Deriving Real-Time Monitors from System Requirements Documentation

By
Dennis K. Peters, B. Eng., M. Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Doctor of Philosophy

McMaster University

DOCTOR OF PHILOSOPHY (2000) McMaster University
(Electrical and Computer Engineering) Hamilton, Ontario

TITLE: Deriving Real-Time Monitors from System Requirements Documentation

AUTHOR: Dennis K. Peters, B. Eng. (Memorial University of Newfoundland),
M. Eng. (McMaster University)

SUPERVISOR: Dr. David L. Parnas

NUMBER OF PAGES: xiv, 157

Abstract

When designing safety- or mission-critical real-time systems, a specification of the required behaviour of the system should be produced and reviewed by domain experts. Also, after the system has been implemented, it should be thoroughly tested to ensure that it behaves correctly. This, however, can be difficult if the requirements are complex or involve strict time constraints. A *monitor* is a system that observes the behaviour of a target system and reports if that behaviour is consistent with the requirements. Such a monitor can be used as an oracle during testing or as a supervisor during operation. This thesis presents a technique and tool for generating software for such a monitor from a system requirements document.

A system requirements documentation technique, based on [102], is presented, in which the required system behaviour is described in terms of the environmental quantities that the system is required to observe and control, which are modelled as functions of time. The relevant history of these quantities is abstracted as the initial conditions and a sequence of events. The required value of all controlled quantities is specified, possibly using *modes*—equivalence classes of histories—to simplify the presentation. Deviations from the ideal behaviour are described using either tolerance or accuracy functions.

The monitor will be affected by the limitations of the devices it uses to observe the environmental quantities, resulting in the potential for false negative or positive reports. The conditions under which these occur are discussed.

The generation of monitor software from the requirements documentation for a realistic system is presented. This monitor is used to test an implementation of the system, and is able to detect errors in the behaviour that were not detected by previous testing. For this example the time required for the monitor software to evaluate the behaviour is less than the interval between events.

Acknowledgements

I sincerely thank my supervisor, Dr. David L. Parnas, for his thoughtful guidance, constructive criticism and mentorship throughout my many years under his tutelage. In addition I am indebted to my supervisory committee, Drs. Ryszard Janicki, Martin von Mohrenschildt, Emil Sekerinski and Jeffery Zucker, who have each taken the time to offer suggestions and guidance to help to improve the quality of this work. Dr. von Mohrenschildt collaborated with me on the initial versions of the specification in Section A.4, and was the instructor responsible for the systems tested in Chapter 7.[81, 104]

Many friends and colleagues who have been associated with the Software Engineering Research Group, either as faculty, staff, students or visiting scholars, have offered support and suggestions that have undoubtedly contributed to this work. In particular, Dr. Jan Bredereke helped to clarify the event class notation and offered many helpful comments on the specifications in Appendix A. The Monitor Generator tool stands on the shoulders of the many people who have worked on the Table Tool System, to whom I'm very grateful.

Constance Heitmeyer and her group at the US Naval Research Laboratory, Center for High Assurance Computer Systems, were instrumental 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) and McMaster University is gratefully acknowledged.

Finally, over the years it has taken to complete this thesis, Ruth Abraham has played many roles: fellow graduate student, project engineer, consultant, reviewer, friend, confidant and, most importantly, wife. Without her unending support it would certainly not have been possible.

Contents

Abstract iii

Acknowledgements iv

List of Acronyms xiv

1 Introduction 1
- 1.1 The Four Variable Requirements Model . . . 2
  - 1.1.1 System Requirements . . . 3
  - 1.1.2 Environmental Constraints . . . 4
  - 1.1.3 System Design . . . 4
  - 1.1.4 Software Requirements . . . 5
- 1.2 Scope . . . 6
- 1.3 Notation and Terminology . . . 7
  - 1.3.1 Predicates and Relations . . . 7

2 Related Work 9
- 2.1 System Requirements Specification . . . 9
  - 2.1.1 Predicate Logic Based Methods . . . 10
  - 2.1.2 Process Algebras . . . 10
  - 2.1.3 Automata Based Methods . . . 11
  - 2.1.4 SCR . . . 12
- 2.2 Real-Time System Monitoring . . . 13

3 Monitors for Real Time Systems 15
- 3.1 Using Monitors . . . 15

3.1.1 Non-Testable Requirements 16
3.2 Monitor Configuration 16
3.3 Accuracy 20

4 Practical Monitors **22**
4.1 Observation Errors 22
4.1.1 Discrete Time 23
4.1.2 Quantization and Measurement Error 25
4.2 Non-determinism 26
4.3 Response Time 27
4.4 Computational Resources 27

5 Specifying System Requirements **29**
5.1 Assumptions and Definitions 30
5.1.1 Environmental Quantities 30
5.1.2 Conditions 31
5.1.3 Events 32
5.1.4 History 32
5.1.5 Modes and Mode Classes 33
5.1.6 Tolerance and Accuracy 34
5.2 Specification Notation 36
5.2.1 Document Sections 37
5.2.2 Monitored and Controlled Quantities 39
5.2.3 Tabular Expressions 40
5.2.4 Conditions 43
5.2.5 Event Classes 45
5.2.6 Mode Classes 47
5.2.7 Controlled Value Relations 49
5.2.8 Merit Functions 50
5.2.9 Environmental Constraints 52
5.2.10 Tolerance and Accuracy Relations 53
5.2.11 Standard Functions 58
5.3 Limitations 60
5.3.1 Environmental Quantities 60

5.3.2 Requirements Relation 61

6 Monitor Generation 63

6.1 Input Format 63
6.2 Monitor Modularization 65
6.3 System Behaviour Module Components 65
6.3.1 Monitor Library 65
6.3.2 Behaviour Module 68
6.4 Code Generation 71
6.4.1 TTS Modifications 71
6.5 Monitor Generator Modules 73
6.5.1 Monitor Constructor 73
6.5.2 Requirements 74
6.5.3 Specification Objects 74
6.5.4 TOG Interface 75
6.5.5 Exceptions 76
6.6 Limitations 76
6.6.1 Real-Time Evaluation 77

7 Sample Application 78

7.1 Monitor Implementation 78
7.1.1 Monitor Control 78
7.1.2 System Interface 79
7.1.3 Behaviour Module 82
7.2 Discussion 83
7.2.1 Monitor Execution Time 84
7.2.2 Alternative System Interface Design 85

8 Conclusions 87

8.1 Contributions 87
8.2 Applicability of This Work 88
8.3 Future Work 89
8.3.1 Monitor Generator and Library Enhancements 89
8.3.2 Verification 91

A Example Requirements Documents **93**
A.1 Notational Conventions 93
A.2 Generalized Railroad Crossing 94
A.2.1 Preliminaries 94
A.2.2 Environmental Quantities 95
A.2.3 Mode Classes 96
A.2.4 Controlled Value Functions 96
A.2.5 Environmental Constraints 96
A.2.6 Dictionary 97
A.3 Steam-Boiler Control 98
A.3.1 Preliminaries 98
A.3.2 Environmental Quantities 99
A.3.3 Mode Classes 100
A.3.4 Controlled Value Functions 101
A.3.5 Environmental Constraints 102
A.3.6 Dictionary 102
A.4 Maze-tracing Robot 105
A.4.1 Preliminaries 105
A.4.2 Environmental Quantities 110
A.4.3 Mode Classes 111
A.4.4 Controlled Value Functions 112
A.4.5 Environmental Constraints 113
A.4.6 Dictionary 114

B Monitor Generator Users' Guide **118**
B.1 SRD Format 118
B.1.1 Environmental Variables 118
B.1.2 Conditions 119
B.1.3 Histories 119
B.1.4 Mode Classes 120
B.1.5 Behaviour 120
B.1.6 Dictionary 120
B.2 Users' Interface 123

C Monitor Generator Design Documentation 124
C.1 Monitor Generator Interface 124
C.1.1 Requirements 124
C.1.2 Monitor Constructor 125
C.2 Monitor Library Interface 126
C.2.1 Timestamp 128
C.2.2 Time Function 132
C.2.3 Condition 139
C.2.4 Mode Class 144

List of Figures

1.1 System Design 5

3.1 Software Monitor 17
3.2 System Monitor 18

4.1 Time Accuracy 24
4.2 Quantization and Error 25
4.3 Event Resolution 27

5.1 Logic Probe System Overview 37
5.2 Logic Probe Environmental Quantities 40
5.3 Example Tabular Expression 41
5.4 Table Layout and Numbering Conventions 43
5.5 Logic Probe Conditions 44
5.6 Logic Probe Mode Class—Direct Definition 48
5.7 Logic Probe Mode Class—FSA Definition 49
5.8 Logic Probe Controlled Values 52
5.9 Logic Probe Mode Classes, with Delay Specification 55
5.10 Logic Probe Complete Mode Classes 56
5.11 Time Resolution 58

A.1 Robot and Maze Parameters 107
A.2 Sample Maze 108

C.1 Monitor Generator Class Diagram 125
C.2 Requirements Object Constructor (`MG_Req(ch)`) Algorithm 126
C.3 Monitor Generation (`MG_monBuild`) Algorithm 127

C.4 Monitor Library Class Diagram 127

List of Tables

1.1 Fonts and Notational Conventions 8
1.2 Operations on Binary Relations 8

5.1 Notational Conventions 39
5.2 Cell Connection Graph Cases 42
5.3 Event Class Notation 46
5.4 Mode Transition Table 1 49
5.5 Mode Transition Table 2 50
5.6 Mode Transition Table 3 50
5.7 Mode Transition Table 3, all cells 51
5.8 Possible Simultaneous Changes in Conditions 53

6.1 Condition Module Programs 66
6.2 Mode Class Members 67
6.3 Mode Class Parameter Members 68
6.4 Time Function Members 69
6.5 Time Stamp Members 70
6.6 Behaviour Module Programs 70
6.7 Table Evaluation Module Members 72
6.8 TOG_Code Changes 73
6.9 Requirements Members 74
6.10 Specification Object Members 75

7.1 Robot Interface Programs 80
7.2 Maze Class Members 81
7.3 Geometry Class Members 81

7.4 Position Class Members 82
7.5 Position Operations 82
7.6 Maze-Tracer Test Results 84
7.7 Maze-Tracer Monitor Timing 85

A.1 Notational Conventions 93
A.2 Railroad Crossing Environmental Quantities 95
A.3 Steam-boiler Environmental Quantities 99
A.4 Summary of Functions 103
A.5 Maze-tracer Robot Environmental Quantities 110
A.6 Constants 117

B.1 Maze-Tracer SRD TTS Context File 121

List of Acronyms

Acronym	Description	Page (definition)
CCG	Cell Connection Graph	40
CCS	Calculus of Communicating Systems	10
CSP	Communicating Sequential Processes	10
FSA	Finite State Automaton	11
GRC	Generalized Railroad Crossing	94
GTS	Generalized Table Semantics	71
IC	Integrated Circuit	37
IDP	Inductively Defined Predicate	77
LCD	Liquid Crystal Display	37
MG	Monitor Generator	63
NRL	Naval Research Laboratory	12
RTL	Real-Time Logic	10
RML	Requirements Modelling Language	10
RSML	Requirements State Machine Language	12
SBC	Steam-Boiler Control	98
SCR	Software Cost Reduction	12
SRD	System Requirements Document	7
STL	Standard Template Library	83
TOG	Test Oracle Generator	71
TTL	Transistor-Transistor Logic	37
TTS	Table Tool System	63
UML	Unified Modelling Language	124

Chapter 1

Introduction

Computer systems are increasingly being used in situations where their correct behaviour is essential for the safety of people, equipment, the environment and businesses. In many cases there are *real-time* requirements on the behaviour 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 behaviour 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 behavioural requirements documentation:

- expresses the required behaviour 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.[45]

Also, after the system has been implemented, it should be tested to ensure that its behaviour satisfies the requirements. In safety-critical applications the system should be constantly observed by an independent safety system to ensure continued correct behaviour. To achieve these goals there must be a means of quickly determining if the observed behaviour is acceptable or not, which can be quite difficult

for complex real-time systems. Several authors (e.g., see [100]) have suggested that a practical approach to analysing the behaviour of a real-time system is to use a *monitor*: a system that observes and analyses the behaviour of another system (the *target system*). This work investigates techniques for using the system requirements documentation to generate a monitor automatically. Such a monitor could be used either as an 'oracle'[106] during system testing, or as a 'supervisor'[95] to detect and report system failure during operation.

1.1 The Four Variable Requirements Model

As pointed out, e.g. in [102, 103, 108], it is important when specifying system and software requirements to distinguish 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. The "Four Variable Model", introduced in [77, 102, 103], addresses this issue and is adopted here as a framework for describing requirements. According to this model, 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. (The requirements for the *software* can be expressed in terms of variables internal to the system, see Section 1.1.4.) 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 settings of controlled devices.

Also, it is widely accepted (e.g., see [46, 77, 85, 103]) that environmental quantities can be modelled 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 behaviour 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

behaviour of the system.[1] Assuming that the monitored quantities are $q_1, q_2, \ldots, q_i$, the *monitored state function*, $\underline{m}^t$: **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, if the controlled quantities are $q_j, q_{j+1}, \ldots, q_n$, the *controlled state function*, $\underline{c}^t$: **Real** $\rightarrow \mathbf{Q_j} \times \mathbf{Q_{j+1}} \times \ldots \times \mathbf{Q_n}$, is derived. In this thesis, the pair of functions $(\underline{m}^t, \underline{c}^t)$ will be used to denote an environmental state function. With respect to a particular target system, **M** denotes the set of functions of type **Real** $\rightarrow \mathbf{Q_1} \times \mathbf{Q_2} \times \ldots \times \mathbf{Q_i}$, (type correct for a monitored state function) and **C** denotes the set of functions of type **Real** $\rightarrow \mathbf{Q_j} \times \mathbf{Q_{j+1}} \times \ldots \times \mathbf{Q_n}$ (type correct for a controlled state function).

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 *behaviour* of the system. A behaviour is *acceptable* if it describes a situation in which the system is operating correctly.

1.1.1 System Requirements

The *system behavioural requirements* (or, where the meaning is clear from context, *system requirements*) characterize the set of acceptable behaviours. 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, **REQ** $\subseteq \mathbf{M} \times \mathbf{C}$. A behaviour is acceptable if and only if 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, **REQ** will not be functional for real systems, i.e., there will be more than one acceptable $\underline{c}^t$ for a given $\underline{m}^t$.

In many cases **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 behaviours can be represented by the behaviour 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

[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.

time and appropriate functions defined to determine the needed quantities.

1.1.2 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.

1.1.3 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

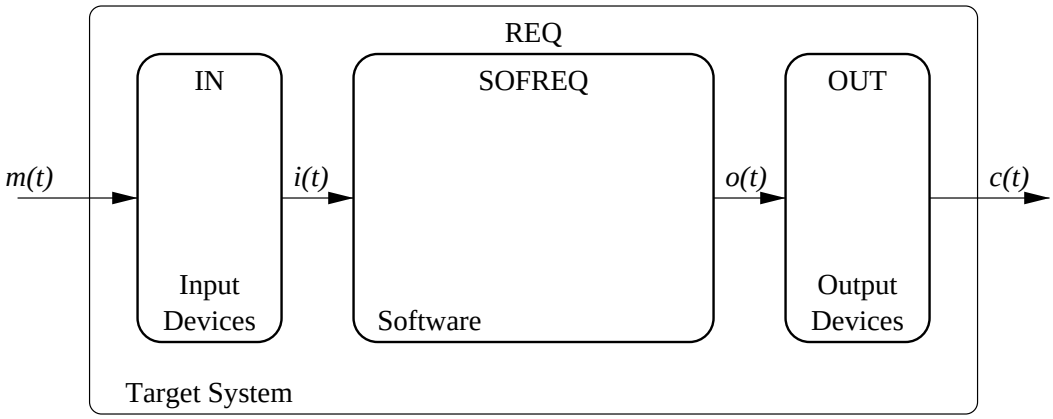

Figure 1.1: System Design

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 behaviour 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 $\underline{o}^t$. This is illustrated in Figure 1.1.

1.1.4 Software Requirements

In [77] the actual software behaviour is described by the *software behaviour relation*, **SOF**, and an expression is given for software acceptability. In this work, as in [39], we are interested in characterizing all acceptable software, so we use the *software requirements relation*, **SOFREQ**, which characterizes the set of acceptable behaviours of the software. This is fully determined by **REQ**, **IN**, **OUT** and **NAT**, as follows

$$\mathbf{SOFREQ} \stackrel{\text{df}}{=} \left\{ \left(\underline{i}^t, \underline{o}^t\right) \mid \left(\forall \underline{m}^t, \underline{c}^t, \begin{pmatrix} \text{IN}\left(\underline{m}^t, \underline{i}^t\right) \wedge \\ \text{OUT}\left(\underline{o}^t, \underline{c}^t\right) \wedge \\ \text{NAT}\left(\underline{m}^t, \underline{c}^t\right) \end{pmatrix} \Rightarrow \text{REQ}\left(\underline{m}^t, \underline{c}^t\right) \right) \right\} \quad (1.1)$$

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 behaviour must be in **NAT**, since, by definition, behaviours not in **NAT** are not possible.

1.2 Scope

This thesis reports the results of an investigation of techniques for using a reviewable form of system behavioural requirements documentation to generate a monitor that will determine if the observed behaviour of a system is consistent with the requirements documentation. It demonstrates the feasibility of producing such a monitor automatically from documentation written in a notation that is both readable and expressive enough to describe realistic system requirements. The following questions are addressed:

1. Under what conditions can an effective monitor be produced from a relational requirements document? What restrictions on the form and content of the documentation must be imposed?

2. What are the useful classes of behavioural properties that can and cannot be:

   (a) specified in relational documentation of this form?

   (b) verified using a monitor derived from the documentation?

3. Under what conditions will such a monitor give useful results?

The term *system* is used to emphasize that this work addresses specifying and monitoring the behaviour of any physical entity. The techniques are not limited to purely software systems (for which they are probably less than ideal) or even to systems based on digital computers. In addition this thesis is concerned only with the behavioural requirements for the system, and does not consider other attributes such as maintainability or cost.

Many of the decisions required when designing a monitor are not influenced by the behavioural requirements, but have to do with the environment in which the monitor is to be used. This work does not present a general notation for describing these decisions, nor does it attempt to automatically generate those parts of the monitor that are dependent on them—only those parts of the monitor that are directly dependent on the system behavioural requirements are generated automatically.

The term *verification* is often used to refer to the process of using a formal proof system to show that a particular description of a system has, or does not have,

certain properties. Although it is the intent that such analysis could be done based on documentation written using the techniqes presented in Chapter 5, this thesis does not present such a formal proof system.

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

Of course, as discussed further in Section 3.2, in some configurations these two systems may share components.

To help simplify the text, font faces and notational conventions are used. These are illustrated in Table 1.1. The symbol "$\stackrel{\text{df}}{=}$" is used to represent "is defined as", so, for example "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:

$$\begin{aligned}
[x, y] &= \{z \in \mathbf{Real} \mid x \leq z \leq y\} \\
(x, y) &= \{z \in \mathbf{Real} \mid x < z < y\} \\
(x, y] &= \{z \in \mathbf{Real} \mid x < z \leq y\} \\
[x, y) &= \{z \in \mathbf{Real} \mid x \leq z < y\}
\end{aligned}$$

This bracketing notation is extended to ranges of fixed-precision numbers by replacing "," with "...", so, for example, $[0.0 \ldots 0.7] = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$.

1.3.1 Predicates and Relations

For a set, $\mathbf{R}$, the *characteristic predicate*, R, is the predicate such that $\mathrm{R}(x)$ is *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 $\mathbf{R}$.[76] Table 1.2 gives notation used for standard operations on binary relations.

Table 1.1: Fonts and Notational Conventions

Item	Notation	Example
function or predicate	math roman	func
standard function	italicised	*kfunc*
sequence	bold	**seq**
set	math bold, capital	$\mathbf{R}$
characteristic predicate of $\mathbf{R}$	set name as predicate	$\mathrm{R}(x)$
tuple	bracketed	(x, y)
named element of tuple v	$\mathsf{v}.name$	$\mathsf{v.x}$
vector function of time	underlined, superscript t.	$\underline{m}^t$

Table 1.2: Operations on Binary Relations

Operation	Definition
Domain	$domain(\mathbf{R}) \stackrel{\text{df}}{=} \{x \mid \exists y, \mathrm{R}(x, y)\}$
Range	$range(\mathbf{R}) \stackrel{\text{df}}{=} \{y \mid \exists x, \mathrm{R}(x, y)\}$
Inverse	$\mathbf{R}^{-1} \stackrel{\text{df}}{=} \{(x, y) \mid \mathrm{R}(y, x)\}$
Composition	$\mathbf{R_1} \circ \mathbf{R_2} \stackrel{\text{df}}{=} \{(x, y) \mid \exists z, \mathrm{R}_1(x, z) \wedge \mathrm{R}_2(z, y)\}$

Chapter 2

Related Work

Two areas of research are most closely related to this work: specifying system requirements and monitoring real-time systems. Some of the most relevant work in these areas is as follows.

2.1 System Requirements Specification

The process through which the system requirements document (SRD) is produced, often called *requirements analysis*, has been the focus of much of the research in this area. As a result, several techniques and notations have been proposed to assist with this analysis, including data flow diagrams, structured analysis[20], object-oriented analysis[15], use cases and scenarios.[19] Documents written using these techniques are useful for communicating with customers and for gaining an understanding of the problem, but they do not describe exactly the set of acceptable system behaviours, so they are not behavioural requirements documents in the sense of this work.

Informal specification techniques, such as structured English or graphical notations that do not have precisely defined behavioural semantics, also do not meet the needs of this work since they cannot be used to unambiguously determine if any system behaviour is or is not acceptable.

There are a number of good collections that illustrate and compare the various formal requirements specification techniques (e.g., [4, 8, 43]), so that work will not be repeated here except to highlight a few of the most relevant techniques.

2.1.1 Predicate Logic Based Methods

The specification technique used in this work is a logic based technique—the SRD expresses, in some logic, the characteristic predicate of the set of acceptable behaviours. There are a number of other such techniques, which differ primarily in the form of logical notation used and the structure of the documents. For example, in Albert II[25, 26, 27] the target system is described graphically as a collection of co-operating *agents*, the behaviour of which is specified using a variation of Real-Time Temporal Logic.[72] In Real-Time Logic (RTL)[9, 52] the behaviour is described in terms of *events* and *actions* using a notation that, like this work, is based on [6, 46], but does not use tabular notations, so it is less readable. Requirements Modelling Language (RML)[34] describes the environment in terms of *entities* and *activities*, which represent, respectively, the physical objects and events of the system in its environment, and *assertions*, which describe behaviour and are expressed using standard logic with some extensions.

Temporal logics introduce special operators to denote that a condition should be true always ($\Box$) or eventually ($\Diamond$). These have been widely studied as a means of describing the temporal behaviour of computer systems that do not have real-time requirements.[28] Several authors have proposed adding bounds to these operators for specifying real-time behaviour (i.e., to assert that a condition should become true within some fixed time period).[8, 22, 58, 72, 85] The temporal operators do not, however, increase the expressiveness of a logic since "always" and "eventually" can be expressed simply as quantification over time ($\forall t$, and $\exists t$, respectively).

2.1.2 Process Algebras

A class of techniques that has received considerable attention, in particular with respect to concurrent systems, is process algebras, such as Calculus of Communicating Systems (CCS)[66, 67], Communicating Sequential Processes (CSP)[47] and COSY[55]. In these techniques the behaviour of the target system and its environment are described as a set of "processes" that participate in sequences of atomic "events." Interaction between processes takes the form of shared events. Extensions to these techniques have been proposed for real-time systems (e.g., Timed CSP[87] and Temporal CCS[69]).

These techniques are not intended for specifying system requirements, in the sense of this work, but rather for abstractly modelling designs so that they can be analysed. The notation, therefore, does not lend itself to review by domain experts.

2.1.3 Automata Based Methods

Most of the 'popular' formal specification techniques (e.g., Hybrid Automata[7, 60], ASTRAL[17], Statecharts[35], LOTOS[49], Petri Nets[61], PAISLey[107] and SDL[50]) are based on automata theory. They model the target system and its environment as one or more finite state automata (FSA), and describe the intended behaviour of the system in terms of operations on that model. While this approach is suitable for describing a system design, it is less than ideal for describing requirements for the following reasons.

- The operational approach does not lend itself well to non-determinism so, some acceptable behaviours are eliminated.
- Precise definition of operational semantics requires a constructive (typically recursive) algorithm, which complicates review (e.g., see [82]).
- Constructing the model requires internal elements (e.g., states, queues), which are not part of the environment, and hence not part of the requirements. These tend to make the requirements less natural to the domain experts, and hence more difficult to review.
- The finite model limits these techniques to discrete event systems—they cannot express continuous behaviour.
- The model implies a design, and hence biases the implementation.

One such technique that has received considerable attention for requirements specification of real-time systems is Statecharts.[35, 36] It extends traditional FSA with nested states (i.e., states can contain other FSA), parallel (AND) or choice (OR) composition of state machines and implicit inter-component communication. Real-time requirements are described using an implicit clock variable and timeout events. The statecharts notation allows relatively complex systems to be described using multiple levels of nesting so that the result is still understandable and visually appealing.

