in one other work [18]. Secondly, it considers system, rather than just software, monitors. That is, it views the monitor as an external device, which does not have access to the internal variables of the target system.

VII. Conclusions

This paper presents a precise definition of a monitor for a real-time system, and identifies some necessary conditions for a monitor to be feasible and useful. The conditions are not particularly surprising, and it would seem likely that they have been observed before, for example in relation to control theory, but we have not been able to find them formalized elsewhere. These conditions can be used to help determine if a particular monitor design is sufficient for the target system.

Monitors, such as described in this work, are well suited to automated testing of systems, where they function as an oracle, reporting if the behavior is acceptable or not. This application offers significant improvement over non-automated testing since test cases can be evaluated quickly and errors in behavior are quickly and reliably detected.

In a similar way, monitors can be used as supervisors to observe the behavior of the target system in operation and report failures as they occur. Such a supervisor could be used as a redundant safety system to initiate corrective or preventative action when a failure is detected.

A. Future Work

We have conducted some investigations using a few software monitors, including one for the Maze-tracing Robot system, that were automatically generated from system requirements documentation.[27] Further study, using different target systems and using the system monitor configuration discussed in Section IV-B would undoubtedly lead to new insight.

Further work is also needed to enhance techniques for specifying the behavior of input and output devices, and to develop analysis techniques that will permit designers to easily determine if a particular set of monitor input devices is sufficient for the monitoring task at hand.

Acknowledgments

Many friends and colleagues at McMaster University have helped to improve this work through informal discussions and helpful comments offered on earlier versions. In particular Drs. Ryszard Janicki, Jan Madey, Martin von Mohrenschildt, Emil Sekerinski and Jeffery Zucker have each offered helpful and constructive comments. Dr. von Mohrenschildt also collaborated in the development of the Maze Tracer system used as an example in this paper. Comments from the anonymous referees have also helped to improve this paper.

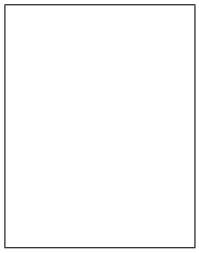
Constance Heitmeyer and her group at the US Naval Research Laboratory, Center for High Assurance Computer Systems, were helpful in the initial formulation of the problem statement.

The financial support received from the Natural Sciences and Engineering Research Council (NSERC), Communications and Information Technology Ontario, (CITO), the Telecommunications Research Institute of Ontario (TRIO), McMaster University and Memorial University of Newfoundland is gratefully acknowledged.