Several other techniques have adapted the nested state notation from statecharts, for example RSML[37, 59] and Modecharts[51]. In addition it has been suggested (e.g., [97]) that Statecharts could be combined with temporal logic to increase its expressive power.

2.1.4 SCR

The "Software Cost Reduction" (SCR) approach[41], which originated in a project at the US Naval Research Laboratory (NRL) to specify the requirements for the operational flight program of the A-7E aircraft[6, 46], is a forerunner of the approach presented in this thesis. In SCR, the behaviour of the system is described by a set of mode classes, which are concurrently executing finite state automata in which each state corresponds to a system mode. The transitions in the FSA are triggered by *events*—changes in the value of variables, modes or terms. The value of each controlled variable is specified by a function defined in terms of the system modes, monitored variables and terms. The use of SCR for hybrid systems is discussed in [38].

This approach has been shown to be effective for a number of real examples (e.g. [13, 44, 45, 57, 105]) and to satisfy some industrial expectations of requirements documentation.[31] A tool-set for specifying and analysing requirements documents that are written using the NRL version of this approach has been developed [39]. This tool-set does not support monitor generation.

The SCR method differs from the requirements specification technique used in this work in two significant ways:

1. SCR defines *events* as changes in the value of conditions, whereas we define events as instants when one or more conditions change value, together with the status of all conditions at that time, which avoids the need for special "conditioned events" and simplifies specification of requirements in the presence of so called "simultaneous events".

2. SCR defines a *mode* as a state of a FSA, whereas we define it as an equivalence class of system histories. This avoids the problems associated with operational semantics, such as eliminating non-determinism and needing to define mode transitions that are triggered by other transitions.

2.2 Real-Time System Monitoring

Much of the work related to monitoring of real-time systems addresses the challenges associated with gathering accurate and sufficiently precise information about the run-time behaviour of the target system without changing its behaviour (e.g., [33, 62, 63, 99, 92]). This work does not address those problems, except to discuss in Sections 3.3 and 4.1 how precision and accuracy effect the conclusions that can be drawn using the monitor.

In [16] 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 behaviour 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 [32] propose that *requirements monitors* be installed as components of systems. These monitors collect and report information about the run-time behaviour 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 behaviour that are likely to indicate that assumptions about the environment are no longer valid. They do not discuss derivation of the monitor directly from the requirements specification.

Jahanian *et al.*[53] and Mok and Liu [68] describe techniques for detecting if a system violates timing assertions—restrictions on the time that may elapse between particular events—expressed as RTL formulae. They do not monitor for "functional" constraints.

Similarly, Auguston and Fritzson[12] describe a notation called PARFORMAN, which is based on formal grammars and can be used to describe desired or undesired sequences of events. They advocate that actual behaviour be compared against assertions written in PARFORMAN to determine if the system is behaving correctly. These assertions are not the system requirements documentation. Also, formal gram-

mar based specification techniques have the same limitations as automata based techniques, as discussed above.

Other authors have presented techniques for comparing behaviour of real-time systems with a specification. Bochmann, Dssouli and Zhao [14], for example, discuss the automatic derivation of a "trace analysis module" which functions as a monitor for some component taking part in a communication protocol. The specification forms used in that work are finite state machine based models (e.g., written in SDL or Estelle), so the monitor is implemented by 'executing' the specification on the observed trace. In a similar manner Diaz *et al.*[21], and Seviora *et al.*[78, 91, 95] discuss deriving monitors (apparently not automatically, however) from specifications written, respectively, using Petri Nets and SDL. All of these groups use model-based techniques for specifying behaviour, which, as discussed above, we find to be less than ideal for requirements specification. In addition, since our specification can be interpreted as the characteristic predicate of all acceptable behaviours, we do not have to execute the specification and hence we avoid the problems caused by non-determinism of the specification, which are central to the work of Sevioura and Bochmann[14, 78].

In much of the literature, the term *test oracle* is used to refer to what we call a monitor. In [23, 24], Dillon *et al.* illustrate a technique for generating oracles from temporal specifications expressed in a variety of temporal logics. They do not, however, support the bounded forms of the temporal operators in these logics, so they cannot express real-time properties.

In [89], Richardson *et al.* discuss procedures for deriving test oracles from formal specifications of reactive systems (i.e., systems that are "largely event-driven and continuously reacting to external stimuli and internal events"). Automatic derivation of oracles from specifications written in Graphical Interval Logic is discussed in [70], and [88] discusses tools to support testing using oracles. The specifications used in that work are design specifications, and the implementation variables and states are mapped onto the corresponding specification entities, which requires that such a mapping exist. Since in this work we generate monitors from system requirements, all expressions are in terms of environmental quantities so we do not make any such assumption about the implementation.

Chapter 3

Monitors for Real Time Systems

Testing a real-time system typically involves running the target system in a test environment, observing its behaviour 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 behaviour of a system and determines if it is consistent with a given specification. That is, an ideal monitor reports the value of REQ $(\underline{m}^t, \underline{c}^t)$.

3.1 Using Monitors

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

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

is also a behaviour of the system. Thus, if $\left(\hat{\underline{m}}^t, \hat{\underline{c}}^t\right) \notin \mathbf{REQ}$ the system has behaved unacceptably and the monitor should report a failure.

This interpretation restricts these techniques to what [5] calls *safety properties*—if a behaviour is unacceptable then no extension of that behaviour is acceptable. Once

a monitor has detected a failure, no further analysis of that behaviour will give a different result. In applications such as supervision[95], where continued analysis of the behaviour is needed following detection of a failure, the monitor, and presumably the target system, will need to be restarted (i.e., a new behaviour begins).

3.1.1 Non-Testable Requirements

In [5], safety properties are distinguished from *liveness properties*—those requirements such that, for a given requirement and any finite duration behaviour, the behaviour 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 behaviour (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 infinite behaviours (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 behaviours (e.g., the system must respond to requests within a fixed time limit), which can be checked by a monitor.

3.2 Monitor Configuration

In this thesis, the monitor is assumed to consist of some software running on a computer system. The monitor 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 behaviour of the monitor input devices is characterized by the monitor input relation, $\mathbf{IN_{mon}} \subseteq (\mathbf{M} \times \mathbf{C}) \times \mathbf{S}$. $((\underline{m}^t, \underline{c}^t), \underline{s}^t) \in \mathbf{IN_{mon}}$ if and only if $\underline{s}^t$ is a possible

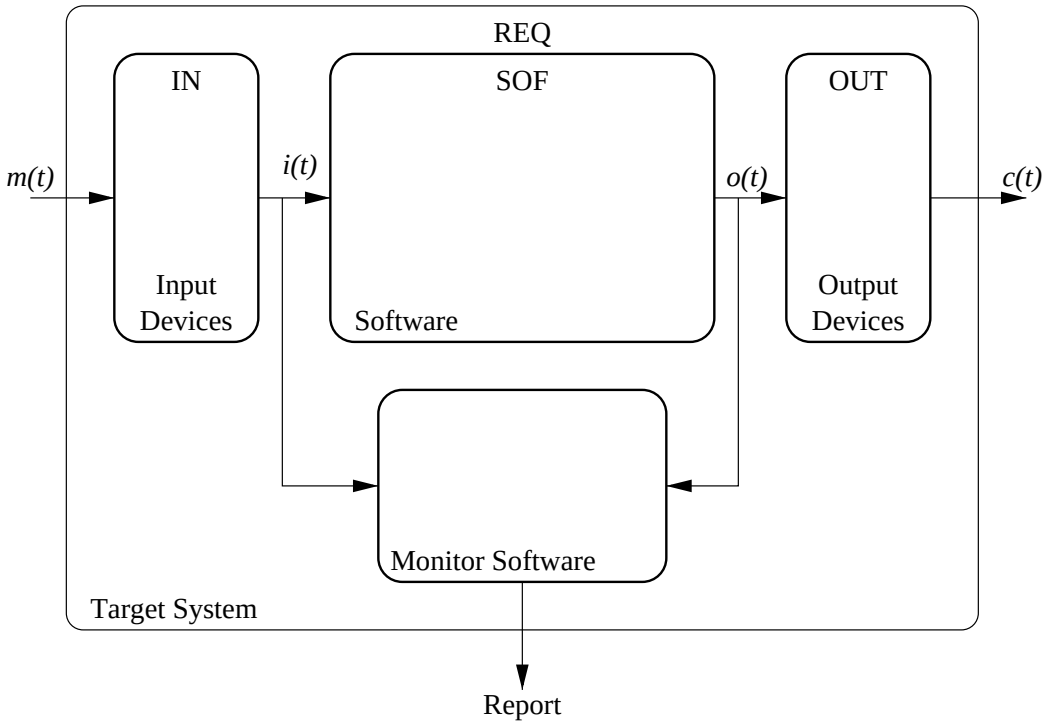

Figure 3.1: Software Monitor

monitor input state function for the environmental state function represented by $(\underline{m}^t, \underline{c}^t)$. Since the monitor must observe all acceptable behaviours, 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:

Software Monitor 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 3.1. In this case $\mathbf{IN_{mon}}$ is related to **IN** and **OUT** as follows.

$$\mathbf{IN_{mon}} = \left\{ \left(\left(\underline{m}^t, \underline{c}^t \right), \left(\underline{i}^t, \underline{o}^t \right) \right) \mid \mathrm{IN} \left(\underline{m}^t, \underline{i}^t \right) \wedge \mathrm{OUT} \left(\underline{o}^t, \underline{c}^t \right) \right\} \tag{3.1}$$

Software monitors include all of the monitor "architectures" discussed in [99].

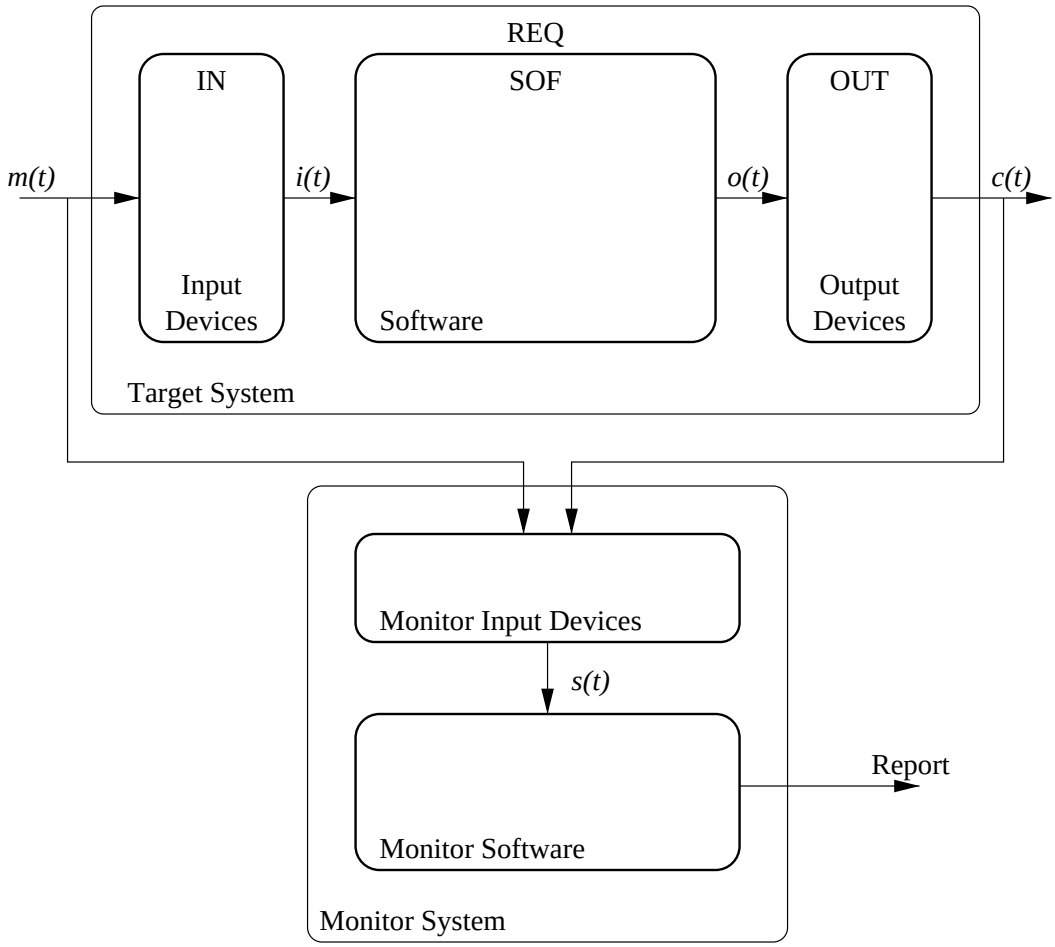

Figure 3.2: System Monitor

System Monitor A *system monitor* is a monitor that observes $(\underline{m}^t, \underline{c}^t)$ using its own input devices as illustrated in Figure 3.2.

The monitor software determines if the target system behaviour is consistent with **REQ** under the assumption that the monitor system's input devices are functioning correctly, as described in $\mathbf{IN_{mon}}$. That is, the monitor software determines if an observation of the target system behaviour, $\underline{s}^t$, is in the monitor relation, $\mathbf{MON_{pe}}$,

which is defined as

$$\mathbf{MON_{pe}} \stackrel{\text{df}}{=} \left\{ \underline{s}^t \in \mathit{range}(\mathbf{IN_{mon}}) \mid \left(\forall \left(\underline{m}^t, \underline{c}^t\right) \in \mathbf{M} \times \mathbf{C}, \begin{pmatrix} \text{IN}_{\text{mon}}\left(\left(\underline{m}^t, \underline{c}^t\right), \underline{s}^t\right) \\ \wedge\ \text{NAT}\left(\underline{m}^t, \underline{c}^t\right) \\ \Rightarrow \text{REQ}\left(\underline{m}^t, \underline{c}^t\right) \end{pmatrix} \right) \right\} \tag{3.2}$$

In the case of the software monitor configuration, and neglecting impossible behaviours (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.

Note that Eq. (3.2) takes a pessimistic view of the system: it requires that *all* behaviours that could have resulted in a particular $\underline{s}^t$ be in **REQ**. If $\text{MON}_{\text{pe}}(\underline{s}^t)$ is $\underline{\mathit{true}}$ then the behaviour is certainly acceptable, i.e., $(\text{MON}_{\text{pe}}(\underline{s}^t) \wedge \text{IN}_{\text{mon}}\left(\left(\underline{m}^t, \underline{c}^t\right), \underline{s}^t\right)) \Rightarrow \text{REQ}\left(\underline{m}^t, \underline{c}^t\right)$. A more optimistic view would be to check if *any* behaviour that could have resulted in $\underline{s}^t$ is in **REQ**. The optimistic monitor relation, $\mathbf{MON_{op}}$, is defined as

$$\mathbf{MON_{op}} \stackrel{\text{df}}{=} \left\{ \underline{s}^t \in \mathit{range}(\mathbf{IN_{mon}}) \mid \left(\exists \left(\underline{m}^t, \underline{c}^t\right) \in \mathbf{M} \times \mathbf{C}, \begin{pmatrix} \text{IN}_{\text{mon}}\left(\left(\underline{m}^t, \underline{c}^t\right), \underline{s}^t\right) \\ \wedge\ \text{NAT}\left(\underline{m}^t, \underline{c}^t\right) \\ \wedge\ \text{REQ}\left(\underline{m}^t, \underline{c}^t\right) \end{pmatrix} \right) \right\} \tag{3.3}$$

and includes those observations that may, but do not necessarily, represent acceptable behaviour. A monitor that evaluates $\mathbf{MON_{op}}$ will not give false negative results—reports that an acceptable behaviour is unacceptable—but is not appropriate for safety-critical systems since it may give false positive results—unacceptable behaviour 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 $\mathbf{IN_{mon}}$.

From the above definitions, it is apparent that neither $\text{MON}_{\text{pe}}(\underline{s}^t) \Rightarrow \text{MON}_{\text{op}}(\underline{s}^t)$ nor $\text{MON}_{\text{op}}(\underline{s}^t) \Rightarrow \text{MON}_{\text{pe}}(\underline{s}^t)$ holds in general. To see this, consider the equivalent form:

$$\text{MON}_{\text{pe}}(\underline{s}^t) \equiv \neg\exists \left(\underline{m}^t, \underline{c}^t\right) \in \mathbf{M} \times \mathbf{C}, \begin{pmatrix} \text{IN}_{\text{mon}}\left(\left(\underline{m}^t, \underline{c}^t\right), \underline{s}^t\right) \wedge \\ \text{NAT}\left(\underline{m}^t, \underline{c}^t\right) \wedge \\ \neg\text{REQ}\left(\underline{m}^t, \underline{c}^t\right) \end{pmatrix}$$

which clearly neither implies, nor is implied by $\text{MON}_{\text{op}}(\underline{s}^t)$. While $\text{MON}_{\text{op}}(\underline{s}^t) \not\Rightarrow \text{MON}_{\text{pe}}(\underline{s}^t)$ is intuitively satisfying—the pessimistic monitor should reject some be-

haviours accepted by the optimistic one—the reverse, $\mathrm{MON}_{\mathrm{pe}}(\underline{s}^t) \not\Rightarrow \mathrm{MON}_{\mathrm{op}}(\underline{s}^t)$, is not—if a behaviour is acceptable using the pessimistic approach, then it seems it should be acceptable using an optimistic approach. However, if we make the assumption 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: $\exists\,(\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C}, (\mathrm{IN}_{\mathrm{mon}}\,((\underline{m}^t, \underline{c}^t)\,, \underline{s}^t) \wedge \mathrm{NAT}\,(\underline{m}^t, \underline{c}^t))$, then the implication holds, i.e., $\mathrm{MON}_{\mathrm{pe}}(\underline{s}^t) \Rightarrow \mathrm{MON}_{\mathrm{op}}(\underline{s}^t)$.

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 Chapter 4, for real input devices and discrete time systems, $\mathbf{IN_{mon}}$ and $\mathbf{IN_{mon}}^{-1}$ are both non-functional relations in practice.

3.3 Accuracy

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

$$\begin{aligned} \mathbf{MONNEG} &\stackrel{\mathrm{df}}{=} \left\{ \left(\underline{m}^t, \underline{c}^t\right) \in \mathbf{M} \times \mathbf{C} \mid \left(\exists \underline{s}^t \in \mathbf{S}, \begin{array}{l} \mathrm{IN}_{\mathrm{mon}}\left((\underline{m}^t, \underline{c}^t)\,, \underline{s}^t\right) \\ \wedge \neg \mathrm{MON}_{\mathrm{pe}}(\underline{s}^t) \end{array} \right) \right\} \\ &= \left\{ \left(\underline{m}^t, \underline{c}^t\right) \in \mathbf{M} \times \mathbf{C} \mid \left(\mathbf{IM}\left(\underline{m}^t, \underline{c}^t\right) \cap \overline{\mathbf{REQ}}\right) \neq \oslash \right\} \end{aligned} \tag{3.4}$$

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

$$\mathbf{IM}\left(\underline{m}^t, \underline{c}^t\right) \stackrel{\mathrm{df}}{=} \left\{ \left(\hat{\underline{m}}^t, \hat{\underline{c}}^t\right) \mid \left(\left(\underline{m}^t, \underline{c}^t\right), \left(\hat{\underline{m}}^t, \hat{\underline{c}}^t\right)\right) \in \left(\mathbf{IN_{mon}} \circ \mathbf{IN_{mon}}^{-1}\right) \right\} \tag{3.5}$$

and thus, **FN** is

$$\begin{aligned} \mathbf{FN} &\stackrel{\mathrm{df}}{=} \mathbf{REQ} \cap \mathbf{MONNEG} \\ &= \left\{ \left(\underline{m}^t, \underline{c}^t\right) \in \mathbf{M} \times \mathbf{C} \mid \mathrm{REQ}\left(\underline{m}^t, \underline{c}^t\right) \wedge \left(\mathbf{IM}\left(\underline{m}^t, \underline{c}^t\right) \cap \overline{\mathbf{REQ}}\right) \neq \oslash \right\} \end{aligned} \tag{3.6}$$

Consider **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 monitor software. In this case, $\mathbf{IM}\,(\underline{m}^t, \underline{c}^t) = \{(\underline{m}^t, \underline{c}^t)\}$, $\mathbf{MONNEG} = \overline{\mathbf{REQ}}$ and $\mathbf{FN} = \oslash$—the monitor software can detect exactly if the behaviour 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 $\mathbf{IM}\,(\underline{m}^t, \underline{c}^t) = \mathit{domain}(\mathbf{IN_{mon}})$, $\mathbf{MONNEG} = \mathit{domain}(\mathbf{IN_{mon}})$ and $\mathbf{FN} = \mathbf{REQ}$—under no circumstances can the monitor be sure that the behaviour is acceptable. In this case the monitor will be *infeasible.*

Definition 3.1 *A monitor is said to be* feasible *with respect to a monitor input relation,* $\mathbf{IN_{mon}}$*, system requirements relation,* **REQ***, and environmental constraints,* **NAT***, if and only if* $\mathbf{MON_{pe}} \neq \oslash$.

Clearly if $\mathbf{IN_{mon}}$ is the identity relation, then $\mathbf{MON_{pe}} = \mathbf{REQ}$ and so, assuming a non-empty **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. (3.3), as follows.

$$\begin{aligned} \mathbf{MONPOS} &\stackrel{\text{df}}{=} \left\{ (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C} \mid \left(\exists \underline{s}^t \in \mathbf{S}, \begin{array}{l} \mathrm{IN}_{\mathrm{mon}}\left((\underline{m}^t, \underline{c}^t), \underline{s}^t\right) \\ \wedge \mathrm{MON}_{\mathrm{op}}(\underline{s}^t) \end{array} \right) \right\} \\ &= \left\{ (\underline{m}^t, \underline{c}^t) \mid \left(\mathbf{IM}\,(\underline{m}^t, \underline{c}^t) \cap \mathbf{REQ} \right) \neq \oslash \right\} \end{aligned} \tag{3.7}$$

where **IM** is as defined in Eq. (3.5). The false positive set, **FP**, is thus

$$\begin{aligned} \mathbf{FP} &\stackrel{\text{df}}{=} \overline{\mathbf{REQ}} \cap \mathbf{MONPOS} \\ &= \left\{ (\underline{m}^t, \underline{c}^t) \in \mathbf{M} \times \mathbf{C} \mid \neg \mathrm{REQ}\,(\underline{m}^t, \underline{c}^t) \wedge \left(\mathbf{IM}\,(\underline{m}^t, \underline{c}^t) \cap \mathbf{REQ} \right) \neq \oslash \right\} \end{aligned} \tag{3.8}$$

Considering the $\mathbf{IN_{mon}}$ scenarios from above, in the best case $\mathbf{FP} = \oslash$ and in the worst case $\mathbf{FP} = \mathit{domain}(\mathbf{IN_{mon}})$—the monitor will report all observations as acceptable behaviour.

For realistic cases **FN** and **FP** will be small but 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 Chapter 4.

Chapter 4

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 behaviour, which influence the conclusions that can be drawn from the monitor output. This chapter discusses these characteristics, and states some conditions which must hold in order for the monitor to produce meaningful results.

4.1 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 behaviour in a non-intrusive manner is beyond the scope of this work—readers interested in that topic are referred to [92] 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 behaviour:

- Sample (i.e., observe the instantaneous value of) the relevant quantities at intervals.
- Modify the behaviour of the target system, and/or the systems that interact with it, to have them notify the monitor system of the values of relevant quan-

tities ($\underline{m}^t, \underline{c}^t, \underline{i}^t$ 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.

4.1.1 Discrete Time

Regardless of whether sampling or notification is used, the time measurement is discrete: 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 using the notification approach the monitor receives notifications for all relevant changes, then this approach is not significantly different from the sampling approach. Thus, the results from sampling theory (e.g., see [83]) can be applied here to show that, for infinite duration signals (behaviours), 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*. Since this work is concerned with real-time systems, such errors in measuring time are particularly important.

Consider the behaviours illustrated in Figure 4.1, 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 monitor software and the shaded regions represent the image of these changes under $\mathbf{IN_{mon}}^{-1}$. Let δ_{mon} represent the monitor sampling interval (i.e., $m_i - m_{i-1}$). 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 behaviour 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 must be less than* $\frac{1}{2}min(\mathbf{Delay})$*, where* **Delay** *is the set of maximum delay*

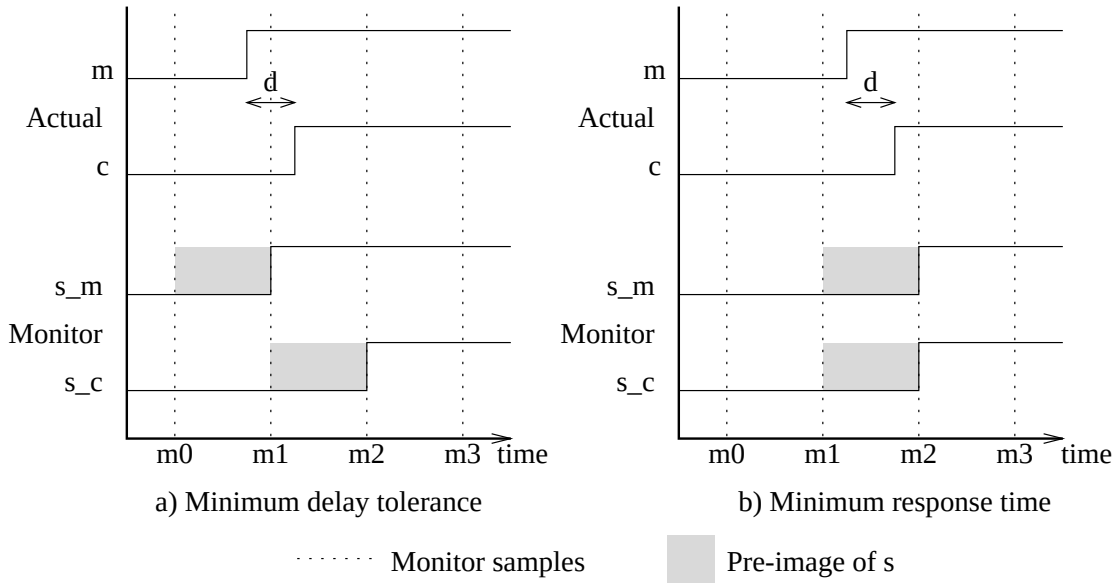

Figure 4.1: Time Accuracy

tolerances for the dependent[1] *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), and hence this behaviour 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*

[1] 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.

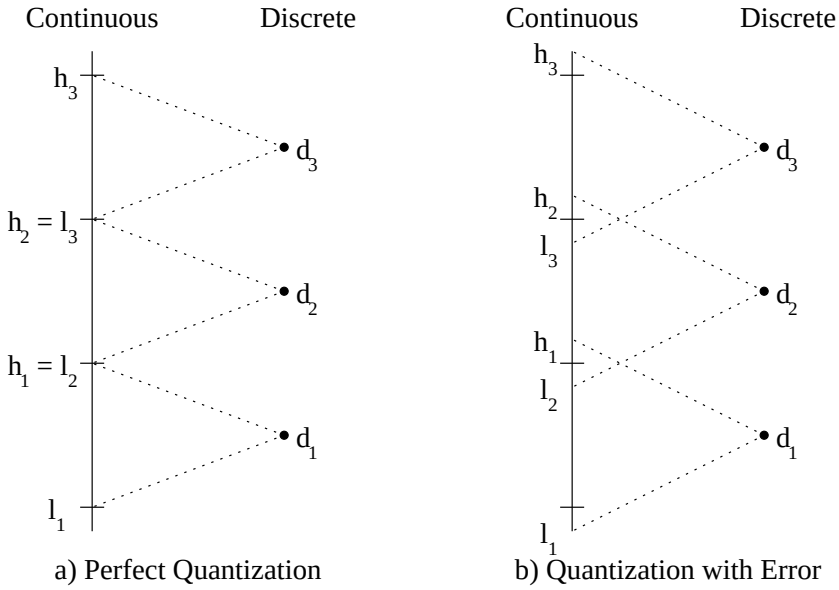

Figure 4.2: Quantization and Error

less than the minimum time in which the target system might respond to that event.

A monitor system that does not satisfy Condition 1 will be infeasible. A system that does not satisfy Condition 2 may give false negative results for target systems responding too quickly.

4.1.2 Quantization and Measurement Error

As for time, other values observed by the monitor software must be of finite precision, so **Real** valued environmental quantities must be quantized, such that, for example, discrete value d_i represents all continuous values, x, such that $l_i < x \leq h_i$. As illustrated in Figure 4.2, 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 non-functional.

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 error in observing a controlled quantity must be less than the non-determinism in* **REQ** *with respect to that quantity.*

4.2 Non-determinism

As mentioned in Section 1.1.1, practical requirements documents will be non-functional to 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, **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 is discussed further in Section 5.2.10. The monitor system must take this non-determinism into account when evaluating behaviour.

Consider the behaviour illustrated in Figure 4.3, and the target time resolution as indicated. The requirements must allow the changes in C1 and C2 to 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 or not, 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, 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 behaviour in which the target system interprets them in either way. The monitor software must take this non-determinism into account.

In the case of the software monitor configuration, as illustrated in Figure 3.1, the monitor software and the target system software 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 behaviour.

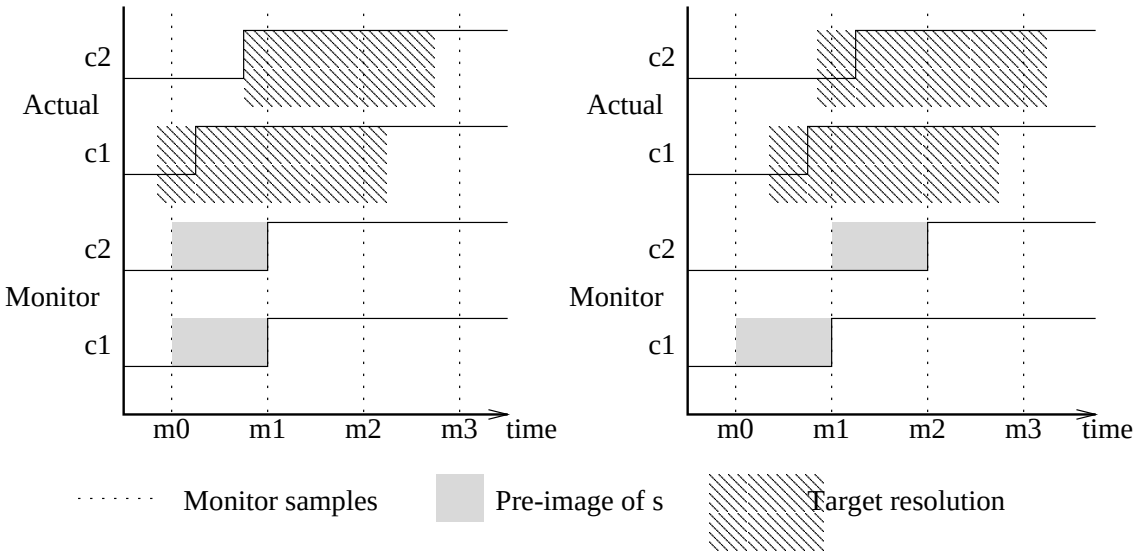

Figure 4.3: Event Resolution

4.3 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 monitor software, 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 a robot has touched a wall) whereas a different choice of input devices would require that the monitor software perform some, possibly expensive, calculations (e.g., search a list of wall locations to determine if the robot is touching any).

4.4 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 [79], 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.

Chapter 5

Specifying System Requirements

This chapter presents a method for documenting system requirements that is based on the method presented in [102], which developed from the A-7E project.[46] This method has a great deal in common with the SCR method discussed in Section 2.1.4, but enhances it in a number of ways:

- It gives an interpretation of the notation for behaviours as functions of continuous time. Discrete time is used in [46] and [102].
- It clarifies the notations for event classes so that real-time properties can be more easily specified.
- It makes a distinction between environmental and system modes, which clarifies the interpretation with respect to environmental quantities, eliminates the need for an assumption of initial mode, and simplifies the description of allowable delays. [46] and [102] use only system modes.
- It introduces notations (e.g., **WHILE**(c) and **CONT**(c)) and simple semantics to specify behaviour for what are usually called "simultaneous events".
- It introduces "merit functions", which allow specifiers to rank the acceptable behaviours, indicating which are preferable over others. Previous work on SCR has not used such functions.
- It allows monitor generation.

One goal in this work is that this method be useful in a wide variety of application domains without imposing new or restricted notations where they are not needed. Rather than explicitly stating a, necessarily incomplete, specification language, the notation presented here is intended as an extension to the (presumably well defined) mathematical notation appropriate to the application domain. This thesis does not discuss restrictions that will ensure that expressions satisfy the assumptions of the next section. The notation accepted by the Monitor Generator tool, and how it can be extended, is discussed in Chapter 6, Appendix B, and [2, 79, 96].

5.1 Assumptions and Definitions

As discussed in Section 1.1, the SRD describes the relation $\mathbf{REQ} \subseteq \mathbf{M} \times \mathbf{C}$, on functions of time. In this thesis the following definitions are used, and assumptions made.

5.1.1 Environmental Quantities

All environmental quantities are assumed to be represented by functions of time that, depending on the value type of the quantity, fall into one of the following categories.

- For **Real** valued quantities, the function is *piecewise-continuous* with respect to time. That is, the quantity is modelled by a function, $\mathsf{v} : \mathbf{Real} \rightarrow \mathbf{Real}$, such that, on any finite period of system operation, $[t_i, t_f]$,

  1. $\lim_{t \rightarrow t_0^+} v(t)$ is defined for all $t_0 \in [t_i, t_f)$,
  2. $\lim_{t \rightarrow t_0^-} v(t)$ is defined for all $t_0 \in (t_i, t_f]$, and
  3. The set of *discontinuities* of v, $\left\{ t_0 \in (t_i, t_f) \mid \lim_{t \rightarrow t_0^+} v(t) \neq \lim_{t \rightarrow t_0^-} v(t) \right\}$, is finite.

- For discrete valued quantities, the function is *finitely variable* with respect to time. That is, for any finite period of system operation, $[t_i, t_f]$, there is at most a finite set of instants at which the function value changes.[8]

As a notational convenience, v is taken to be a shorthand for $\mathsf{v}(t_f)$ where t_f is the "current" time, i.e., the final point of the behaviour being considered.

The notations $\grave{\mathsf{v}}(t_0)$ and $\mathsf{v}'(t_0)$ are used to denote the value of v immediately before and after, respectively, t_0. This notation is formally defined by Definitions 5.1 through 5.4, below, depending on the value type of the quantity, v.

Definition 5.1 *For a* **Real** *valued quantity,* v*, that is piecewise-continuous on an interval* $[t_i, t_f]$*, and an instant* $t_0 \in (t_i, t_f]$*,*

$$\grave{\mathsf{v}}(t_0) \stackrel{\text{df}}{=} \lim_{t \to t_0^-} \mathsf{v}(t)$$

Definition 5.2 *For a* **Real** *valued quantity,* v*, that is piecewise-continuous on an interval* $[t_i, t_f]$*, and an instant* $t_0 \in [t_i, t_f)$*,*

$$\mathsf{v}'(t_0) \stackrel{\text{df}}{=} \lim_{t \to t_0^+} \mathsf{v}(t)$$

Definition 5.3 *For a discrete valued quantity,* v*, that is finitely variable on an interval* $[t_i, t_f]$*, and an instant* $t_0 \in (t_i, t_f]$*,*

$$\grave{\mathsf{v}}(t_0) \stackrel{\text{df}}{=} x \mid (\exists \delta > 0, (\forall t \in (t_0 - \delta, t_0), \mathsf{v}(t) = x))$$

Definition 5.4 *For a discrete valued quantity,* v*, that is finitely variable on an interval* $[t_i, t_f]$*, and an instant* $t_0 \in [t_i, t_f)$*,*

$$\mathsf{v}'(t_0) \stackrel{\text{df}}{=} x \mid (\exists \delta > 0, (\forall t \in (t_0, t_0 + \delta), \mathsf{v}(t) = x))$$

Note that since v may be discontinuous at t_0, it is possible that $\grave{\mathsf{v}}(t_0) \neq \mathsf{v}'(t_0)$.

5.1.2 Conditions

All systems are required to change the value of their controlled quantities in response to changes in the monitored quantities, e.g., a user pushing a button to request some action, or a physical quantity crossing some threshold. As illustrated in [45], such behaviour can often be effectively described in terms of conditions and events.

Definition 5.5 *A* condition *is a function* **Real** $\rightarrow$ **Boolean***, defined in terms of the environmental state function, that is finitely variable on all intervals of system operation.*

With respect to a particular system being specified, we assume a finite set of conditions, $p_1, p_2, \ldots, p_n$, and denote a tuple of **Boolean** of length n by **Cnd**. For the purposes of the following discussion, we assume that the conditions are assigned a fixed order and so can be referred to simply by their index in that order (i.e., 2 refers to p_2, etc.). The notations of Section 5.1.1 are extended to conditions.

5.1.3 Events

The instants when conditions change value are significant to the behaviour of the system. These are referred to as *events*, and defined as follows.

Definition 5.6 *An* event, *e, is a pair,* (t, c)*, where* $e.t \in$ **Real** *is a time at which one or more conditions change value and* $e.c \in \{\mathrm{T}, \mathrm{F}, @\mathrm{T}, @\mathrm{F}\}^n$ *indicates the status of all conditions at e.t, as follows:*

$e.c[i]$	p_i
T	$`p_i(e.t) \wedge p_i'(e.t)$
F	$\neg `p_i(e.t) \wedge \neg p_i'(e.t)$
@T	$\neg `p_i(e.t) \wedge p_i'(e.t)$
@F	$`p_i(e.t) \wedge \neg p_i'(e.t)$

The type **EvSp** $\stackrel{\mathrm{df}}{=}$ **Real** $\times \{\mathrm{T}, \mathrm{F}, @\mathrm{T}, @\mathrm{F}\}^n$ is the set of all possible events—the *event space*—relevant to a particular system. Since all conditions are finitely variable on all intervals of system operation, any particular finite duration behaviour defines a finite set of events **Ev** $\subset$ **EvSp**.

5.1.4 History

For systems that respond to changes in conditions, the relevant aspects of the environmental state function on some time interval, $[t_i, t_f]$, can often be concisely described by giving the value of the relevant conditions at t_i (the *initial conditions*) and listing the sequence of events between t_i and t_f. Such an abstraction of an environmental state function with respect to a particular tuple of conditions, $p_1, p_2, \ldots, p_n$, is known as a *history*.

Definition 5.7 *With respect to a particular system and conditions $p_1, p_2, \ldots, p_n$, a* history*, H, between t_i and t_f is a pair, $(I, \mathbf{E})$, where $H.I \stackrel{\text{df}}{=} (p_1(t_i), p_2(t_i), \ldots, p_n(t_i))$ is the value of the initial conditions, and $H.\mathbf{E} : [1 \ldots l] \to \mathbf{EvSp}$, for some $l \geq 0$, is the sequence of events occurring on $(t_i, t_f]$, such that*

$$\forall i \in [1 \ldots l) \left(\begin{array}{l} H.\mathbf{E}[i].t < H.\mathbf{E}[i+1].t \wedge \\ \forall j \in [1 \ldots n] \left(\begin{array}{l} H.I[j] \Leftrightarrow H.\mathbf{E}[1].c[j] \in \{\mathrm{T}, @\mathrm{F}\} \ \wedge \\ H.\mathbf{E}[i].c[j] \in \{\mathrm{T}, @\mathrm{T}\} \Leftrightarrow H.\mathbf{E}[i+1].c[j] \in \{\mathrm{T}, @\mathrm{F}\} \end{array} \right) \end{array} \right).$$

The type **Hist** $\subseteq$ **Cnd** $\times$ sequence of **EvSp** is the set of histories relevant to the system. A history, H, is said to be *possible* with respect to a particular system and environmental constraints, **NAT**, if $\exists\, (\underline{m}^t, \underline{c}^t) \in \mathbf{NAT}$ such that H is a correct abstraction of $(\underline{m}^t, \underline{c}^t)$ with respect to the conditions $p_1, p_2, \ldots, p_n$.

5.1.5 Modes and Mode Classes

In [45, 46], it was noted that

- it is frequently the case that many histories are equivalent with respect to their impact on future behaviour, and
- each controlled quantity is usually affected by a small number of conditions.

The following concepts are introduced here to take advantage of these facts.

Definition 5.8 *An* environmental mode class *(or simply* mode class*) is an equivalence relation on possible histories,* **MC** $\subseteq$ **Hist** $\times$ **Hist***, such that, if* $\mathrm{MC}(H_1, H_2)$*, and $\hat{H}_1$ and $\hat{H}_2$ are the extensions of H_1 and H_2 by the same event, then* $\mathrm{MC}(\hat{H}_1, \hat{H}_2)$*.*

Note that the same change in conditions occurring at different times are different events. Also, if, for example, the time elapsed since a previous event is significant to the behaviour, then this is a condition relevant to the system, and is included in the events.

A mode class defines a set of equivalence classes of histories, known as *modes*, that partitions the set of possible histories (i.e., every possible history is in exactly one mode in the mode class).

Modes and mode classes provide a mechanism for separation of concerns in the requirements document. The mode definitions clearly identify how the system behaviour depends on previous events, and the controlled state functions can be specified in terms of the current mode in one or more mode classes.

Events, histories, modes and mode classes are neither a departure from, nor an extension to, the "four variable model" as described in Section 1.1, but rather provide a foundation for concise descriptions of required system behaviour (**REQ**) by allowing us to describe those aspects of $(\underline{m}^t, \underline{c}^t)$ that are relevant, and ignore those that are not. These techniques are clearly not appropriate for all systems, and should only be used where they add value. For example, the requirements for a simple integrator circuit—for which the voltage at the output is the definite integral of the voltage at the input over some period—are not made more concise by using events, histories or mode classes. Such a system can be more effectively specified using traditional engineering mathematics.

5.1.6 Tolerance and Accuracy

As pointed out in [102], among others, it is often helpful to describe the system requirements by first describing the behaviour of an "ideal" (i.e., infinitely accurate and fast) system and then specifying acceptable deviations from that behaviour. This technique is common practice in most engineering disciplines (e.g., "2.00 m$\pm$0.01 m", "3.3 kΩ, 10% tolerance").

Ideal system requirements are useful for requirements capture and for construction of models for verification or validation, but they are not sufficient for detailed system design, implementation or testing. For these tasks we must recognize that, because of the limitations of the physical world, any implemented system can only approximate the ideal behaviour. The requirements specification must clearly state exactly which approximations of the ideal behaviour are sufficient for the task at hand. In the four variable model this appears as **REQ** being a non-functional relation (i.e., there is more than one acceptable value of $\underline{c}^t$ for any value of $\underline{m}^t$).

Such specifications help the implementer to decide where effort and expense should be invested to reduce the impact of technological limitations on system behaviour. Some examples of limitations that typically affect computer systems are

- all measurements contain some error,
- calculations performed on digital computers are finite precision and subject to numerical errors, and
- no computer system can respond instantaneously to changes in its environment.

The concepts of *tolerance* and *accuracy*, which are adapted here from [102, 103], provide a model for such specifications. Note that by choosing one or the other of these techniques the specification author is not identifying sources of error, but is rather describing what cumulative effect of all sources of error on the values of the controlled quantities is considered acceptable.

The most common, and intuitively satisfying, means of describing deviations from ideal behaviour is to state how much the actual value of a controlled quantity is permitted to deviate from the ideal value. This can be expressed in the form of a tolerance relation.

Definition 5.9 *A* tolerance relation, $\mathbf{T} \subseteq \mathbf{C} \times \mathbf{C}$, *is a relation that relates an ideal controlled state function to the controlled state functions representing acceptable actual values of the controlled quantity. A pair of functions is in the relation if and only if the second element of the pair is an acceptable actual value of* $\underline{c}^t$ *when the first element of the pair is the ideal value.*

An alternative technique for describing deviation from ideal behaviour is to use an *accuracy relation* to state how the values of monitored variables used in the specification may deviate from their "true" values. In [102, 103] this is called *precision*, but *accuracy* is used here to avoid confusion with other meanings of precision, such as the number of bits used in the representation.

Definition 5.10 *An* accuracy relation, $\mathbf{A} \subseteq \mathbf{M} \times \mathbf{M}$, *relates an actual value of a monitored state function to the functions representing acceptable perceived monitored state functions.*

Thus, as stated in [102, 103], the set of acceptable behaviours, **REQ**, can be calculated by the composition of the accuracy relation, **A**, the ideal behaviour, **I**, and the tolerance, **T**: $\mathbf{REQ} = \mathbf{A} \circ \mathbf{I} \circ \mathbf{T}$, or, more explicitly:

$$\mathbf{REQ} = \left\{ \left(\underline{m}^t, \underline{c}^t\right) \mid \exists \hat{\underline{m}}^t, \hat{\underline{c}}^t, \left(\mathrm{A}\left(\underline{m}^t, \hat{\underline{m}}^t\right) \wedge \mathrm{T}\left(\hat{\underline{c}}^t, \underline{c}^t\right) \wedge \mathrm{I}\left(\hat{\underline{m}}^t, \hat{\underline{c}}^t\right)\right)\right\} \tag{5.1}$$

Clearly, there are many choices of combinations of ideal behaviour, accuracy and tolerance to specify the same set of acceptable behaviours. In particular, by choosing the identity relation for both accuracy and tolerance, Eq. (5.1) reduces to $\mathbf{REQ} = \{(\underline{m}^t, \underline{c}^t) \mid \mathrm{I}(\underline{m}^t, \underline{c}^t)\} = \mathbf{I}$. (Of course, in this case $\mathbf{I}$ would not be what is typically called "ideal" behaviour—it would have to allow for delays etc.) Thus accuracy and tolerance relations do not increase the expressiveness of the method; however, they have been found to be convenient for concisely specifying ranges of behaviours. Similarly, there is no theoretical advantage to allowing both accuracy and tolerance relations to be used when either one would suffice and, since the method is controlled value driven, tolerance seems to be the most intuitive. However, there are situations where using accuracy relations leads to a more concise and readable specification. For example, consider a system that measures a quantity, displays its value and turns on a light if the value is above some threshold. The SRD author may want to require that the displayed value and light are consistent (i.e., the light is not on unless the displayed value is above the threshold), but be willing to accept some error introduced in the measurement of the monitored quantity. This behaviour can be clearly stated using an accuracy relation rather than tolerance.

5.2 Specification Notation

The SRD is interpreted as the characteristic predicate of **REQ**, and so expressions are implicitly taken to refer to a particular behaviour of the system (i.e., environmental state function on some finite interval). Throughout this section the behaviour will be assumed to be on the interval $[t_i, t_f]$.

In cases where the controlled quantities are simple functions of the current or past values of monitored quantities, no special notation is required. For example, for a simple amplifier with one controlled quantity, ${}^{c}\mathsf{o}$, that is always required to be within ${}^{C}\mathsf{TOL}$ of twice the value of the monitored quantity, ${}^{m}\mathsf{i}$, the controlled value relation is simply stated "${}^{c}\mathsf{o} = 2 * {}^{m}\mathsf{i} \pm {}^{C}\mathsf{TOL}$". In cases where the system responds differently depending on past events, however, notation to describe conditions, events, modes and mode classes can be helpful.

This section presents a system requirements documentation technique, based on [102] and the SCR method, that could be used to specify target system behaviour.

The next sub-section gives an overview of the components of a system requirements document (SRD). The remaining sub-sections of this section present notations for the main components of an SRD in more detail, using a simple logic probe, of which a system overview is given in Figure 5.1, as a running example. Appendix A contains several complete realistic SRD examples.

System Overview

The logic probe is a device for testing and debugging TTL logic circuits. It consists of a pen-shaped body with a short grounding wire, which has an alligator clip on the end. One end of the body is a finely pointed metalic tip so that it can be easilly placed against a circuit on a printed circuit board or an IC chip leg. The other end of the body has a tri-state light on it that glows red to indicate a logic "high" signal, green for logic "low" or is off for an invalid logic level. The actual voltage measured at the tip is displayed on a small LCD display on the probe body.

There is a momentary contact push button labelled "Pulse" on the side of the probe body. When this button is pushed the probe tip will be "pulled low" (i.e., shunted through a low resistance to ground) for a period of 100 ms.

Figure 5.1: Logic Probe System Overview

5.2.1 Document Sections

Four sections are required in the system requirements document, and three additional sections are optional, as follows.[102]

Required Sections

Environmental Quantities : This section defines the system boundaries by describing the set of quantities that are either monitored or controlled by the system. Variables are defined to refer to each quantity and the physical meaning of these are precisely described.

System Behaviour : This section defines the required behaviour of the system by stating the value of each controlled quantity for any possible values (and history) of the monitored quantities.

Environmental Constraints : This section describes the behaviour of the environment in which the system operates by describing the constraints on the monitored and controlled quantities imposed by the environment. These constraints both limit the possible behaviour of the system and form assumptions that can be made by system designers. These are expressed in terms of the values of the monitored and controlled variables and define the relation **NAT**.

Dictionary The dictionary contains definitions of terms, which are mathematical functions or relations, and words that are either not common English (or whatever natural language is used) or have special meanings in the application domain. If no such definitions are needed the dictionary may be empty.

Optional Sections

System Overview : This section gives an informal description of the system requirements, possibly including non-behavioural requirements. Figure 5.1 is the system overview for the logic probe example.

Notational Conventions : If conventions or notations are used that are not standard in the application domain, then they should be explained here. For example, as illustrated in [45, 46, 102], it is often useful to use special bracketing, font, or variable naming conventions as mnemonics to aid readability. The conventions illustrated in Table 5.1 are adopted for the examples in this thesis.

Anticipated Changes As discussed in [74], a major consideration when designing a system is the set of changes to the requirements that are most likely to occur. This information can be used to help limit the amount of re-design and implementation modification needed to effect these changes. While it is impossible to anticipate or isolate the effects of all changes, it is useful to consider which are most likely, and to record this in the SRD.

Table 5.1: Notational Conventions

Item	Notation	Example
Monitored variable	prefix superscript m	^{m}mvar
Controlled variable	prefix superscript c	^{c}cvar
Monitored and controlled variable	prefix superscript mc	^{mc}mcvar
Condition	prefix superscript p	^{p}cond
Mode Class	prefix superscript Cl	^{Cl}mclass
Mode	prefix superscript Md	^{Md}mode
Constant	prefix superscript C	^{C}const

5.2.2 Monitored and Controlled Quantities

The description of the environmental quantities defines the range of the environmental state function, denoted **St**. The following information must be given for each environmental quantity that is relevant to the system.