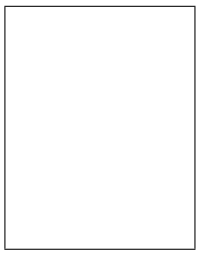
References

[1] B. Alpern and F. B. Schneider, "Defining Liveness," *Information Processing Letters*, vol. 21, pp. 181–185, Oct. 1985.
[2] M. Auguston and P. Fritzson, "PARFORMAN—An Assertions Language for Specifying Behavior When Debugging Parallel Applications," *Int'l J. of Software Engineering and Knowledge Engineering*, vol. 6, no. 4, pp. 609–640, 1996.
[3] L. Baresi and M. Young, "Test Oracles," Tech. Rep. CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Aug. 2001. Available at http://www.cs.uoregon.edu/~michal/pubs/oracles.html.
[4] J. P. Bowen and M. Hinchey, eds., *ZUM The Z Formal Specification Notation*, no. 260 in Lecture Notes in Computer Science, Springer, 1995.
[5] M. Brockmeyer, F. Jahanian, C. Heitmeyer, and B. Labaw, "An Approach to Monitoring and Assertion-Checking of Real Time Specifications in Modechart," in *Proc. Workshop on Parallel and Distributed Real-Time Systems*, pp. 236–243, Apr. 1996.
[6] M. Diaz, G. Juanole, and J. Courtiat, "Observer—A Concept for Formal On-Line Validation of Distributed Systems," *IEEE Trans. Software Engineering*, vol. 20, no. 12, pp. 900–912, Dec. 1994.
[7] L. K. Dillon and Y. S. Ramakrishna, "Generating Oracles from Your Fravorite Temporal Logic Specificaitons," in *Symposium on the Foundations of Software Engineering*, ACM SIGSOFT, Oct. 1996. published in Software Engineering Notes, vol. 21, no. 6.
[8] L. K. Dillon and Q. Yu, "Oracles for Checking Temporal Properties of Concurrent Systems," in *Symposium on the Foundations of Software Engineering*, pp. 140–153, ACM SIGSOFT, Dec. 1994. published in Software Engineering Notes, vol. 19, no. 5.
[9] S. Fickas and M. Feather, "Requirements Monitoring in Dynamic Environments," in *Proc. Int'l Symp. Requirements Eng. (RE '95)*, pp. 140–147, IEEE, Mar. 1995.
[10] C. Fidge, "Fundamentals of Distributed System Observation," *IEEE Software*, vol. 13, no. 6, pp. 77–83, Nov. 1996.
[11] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A Reference Model for Requirements and Specifications," *IEEE Software*, pp. 37–43, May/June 2000.
[12] C. L. Heitmeyer, A. Bull, C. Gasarch, and B. G. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," in *Proc. Conf. Computer Assurance (COMPASS)*, (Gaithersburg, MD), pp. 109–122, National Institute of Standards and Technology, June 1995.
[13] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and their Application," *IEEE Trans. Software Engineering*, vol. SE-6, no. 1, pp. 2–13, Jan. 1980.
[14] K. L. Heninger, D. L. Parnas, J. E. Shore, and J. Kallander, "Software Requirements for the A-7E Aircraft," Tech. Rep. MR 3876, Naval Research Laboratory, 1978.
[15] M. Hlady, R. Kovacevic, J. J. Li, B. R. Pekilis, D. Prairie, T. Savor, and R. E. Seviora, "An Approach to Automatic Detection of Software Failures," in *Proc. Int'l Symp. Software Reliability Eng. (ISSRE)*, pp. 314–323, IEEE Computer Society Press, Oct. 1995.