- Physical interpretation
- Variable name
- Role: monitored/controlled/both
- Type (set of possible values)

Typically this information is summarized in a table, such as in Figure 5.2, and notes or separate sub-sections are used to give further details. As is common in other fields of science and engineering, the description of the physical interpretation is usually informal, but it should be as precise and accurate as possible, using diagrams or mathematical notation where appropriate.

The set of possible values for a particular quantity can also be determined by examining the domain or range of the environmental constraints, **NAT**, but it is stated explicitly in this section because it can be used to determine the data types needed to model the quantities. This set of values may be expressed in a wide variety of ways, including a (open or closed) range of the real numbers or integers, an enumerated set (e.g., for button positions) or a sequence of characters (e.g., command typed).

Environmental Quantities

Variable	Description	Monitored	Controlled	Type	Notes
^{m}t	Current time	•		**Real**	1
^{m}Vtip	Potential (voltage) at the probe tip with respect to the ground clip.	•		[-15, +15] V	
^{m}Pulse	Position of the pushbutton on the probe body labelled "Pulse".	•		{^{s}Up, ^{s}Down}	
^{c}Vdisp	Value displayed on display panel.		•	$[-9.9, -9.8 \ldots + 9.9]$	
^{c}Light	State of the probe light.		•	{^{s}Off, ^{s}Red, ^{s}Green}	
^{c}Requiv	Equivalent resistance at probe tip.		•	$[0\ \Omega, 10\ \mathrm{M}\Omega]$	2

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).

2. Equivalent resistance of the probe, modelled as a resistive shunt to ground.

Figure 5.2: Logic Probe Environmental Quantities

5.2.3 Tabular Expressions

Several industrial projects and research efforts (e.g., [39, 46, 48, 59, 75]) have demonstrated the utility of tabular notations for representing the kinds of functions that occur regularly in computer system documentation. Tabular expressions are used in this work and the generalized tabular semantics model and notation of [2, 54], which are based on [56, 75], are used to define how they are to be interpreted. In this model, the semantics of each form of tabular expression are described by the *cell connection graph (CCG) case* and two "table rules": the *table predicate rule*, $\mathrm{p_T}$, which determines the domain of the expression, and the *table relation rule*, $\mathrm{r_T}$, which determines the value of the expression.

An n-dimensional table consists of n *headers*, denoted $H_1, H_2, \ldots H_n$, and an n-dimensional *main grid*, G, such that the length of G in any dimension, i, is equal to the length of the corresponding header, H_i. In the table in Figure 5.3, for example, H_1 is the row header containing the expressions $y < 0$, $y = 0$, and $y > 0$. An *index*, α, is a tuple of length n such that $\forall i, 1 \leq i \leq n \Rightarrow 1 \leq \alpha[i] \leq \mathit{length}(H_i)$, where $\alpha[i]$ represents the element at position i of index α. An index identifies a unique cell in each header and in the main grid. For example, the tuple $(2, 1)$ is a valid index for the table in Figure 5.3 and identifies the expression $y = 0$, in H_1, $x < 0$ in H_2 and π in G.

$\mathrm{ang}(x, y)$

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1 \wedge H_2$ $\mathrm{r_T} : G$ Normal	$x < 0$	$x = 0$	$x > 0$
$y < 0$	$arctan(\frac{y}{x}) - \pi$	$-\frac{\pi}{2}$	$arctan(\frac{y}{x})$
$y = 0$	π	0	0
$y > 0$	$arctan(\frac{y}{x}) + \pi$	$\frac{\pi}{2}$	$arctan(\frac{y}{x})$

Figure 5.3: Example Tabular Expression

The cells of those headers and grids mentioned in the table predicate rule (H_1 and H_2 in Figure 5.3) and table relation rule (G in Figure 5.3) are respectively referred to as *guard* and *value* cells. The guard cells are combined according to $\mathrm{p_T}$ to form a guard expression, $\mathrm{p}^{\mathrm{T}}_{\alpha}$, for a particular index α. For example, for $\alpha = (2, 1)$, in Figure 5.3, $\mathrm{p}^{\mathrm{T}}_{\alpha} \stackrel{\text{df}}{=} y = 0 \wedge x < 0$. The conjunction of $\mathrm{p}^{\mathrm{T}}_{\alpha}$ with the value expression for that index, $\mathrm{r}^{\mathrm{T}}_{\alpha}$, forms a *raw element relation*, R_{α} (e.g., $\mathrm{R}_{(2,1)} \stackrel{\text{df}}{=} y = 0 \wedge x < 0 \wedge \mathsf{ang} = \pi$).

The cell connection graph case determines how the raw element relations are to be combined to construct the table relation, $\mathbf{R_T}$, as described in Table 5.2. In Table 5.2, $\mathbf{I}$ is the set of possible indices for the table and v is the number of a special header know as the *vector header*. $\mathbf{I^D}$ is the index set for the table with the vector header index removed, and $\alpha \mid v$ is the index formed by deleting the v^{th} element from α. n is the length of the vector header. The operator "$\otimes$", defined in [54], is a variation of 'join' from relational databases, and is used to merge relations to form a single 'vector' relation. For example, if $\mathbf{A} \subseteq \mathbf{U_0} \times \mathbf{U_1}$ and $\mathbf{B} \subseteq \mathbf{U_0} \times \mathbf{U_2}$, then

$\mathbf{A} \otimes \mathbf{B} = \{(x_0, x_1, x_2) \mid (x_0, x_1) \in \mathbf{A} \wedge (x_0, x_2) \in \mathbf{B}\}$. Readers interested in a more formal and complete treatment of tabular expression semantics are referred to [54].

As an example of interpreting a tabular expression, consider Figure 5.3 in which $\mathrm{p_T} : H_1 \wedge H_2$, $\mathrm{r_T} : G$ and CCG case "Normal" specifies that the table is interpreted by choosing i and j such that $H_1[i] \wedge H_2[j]$ is $\underline{true}$, the value of the tabular expression is given by $G[i, j]$, giving the table relation, $\mathrm{R_T}$, as follows.

$$
\begin{aligned}
\mathbf{R_T} &\stackrel{\text{df}}{=} \textstyle\bigcup_{j=1}^{3} \bigcup_{i=1}^{3} \mathbf{R}_{(i,j)} \\
&= \mathbf{R}_{(1,1)} \cup \mathbf{R}_{(2,1)} \cup \mathbf{R}_{(3,1)} \cup \mathbf{R}_{(1,2)} \cup \mathbf{R}_{(2,2)} \cup \mathbf{R}_{(3,2)} \\
&\quad \cup \mathbf{R}_{(1,3)} \cup \mathbf{R}_{(2,3)} \cup \mathbf{R}_{(3,3)} \\
&= \left\{ x, y, \mathsf{ang} \;\middle|\; \begin{array}{l} (y < 0 \wedge x < 0 \wedge \mathsf{ang} = arctan(\frac{y}{x}) - \pi) \\ \vee (y = 0 \wedge x < 0 \wedge \mathsf{ang} = \pi) \\ \vee (y > 0 \wedge x < 0 \wedge \mathsf{ang} = arctan(\frac{y}{x}) + \pi) \\ \vee (y < 0 \wedge x = 0 \wedge \mathsf{ang} = -\frac{\pi}{2}) \\ \vee (y = 0 \wedge x = 0 \wedge \mathsf{ang} = 0) \\ \vee (y > 0 \wedge x = 0 \wedge \mathsf{ang} = \frac{\pi}{2}) \\ \vee (y < 0 \wedge x > 0 \wedge \mathsf{ang} = arctan(\frac{y}{x})) \\ \vee (y = 0 \wedge x > 0 \wedge \mathsf{ang} = 0) \\ \vee (y > 0 \wedge x > 0 \wedge \mathsf{ang} = arctan(\frac{y}{x}) \end{array} \right\}
\end{aligned}
$$

Table 5.2: Cell Connection Graph Cases

CCG Case	Relation, $\mathbf{R_T} =$	Intuition
Normal, Inverted[1]	$\bigcup_{\alpha \in \mathbf{I}} \mathbf{R}_\alpha$	Evaluate the value expression for the index that makes the guard expression $\underline{true}$.
Vector	$\bigotimes_{i=1}^{n} \left(\bigcup_{\substack{\alpha_v = i \\ \alpha \in \mathbf{I}}} \mathbf{R}_\alpha \right)$	'Join' the value expressions for all indices (a row or column) that make the guard expression $\underline{true}$.
Decision	$\bigcup_{\beta \in \mathbf{I^D}} \left(\bigcap_{\substack{\alpha_v = 1 \\ \alpha \mid v = \beta}}^{n} \mathbf{R}_\alpha \right)$	Evaluate the value expression for the index that make the conjunction of the guard expressions in that row or column $\underline{true}$.

The table layout conventions for tabular expressions are illustrated in Figure 5.4. In this figure, $\mathbf{R}$ represents the table rule information (p_T, r_T, and CCG case), $\mathbf{H_i}[j]$ represents the j^{th} cell in header i and $\mathbf{G}[x]$ represents the cell at position x in the main grid. The two tables on the left of the figure each have a one-dimensional main grid and only one header, $\mathbf{H_1}$. The middle two tables in the figure each have a two-dimensional main grid, and two headers: $\mathbf{H_1}$, the row header, and $\mathbf{H_2}$, the column header. The table on the right of the figure illustrates how a three-dimensional table can be printed by printing 'slices' of the table and repeating the cells of $\mathbf{H_3}$ for each slice. Header grids are denoted by being separated from the main grid by double or heavy lines. Where possible, the table rule information is printed in an otherwise unused cell, as indicated.

$\mathbf{R}$	
$\mathbf{H_1}[1]$	$\mathbf{G}[1]$
$\mathbf{H_1}[2]$	$\mathbf{G}[2]$

$\mathbf{H_1}[1]$	$\mathbf{H_1}[2]$
$\mathbf{G}[1]$	$\mathbf{G}[2]$

$\mathbf{R}$	$\mathbf{H_2}[1]$	$\mathbf{H_2}[2]$	$\mathbf{H_2}[3]$
$\mathbf{H_1}[1]$	$\mathbf{G}[1,1]$	$\mathbf{G}[1,2]$	$\mathbf{G}[1,3]$
$\mathbf{H_1}[2]$	$\mathbf{G}[2,1]$	$\mathbf{G}[2,2]$	$\mathbf{G}[2,3]$

$\mathbf{H_1}[1]$	$\mathbf{G}[1,1]$	$\mathbf{G}[1,2]$	$\mathbf{G}[1,3]$
$\mathbf{H_1}[2]$	$\mathbf{G}[2,1]$	$\mathbf{G}[2,2]$	$\mathbf{G}[2,3]$
$\mathbf{R}$	$\mathbf{H_2}[1]$	$\mathbf{H_2}[2]$	$\mathbf{H_2}[3]$

$\mathbf{R}$	$\mathbf{H_2}[1]$	$\mathbf{H_2}[2]$	
$\mathbf{H_1}[1]$	$\mathbf{G}[1,1,1]$	$\mathbf{G}[1,2,1]$	$\mathbf{H_3}[1]$
	$\mathbf{G}[1,1,2]$	$\mathbf{G}[1,2,2]$	$\mathbf{H_3}[2]$
$\mathbf{H_1}[2]$	$\mathbf{G}[2,1,1]$	$\mathbf{G}[2,2,1]$	$\mathbf{H_3}[1]$
	$\mathbf{G}[2,1,2]$	$\mathbf{G}[2,2,2]$	$\mathbf{H_3}[2]$

Figure 5.4: Table Layout and Numbering Conventions

To reduce wasted space when printing 3-D tables, rows may be eliminated from slices where the table predicate rule is trivially *false* for the particular index values. This is illustrated for a particular example in Table 5.6 and Table 5.7.

5.2.4 Conditions

As discussed in Section 5.1.2, conditions are simply Boolean functions of time defined in terms of the monitored and controlled quantities. These definitions can be written using constants, the environmental quantities, and functions of them together with standard relational (e.g, $<, >$) and logic (e.g., $\wedge, \vee, \neg$) operators and tabular expres-

[1] The "Normal" and "Inverted" CCG cases have the same interpretation but they are distinguished for historical and technical reasons.

sions. By adopting a logic such as presented in [76] the definitions can include partial functions.

Conditions that differ at finitely many points (instants) are considered to be equivalent, so, for example, for a **Real** valued and continuously changing quantity, ${}^{m}\mathsf{x}$, ${}^{m}\mathsf{x} = 0 \equiv \underline{\mathit{false}}$. While this may seem counter-intuitive at first, it makes sense since no system can determine if ${}^{m}\mathsf{x} = 0$ *exactly*, but can only determine that it is within some tolerance of it, i.e., $-\delta < {}^{m}\mathsf{x} < \delta$, for some small δ. To help make the documentation concise shorthand notations could be defined, e.g., $x \approx_{\delta} y \stackrel{\text{df}}{=} |x - y| < \delta$, although this form will not be used in this thesis.

A full treatment of the logic of conditions is beyond the scope of this thesis, so in the sequel they are treated as environmental quantities, and assumed to be finitely variable, as defined in Section 5.1.1.

Figure 5.5 illustrates some condition definitions.

Conditions

Name	Condition
${}^{p}\mathsf{low}$	${}^{m}\mathsf{Vtip} < 0.8$ V
${}^{p}\mathsf{float}$	0.8 V $< {}^{m}\mathsf{Vtip} < 2.0$ V
${}^{p}\mathsf{high}$	${}^{m}\mathsf{Vtip} > 2.0$ V

Figure 5.5: Logic Probe Conditions

The behaviour of real-time systems is, by definition, dependent on time, and hence there is often a need to specify conditions dependent on time. If this is the case, time is a monitored variable and no special notation is required. For example, the condition that becomes $\underline{\mathit{true}}$ 100 ms after the ${}^{m}\mathsf{Pulse}$ button was pressed is "$(t_f - \mathit{Last}(@\mathbf{T}\,({}^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down}))) > 100$ ms". For systems where time of day or date are relevant, time can be represented by the time elapsed since some fixed time prior to the system being turned on. If time of day and date are not relevant to the system behaviour then time can be represented by time elapsed since some arbitrary time prior to the system being turned on.

5.2.5 Event Classes

In many cases the description of system behaviour can be stated concisely by considering sets of similar changes in conditions. Such sets of events are know as *event classes.*

Definition 5.11 *An* event class, **EC**, *is a subset of the events relevant to the system:* $\mathbf{EC} \subseteq \mathbf{Ev}$.

For example, the set of instants when the "Pulse" button is pressed is an event class used in the logic probe SRD. This event class is the set of events in which the condition ${}^{m}\mathsf{Pulse}(t) = {}^{s}\mathsf{Up}$ becomes *false* (or, alternatively, ${}^{m}\mathsf{Pulse}(t) = {}^{s}\mathsf{Down}$ becomes *true*).

SCR uses the notations "@T (p)" and "@F (p)" to characterize the event of condition p becoming *true* or *false*, respectively, and "@T (p_1) WHEN(p_2)" and "@F (p_1) WHEN(p_2)", to characterize the "conditioned event" of p_1 becoming *true* or *false*, respectively, under the constraint that p_2 is *true*. We adopt this notation and give its semantics in terms of event classes, as follows.

Definition 5.12 *An* event class expression *is a function,* $f : \mathbf{Ev} \rightarrow \mathbf{Boolean}$, *that characterizes an event class.*

Some simple event class expressions are defined in Table 5.3. In keeping with the notation described in Section 1.3, the bolded form of the event class expression is used to denote the class, and, by convention, the event argument is omitted from event class expressions, i.e., $\mathbf{@T}\,(p_i) \stackrel{\text{df}}{=} \{e \in \mathbf{Ev} \mid e.c[i] = \text{@T}\}$. The juxtaposition of two or more event class expressions denotes the conjunction of the expressions (intersection of the event classes), e.g., "@T (p_1) WHEN(p_2)" denotes "@T $(p_1) \wedge$ WHEN(p_2)" ("$\mathbf{@T}\,(p_1)$ $\mathbf{WHEN}(p_2)$" denotes "$\mathbf{@T}\,(p_1) \cap \mathbf{WHEN}(p_2)$").

Note that since an event describes all changes that occur at an instant, the common term "simultaneous events" doesn't make sense. Instants when two or more significant changes occur at the same time are cases where an event is in two or more relevant event classes. This can lead to ambiguity if an event class is under specified, for example the event classes "$\mathbf{@T}\,(p_1)$", "$\mathbf{@T}\,(p_1)$ $\mathbf{WHEN}(\neg p_2)$",

Table 5.3: Event Class Notation

Notation		Event Class
scalar	tabular	Expression
	p_i	
@T (p_i)	@T	$e.c[i] = @T$
@F (p_i)	@F	$e.c[i] = @F$
WHILE(p_i)	T	$e.c[i] = T$
WHILE$(\neg p_i)$	F	$e.c[i] = F$
WHEN(p_i)	t	$e.c[i] = T \vee e.c[i] = @F$
WHEN$(\neg p_i)$	f	$e.c[i] = F \vee e.c[i] = @T$
	t′	$e.c[i] = T \vee e.c[i] = @T$
	f′	$e.c[i] = F \vee e.c[i] = @F$
CONT(p_i)	—	$e.c[i] = F \vee e.c[i] = T$
	$*$	$\underline{true}$
	$\oslash$	$\underline{false}$

"@**T** (p_2)" and "@**T** (p_2) **WHEN**$(\neg p_1)$" all contain instants when both p_1 and p_2 become $\underline{true}$ at the same time. The "**WHILE**(p)" and "**CONT**(p)" notations, which are not used in SCR, are introduced to help in specifying disjoint event classes so that the required behaviour can be concisely described. For example, if the system response to ^{m}buttonA and/or ^{m}buttonB being pressed is specified using the event classes "@**T** $(^m\mathsf{buttonA} = {}^s\mathsf{Down})$" and "@**T** $(^m\mathsf{buttonB} = {}^s\mathsf{Down})$ **WHILE**$(^m\mathsf{buttonA} = {}^s\mathsf{Up})$", then the reader can conclude that

- the required response to the buttons being pressed simultaneously is as specified for "@T $(^m\mathsf{buttonA} = {}^s\mathsf{Down})$", since clearly @**T** $(^m\mathsf{buttonA} = {}^s\mathsf{Down}) \supseteq$ @**T** $(^m\mathsf{buttonA} = {}^s\mathsf{Down})$ @**T** $(^m\mathsf{buttonB} = {}^s\mathsf{Down})$, and

- if a different response is specified for "@T $(^m\mathsf{buttonB} = {}^s\mathsf{Down})$ WHILE$(^m\mathsf{buttonA} = {}^s\mathsf{Up})$" then it does not result in a contradictory specification since $\begin{pmatrix} @\mathbf{T}\,(^m\mathsf{buttonB} = {}^s\mathsf{Down})\ \mathbf{WHILE}(^m\mathsf{buttonA} = {}^s\mathsf{Up}) \cap \\ @\mathbf{T}\,(^m\mathsf{buttonA} = {}^s\mathsf{Down}) \end{pmatrix} = \oslash$.

Note also that these event classes include all events where either button is being pressed, so the specification covers all cases. This is discussed further with respect to mode transition relations in the next sub-section.

Event classes are sets, so standard set notation can be used to define new event classes in terms of previously defined classes. Set notation can also be used to define functions that describe properties of event classes.

5.2.6 Mode Classes

The description of a mode class consists of the set of mode names, $\{{}^{Md}\mathsf{m}_1, {}^{Md}\mathsf{m}_2, \ldots, {}^{Md}\mathsf{m}_k\}$, and the function, M : **Hist** $\rightarrow \{{}^{Md}\mathsf{m}_1, {}^{Md}\mathsf{m}_2, \ldots, {}^{Md}\mathsf{m}_k\}$, mapping each possible history to a mode in the mode class. The mode name is used to represent the characteristic predicate of the mode, so for a mode, m, and a time, t, $m(t)$ is a condition that is *true* if and only if the history on $[t_i, t]$ is in m. Note that by convention t is implicitly t_f, so this condition is denoted by "m".

In this thesis we permit two forms of definition for M, as follows.

Direct Definition

In cases where the current mode is a reasonably simple function of recent events, it may be best to define the current mode function using the standard logic operators and tabular expressions. For example, as described in Figure 5.6, the logic probe SRD uses one mode class ${}^{Cl}\mathsf{probe}$ comprised of two modes: ${}^{Md}\mathsf{test}$ and ${}^{Md}\mathsf{pulse}$. The mode ${}^{Md}\mathsf{pulse}$ contains those histories at the end of which the probe tip is being "pulled low" ($<$ 100 ms after the pulse button was pressed). All other histories are in ${}^{Md}\mathsf{test}$. SCR does not use this form of mode class definition.

Finite State Automaton

An alternative form of mode class description, which is used in SCR, is to define a finite state automaton (FSA) in which the states represent the modes; the transition function specifies the possible next mode for any combination of current mode and event; and the initial state of the FSA is defined by a function on the initial conditions. Thus, $MC = \left({}^{Cl}\mathsf{M}, \delta, m_0\right)$ where

- ${}^{Cl}\mathsf{M}$ is the set of modes in the mode class,
- $\delta : {}^{Cl}\mathsf{M} \times \mathbf{Ev} \rightarrow {}^{Cl}\mathsf{M}$ is the mode transition function, and

Mode Class : ^{Cl}probe

Modes : ^{Md}test, ^{Md}pulse

Current Mode function :

$p_T : H_1$ $r_T : G$ Normal	
$@\mathbf{T}\,(^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down}) = \oslash \vee$ $Since(@\mathbf{T}\,(^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down})) > 100$ ms	^{Md}test
$@\mathbf{T}\,(^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down}) \neq \oslash \wedge$ $Since(@\mathbf{T}\,(^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down})) \leq 100$ ms	^{Md}pulse

Figure 5.6: Logic Probe Mode Class—Direct Definition

- $m_0 : \mathbf{Cnd} \rightarrow {}^{Cl}\mathsf{M}$ is the initial mode function.

Mode transition functions can be described using any of three types of SCR mode transition tables. In [2] two of these were said to "not fit the [semantic] model of tables." They can be interpreted in the model, however, as follows. The form illustrated in Table 5.5 is a re-arrangement of the form illustrated in Table 5.4, which does fit the model, so the same semantics are used. The form illustrated in Table 5.6 can be interpreted in the model by treating it as 3-dimensional table that has been flattened out for printing as illustrated in Figure 5.4, and rows containing the empty event class (i.e., transitions that never occur) have been deleted. (Table 5.7 illustrates Table 5.6 showing all cells.) In this style of table the short-hand notations from the second column of Table 5.3 are used to denote events. Tables 5.4, 5.5 and 5.6 all describe the mode transition relation for ^{Cl}probe.

One potential difficulty with modelling a mode class as a FSA is that it is potentially ambiguous in the presence of "simultaneous events"—an event that is in two or more relevant event classes. If the mode transition function gives different transitions for two or more event classes that are not disjoint, then the specification is ambiguous as to the next mode at the instants where the event classes intersect. Such a FSA does not correctly describe a mode class since, according to Definition 5.8, the modes

Mode Class : ^{Cl}probe

Modes : ^{Md}test, ^{Md}pulse

Initial Mode : ^{Md}test

Transition Relation :

Mode	Event	New mode
^{Md}test	@T (^{m}Pulse = ^{s}Down)	^{Md}pulse
^{Md}pulse	@T $\left(Since(@\mathbf{T}\left(^{Md}\text{pulse}\right)) > 100 \text{ ms}\right)$	^{Md}test

Figure 5.7: Logic Probe Mode Class—FSA Definition

Table 5.4: Mode Transition Table 1

^{Md}test	*false*	@T (^{m}Pulse = ^{s}Down)
^{Md}pulse	@T $\left(Since(@\mathbf{T}\left(^{Md}\text{pulse}\right)) > 100 \text{ ms}\right)$	*false*
$p_T : H_1 \wedge G$ $r_T : H_2$ Inverted	^{Md}test	^{Md}pulse

in a class must be disjoint. This potential source of specification errors has been recognized for some time with respect to the SCR method and several techniques have been suggested to help avoid such errors (e.g., [11, 10, 29, 30]). As discussed in Section 5.2.5, this work extends the event class notation of SCR to make description of disjoint event classes easier, so this problem is not as pronounced here as it is in earlier work. Also, since we do not define mode classes in terms of the FSA model, but rather use it as one means to describe equivalence classes of histories, we are free to use other techniques to describe mode classes where they lead to more concise or readable descriptions, as illustrated in Figure 5.6.

5.2.7 Controlled Value Relations

The characteristic predicate of the acceptable values of controlled values is given using standard predicate and relational operators and tabular expressions. These are

Table 5.5: Mode Transition Table 2

Mode	Event	New mode
${}^{Md}\mathsf{test}$	$@\mathrm{T}\,({}^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down})$	${}^{Md}\mathsf{pulse}$
${}^{Md}\mathsf{pulse}$	$@\mathrm{T}\left(\mathit{Since}(@\mathbf{T}\left({}^{Md}\mathsf{pulse}\right)) > 100\text{ ms}\right)$	${}^{Md}\mathsf{test}$

Table 5.6: Mode Transition Table 3

$\mathrm{p_T} : H_1 \wedge G$ $\mathrm{r_T} : H_3$ Decision	${}^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down}$	$\mathit{Since}(@\mathbf{T}\left({}^{Md}\mathsf{pulse}\right)) > 100\text{ ms}$	
${}^{Md}\mathsf{test}$	$@\mathrm{T}$	$*$	${}^{Md}\mathsf{pulse}$
${}^{Md}\mathsf{pulse}$	$*$	$@\mathrm{T}$	${}^{Md}\mathsf{test}$

expressed in terms of the previous behaviour, current mode in one or more mode classes, and condition values. Note that controlled value relations may be defined in terms of past values of monitored or controlled quantities. For example the value of ${}^{m}\mathsf{i}$ at time t_j is denoted ${}^{m}\mathsf{i}(t_j)$, and its value δ ago is ${}^{m}\mathsf{i}(t_f - \delta)$. In the logic probe example, the probe light is required to be ${}^{s}\mathsf{Red}$ whenever the measured voltage represents a logic "high" signal (${}^{m}\mathsf{Vtip} \geq 2.0$ V), which could be written as "${}^{c}\mathsf{Light} = {}^{s}\mathsf{Red} \Leftrightarrow {}^{m}\mathsf{Vtip} \geq 2.0$ V". Figure 5.8 illustrates controlled value relations for the logic probe example.

5.2.8 Merit Functions

It is often the case that, although all behaviours in **REQ** are acceptable, some are preferable over others. For example, the following may be figures of merit for a

Table 5.7: Mode Transition Table 3, all cells

$p_T : H_1 \wedge G$ $r_T : H_3$ Decision	${}^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down}$	$Since(@\mathbf{T}\left({}^{Md}\mathsf{pulse}\right)) > 100$ ms	
${}^{Md}\mathsf{test}$	@T	$*$	${}^{Md}\mathsf{pulse}$
	$\oslash$	$\oslash$	${}^{Md}\mathsf{test}$
${}^{Md}\mathsf{pulse}$	$\oslash$	$\oslash$	${}^{Md}\mathsf{pulse}$
	$*$	@T	${}^{Md}\mathsf{test}$

system:

Processing speed : Although a range of delays are acceptable, quicker responses are preferred.

Soft real-time constraints : Failure to respond within a specified time is not catastrophic, but it is undesirable.

Safety margins : It is acceptable for controlled values to approach, but not cross, certain thresholds, but the larger the safety margin (i.e., difference between the actual value and the threshold) the better.

Stability : Large oscillations in the controlled values are undesirable.

Definition 5.13 *A* merit function *is a function of a behaviour that indicates which behaviours are preferred over which others—the higher the merit function value the more preferred the behaviour.*

Controlled Values

${}^{c}\mathsf{Vdisp}$

$= {}^{m}\mathsf{Vtip}$

${}^{c}\mathsf{Light}$

$=$

${}^{p}\mathsf{low}$	${}^{p}\mathsf{float}$	${}^{p}\mathsf{high}$
${}^{s}\mathsf{Green}$	${}^{s}\mathsf{Off}$	${}^{s}\mathsf{Red}$

${}^{c}\mathsf{Requiv}$

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : G$ Vector	${}^{Md}\mathsf{test}$	${}^{Md}\mathsf{pulse}$
${}^{c}\mathsf{Requiv}$	${}^{c}\mathsf{Requiv} \geq 500\ \mathrm{k}\Omega$	${}^{c}\mathsf{Requiv} \leq 320\ \Omega$

Figure 5.8: Logic Probe Controlled Values

Merit functions allow the specifier to indicate which aspects of the behaviour are relevant to choosing between behaviours, and to state quantitatively their relationship. This is quite similar to what is done in control systems and optimization where the goal is to minimize or maximize the value of an *objective function.*

5.2.9 Environmental Constraints

The **NAT** relation characterizes the set of possible environmental state functions as determined by the constraints imposed by the environment. This section presents some techniques for expressing **NAT** by describing constraints on the monitored and controlled quantities. **NAT** is the weakest relation (largest set) satisfying all the stated constraints.

Exclusive Conditions

Often some of the conditions used in an SRD are mutually exclusive, i.e., one being *true* implies that the other is *false*. Where these exclusions are not obvious they should be stated explicitly using logical formulae, e.g., ${}^{p}\mathsf{C}_1 \Rightarrow \neg {}^{p}\mathsf{C}_2$.

Possible Simultaneous Changes

As discussed in Section 5.2.6, SRD authors must ensure that the event classes used to define mode transition functions are disjoint. Often event classes that may seem from their definitions to intersect, are actually disjoint due to environmental constraints that, for example, preclude certain conditions from changing value simultaneously. Knowledge of the possibility or impossibility of such simultaneous changes can be essential for checking that all possibilities are addressed by the SRD.[30] One simple notation for describing possible subsets of changes is a matrix representation (similar to an adjacency matrix used to represent a graph) that indicates which pairs of conditions can and cannot change simultaneously. For example Table 5.8 denotes that the following sets of conditions may change simultaneously: $\{\{C_1, C_2\}, \{C_1, C_4\}, \{C_1, C_2, C_4\}, \{C_2, C_3\}, \{C_2, C_4\}\}$. This notation is not general enough to describe all restrictions on simultaneous changes, for example it can't express that the number of simultaneous changes is limited (e.g., only ten buttons can be pressed at the same time because the operator only has ten fingers), but it is sufficient for many practical situations. In cases where there are more complicated restrictions different notation will be needed.

Table 5.8: Possible Simultaneous Changes in Conditions

	C_2	C_3	C_4
C_1	Y	N	Y
C_2		Y	Y
C_3			N

5.2.10 Tolerance and Accuracy Relations

The following are some techniques for specifying tolerance and accuracy relations.

Transition Modes

Environmental modes and mode classes are distinct from the "system modes" and "system mode classes" used in the SCR project and much of the work that has followed from it (e.g., [11, 29, 41]). System modes are defined to be equivalence

classes of (target) system states whereas environmental modes are equivalence classes of histories. Changes in the environmental mode depend only on the occurrence of events; changes in the system mode depend on event detection and system response to it. In addition, the initial system mode can be assumed to be fixed, whereas the initial environmental mode depends on what has happened before the system was turned on. For the ideal system requirements the system and environmental modes should be equivalent.

A common form of tolerance relation arises from the fact that it is not possible for realistic systems to respond instantaneously to changes in the environment, but it is convenient to describe the modes as if it is, and to specify the controlled values in terms of those ideal modes, as in Figure 5.8. This form of tolerance relation specifies the acceptable duration of the periods following a mode transition when the actual controlled values are permitted to differ from those specified in the ideal controlled value relations. One useful technique for doing this is to treat the system modes as controlled variables, and to modify an environmental mode class to include *transition modes* corresponding to the periods when the system mode differs from the environmental mode. The behaviour of the system in the transition modes, including the maximum (or minimum) period it can remain in the mode and its response to environmental changes, can be specified in the same manner as for other modes.

In many cases the required behaviour while in the transition mode is the same as for the source of the transition and only the maximum duration of the transition is important. Such requirements can be concisely specified by giving either a single maximum delay for all transitions as part of the mode class definition, as is done in Figure 5.9, or, if different delays are permissible for different transitions, the maximum delay for each transition can be presented in a tabular form, similar to the mode transition table.

Formally this is interpreted as follows. Each mode, m_j, in the ideal mode class is partitioned into a set of sub-modes, one transition mode corresponding to each transition in the ideal mode transition relation that has m_j as its destination, plus one extra, $\widehat{m_j}$, known as the *non-transition mode* (i.e., $m_j = \bigcup_{((m_i,e),m_j)\in\delta} \left(m_{(i,e,j)}\right) \cup \widehat{m_j}$).

Mode Class : ^{Cl}probe

Modes : ^{Md}test, ^{Md}pulse

Initial Mode : ^{Md}test

Transition Relation :

Mode	Event	New mode
^{Md}test	@T (^{m}Pulse = ^{s}Down)	^{Md}pulse
^{Md}pulse	@T $\left(Since(@\mathbf{T}\left(^{Md}\text{pulse}\right)) > 100 \text{ ms}\right)$	^{Md}test

Maximum Delay : 2 ms

Figure 5.9: Logic Probe Mode Classes, with Delay Specification

These sub-modes are defined as follows,

$$m_{(i,e,j)} \stackrel{\text{df}}{=} \left\{ \mathsf{H} \in \mathbf{Hist} \;\middle|\; \begin{array}{l} (\forall t, Last(@\mathbf{F}\,(m_i)) < t \leq t_f \Rightarrow \mathsf{H}\,[t] \in m_j) \\ \wedge Since(@\mathbf{F}\,(m_i)) < \text{Delay}(m_i, e, m_j) \\ \wedge {}^{c}\mathsf{mode} \neq m_j \end{array} \right\} \tag{5.2}$$

and

$$\widehat{m_j} \stackrel{\text{df}}{=} m_j \setminus \bigcup_{((m_i,e),m_j)\in\delta} m_{(i,e,j)} \tag{5.3}$$

where $\mathsf{H}\,[t]$ denotes the history on the interval $[t_i, t]$, ^{c}mode represents the system mode, and Delay is the maximum delay specified for the given transition. The mode $m_{(i,e,j)}$ represents the period between the event, e causing a transition from m_i to m_j, and the system's response to e. The mode $\widehat{m_j}$ represents the periods when the environmental and system modes are both m_j.

For example, in Figure 5.9, the maximum delay for a transition in ^{Cl}probe is specified to be 2 ms, which means that the probe may take up to 2 ms to respond to a change in the environmental mode. For illustration, the complete mode class is described in Figure 5.10, although such a mode class description would not normally be written in the SRD.

Mode Class : ${}^{Cl}\mathsf{probe}'$

Modes : ${}^{Md}\widehat{\mathsf{test}}$, ${}^{Md}\widehat{\mathsf{pulse}}$, ${}^{Md}\mathsf{test-pulse}$, ${}^{Md}\mathsf{pulse-test}$

Initial Mode : ${}^{Md}\widehat{\mathsf{test}}$

Transition Relation :

Mode	Event	New mode
${}^{Md}\widehat{\mathsf{test}}$	@T $({}^{m}\mathsf{Pulse} = {}^{s}\mathsf{Down})$	${}^{Md}\mathsf{test-pulse}$
${}^{Md}\mathsf{test-pulse}$	@T $({}^{c}\mathsf{Requiv} \leq 320\ \Omega)$	${}^{Md}\mathsf{pulse}'$
	@T $\left(\mathit{Since}(@\mathbf{F}\left({}^{Md}\widehat{\mathsf{test}}\right)) \geq 2\text{ ms}\right)$	
${}^{Md}\widehat{\mathsf{pulse}}$	@T $\left(\mathit{Since}(@\mathbf{T}\left({}^{Md}\widehat{\mathsf{pulse}}\right)) > 100\text{ ms}\right)$	${}^{Md}\mathsf{pulse-test}$
${}^{Md}\mathsf{pulse-test}$	@T $({}^{c}\mathsf{Requiv} \geq 500\ \mathrm{k}\Omega)$	${}^{Md}\widehat{\mathsf{test}}$
	@T $\left(\mathit{Since}(@\mathbf{F}\left({}^{Md}\widehat{\mathsf{pulse}}\right)) \geq 2\text{ ms}\right)$	

Figure 5.10: Logic Probe Complete Mode Classes

When this form of transition mode specification is used, the required behaviour, including response to environmental changes, during the transition modes is the same as for the source mode of the transition. Thus, for example, if an environmental change occurs during the transition mode resulting in another mode transition, the system mode may never be in the destination mode—the first transition may appear to be skipped by the system.

Clearly not all transition behaviours can be expressed using such a simple extension of the ideal mode class. In cases where the behaviour during transition is important it may be necessary to explicitly identify these periods (as modes) and specify the acceptable behaviour during them. In such cases the mapping between the ideal mode classes and the acceptable mode classes may be complex.

Note also that, since the definition of transition modes treats system modes as a controlled quantity, a monitor system must be able to determine the current system mode in order for it to evaluate the behaviour of a system specified using this form. A target system is said to be *mode apparent* with respect to a mode class ${}^{Cl}\mathsf{C}$, if it is always possible to determine the system mode in that class by the value of the

environmental state function. That is, there is a predicate isSysMode(m), expressed in terms of the monitored and controlled variables, such that, for every mode ${}^{Md}\mathsf{m_i}$ in ${}^{Cl}\mathsf{C}$, isSysMode(${}^{Md}\mathsf{m_i}$) is *true* if and only if the current system mode is ${}^{Md}\mathsf{m_i}$.

Timing Requirements

For real-time systems, time relative to some initial time is always a monitored variable, whether it is explicitly used by the system or not. Although a continuous model of time is used in this work, the requirements should be such that it is possible to produce acceptable implementations of the target systems using a discrete clock. Such requirements can be concisely stated by specifying the required *resolution*—smallest significant increment—of time. Specifying that the resolution of time is δ implies:

- the system clock frequency is required to be at least $\frac{1}{\delta}$,
- it is sufficient to sample monitored quantities at a rate of $\frac{1}{\delta}$,
- changes in the environment that occur within δ of each other may be considered to be simultaneous,
- the system can only be required to detect conditions that have held for at least δ,
- instants may be measured with a maximum accuracy of $+0/-\delta$,
- durations of time intervals can be measured with a maximum accuracy of $\pm\delta$, and
- the delay tolerance for response to any event must be at least δ.

These points have some subtle implications with regards to what can be implemented in real systems. Consider the time lines in Figure 5.11, in which the vertical dashed lines represent hypothetical sample points. If the system is required to respond to ${}^{p}\mathsf{C1}$ being true for the duration $d-a$, then it must also be allowed to respond to it being true for duration $g-f$ and cannot be required to detect duration $j-i$—i.e., it must allow non-deterministic behaviour for durations, D, such that $0 < D < \delta$. Similarly, if the system is required to not respond to ${}^{p}\mathsf{C2}$ being true for duration $c-b$,

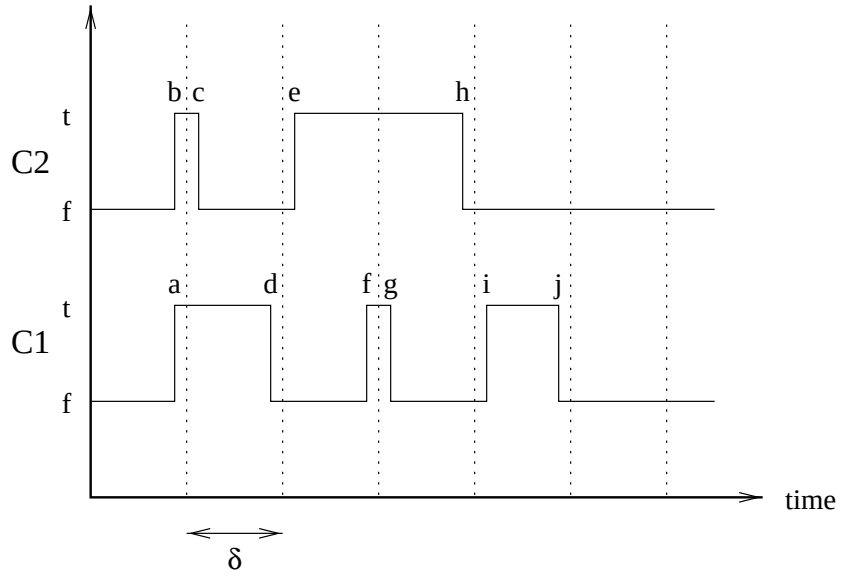

Figure 5.11: Time Resolution

then it must be allowed to not respond to it being true for duration $h - e$. In general any time interval, Δ, in the specification should be such that $\Delta = n\delta$ for some integer n and the system must be allowed to behave non-deterministically for intervals, D such that $\Delta - \delta \leq D \leq \Delta + \delta$. Note that from the point of view of the given sample points the histories for ${}^{p}\mathsf{C1}$ and ${}^{p}\mathsf{C2}$ are the same.

In the logic probe example, specifying the required resolution for time as 0.1 ms would mean that the probe is allowed to not respond to the ${}^{m}\mathsf{Pulse}$ button being pressed for periods of less than 0.1 ms and to respond to events in the class "$@\mathbf{T}\left(\mathit{Since}(@\mathbf{T}\left({}^{Md}\mathsf{pulse}\right)) > 100\text{ ms}\right)$" anywhere between 100.0 ms and 100.1 ms after the events in "$@\mathbf{T}\left({}^{Md}\mathsf{pulse}\right)$".

5.2.11 Standard Functions

The following standard functions, some of which are adapted from [102], are useful and will be used in the sequel. As stated above, all these functions are implicitly interpreted with respect to a particular behaviour on the interval $[t_i, t_f]$. By convention, when referring to the "current" time (i.e., t_f) the time argument will be omitted.

Definition 5.14 *For an event class, e, and time, t, $Prev(e,t)$ is the set of events in e that occur prior to t, i.e.,*

$$Prev(e,t) \stackrel{\text{df}}{=} \{x \in e \mid x.t < t\}$$

Definition 5.15 *For an event class, e, and time, t, $Last(e,t)$ is the time of the latest event from e before t.*

$$Last(e,t) \stackrel{\text{df}}{=} \begin{array}{|c|c|}\hline Prev(e,t) \neq \oslash & Prev(e,t) = \oslash \\ \hline \max(\{x \mid \exists y \in Prev(e,t), y.t = x\}) & 0 \\ \hline \end{array}$$

Definition 5.16 *For an event class, e, and time, t, $First(e,t)$ is the time of the earliest event from e before t.*

$$First(e,t) \stackrel{\text{df}}{=} \begin{array}{|c|c|}\hline Prev(e,t) \neq \oslash & Prev(e,t) = \oslash \\ \hline \min(\{x \mid \exists y \in Prev(e,t), y.t = x\}) & 0 \\ \hline \end{array}$$

Definition 5.17 *For a condition, p_i, and time, t, such that $t_i < t \leq t_f$, $Drtn(p_i,t)$ is the duration of time that p_i has been continuously <u>true</u> if $p_i(t)$ is <u>true</u>, otherwise, if $p_i(t)$ is <u>false</u>, then $Drtn(p_i,t) = 0$.*

$$Drtn(p_i,t) \stackrel{\text{df}}{=} \begin{array}{|l||c||c|}\hline \begin{array}{l}\mathrm{p_T} : H_1 \wedge H_2 \\ \mathrm{r_T} : G \\ \text{Normal}\end{array} & p_i(t) & \neg p_i(t) \\ \hline\hline Prev(@\mathbf{T}(p_i), t) \neq \oslash & t - Last(@\mathbf{T}(p_i), t) & 0 \\ \hline Prev(@\mathbf{T}(p_i), t) = \oslash & t - t_i & 0 \\ \hline \end{array}$$

Definition 5.18 *For a condition, p_i, and times, t_1 and t_2 such that $t_i \leq t_1 \leq t_2 \leq t_f$, $totalDrtn(p_i, t_1, t_2)$ is the total amount of time that p_i has been <u>true</u> between t_1 and t_2.*

$$totalDrtn(p_i, t_1, t_2) \stackrel{\text{df}}{=} \int_{t_1}^{t_2} onTime(p_i, t)\, dt$$

where $onTime(p_i, t)$

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1$
$\mathrm{r_T} : G$
Normal

$p_i(t)$	1
$\neg p_i(t)$	0

Definition 5.19 *For an event class, e, and time, t, $Since(e, t)$ is the time elapsed since the latest event in e before t.*

$$Since(e, t) \stackrel{\text{df}}{=} t - Last(e, t)$$

5.3 Limitations

The chosen requirements model, as described in Section 1.1, limits the classes of properties that can be described using these techniques. In order for the requirements to be expressible using this method, both of the following statements must be true, as discussed below.

1. The environmental quantities can be expressed as functions of time that are either piecewise-continuous, for **Real** valued quantities, or finitely variable for discrete valued quantities.

2. The acceptable behaviour can be characterized by a relation on the environmental quantities.

5.3.1 Environmental Quantities

Piecewise-continuous functions of time are an appropriate model for describing environmental quantities relevant to many engineering systems, such as process control, automation, and embedded systems, where the environmental quantities represent

physical properties (e.g., temperature, pressure, current, position). Quantities that are modelled as partial functions of time can be easily modelled by total functions, for example, with range consisting of pairs, where one element is a Boolean indicating if the other element is meaningful at that time.

For other systems, particularly so called "information processing" systems, some or all of the relevant environmental quantities cannot be modelled because they either cannot be effectively described mathematically, or are not usefully viewed as functions of time.

The Monitor Generator tool, described in Chapter 6, is an example of such a system that cannot be effectively modelled using these techniques. In that case the relevant environmental quantities are specifications and program code, which cannot be effectively related using mathematical and logic notation. These quantities can be modelled, for example, as arrays of characters, and many mathematical statements could be made about them, but such expressions would not provide a useful model for describing the behaviour of the tool. Also, only two instants are relevant to the tool behaviour: when it is started, and when it terminates.

5.3.2 Requirements Relation

Clearly non-behavioural properties (e.g., maintainability, code size) cannot be expressed in **REQ**, and internal properties (e.g., restrictions on the number of times a particular instruction is invoked) can be expressed only if appropriate quantities, which may not be externally observable, are included in the environmental quantities (e.g., count of the number of invocations of a particular instruction). A less obvious limitation is imposed by the interpretation of **REQ**—if $(\underline{m}^t, \underline{c}^t) \in \mathbf{REQ}$ then $(\underline{m}^t, \underline{c}^t)$ is an acceptable behaviour. As is pointed out in [1] and [64], this interpretation is not sufficient for those requirements that are not "preserved under sub-setting" (i.e., if the possible behaviours of system A are a subset of the possible behaviours of system B, B may be acceptable but A not).

One form of such requirements specifies properties of all behaviours actually exhibited or possible, such as "the average response time must be T". Note that this is different from properties on the lifetime of a particular system, which can be modelled as a single behaviour, or properties that must be always true for a particular

execution (e.g, "at all times the average of the response times must be $T \pm 10\%$"), both of which can be expressed. These requirements constrain the set of possible behaviours, rather than any particular behaviour.

While statistical properties such as "average response time" can usually be approximated, and specified, reasonably well with reference to one sufficiently lengthy execution, there are some requirements for which this is not sufficient. For example, requirements—called *possibilistic* properties in [64]—such as "if behaviour A is possible, then behaviour B must also be possible". Such requirements are necessary when security is a concern, so that, for example, intruders cannot infer information from the possibility of A and not B. Although it is easy enough to express that $A \in \mathbf{REQ} \Leftrightarrow B \in \mathbf{REQ}$, this does not have the desired interpretation—it only states that A is an *acceptable* behaviour if and only if B is acceptable. Several examples of this class of requirement are discussed in [64], including non-interference, integrity and availability.

Chapter 6

Monitor Generation

This chapter describes the technique and prototype tool—the Monitor Generator (MG)—that, using an SRD as input, generates those modules of a monitor that are directly dependent on it. The tool is implemented as part of the Table Tool System (TTS) project[73, 96] and makes extensive use of the services that it provides. A particular monitor is described in Chapter 7.

6.1 Input Format

The following is an overview of the format of the input to the monitor generator. A more complete description is given in Appendix B.

The input to the monitor generator consists of the following:

1. A "TTS context" file (see [96]), which contains the expressions that make up the SRD, and

2. optionally, the name of a file containing the System Interface Declarations.

The context file contains the following expressions:

Environmental variables For each environmental variable used in the SRD an expression must be given containing just that variable. This serves to inform the MG of the set of environmental variables.

Conditions All conditions that are used in defining event classes in the SRD must be named and defined as separate expressions.

History expressions All expressions (typically simply variables), excluding conditions, for which 'previous' values are needed in the SRD must be listed separately. Approximation of the value of variables at times prior to the most recent sample point is accomplished by the `timefunc` class template as described in Section 6.3.1. A few functions of past values are also provided (e.g., `dBYdt`, `before`), and other functions (e.g., numerical integration) could be added if needed.

Dictionary expressions Any non-standard functions or predicates that are defined in the dictionary section of the SRD must be given.

Mode Classes For each mode class a special mode class definition expression must be given. It lists the set of modes in the mode class. In addition, expressions must be given to define the initial mode, mode transition relation, transition delay relation and mode apparency predicate.

Controlled Value Relations Definitions must be given for all controlled value relations used in the SRD.

Merit Function (optional) If a merit function is used its definition must be given.

Behaviour expression An expression, labelled "Behaviour", that gives the characteristic predicate of the acceptable behaviour. This expression is usually simply the conjunction of the characteristic predicates of the controlled value relations.

The MG, like all tools in the TTS, manipulates and interprets expressions using the "table holder" modules, which store an abstract representation of expressions. The exact syntax of the expressions used in the SRD is thus determined by other componnents of the TTS. In particular, since the test oracle generator is the primary tool used to convert expressions into C++ code, it limits the form of expressions that can be used as described in [79]. Expressions are entered into the TTS using the "table construction tool", so that tool determines the syntax accepted [96].

6.2 Monitor Modularization

The monitor software is divided into three top level information hiding modules,[74] as follows.