[16] H. Hörcher, "Improving Software Tests using Z Specifications," in Bowen and Hinchey [4], pp. 152–166.
[17] F. Jahanian, R. Rajkumar, and S. C. V. Raju, "Runtime Monitoring of Timing Constraints in Distributed Real-Time Systems," *Real-Time Systems*, vol. 7, no. 3, pp. 247–273, 1994.
[18] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime Assurance Based On Formal Specifications," in *Proc. of Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, June 1999.
[19] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe, *ANNA A Language for Annotating Ada Programs Reference Manual*. No. 260 in Lecture Notes in Computer Science, Springer-Verlag, 1987.
[20] M. Mansouri-Samani and M. Sloman, "Monitoring Distributed Systems (A Survey)," Research Report DOC92/23, Imperial College, Dept. of Computing, 180 Queen's Gate, London SW7 2BZ, UK, Apr. 1993.
[21] M. Mansouri-Samani and M. Sloman, "GEM: A Generalised Event Monitoring Language for Distributed Systems," Tech. Rep. DOC95/8, Imperial College, Dept. of Computing, 180 Queen's Gate, London SW7 2BZ, UK, July 1995.
[22] E. Mikk, "Compilation of Z Specifications into C for Automatic Test Result Evaluation," in Bowen and Hinchey [4], pp. 167–180.
[23] A. K. Mok and G. Liu, "Efficient Run-Time Monitoring of Timing Constraints," in RTAS '97 [33].
[24] D. L. Parnas and J. Madey, "Functional Documentation for Computer Systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, Oct. 1995.
[25] B. R. Pekilis and R. E. Seviora, "Detection of Response Time Failures fo Real-Time Software," in *Proc. Int'l Symp. Software Reliability Eng. (ISSRE)*, IEEE Computer Society Press, Nov. 1997.
[26] D. K. Peters, "Generating a Test Oracle from Program Documentation," M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, Apr. 1995.
[27] D. K. Peters, *Deriving Real-Time Monitors from System Requirements Documentation*. PhD thesis, McMaster University, Hamilton ON, Jan. 2000.
[28] D. K. Peters and D. L. Parnas, "Using Test Oracles Generated from Program Documentation," *IEEE Trans. Software Engineering*, vol. 24, no. 3, pp. 161–173, Mar. 1998.
[29] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing Principles, Algorithms and Applications*. Maxwell Macmillan, second ed., 1992.
[30] A. P. Ravn, H. Rischel, and K. M. Hansen, "Specifying and Verifying Requirements of Real-Time Systems," *IEEE Trans. Software Engineering*, vol. 19, no. 1, pp. 41–55, Jan. 1993.
[31] D. J. Richardson, S. L. Aha, and T. O. O'Malley, "Specification-based Test Oracles for Reactive Systems," in *Proc. Int'l Conf. Software Eng. (ICSE)*, pp. 105–118, May 1992.
[32] D. S. Rosenblum, "A Practical Approach to Programming With Assertions," *IEEE Trans. Software Engineering*, vol. 21, no. 1, pp. 19–31, Jan. 1995.
[33] *Real-Time Technology and Applications Symposium*, June 1997.
[34] T. Savor and R. E. Seviora, "An Approach to Automatic Detection of Software Failures in Real-Time Systems," in RTAS '97 [33].
[35] U. Schmid, "Monitoring Distributed Real-Time Systems," *Real-Time Systems*, vol. 7, no. 1, pp. 33–56, July 1994.
[36] D. Simser and R. E. Seviora, "Supervision of Real-Time Systems Using Optimistic Path Prediction and Rollbacks," in *Proc. Int'l Symp. Software Reliability Eng. (ISSRE)*, pp. 340–349, IEEE Computer Society Press, Oct. 1996.
[37] P. Stocks and D. Carrington, "Test Template Framework: A Specification-Based Testing Case Study," in *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '93)*, pp. 11–18, ACM SIGSOFT Software Engeering Notes, vol. 18, no. 3, June 1993.
[38] Sun Microsystems Inc., *ADL Language Reference Manual for ANSI C programmers, Release 1.1*, document reference miti/0002/d/r1.1 ed., Dec. 1996.
[39] J. J. Tsai, Y. Bi, S. J. H. Yang, and R. A. W. Smith, eds., *Distributed Real-Time Systems : Monitoring, Visualization, Debugging, and Analysis*. John Wiley & Sons, 1996.
[40] J. J. Tsai and S. J. Yang, eds., *Monitoring and Debugging of Distributed Real-Time Systems*. IEEE Computer Society Press, 1995.
[41] A. J. van Schouwen, "The A-7 Requirements Model: Re-examination for Real-Time Systems and An Application to Monitoring Systems," Tech. Rep. TR 90-276, Queen's University, Kingston, Ontario, 1990. also printed as CRL Report No. 242, Telecommunications Research Institute of Ontario (TRIO).
[42] A. J. van Schouwen, D. L. Parnas, and J. Madey, "Documentation of Requirements for Computer Systems," in *Proc. Int'l Symp. Requirements Eng. (RE '93)*, pp. 198–207, IEEE, Jan. 1993.
[43] E. J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.
[44] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 1, pp. 1–30, Jan. 1997.



Dennis K. Peters received the BEng (electrical) degree from Memorial University of Newfoundland in 1990. He received the MEng degree (computer) and the PhD degree from the Department of Electrical and Computer Engineering at McMaster University, Ontario, in 1995 and 2000, respectively. He is an assistant professor of Electrical and Computer Engineering at Memorial University of Newfoundland, where he has been since 1998. He is currently researching documentation, design and analysis techniques for software and computer systems, with particular interests in real-time systems. He also has a strong interest in software engineering education. He is a licensed Professional Engineer in the province of Newfoundland and a member of the IEEE, IEEE Computer Society, ACM, and ACM SIGSOFT.



David Lorge Parnas received his PhD in electrical engineering from Carnegie Mellon University, and honorary doctorates from the ETH in Zurich, Switzerland, and the Catholic University of Louvain, Belgium. Dr. Parnas is a professor in the Faculty of Engineering Computing and Software Department at McMaster University, Ontario, where he is Director of the Software Engineering Program; he is also an associate member of the Department of Electrical and Computer Engineering. The author of more than 190 papers and reports, Dr. Parnas is interested in most aspects of computer system design. Dr. Parnas won an ACM Best Paper Award in 1979, and two Most Influential Paper awards from the International Conference on Software Engineering. He is the 1998 winner of ACM SIGSOFT's Outstanding Research award. He is licensed as a Professional Engineer in the province of Ontario. Dr. Parnas is a fellow of the Royal Society of Canada, a fellow of the ACM, a senior member of the IEEE, and a member of the IEEE Computer Society.