Monitor Control This module hides the design decisions relating to how users (possibly including other software) can interact with the monitor and how the user's commands are implemented through calls to other monitor components. This module may need to be changed if the manner in which the monitor is to be used is changed.

System Behaviour This module hides the design decisions related to the required behaviour of the system to be monitored. This module will need to be modified if either the definition of acceptable behaviour (i.e., **REQ**) of the system, or the interface to the System Interface module are changed.

System Interface This module hides the interface between the monitor system and the target system. It will need to be changed if the set of monitored and controlled quantities is changed, or if the means of observing the value of these quantities (i.e., $\mathbf{IN_{mon}}$) is changed. The implications of the system interface design on the effectiveness of the monitor are discussed in Sections 3.3 and 4.1.

6.3 System Behaviour Module Components

The system behaviour module is made up of the Behaviour module, which contains all of the automatically generated code, and the Monitor Library, which is not automatically generated and contains class and class template definitions used by the Behaviour module. Both of these are implemented in C++, so the terminology of that language is used in the following sections.

6.3.1 Monitor Library

The monitor library defines two classes, `Condition` and `Timestamp`, and two class templates,[1] `Timefunc` and `ModeClass`, which implement the standard functions defined

[1] A *class template* is a class that has types, possibly including other classes, as parameters.

in Section 5.2.11 and the interpretation of mode classes. These are briefly described below. Detailed documentation is given in Appendix C.

Conditions

Secret Representation of condition histories.

Service Allows programs to determine if an event is in a given event class.

Modules used Time Functions, Time Stamp.

Files `condition.cc`, `condition.h`

This module implements the `Condition` class, which is derived from `timefunc<bool>` (a Boolean valued Time Function), for keeping a history of changes in the value of a condition. In addition, the programs described in Table 6.1 are implemented.

Table 6.1: Condition Module Programs

Program	Description
`Timestamp Last(Change ch, Condition& cnd)`	Most recent event in the given event class. `ch` is either `AtT` (representing @**T** (`cnd`)) or `AtF` (representing @**F** (`cnd`)).
`Timestamp First(Change ch, Condition& cnd)`	Earliest event in the given event class.
`double Since (Change ch, Condition& cnd, Timestamp& t)`	Time elapsed between an event in the given event class and time `t`.
`double Drtn (Condition& cnd, Timestamp& t)`	Amount of time that `cnd` has been continuously *true* up to time `t`.
`double totalDrtn (Condition& cnd, Timestamp& t1, Timestamp& t2)`	Total amount of time that `cnd` has been true between times `t1` and `t2`.
`bool AtTrue(Condition& cnd, Timestamp& t)`	$\exists e \in @\mathbf{T}(\texttt{cnd}), \texttt{t} = e.t.$
`bool AtFalse(Condition& cnd, Timestamp& t)`	$\exists e \in @\mathbf{F}(\texttt{cnd}), \texttt{t} = e.t.$
`bool When(Condition& cnd, Timestamp& t)`	$\exists e \in \mathbf{WHEN}(\texttt{cnd}), \texttt{t} = e.t.$
`bool While(Condition& cnd, Timestamp& t)`	$\exists e \in \mathbf{WHILE}(\texttt{cnd}), \texttt{t} = e.t.$
`bool WhenF(Condition& cnd, Timestamp& t)`	$\exists e \in \mathbf{WHEN}(\neg\texttt{cnd}), \texttt{t} = e.t.$
`bool WhileF(Condition& cnd, Timestamp& t)`	$\exists e \in \mathbf{WHILE}(\neg\texttt{cnd}), \texttt{t} = e.t.$

Mode Classes

Secret Method for determining the actual system mode from an ideal mode transition function and tolerance function.

Service Provides access to the current mode in each class.

Modules used Time Functions.

File `modeclass.h`

This module implements `ModeClass<T>`, a class template that models a mode class, including transition modes. The initial mode function, ideal mode transition function, transition delay tolerance function and mode apparency function are assumed to be defined in the parameter class, as described in Table 6.3. The public members are listed in Table 6.2.

Table 6.2: Mode Class Members

Member	Description
`M update(void)`[2]	Update the system mode.
`M operator() (void)`	Returns the current system mode.
`double Since (ModeChange ch, M& m, Timestamp& t)`	Returns the time elapsed since the most recent occurrence of the given change. `ch` is either `Enter` or `Leave`, and `m` is the mode.
`double Drtn (const M& m, Timestamp& t)`	If the current system mode is `m` then it returns the time elapsed in that mode.

Time Functions

Secret Method for maintaining a history of samples of a function of time.

Service Provides access to the value of the function at any time in the past.

Modules used Time Stamp.

File `timefunc.h`

[2] `M` is a type name for `T::mode`.

Table 6.3: Mode Class Parameter Members

Member	Description
`mode`	Enumerated type specifying the possible modes.
`mode init(void)`	Initial mode function.
`mode trans(mode m)`	Ideal mode transition function. Global conditions are evaluated to determine the event classes containing the current event.
`double delay(mode s, mode d)`	Maximum delay for a transition between mode `s` and `d`.
`bool isSysMode(mode m)`	Mode apparency function. True if and only if the current system mode is `m`.

This module implements the `Timefunc<T>` class template that retains a sequence of samples of a quantity, of type `T`, so that it can be treated as a function of time. Clearly, since a sampled representation of the function is used, the value of the function between samples can only be approximated. For instants between two samples, the value is approximated by the value of the earlier sample. The public members are given in Table 6.4. The parameter type must have a copy constructor, assignment operator and stream output operator (`<<`).

Time Stamp

Secret Representation of time in the system.

Service Operations on time values (e.g., addition, subtraction, comparison).

Modules used none.

Files `timestamp.cc`, `timestamp.h`

This module implements the `Timestamp` class, which is used to represent time values within the monitor. The public members are given in Table 6.5.

6.3.2 Behaviour Module

The behaviour module contains all of the code directly generated from the SRD. The interface to this module consists of three access programs, as described in Table 6.6,

Table 6.4: Time Function Members

Member	Description
`Timefunc(int len, bool keepInit)`	Constructor. Keep `len` samples. If `keepInit` is true always keep the oldest sample. If `len` is 0 then keep all samples.
`Timefunc(double oldest, bool keepInit)`	Constructor. Keep all samples less than `oldest` seconds old.
`Timefunc& addPoint(Timestamp t, T v)`	Add a sample point to the history.
`T operator() (void)`	Get the most recent value.
`T operator() (Timestamp t)`	Get the value at time `t`.
`T after(Timestamp t)`	Get the value immediately after time `t`
`T before(Timestamp t)`	Get the value immediately before time `t`
`T oldest(void)`	Get oldest value.
`T dBYdt(void)`	Return an estimate of the rate of change of the function using the most recent two function values.
`bool empty()`	True if there are no sample points in the history.
`iterator begin()`	Iterator pointing to the start of the history.
`iterator end()`	Iterator pointing past the end of the history.

and the variables representing the environmental quantities, which can be accessed directly.

The implementation of the module contains the following sections.

System Interface Declarations Code needed so that the behaviour module can access the system interface module is copied directly from a file specified by the user. This allows any appropriate C++ type or function to be defined elsewhere and used by the behaviour module. This is similar to the "user definitions" included in a test oracle implementation in [79].

Environmental Variables Each environmental variable is represented by a global variable declared here.

Conditions A `Condition` object is declared to represent each condition that is used in the SRD.

History An appropriate `Timefunc` object is declared to represent the past values of each environmental variable for which previous values are used in the SRD.

Table 6.5: Time Stamp Members

Member	Description
`Timestamp(int sec = 0, int usec = 0)`	Constructor.
`Timestamp(timeval t)`	Constructor.
`Timestamp(double d)`	Constructor.
`int seconds(void)`	Get the seconds component of the time stamp.
`int useconds(void)`	Get the microseconds component of the time stamp.
`Timestamp& set(int s, int u)`	Set the value of the timestamp.
`Timestamp& operator+=(Timestamp t)`	Increment.
`Timestamp& operator+=(double sec)`	Increment.
`Timestamp& operator-=(Timestamp t)`	Decrement.
`Timestamp& operator-=(double sec)`	Decrement.
`operator double()`	Convert to floating point representation.

Table 6.6: Behaviour Module Programs

Program	Description
`void MON_update(void)`	Update all time functions, conditions, and mode classes using the current values of the environmental quantities.
`bool MON_behaviour(void)`	Returns true iff the current behaviour is acceptable according to the SRD.
`void initOracle(void)`	Initialize the behaviour module.

Mode Classes A class with members as described in Table 6.3 is defined for each mode class, and it is used as a parameter, `T`, for the `ModeClass<T>` template to instantiate an object to represent the mode class.

Controlled Value Relations The characteristic predicate of each of the controlled value relations defined in the SRD is implemented as a separate C++ function.

Dictionary Each of the functions and predicates defined in the dictionary section of the SRD is implemented as a C++ function.

Interface programs The implementation of the interface programs described in Table 6.6.

6.4 Code Generation

In my previous work [79, 80], I have shown how an oracle can be automatically generated from relational program documentation, which gives the characteristic predicate of the relation describing the acceptable start and stop state pairs for a single program. This oracle determines if the values of the program variables in the starting and stopping states of the program execution are in the relation by evaluating its characteristic predicate. A major component of that work is a procedure for generating C++ code to evaluate the predicates or functions defined using tabular notations. A slightly modified version of that procedure is used here to generate code for evaluation of the functions and predicates defined in the SRD.

The modifications to existing TTS modules are described in the remainder of this section. The following section describes the new Monitor Generator modules.

6.4.1 TTS Modifications

Three TTS modules are involved in generation and execution of code that evaluates tabular expressions: the Table Evaluation Module (`liboracle.a`), Test Oracle Generator (TOG), and Generalized Table Semantics (GTS). These have been modified to support a wider variety of tables, as follows.

- Cell expressions may include constants representing other grids. To evaluate the table these constants are replaced by the appropriate expression from that grid. The cell expression can thus be constructed from expressions from more than one grid.

- Table rules may contain variables, function applications and arbitrary scalar logical operators.

The primary motivating example for this change is the form of mode transition table, illustrated in Table 5.6, in which the notation "@T" in the main grid is taken as a shorthand notation for "@T (H_2)", where H_2 is a constant representing the header that contains conditions in its cells. With these extensions, the condition from the appropriate cell of H_2 is substituted into the main grid expression to form a mode class expression. The changes needed to effect these extensions are described below.

Table Evaluation Module

The table evaluation module has been completely replaced by a new version with members as described in Table 6.7. This version makes extensive use of the C++ template facility and the Standard Template Library.[94, 98] An instance of the `Cell<T>` class is used to represent each of the guard and value components of each raw element for the table. These are stored in `map<Index, Cell<T> >` objects, which allow the index for a true guard element to be found and used to select the appropriate value expression.

Table 6.7: Table Evaluation Module Members

Member	Description
`template<class T> class Cell`	Cell class template. Represents the relation defined by a raw table element.
`T operator ()(int ...)`	Cell value function.
`T operator ()(va_list)`	Cell value function.
`Index`	Table index type.
`template<class T> class Table`	Table class template.
`T operator ()(int ...)`	Returns value of table.
`void setGuardCell(Index& indx, Cell<bool>& c)`	Set the guard function for `indx` to be `c`.
`void setValCell(Index& indx, Cell<T>& c)`	Set the value function for `indx` to be `c`.

Test Oracle Generator

The only interface change required was to the code generation sub-module of TOG (`TOG_code`), which has been substantially revised in those aspects relating to implementing tables, as summarized in Table 6.8. Extensive internal changes were required in the expression sub-module (`TOG_expn`) with respect to the interpretation of tables.

Generalized Table Semantics

Table components (grids) are represented in the rule expressions used by the GTS semantics module (`GTS_semantics`) by constants "H1", "H2", ..., "H10" and "G", rather than by their grid number as stated in [2]. This is required to allow numbers

Table 6.8: TOG_Code Changes

Program	Description/Change
`TOG_codeNewTable`	Generates code to instantiate an appropriate instance of `Table<T>`.
`TOG_codeTableCell`	Implements a table raw element expression as an instance of `Cell<T>`.
`TOG_codeTableSetRules`	New program. Generates code to invoke `setValCell` and `setGuardCell` as appropriate for the table.
`TOG_codeTableGuardInterp`	Removed.
`TOG_codeTableValueInterp`	Removed.
`TOG_codeTableSetHeader`	Removed.
`TOG_codeTableSetMain`	Removed.
`TOG_codeInterpSetGuard`	Removed.
`TOG_codeInterpSetValue`	Removed.

to appear in the table cell and rule expressions without them being treated as representing table grids. The constants "H1" etc., however, are keywords and should only appear in expressions where they represent grids.

6.5 Monitor Generator Modules

This section gives an overview of the design of the monitor generator tool, which is part of the TTS. Appendix C gives more detailed design documentation.

6.5.1 Monitor Constructor

Secret Procedure for generating a monitor from a SRD.

Service Monitor generation.

Modules used Requirements, Specification Object, TOG Interface, Exceptions, TOG.

Files `MG_monitor.h`, `MG_monitor.cc`

This is the main monitor generator module. It has only one access program, `MG_monBuild`, which generates a monitor using the given `MG_Req` object. It invokes the

specification object methods to generate appropriate code for each, and invokes the TOG to implement the auxiliary definitions.

6.5.2 Requirements

Secret Interpretation of a TTS context as an SRD.

Services Implements `MG_Req` class. Allows access to the components of the SRD.

Modules used Specification Objects, TTS Context Manager.

Files `MG_requirements.cc`, `MG_requirements.h`

This module is responsible for the interpretation of the TTS context file as described in Section 6.1. It determines the appropriate `MG_SpecObj` derived class for each expression and instantiates an appropriate object. The public members are given in Table 6.9.

Table 6.9: Requirements Members

Member	Description
`MG_Req(const CHandle c)`	Constructor. Interprets `c` as an SRD.
`CHandle getAuxDefns(void)`	Returns the auxiliary function and predicate definitions from the SRD dictionary section.
`Expn getBehaviour(void)`	Returns the behaviour predicate from the SRD.
`iterator`	Iterator type. Dereference to pointer to `MG_SpecObj`.
`iterator begin(void)`	First specification object in SRD.
`iterator end(void)`	Past the last specification object in SRD.
`bool empty(void)`	`true` if the SRD contains no objects.

6.5.3 Specification Objects

Secret The method for declaring and implementing the class of specification object.

Service Code generation for each type of object.

Modules used TOG Interface, Exceptions.

Files `MG_specObj.cc`, `MG_specObj.h`, `MG_condDefn.cc`, `MG_condDefn.h`, `MG_histDefn.cc`, `MG_histDefn.h`, `MG_envVar.cc`, `MG_envVar.h`, `MG_modeClass.cc`, `MG_modeClass.h`,

This module implements a base class, `MG_SpecObj`, and four derived classes, `MG_EnvVar`, `MG_CondDefn`, `MG_HistDefn` and `MG_ModeClass`, for environmental variables, conditions, history expressions, and mode classes, respectively. Each derived class overrides the `codeDefinition`, `codeDeclaration` and `codeUpdate` methods to generate appropriate code for each type of specification object. Any expression implementation is done by invoking the TOG.

Table 6.10: Specification Object Members

Member	Description
`MG_SpecObj(const char *n, const Expn e)`	Constructor. `n` is the name, `e` is the definition.
`MG_SpecObj(const MG_SpecObj&)`	Copy constructor.
`operator= (const MG_SpecObj&)`	Assignment operator.
`Id getId(void)`	Return the `Id` of the defined object.
`Expn getDefn(void)`	Return the definition of the object.
`string getName(void)`	Return the name of the object.
`ostream & codeDefinition(ostream & s, const TOG_Cntxt cntxt)`	Write the object implementation to `s`.
`ostream & codeDeclaration(ostream & s, const TOG_Cntxt cntxt)`	Write the object declaration to `s`.
`ostream & codeUpdate(ostream & s, const TOG_Cntxt cntxt)`	Write to `s` the code to update the object's state.

6.5.4 TOG Interface

Secret Test Oracle Generator data types.

Service Access to TOG.

Modules used TOG.

Files `MG_tog.cc`, `MG_tog.h`

This module is essentially a wrapper on TOG to allow the monitor generator to use it in a C++ style. It implements one class, `MG_TogProc`, which encapsulates a `TOG_Cntxt`, and two access programs: `MG_togExpn`, which is analogous to its TOG counterpart, `TOG_expn`, only it writes to a stream rather than `TOG_Line`, and `MG_togVarDecl`, which uses TOG to implement a variable declaration.

6.5.5 Exceptions

Secret Handling of TTS error status.

Service Means to throw exceptions that carry TTS status tokens.

Modules used TH_error, TOG_error, TIF_error.

File `MG_exceptions.cc`, `MG_exceptions.h`

This module implements exception classes `MG_Fail` (base class), `TIF_Fail`, `TOG_Fail` and `TH_Fail`, which are thrown by MG modules with appropriate TTS status information when errors occur.

6.6 Limitations

Some of the limitations of this prototype system are as follows.

- Transition modes (see Section 5.2.10) are the only form of tolerance relation for which the tool provides explicit support. Other forms of tolerance and accuracy relations can be used, but they must be explicitly written in the SRD expressions (e.g., "$^{c}\mathsf{x} = {}^{m}\mathsf{y} \pm 0.01$" must be written as "$^{m}\mathsf{y} - 0.01 \leq {}^{c}\mathsf{x} \leq {}^{m}\mathsf{y} + 0.01$"). Support for such standard forms of tolerance relation could reasonably easily be added in future versions.

- Since behaviours not in **NAT** are not possible, the MG does not consider **NAT**.

- Since all expressions in the SRD are implemented using the code generation modules of the TTS, the form of the expressions must be acceptable to those modules[2, 79] with the extensions described in Section 6.4.1. In particular all

variables must be given appropriate C++ data types, and only available operations on those types are permitted (although, since C++ has powerful type and operation definition mechanisms, this is not a significant restriction). Also quantification is limited to the Inductively Defined Predicate (IDP) form described in [79]. While this is a limitation, it is also a feature of the tool, since any future enhancements to the TTS code generation facilities will automatically be available to the monitor generator.

- The monitor is assumed to be a software monitor as described in Section 3.2. For this reason it does not consider non-determinism due to event orders being perceived differently by the monitor and target system, and it does not take into account any error introduced by $\mathbf{IN_{mon}}$. The monitor is thus an optimistic monitor in the sense of Section 3.2 unless the error introduced by the input devices is accounted for by the system interface module or in the controlled value relations.

6.6.1 Real-Time Evaluation

Since the application domain for this work is real-time systems, it would be useful if the monitor could be assured to meet some real-time constraints, for example reporting if the behaviour is acceptable or not within a fixed time. Unfortunately it is impossible to make any such claims, for the following reasons.

1. The computation required to evaluate the behaviour is not, in general, bounded, as discussed in Section 4.4.
2. The choice of system interface will have significant impact on the computation required by the monitor software, as discussed in Section 4.3.
3. The relevant deadlines are dependent on the target system.
4. The monitor operating environment (i.e., operating system, processor, etc.) is unknown.

However, as illustrated in Chapter 7, the set of target systems for which the monitor does operate in real-time is likely to include many practical systems.

Chapter 7

Sample Application

This chapter presents a monitor for a realistic target system—the Maze-tracing Robot—and discusses its use in testing of implementations of that system. The implementations were produced by undergraduate students, under the instruction of Dr. Martin von Mohrenschildt, as the course project for Computer Engineering 3VA3 in the Fall semester of 1997. The students were given a version of the SRD in Section A.4 and required to implement software to control a robot such that it behaved consistent with that specification. The behaviour of the student projects was recorded to be used as input to a monitor.

The implementation of the monitor is presented in the following section. Section 7.2 discusses the results of testing the student projects using the monitor.

7.1 Monitor Implementation

The monitor described in this section is a software monitor in the sense of Section 3.2—it observes the values of the target software input and output variables as described in Section 7.1.2.

7.1.1 Monitor Control

The monitor control module is quite simple for this example. After initialization of the behaviour module, it uses the system interface module to read the maze description from a file, and then read each sample in turn from the file containing the recorded

target system behaviour. After reading each sample it calls `MON_update` to update the monitor and then checks that the behaviour is acceptable by calling `MON_behaviour`. If the behaviour is not acceptable, it stops reading samples, reports the time of the failure and exits. If it gets to the end of the file, it outputs the value of the merit function.

7.1.2 System Interface

For the course project the students were provided with a library of programs, as described in Table 7.1, through which their software could control the robot and read the state of the buttons.[1] The behaviour of each target system was recorded using a version of this library modified to write the values of all of the state variables, including the current time, to a log file on completion of each invocation of an access program. The resulting log files contain a series of samples of $(\underline{i}^t, \underline{o}^t)$ in the form of a tuple: (`i_t`, `arms`, `o_penDown`, `o_powerOn`, `i_stopB`, `i_homeB`, `i_backB`, `o_message`). These samples are assumed to be related to $(\underline{m}^t, \underline{c}^t)$ by Eq. (7.1). The value of ^{c}message is assumed to indicate the system mode.

p_T : *true* r_T : $H_1 = \lvert G$ Vector	
`i_t` =	^{m}t $\pm$ 0.01 seconds
`arms` |	$\lvert {}^{mc}$penPos $-$ `Posn(arms)`$\rvert \leq 0.5$ mm
`o_penDown` =	^{mc}penDown
`o_powerOn` =	^{c}powerOn
`i_stopB` =	^{m}stopButton
`i_homeB` =	^{m}homeButton
`i_backB` =	^{m}backButton
`o_message` =	^{c}message

(7.1)

The description of the maze (i.e., the values of ^{m}mazeWalls, ^{m}mazeStart and ^{m}mazeEnd) was provided in the form of a data file that could be read by the student programs.

[1] The library and their description in Table 7.1 are due to Dr. von Mohrenschildt.

Table 7.1: Robot Interface Programs

Program	Description
`short o_init(void)`	Initialize the library and robot.
`short o_power(int pow)`	Turn the robot power on (`pow` = 1) or off (`pow` = 0). Returns 0 if successful, non-zero otherwise.
`short o_penDown(int pen)`	Move the robot pen down (`pen` = 1) or up (`pen` = 0). Returns 0 if successful, non-zero otherwise.
`short o_penPos(short a, short b)`	Move arm 1 to angle `a` and arm 2 to angle `b`, where $-\frac{1000\pi}{2} \leq \mathtt{a}, \mathtt{b} \leq \frac{1000\pi}{2}$. Returns 0 if successful, non-zero otherwise.
`short o_message(char* mesg)`	Output a message.
`short i_homeButton(void)`	Return 1 if the Home button is pressed, 0 otherwise.
`short i_stopButton(void)`	Return 1 if the Stop button is pressed, 0 otherwise.
`short i_reverseButton(void)`	Return 1 if the Reverse button is pressed, 0 otherwise.

The monitor accesses this data using three sub-modules: Maze, Geometry and Position, as described below.

Maze module

Secret Representation of the maze.

Service Algorithms for reading a maze description. Programs to evaluate predicates on the maze.

Modules used Position.

Files `maze.cc`, `maze.h`

This module provides access to ^{m}mazeWalls, ^{m}mazeStart and ^{m}mazeEnd, which it gets by reading the maze description from the same data file as was used by the target system software. It also implements two predicates used in the SRD: "wall" and "connected". The class members are described in Table 7.2.

Geometry module

Secret Geometry of robot as it determines the relationship between angular positions of the robot arms and the position of the pen tip (^{mc}penPos).

Table 7.2: Maze Class Members

Member	Description
`Maze(istream& ins)`	Constructor. Read the maze description from `ins`.
`void addWall(const Posn& e1, const Posn& e2)`	Add all points on a straight line between `e1` and `e2` (inclusive) to the set of maze wall points (^{m}mazeWalls).
`istream& operator >> (istream& ins, Maze& m)`	Read the maze description from `ins`.
`Posn mazeStart(void)`	Return the start position for the maze (^{m}mazeStart).
`Posn mazeEnd(void)`	Return the finish position for the maze (^{m}mazeEnd).
`bool wall(const Posn& p)`	wall(p)
`bool atStart(const Posn& p)`	p $\in$ tol(^{m}mazeStart)
`bool atFinish(const Posn& p)`	p $\in$ tol(^{m}mazeEnd)
`bool safeMove(const Posn& e1, const Posn& e2)`	$\neg$ ($\exists q \in$ straight line between `e1` and `e2`) (wall(q))
`bool connected(void)`	connected(^{m}mazeStart, ^{m}mazeEnd)

Service Algorithms to convert between angular positions of arms and ^{m}penPos.

Modules used Position.

Files `geometry.cc`, `geometry.h`

This module implements the `RobotArm` class, the public members of which are described in Table 7.3.

Table 7.3: Geometry Class Members

Member	Description
`RobotArm(void)`	Constructor.
`operator Posn (void)`	Convert angular positions to **Posn**.
`istream& operator >> (istream &s, RobotArm& a)`	Read angular positions.

Position module

Secret Representation of a point in space (**Posn**), interpreted as a vector.

Service Algorithms for operating on positions.

Modules used none.

Files `position.cc`, `position.h`

This module implements the `Posn` class, the public members of which are described in Table 7.4. In addition it implements the operations on positions described in Table 7.5.

Table 7.4: Position Class Members

Member	Description
`Posn(int x, int y)`	Constructor.
`Posn& operator+=(const Posn& p)`	Increment.
`Posn& operator-=(const Posn& p)`	Decrement.
`Posn& operator/= (double d)`	Divide.
`void set(int x, int y)`	Set value.
`int x(void)`	Return x component.
`int y(void)`	Return y component.
`double mag(void)`	Calculate distance from origin.

Table 7.5: Position Operations

Program	Description
`ostream& operator << (ostream& s, Posn t)`	Output operator.
`istream& operator >> (istream& s, Posn& t)`	Input operator.
`Posn operator + (const Posn& l, const Posn& r)`	Addition.
`Posn operator - (const Posn& l, const Posn& r)`	Subtraction.
`Posn operator / (const Posn& n, double d)`	Division.
`bool operator < (const Posn& l, const Posn &r)`	Less than.
`bool operator == (const Posn& l, const Posn &r)`	Equality.

7.1.3 Behaviour Module

The majority of the Behaviour module is automatically generated from the SRD as it appears in Section A.4. The exceptions to this are the predicates wall and connected,

and the functions tol and path. These were implemented manually since an automatically generated version of these would be highly inefficient. This is particularly true of path, which involves constructing a set of points from the history of ${}^{mc}\mathsf{penPos}$—an operation that can be done quite easily using the C++ STL iterator concept.

7.2 Discussion

The monitor was used to evaluate 10 test executions of the same target system—considered to be the best of those submitted by teams of students enrolled in the course—using eight different mazes. The results are summarized in Table 7.6, in which **Fail** indicates that the monitor reported that the behaviour is unacceptable. This target system had been evaluated by the course instructor and teaching assistants using manual testing methods—visual observation of the behaviour—and was thought to conform with the requirements. As can be seen from the table, this system does not, in fact, behave in an acceptable manner in many cases. Each of these failures is confirmed by examining the log files at the point of failure, and closer investigation suggests a possible source of the problem: in all cases the failure occurred when ${}^{mc}\mathsf{penPos}.x = 0$—a point of discontinuity in the equations for calculating the angles of the arms. These results clearly illustrate a strength of such automated testing methods over manual techniques: they detect subtle and short-lived violations of the requirements, which humans will often miss (in this case all of the violations lasted about 0.5 seconds or less).

Unfortunately, since this testing was done using recorded system behaviour when the target system was no longer available, it was not possible to attempt to correct the behaviour and conduct further testing.

This trial also highlighted the strengths of these techniques as a method for syntax and type-consistency verification and validation of the SRD. Since the monitor code is generated directly from the SRD, syntactic and type-consistency errors are either detected by the monitor generator tool, or propagated to the monitor code where they are detected by the compiler used to compile the monitor. Using the monitor to evaluate behaviour that is known to be acceptable or not helps to gain confidence that the specification is correct, or finds errors when it is not. Initial runs of the test cases for this system helped to isolate several errors that remained in the Maze Tracer

Table 7.6: Maze-Tracer Test Results

Log File	Results
maze1gen	**Fail** at i_t = 63036.280. forward is false due to pen re-tracing over some points.
maze1agen	**Fail** at i_t = 61521.330. forward is false (as above).
maze2gen	**Fail** at i_t = 61349.410. wall is true—pen tip comes too close to a wall.
maze3gen	**Fail** at i_t = 61412.190. forward is false (as above).
maze4gen	**Fail** at i_t = 39064.930. forward is false (as above).
maze5gen	**Fail** at i_t = 39157.040. forward is false (as above).
maze6gen	**Success**. ^{c}merit = 96.52.
maze9gen	**Success**. ^{c}merit = 0 (no path).
maze12gen	**Fail** at i_t = 39379.210. forward is false (as above).
maze3fudge	**Success**. ^{c}merit = 99.87.

SRD despite it being previously carefully reviewed by several well qualified people.

7.2.1 Monitor Execution Time

As discussed in Section 7.1.1, the monitor control module calls `MON_behaviour` after each sample is read to determine if the behaviour up to that time is acceptable. For this specification, the amount of processing done by this function, and hence its execution time, varies significantly depending on the history. For example, if $^{Cl}\mathsf{direction} = {}^{Md}\mathsf{Starting}$ then the processing is trivial, whereas if $^{Cl}\mathsf{direction} = {}^{Md}\mathsf{Forward} \wedge \neg {}^{p}\mathsf{Holding}$ then the "forward" predicate will be evaluated using the "path" function, which has $O(n \log n)$ complexity, where n is the number of sample points since $\mathit{First}(@\mathbf{T}\left({}^{Md}\mathsf{Forward}\right))$.

To determine if the monitor could evaluate the behaviour in the time that it takes to occur, the execution time of the monitor is measured and compared with the elapsed time between samples in the log file. Processing of a sample, s_j, is said to be *on-time* if for all previous samples, s_i, the time elapsed between the monitor processing s_i and s_j is less than the timestamp difference, $s_j.\mathtt{i\_t} - s_i.\mathtt{i\_t}$ for the samples. The initial sample is on-time. If processing for a sample is not on-time then it is *over-time*. The percentage of the samples that are over-time for the test cases are given in Table 7.7. Also the total execution times for the monitor and target system are given. All times are in seconds, and for the monitor running on a

266 MHz Pentium-based PC running Linux 2.0.34. The optimized results are for the monitor compiled using the optimization option of the compiler (GNU gcc version egcs-2.91.66, egcs release 1.1.2). In this case all processing was on-time.

Table 7.7: Maze-Tracer Monitor Timing

Log File	Samples	% over-time (non-optimized)	Robot Time	Monitor Time	
				Non-optimized	Optimized
maze1gen	210	5.7	29.05	3.23	0.60
maze1agen	588	30.4	59.71	22.55	3.92
maze2gen	157	0	22.30	0.33	0.11
maze3gen	108	0	15.49	0.29	0.09
maze4gen	147	0	20.54	0.46	0.13
maze5gen	196	0	25.05	1.36	0.29
maze6gen	300	22.3	42.24	11.63	2.02
maze9gen	2	0	0.38	0.004	0.002
maze12gen	153	0	20.71	0.49	0.14
maze3fudge	299	4.3	41.03	3.50	0.68

No generally applicable conclusions can be drawn from these execution time results, only that this monitor can operate on-time for these test cases, under these conditions. However, since this monitor requires relatively complex computation, it does suggest that there is a large class of systems for which these techniques will produce monitors that can operate in real-time. Applications where real-time processing is critical will require careful analysis of the worst-case processing times for each sample, which cannot be guaranteed in this case since the monitor is executed on a pre-emptive operating system.

7.2.2 Alternative System Interface Design

As mentioned in Section 4.3, the choice of system interface design can have a significant impact on the processing that the monitor software must do. An alternative design for this system, which would significantly improve performance, would use sensors to directly evaluate the following predicates.

- ${}^{mc}\mathsf{penPos} \in \mathrm{tol}(\{{}^{m}\mathsf{mazeStart}\})$

- ${}^{mc}\mathsf{penPos} \in \mathrm{tol}(\{{}^{m}\mathsf{mazeEnd}\})$
- ${}^{mc}\mathsf{penPos} \in \mathrm{tol}(\{({}^{C}\mathsf{HOME\_X}, {}^{C}\mathsf{HOME\_Y})\})$
- $\mathrm{wall}({}^{mc}\mathsf{penPos})$

Chapter 8

Conclusions

This work has demonstrated that practical monitors for real-time systems can be automatically derived from system requirements documents written in a notation that is both expressive and reletively readable. Such monitors are useful both as oracles for system testing, where they are clearly superior to manual methods, or as redundant supervisors for safety critical systems.

8.1 Contributions

The main contributions of this work are as follows.

- It gives a precise definition of a monitor for a real-time system, and identifies some necessary conditions for a monitor to be feasible and useful.
- It describes a notation for writing system requirements documentation method based on [102] and gives an interpretation of this notation on behaviours as functions of continuous time.
- It introduces the use of *transition modes* as a technique for describing permitted deviations from ideal behaviour.
- It demonstrates a technique for generating a monitor from system requirements documentation and illustrates how such a monitor can be used to evaluate the behaviour of a realistic system.

8.2 Applicability of This Work

The specification techniques presented in Chapter 5 are best suited to so called "reactive systems"—those that continuously observe environmental quantities and respond to changes in their value. Such systems are common in process control and industrial automation applications, where they are often safety critical. These techniques are less well suited to "information processing systems", which transform or manipulate data.

Monitors, such as described in this work, are well suited to automated testing of systems, where they function as an oracle, reporting if the behaviour is acceptable or not. This application offers significant improvement over non-automated testing since test cases can be evaluated quickly and errors in behaviour are quickly and reliably detected. It also offers significant improvements over methods where the oracle is implemented manually since testers can be assured that the oracle and the SRD agree, and any changes in the requirements can be quickly translated into a new oracle.

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

Monitor generation is also a good way to perform some simple checks on the SRD, since many common types of error (e.g., type consistency, syntax errors) will be detected either by the monitor generator or the compiler used to compile the monitor implementation. In addition, a monitor can be used to validate the SRD by executing it using behaviours that are known to be acceptable or not, thus increasing confidence that the SRD specifies the behaviour correctly.

Since the monitor is generated directly from the documentation, it gives more incentive to get the documentation right, and keep it up to date, so that it can be used for future system testing. Knowing that the system requirements documentation is accurate makes that documentation much more valuable to system designers and maintainers since they can rely on it to accurately describe the required behaviour.

8.3 Future Work

Application of these techniques to the specification and testing or supervision of different systems will further illustrate their usefulness and may allow us to draw more conclusions about what classes of systems can be monitored in real-time. Also, further application will undoubtedly suggest further enhancements to the specification and monitor generation techniques.

Further investigation of the monitor input relation, what classes are typical for realistic devices, and the relationship to monitor feasibility would be useful. Ideally such investigation would lead to a useful expression of necessary and sufficient conditions for monitor feasibility.

8.3.1 Monitor Generator and Library Enhancements

Since the Monitor Generator is a prototype tool, certain features that would be desirable in a more generally applicable tool were intentionally left out in the interest of expediency. The tool could certainly be enhanced by implementing these features, some of which are discussed below.

The most significant weakness of the monitor generator is that it does not take the monitor input relation into account, but requires that this is done by the system interface module or explicitly encoded in the SRD (e.g., by specifying tolerance functions that allow fewer behaviours). It may be possible to enhance the tool to automatically account for the effect of the input devices for a restricted set of device behaviours. For example, if **REQ** allows controlled variable ${}^{c}\mathsf{c}$ to be $x \pm {}^{C}\mathsf{T}$, and $\mathbf{IN_{mon}}$ specifies that $s.c = {}^{c}\mathsf{c} \pm {}^{C}\mathsf{E}$ then, assuming that ${}^{C}\mathsf{T} > {}^{C}\mathsf{E}$, the monitor should check that $s.c = x \pm ({}^{C}\mathsf{T} - {}^{C}\mathsf{E})$.

Another weakness is that the order of expressions in the TTS context file is significant—it is used to determine the order of updates to conditions and mode classes. While determining the required order is usually not difficult, it should also be reasonably easy to automate using dependency analysis techniques such as presented in [39, 41]. Removing the restriction on the order of expressions in the SRD, and thus allowing the author to structure the specification logically will enhance readability.

Some other technical enhancements that would help to make the monitor generator more 'user-friendly' are as follows.

- The "behaviour expression", which is required in the MG input, is usually simply the conjunction of the characteristic predicates of the controlled value relations, in which case it could be generated automatically.

- It may be possible to automatically determine which "condition expressions" are needed by searching the SRD for event class expressions. This would remove the requirement that conditions be named separately in the context file.

- Similarly, it may be possible to automatically extract the "history expressions" from the SRD, although identification of these may not be as easy as for condition expressions.

- The monitor generator libraries support only a very limited set of functions of history expressions. This set could be extended, for example to include numerical integration.

- For both condition and history expressions, the monitor software retains all of the previous values that it has seen (i.e., all of the history). This may impose unnecessary demands on the monitor software, and will certainly eventually lead to problems due to lack of computer memory if a monitor is used to evaluate a very long behaviour. Usually it is not necessary to keep all of this history, so this problem could be overcome in many situations by introducing a method to specify the amount of history needed. In some cases it may also be possible to determine this automatically by analysis of the SRD.

- The use of global variables, as is done in the monitor implementation, is usually seen as a poor programming technique and has particular problems with respect to C++ objects (see, e.g., [65]). This could probably be overcome by either using the notion of "namespace" to encapsulate these variables and objects, or by implementing the behaviour module as one or more object classes, of which the monitor control module would be required to instantiate appropriate instances. This latter technique has the advantage that it would remove the need for the

`initOracle` access program since its behaviour could be encapsulated in the object constructor function.

This work has also highlighted some possible enhancements to the code generation modules of the TTS, as follows.

- In the implementation of tabular expressions the guard expression is, in some cases, trivially always false for a particular table index (e.g., in mode transition relations for source and destination modes for which no such transition occurs). These expressions, and the corresponding value expressions, need not be implemented since they will have no effect on the value of the tabular expression. The new Table Evaluation module, described in Section 6.4.1, does not require that there be a guard expression for all valid table indices, and only requires value expressions for those table indices for which the guard expression may be true. Reducing the number of guard expressions implemented will reduce the search space for table evaluation, and hence will make the evaluation more efficient.

- The "Inductively Defined Predicate" implementation could be significantly improved by using the C++ template mechanisms, which have recently become available, to support arbitrary types. In addition the interface to an IDP could be made to conform with the STL concept of "InputIterator" and the implementation of quantified expressions could be modified to use the STL `find_if` algorithm, which is likely to be highly optimized. These changes would have the following advantages.

  1. Quantification would be supported over all types conforming with the "InputIterator" concept.
  2. IDPs could be used in other STL algorithms.

8.3.2 Verification

One goal when using mathematical documentation techniques, such as those presented in Chapter 5, is that the documentation can be formally verified by proving that it has, or does not have, certain properties. To do so requires that the specification language be strictly constrained and that the underlying logic model be made explicit, which

has not been done in this thesis. The ability to do such analysis, particularly if supported by tools, can greatly enhance a documentation technique and will increase confidence that specifications are correct.

One potential approach to this is to define a translation from the specification notation into the notation used by an existing proof system that is based on an appropriate formal model (e.g., Duration Calculus[85]).

Appendix A

Example Requirements Documents

This appendix illustrates the requirements documentation method used in this thesis by presenting complete requirements documents for some realistic sample problems. Two of these examples are popular sample problems found in the literature, while the third was developed by the author and his colleagues.

A.1 Notational Conventions

The same notational conventions are used in all of the examples, as follows.

Table A.1: Notational Conventions

Item	**Style**
Monitored variable	^{m}mvar
Controlled variable	^{c}cvar
Monitored and controlled variable	^{mc}mcvar
Mode Class	^{Cl}mclass
Mode	^{Md}mode
State	^{s}state
Constant	^{C}const
Set	**set**
Sequence	**seq**
Data type	**datatype**
Function (defined in this document)	func
Standard function	*kfunc*

A.2 Generalized Railroad Crossing

The Generalized Railroad Crossing (GRC) problem was first proposed as a benchmark problem by Heitmeyer *et al.* in [40] and has since been used to illustrate several different methods for specification and verification of real-time systems (e.g. [11, 27, 42, 43]).

A.2.1 Preliminaries

In the following informal description, which is quoted from [43], **I** and **R** are sets of points on the ground.

> The system to be developed operates a gate at a railroad crossing. The railroad crossing **I** lies in a region of interest **R**, i.e., $\mathbf{I} \subseteq \mathbf{R}$. A set of trains travel through **R** on multiple tracks in both directions. A sensor system determines when each train enters and exits region **R**. To describe the system formally, we define a gate function $\mathrm{g}(t) \in [0, 90]$, where $\mathrm{g}(t) = 0$ means the gate is down and $\mathrm{g}(t) = 90$ means the gate is up. We define a set $\{\lambda_i\}$ of occupancy intervals, where each occupancy interval is a time interval during which one or more trains are in **I**. The i^{th} occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the i^{th} entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i).
>
> Given two constants ξ_1 and ξ_2, $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:
>
> **Safety Property** : $t \in \bigcup_i \lambda_i \Rightarrow \mathrm{g}(t) = 0$ (The gate is down during all occupancy intervals.)
>
> **Utility Property** : $t \notin \bigcup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow \mathrm{g}(t) = 90$ (The gate is up when no train is in the crossing.)

A.2.2 Environmental Quantities

Table A.2: Railroad Crossing Environmental Quantities

Variable	Description	Monitored	Controlled	Value Set	Notes
^{m}t	Current time	•		**Real**	1
^{m}trainDist	Distance from train to crossing.	•		[0, ^{C}R_SIZE]	2
^{c}gate	Gate arm position.		•	[0, 90]	3

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. The shortest distance from any point on any train to any point in the crossing. The required resolution is 0.1 m.

3. Angle, in degrees, between gate arm and the horizontal. The required resolution is 2 °.

A.2.3 Mode Classes

^{Cl}train

Modes : ^{Md}InX, ^{Md}Empty, ^{Md}Approach, ^{Md}Leaving, ^{Md}Stopped

Mode function :

$p_T : H_1 \wedge H_2$ $r_T : G$ Normal	$\frac{d}{dt}{}^{m}$trainDist		
	< 0	$= 0$	> 0
^{m}trainDist $= 0$	^{Md}InX	^{Md}InX	^{Md}InX
$0 <$ ^{m}trainDist $<$ ^{C}R_SIZE	^{Md}Approach	^{Md}Stopped	^{Md}Leaving
^{m}trainDist $\geq$ ^{C}R_SIZE	^{Md}Empty	^{Md}Empty	^{Md}Empty

A.2.4 Controlled Value Functions

^{c}gate

^{Md}InX	$\underline{false}$	$\underline{true}$	$\underline{false}$
^{Md}Approach	$\left\lvert \frac{{}^{m}\text{trainDist}}{\frac{d}{dt}({}^{m}\text{trainDist})} \right\rvert \leq \xi_1$	$\underline{false}$	$\left\lvert \frac{{}^{m}\text{trainDist}}{\frac{d}{dt}({}^{m}\text{trainDist})} \right\rvert > \xi_1$
^{Md}Leaving	$Drtn(\neg {}^{Md}\text{InX}) \leq \xi_2$	$\underline{false}$	$Drtn(\neg {}^{Md}\text{InX}) > \xi_2$
^{Md}Empty	$\underline{false}$	$\underline{false}$	$\underline{true}$
^{Md}Stopped	$\underline{true}$	$\underline{false}$	$\underline{false}$
$p_T : H_1 \wedge G$ $r_T : H_2$ Inverted	$\underline{true}$	^{c}gate $= 0$	^{c}gate $= 90$

A.2.5 Environmental Constraints

The following two equations are implicit in the given description, but should be made explicit.

$$\left| \frac{d}{dt}({}^{m}\text{trainDist}) \right| \leq {}^{C}\text{TRAIN\_MAX} \tag{A.1}$$

(Trains move at less than some speed.)

$$\left|\frac{d}{dt}({}^{c}\mathsf{gate})\right| \leq {}^{C}\mathsf{GATE\_MAX} \tag{A.2}$$

(The gate has a fixed maximum speed.)

In addition, we make the following assumptions with respect to the given constants.

$$\frac{{}^{C}\mathsf{R\_SIZE}}{{}^{C}\mathsf{TRAIN\_MAX}} \geq \frac{90}{{}^{C}\mathsf{GATE\_MAX}} \tag{A.3}$$

$$\xi_1 \geq \frac{90}{{}^{C}\mathsf{GATE\_MAX}} \tag{A.4}$$

$$\xi_2 \geq \frac{90}{{}^{C}\mathsf{GATE\_MAX}} \tag{A.5}$$

(The gate can be raised or lowered in the given time limits.)

A.2.6 Dictionary

Constants

Name	Constraints	Interpretation
ξ_1	$\xi_1 > 0$	Amount of time prior to a train entering the crossing that the gate is permitted to not be up.
ξ_2	$\xi_2 > 0$	Amount of time after all trains have left the crossing that the gate is permitted to not be up.
${}^{C}\mathsf{GATE\_MAX}$	${}^{C}\mathsf{GATE\_MAX} > 0$	Maximum (angular) speed of gate.
${}^{C}\mathsf{R\_SIZE}$	${}^{C}\mathsf{R\_SIZE} > 0$	Distance from crossing to edge of "region of interest".
${}^{C}\mathsf{TRAIN\_MAX}$	${}^{C}\mathsf{TRAIN\_MAX} > 0$	Maximum speed of trains.

A.3 Steam-Boiler Control

The Steam-Boiler Control (SBC) problem was developed by LtCol. J. C. Bauer for the Institute for Risk Research of the University of Waterloo, and has been used as a case study for a number of papers and workshops (e.g., [4]). The description presented in [3] gives the software requirements document for the boiler-controller, which includes the input and output relations, and so is more implementation specific than intended for this work. The description given here is the system requirements document for the boiler-controller, which hides the behaviour of the messaging system and inter-component communications specified in [3]. Also, since the "program modes" described in that document are not externally observable (i.e., the system is not mode apparent with respect to those modes), they are not used here.

A.3.1 Preliminaries

The system consists of a boiler equipped with four pumps for adding water, a valve for removing water and sensors to measure the quantity of water in the tank and the rate at which steam is coming out of the tank. In addition there is an operator console where outputs may be displayed and commands issued by a human operator. Each of the devices (pumps, sensors) is capable of detecting and reporting its own operational status (i.e., ${}^{s}\mathsf{Ok}$ or ${}^{s}\mathsf{Fail}$). A failed pump may be pumping or not. A failed sensor gives no useful information. The problem is to control the amount of water in a steam-boiler tank such that it operates safely, and to shut down the boiler if it cannot continue to do so.

During operation the amount of water must always be between a lower and upper limit, ${}^{C}\mathsf{LowCrit}$ and ${}^{C}\mathsf{HighCrit}$, respectively, which are chosen such that the boiler could reach an unsafe condition (assuming maximum flow rates imposed by the system) in five seconds from the time the water volume reaches either ${}^{C}\mathsf{LowCrit}$ or ${}^{C}\mathsf{HighCrit}$. In the event that system failure prevents the water volume from continuously being maintained between these limits, the controller is required to shut down the boiler. Under "normal" circumstances the water volume in the tank should be maintained between the volumes, ${}^{C}\mathsf{LowNorm}$ and ${}^{C}\mathsf{HighNorm}$.

On startup the tank is either drained or filled, as required, to a normal operational volume before the boiler is started.

A.3.2 Environmental Quantities

Table A.3: Steam-boiler Environmental Quantities

Variable	Description	Monitored	Controlled	Value Set	Notes
^{m}t	Current time	•		**Real**	1
^{m}water	Current volume of water in the tank.	•		**Real**, $[0, {}^{C}\mathsf{Cap}]$	2
^{m}steam	Rate at which water is evaporating from the tank.	•		**Real**, $[0, {}^{C}\mathsf{MaxSteam}]$	3
^{m}flow	Rate at which water is being added to the tank.	•		**Real**, $[0, 4{}^{C}\mathsf{PumpRate}]$	3
^{m}request	The operator requested boiler state.	•		{ ^{s}Off, ^{s}On }	
^{m}levelStat	Water level sensor status.	•		{ ^{s}Ok, ^{s}Fail }	
^{m}steamStat	Steam sensor status.	•		{ ^{s}Ok, ^{s}Fail }	
^{m}pumpStat$[i]$	Pump status. $i \in [1 \ldots 4]$	•		{ ^{s}Ok, ^{s}Fail }	
^{m}commStat	Communications network status.	•		{ ^{s}Ok, ^{s}Fail }	
^{c}boiler	The boiler status.		•	{ ^{s}Off, ^{s}On }	
^{c}pump$[i]$	The control state of pump i. $i \in [1 \ldots 4]$		•	{ ^{s}Open, ^{s}Closed }	
^{c}valve	The valve status.		•	{ ^{s}Open, ^{s}Closed }	

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. The volume is represented as litres of water in the tank. The required resolution is 0.1 litres.

3. Flow and evaporation rates are represented in litres of liquid water per second. The required resolution is 0.1 litres/second.

A.3.3 Mode Classes

^{Cl}controller

Modes : ^{Md}Init_Drain, ^{Md}Init_Fill, ^{Md}Run, ^{Md}Off

Initial Mode function : ^{Md}Off

Transition relation :

$p_T : H_1 \wedge G$ $r_T : H_3$ Decision	^{m}request = ^{s}On	estMaxWater $\geq$ ^{C}HighCrit	estMaxWater $\geq$ ^{C}HighNorm	estMinWater $\leq$ ^{C}LowNorm	estMinWater $\leq$ ^{C}LowCrit	^{m}levelStat = ^{s}Ok	^{m}steamStat = ^{s}Ok	^{m}commStat = ^{s}Ok	
^{Md}Off	@T	*	t	f	f	t	t	t	^{Md}Init_Drain
	@T	f	f	t	*	t	t	t	^{Md}Init_Fill
	@T	f	f	f	f	t	t	t	^{Md}Run
^{Md}Init_Drain$\vee$	—	f	f	@F	f	t	t	t	^{Md}Run
^{Md}Init_Fill	—	f	@F	f	f	t	t	t	
	@F	*	*	*	*	*	*	*	^{Md}Off
	—	*	*	*	*	@F	*	*	
	—	*	*	*	*	*	@F	*	
	—	*	*	*	*	*	*	@F	
^{Md}Run	@F	*	*	*	*	*	*	*	^{Md}Off
	—	f	f	t	@T	*	*	*	
	—	@T	t	f	f	*	*	*	
	—	F	*	*	F	@F	f	T	
	—	F	*	*	F	f	@F	T	
	—	F	*	*	F	@F	@F	T	
	—	F	*	*	F	*	*	@F	

Maximum Delay : < 5.0 s

A.3.4 Controlled Value Functions

^{c}boiler

$p_T : H_2$ $r_T : H_1 = G$ Vector	^{Md}Run	$\neg^{Md}$Run
^{c}boiler	^{s}On	^{s}Off

^{c}valve

$p_T : H_2$ $r_T : H_1 = G$ Vector	^{Md}Init_Drain	$\neg^{Md}$Init_Drain
^{c}valve	^{s}Open	^{s}Closed

^{c}pump

$p_T : H_2 \wedge G$ $r_T : H_1$ Inverted	^{Md}Init_Fill	^{Md}Run	^{Md}Init_Drain $\vee$ ^{Md}Off
$\forall i, (1 \leq i \leq 4) \Rightarrow$ ^{c}pump$[i] = {}^{s}$Closed	*false*	estMaxWater $\geq {}^{C}$HighNorm	*true*
$(\exists i, {}^{m}$pumpStat$[i] = {}^{s}$Ok$) \Rightarrow$ $(\exists i,$ pumping$(i))$	*true*	estMinWater $\leq {}^{C}$LowNorm	*false*
true	*false*	estMinWater $>$ ^{C}LowNorm $\wedge$ estMaxWater $<$ ^{C}HighNorm	*false*

Merit

^{c}merit

$\stackrel{\text{df}}{=} 1/card(\cup_{i=1}^{4} @\mathbf{T}\,({}^{c}\text{pump}[i] = {}^{s}\text{Open}))$

A.3.5 Environmental Constraints

All environmental quantities are restricted to the range of values given in the appropriate row of Table A.3. In addition, the following assumptions can be made.

Pump Behaviour

The pumps have the property that "after having been switched on the pump needs five seconds to start pouring water into the boiler." This constrains the value of ${}^{m}\mathsf{flow}$, as follows.

$$card(\{i \mid Drtn({}^{c}\mathsf{pump}[i] = {}^{s}\mathsf{Open}) > 5.0 \wedge {}^{m}\mathsf{pumpStat}[i] = {}^{s}\mathsf{Ok}\}) \leq \frac{{}^{m}\mathsf{flow}}{{}^{C}\mathsf{PumpRate}} < card(\{i \mid {}^{c}\mathsf{pump}[i] = {}^{s}\mathsf{Open} \vee {}^{m}\mathsf{pumpStat}[i] = {}^{s}\mathsf{Fail}\})$$

Steam

The rate of change of ${}^{m}\mathsf{steam}$ is bounded above and below by the quantities ${}^{C}\mathsf{MaxGrad}$ and ${}^{C}\mathsf{MinGrad}$, respectively.

$${}^{C}\mathsf{MinGrad} \leq \frac{d}{dt}({}^{m}\mathsf{steam}) \leq {}^{C}\mathsf{MaxGrad} \tag{A.6}$$

A.3.6 Dictionary

Functions

estMaxFlow :→ **Real**

estMaxFlow

$$\stackrel{\text{df}}{=} {}^{C}\mathsf{PumpRate} \times card\left(\left\{i \,\middle|\, {}^{c}\mathsf{pump}[i] = {}^{s}\mathsf{Open} \ \vee {}^{m}\mathsf{pumpStat}[i] = {}^{s}\mathsf{Fail}\right\}\right)$$

estMinFlow : **Real** → **Real**

estMinFlow(δ)

$$\stackrel{\text{df}}{=} {}^{C}\mathsf{PumpRate} \times card\left(\left\{i \,\middle|\, \begin{array}{l} Drtn({}^{c}\mathsf{pump}[i] = {}^{s}\mathsf{Open}) + \delta > 5.0 \wedge \\ {}^{m}\mathsf{pumpStat}[i] = {}^{s}\mathsf{Ok} \end{array}\right\}\right)$$

estMaxSteam : **Real** → **Real**

estMaxSteam(δ)

$$\stackrel{\text{df}}{=} {}^{m}\mathsf{steam} + {}^{C}\mathsf{MaxGrad} \times \delta$$

Table A.4: Summary of Functions

Name	Interpretation
estMaxFlow	an estimate of the maximum value of ${}^{m}\mathsf{flow}$ at a time in the next 5 seconds.
estMinFlow	an estimate of the minimum value of ${}^{m}\mathsf{flow}$ at a time in the next 5 seconds.
estMaxSteam	an estimate of the maximum value of ${}^{m}\mathsf{steam}$ at a time in the next 5 seconds.
estMinSteam	an estimate of the minimum value of ${}^{m}\mathsf{steam}$ at a time in the next 5 seconds.
estMaxWater	an estimate of the maximum value of ${}^{m}\mathsf{water}$ at 5 seconds from the current time.
estMinWater	an estimate of the minimum value of ${}^{m}\mathsf{water}$ at 5 seconds from the current time.
pumping	*true* if pump is pumping.

$\text{estMinSteam} : \mathbf{Real} \to \mathbf{Real}$

$\text{estMinSteam}(\delta)$

$\quad \stackrel{\text{df}}{=} {}^{m}\mathsf{steam} + {}^{C}\mathsf{MinGrad} \times \delta$

$\text{estMaxWater} :\to \mathbf{Real}$

estMaxWater

$\quad \stackrel{\text{df}}{=} {}^{m}\mathsf{water} + \int_0^5 (\text{estMaxFlow} - \text{estMinSteam}(\delta))\, d\delta$

$\text{estMinWater} :\to \mathbf{Real}$

estMinWater

$\quad \stackrel{\text{df}}{=} {}^{m}\mathsf{water} + \int_0^5 (\text{estMinFlow}(\delta) - \text{estMaxSteam}(\delta))\, d\delta$

$\text{pumping} : \mathbf{Integer} \to \mathbf{Boolean}$

$\text{pumping}(i)$

$\quad \stackrel{\text{df}}{=} \mathit{Drtn}({}^{c}\mathsf{pump}[i] = {}^{s}\mathsf{Open}) \geq 5.0\ \text{s} \wedge {}^{m}\mathsf{pumpStat}[i] = {}^{s}\mathsf{Ok}$

Constants

Name	Constraints	Interpretation
${}^{C}\mathsf{Cap}$	$0 < {}^{C}\mathsf{Cap}$	Capacity of boiler tank, litres.
${}^{C}\mathsf{HighCrit}$	${}^{C}\mathsf{HighNorm} < {}^{C}\mathsf{HighCrit} < {}^{C}\mathsf{Cap}$	Critical maximum volume of water in boiler, litres.
${}^{C}\mathsf{HighNorm}$	${}^{C}\mathsf{LowNorm} < {}^{C}\mathsf{HighNorm} < {}^{C}\mathsf{HighCrit}$	Maximum 'normal' volume of water in boiler, litres.
${}^{C}\mathsf{LowCrit}$	${}^{C}\mathsf{LowCrit} < {}^{C}\mathsf{LowNorm}$	Critical minimum volume of water in boiler, litres.
${}^{C}\mathsf{LowNorm}$	${}^{C}\mathsf{LowCrit} < {}^{C}\mathsf{LowNorm} < {}^{C}\mathsf{HighNorm}$	Minimum 'normal' volume of water in boiler, litres.
${}^{C}\mathsf{MaxGrad}$	$0 < {}^{C}\mathsf{MaxGrad}$	Maximum rate of increase in ${}^{m}\mathsf{steam}$. See Eq. (A.6)
${}^{C}\mathsf{MaxSteam}$	$0 < {}^{C}\mathsf{MaxSteam}$	Maximum rate of steam evacuation, litres/second.
${}^{C}\mathsf{MinGrad}$	$0 > {}^{C}\mathsf{MinGrad}$	Maximum rate of decrease in ${}^{m}\mathsf{steam}$. See Eq. (A.6)
${}^{C}\mathsf{PumpRate}$	$0 < {}^{C}\mathsf{PumpRate}$	Nominal flow rate for pumps, litres/second.

A.4 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).

A.4.1 Preliminaries

The environment relevant to the target system consists of

- a *draw-bot*, which is a robot 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 labelled "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 and the maximum speed of the draw-bot is fixed so a shortest path through the maze should be chosen.

Pen Position

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}penPos.x, ^{mc}penPos.y) of reals, representing the location in the horizontal plane where the pen tip is touching the maze (if ^{mc}penDown is *true*) or would touch the maze if lowered (if ^{mc}penDown is *false*). The location is specified by the distance, in millimetres, 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 ^{C}MIN_X, ^{C}MAX_X, ^{C}MIN_Y and ^{C}MAX_Y. The origin is the centre of the robot base post. The home location of the pen-tip (to which it is returned on initialization of the draw-bot) is (^{C}HOME_X, ^{C}HOME_Y).

Maze

As illustrated in Figure A.1, the maze is contained within a ^{C}M_WIDTH mm × ^{C}M_HEIGHT mm region of the horizontal plane bounded by the lines $x = -{}^{C}$M_X_OFFSET, $y = {}^{C}$M_Y_OFFSET, $x = -{}^{C}$M_X_OFFSET + ^{C}M_WIDTH and $y = {}^{C}$M_Y_OFFSET + ^{C}M_HEIGHT, which are the external walls of the maze. The internal walls of the maze are segments of the lines $x = -{}^{C}$M_X_OFFSET + $n \times {}^{C}$M_CELL_SIZE mm and $y = {}^{C}$M_Y_OFFSET + $n \times {}^{C}$M_CELL_SIZE mm, where n is an integer (i.e., a square grid with line spacing ^{C}M_CELL_SIZE mm). The endpoints of the walls lie at intersections of these grid lines. Figure A.2 is a sample maze (actual size for ^{C}M_WIDTH = ^{C}M_HEIGHT = 150 and ^{C}M_CELL_SIZE = 10), with dashed lines indicating the possible wall locations.

Informal Description

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

When the pen is down (^{mc}penDown = *true*) the pen tip must never come within ^{C}WALL_SPACE mm of a wall point (wall(^{mc}penPos) = *true*).

Messages Whenever a significant event occurs (i.e., a button is pressed or released, the pen reaches a significant point in its journey, or an error is detected) the system must display a diagnostic message describing the event and the system's response to it.

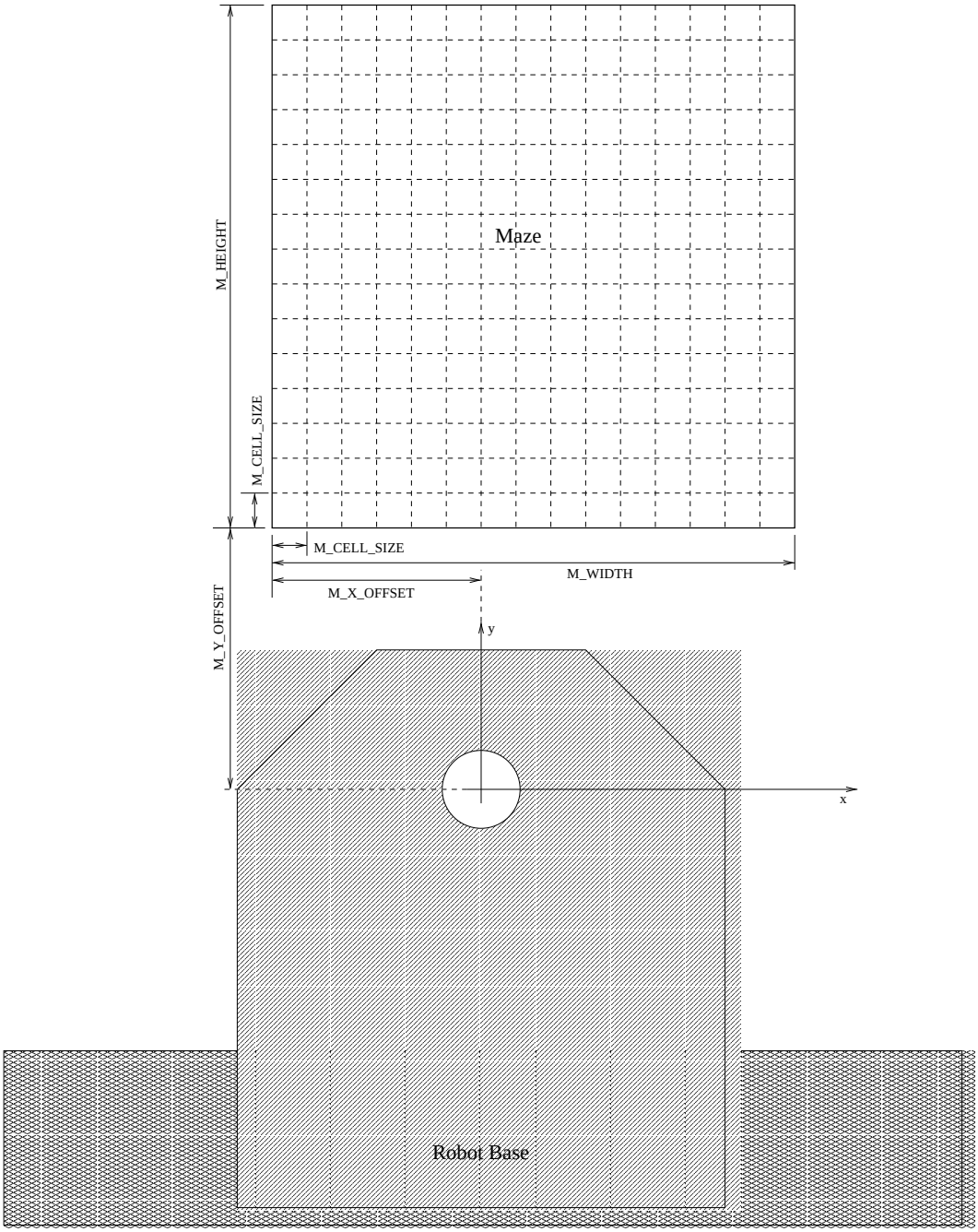

Figure A.1: Robot and Maze Parameters

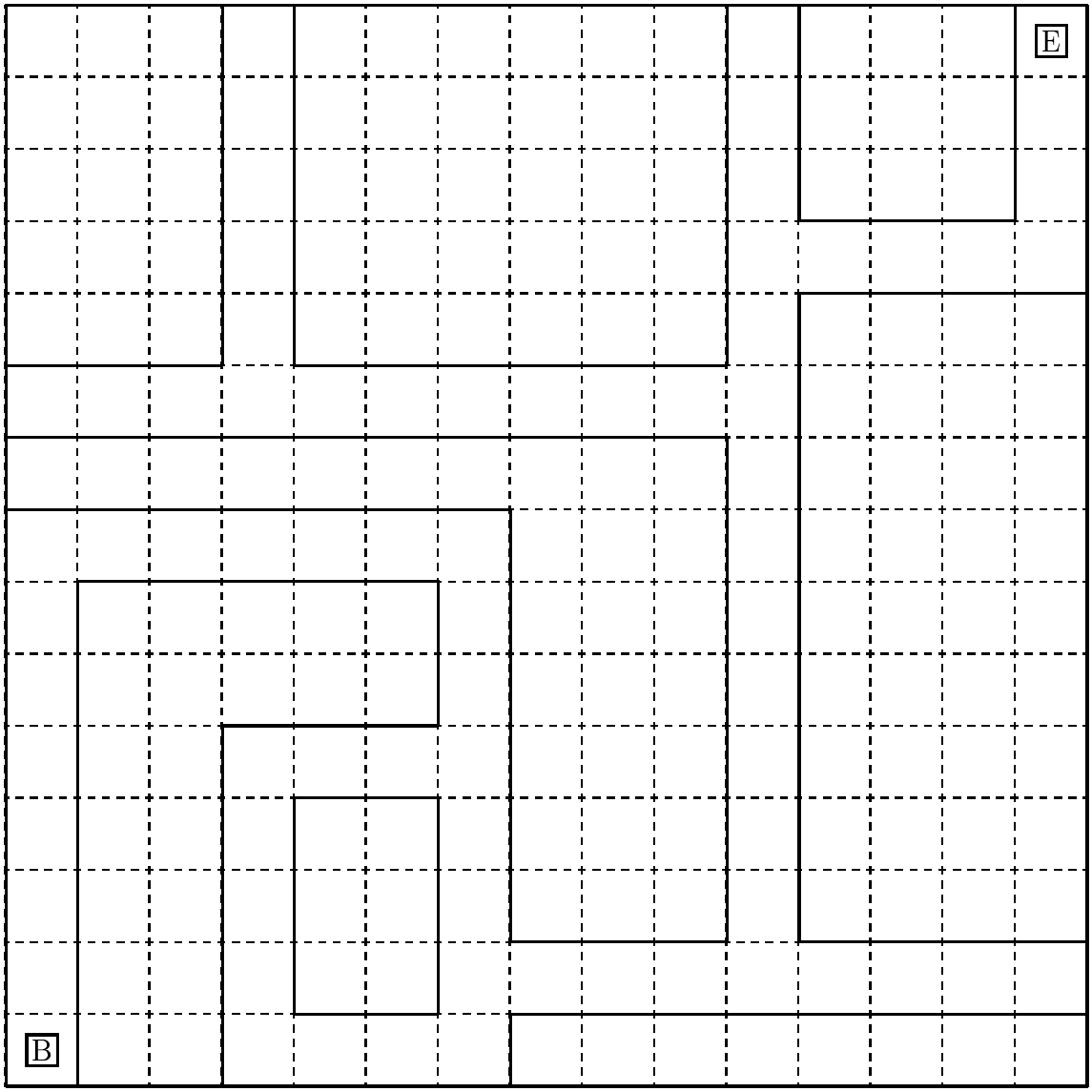

Figure A.2: Sample Maze

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.

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 (${}^{c}\mathsf{powerOn} = \underline{\mathit{false}}$).

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

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

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 ${}^{C}\mathsf{RESPONSE\_TIME}$ seconds and begin to re-trace its path back to the beginning (${}^{mc}\mathsf{penPos} \in \mathrm{tol}(\{{}^{m}\mathsf{mazeStart}\})$). 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.

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 ${}^{C}\mathsf{RESPONSE\_TIME}$ seconds, raise the pen (${}^{mc}\mathsf{penDown} = \underline{\mathit{false}}$) and return to the home position, without making any further marks.

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

A.4.2 Environmental Quantities

Table A.5: Maze-tracer Robot Environmental Quantities

Variable	Description	Monitored	Controlled	Value Set	Notes
$^{m}\mathsf{t}$	Current time	•		**Real**	1
$^{m}\mathsf{mazeWalls}$	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
$^{m}\mathsf{mazeStart}$	Start position for the maze.	•		**Posn**	2
$^{m}\mathsf{mazeEnd}$	Finish position for the maze.	•		**Posn**	2
$^{m}\mathsf{stopButton}$	Status of the "stop" button.	•		**buttonT**	
$^{m}\mathsf{homeButton}$	Status of the "home" button.	•		**buttonT**	
$^{m}\mathsf{backButton}$	Status of the "back" button.	•		**buttonT**	
$^{mc}\mathsf{penPos}$	The position of the pen relative to the origin $(0, 0)$, which is the centre of the robot base post.	•	•	**Posn**	2
$^{mc}\mathsf{penDown}$	$\underline{\mathit{true}}$ iff the pen is touching the plane containing the maze. Assumed to be initially $\underline{\mathit{false}}$.	•	•	**Boolean**	
$^{c}\mathsf{powerOn}$	$\underline{\mathit{true}}$ iff the robot power is on. Assumed to be initially $\underline{\mathit{false}}$.		•	**Boolean**	
$^{c}\mathsf{message}$	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 A.4.1. The required resolution is 0.5 millimetres.

A.4.3 Mode Classes

^{Cl}direction

Modes : ^{Md}Starting, ^{Md}Forward, ^{Md}Reverse, ^{Md}Home, ^{Md}Done

Initial mode :

$p_T : H_1$ $r_T : H_2 = G$ Vector	^{Cl}direction
connected(^{m}mazeStart, ^{m}mazeEnd)	^{Md}Starting
¬connected(^{m}mazeStart, ^{m}mazeEnd)	^{Md}Done

Transition relation :

$p_T : H_1 \wedge G$ $r_T : H_3$ Decision	^{mc}penPos ∈ tol({^{m}mazeStart})	^{mc}penPos ∈ tol({^{m}mazeEnd})	^{mc}penPos ∈ tol({({^{C}HOME_X, ^{C}HOME_Y)}})	^{mc}penDown = $\underline{true}$	^{m}homeButton = ^{s}Down	^{m}backButton = ^{s}Down	
^{Md}Starting	t	*	*	@T	*	*	^{Md}Forward
^{Md}Forward	*	@T	*	*	*	*	^{Md}Home
	*	F	*	*	@T	*	^{Md}Home
	*	F	*	*	F	@T	^{Md}Reverse
^{Md}Reverse	*	*	*	*	@T	*	^{Md}Home
	*	*	*	*	F	@F	^{Md}Forward
^{Md}Home	*	*	@T	*	*	*	^{Md}Done

Maximum Delay : ${}^{C}\mathsf{RESPONSE\_TIME}$ s

A.4.4 Controlled Value Functions

${}^{mc}\mathsf{penPos}$

$$\left| \left(\text{holding} \wedge \tfrac{d}{dt}({}^{mc}\mathsf{penPos}) = 0\right) \vee \right.$$

$$\left(\begin{array}{|l||l|} \hline \begin{array}{l} \mathrm{p_T} : H_1 \\ \mathrm{r_T} : G \\ \text{Normal} \end{array} & \\ \hline\hline {}^{Md}\mathsf{Forward} & \neg\text{wall}({}^{mc}\mathsf{penPos}) \wedge \text{inMaze}({}^{mc}\mathsf{penPos}) \wedge \text{forward}({}^{mc}\mathsf{penPos}) \\ \hline {}^{Md}\mathsf{Reverse} & \neg\text{wall}({}^{mc}\mathsf{penPos}) \wedge \text{inMaze}({}^{mc}\mathsf{penPos}) \wedge \text{reverse}({}^{mc}\mathsf{penPos}) \\ \hline {}^{Md}\mathsf{Starting} \vee {}^{Md}\mathsf{Home} & \underline{\mathit{true}} \\ \hline {}^{Md}\mathsf{Done} & {}^{mc}\mathsf{penPos} \in \text{tol}\left(\left\{ \begin{pmatrix} {}^{C}\mathsf{HOME\_X}, \\ {}^{C}\mathsf{HOME\_Y} \end{pmatrix} \right\}\right) \\ \hline \end{array} \right) \neg\text{holding} \wedge$$

${}^{mc}\mathsf{penDown}$

$$= \begin{array}{|l||l|l|} \hline \begin{array}{l} \mathrm{p_T} : H_2 \\ \mathrm{r_T} : H_1 = G \\ \text{Vector} \end{array} & {}^{Md}\mathsf{Forward} \vee {}^{Md}\mathsf{Reverse} & {}^{Md}\mathsf{Starting} \vee {}^{Md}\mathsf{Home} \vee {}^{Md}\mathsf{Done} \\ \hline\hline {}^{mc}\mathsf{penDown} & \underline{\mathit{true}} & \underline{\mathit{false}} \\ \hline \end{array}$$

${}^{c}\mathsf{powerOn}$

$$\left| \neg \left({}^{Md}\mathsf{Done} \vee {}^{Md}\mathsf{Starting}\right) \Leftrightarrow {}^{c}\mathsf{powerOn}\right.$$

${}^{c}\mathsf{message}$

$|\ (\text{holding} \wedge {}^{c}\mathsf{message} = \text{“Stop button pressed, holding”})$

$\vee(\neg\text{holding} \wedge$

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 = G$ Vector	${}^{c}\mathsf{message}$
${}^{Md}\mathsf{Starting}$	“Path found, starting.”
${}^{Md}\mathsf{Forward}$	“Tracing forward.”
${}^{Md}\mathsf{Reverse}$	“Tracing backwards.”
${}^{Md}\mathsf{Home}$	“Tracing complete, returning to home position.”
${}^{Md}\mathsf{Done}\wedge$ $\text{connected}({}^{m}\mathsf{mazeStart}, {}^{m}\mathsf{mazeEnd})$	“Shutting down.”
${}^{Md}\mathsf{Done}\wedge$ $\neg\text{connected}({}^{m}\mathsf{mazeStart}, {}^{m}\mathsf{mazeEnd})$	“No path found, shutting down.”

)

Merit Function

${}^{c}\mathsf{merit}$

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : G$ Normal	
$= \quad @\mathbf{T}\left({}^{Md}\mathsf{Forward}\right) = \oslash \vee$ $@\mathbf{T}\left({}^{Md}\mathsf{Done}\right) = \oslash$	0
$@\mathbf{T}\left({}^{Md}\mathsf{Forward}\right) \neq \oslash \wedge$ $@\mathbf{T}\left({}^{Md}\mathsf{Done}\right) \neq \oslash$	$\frac{{}^{C}\mathsf{MAX\_TIME}}{\text{movingTime}}$

A.4.5 Environmental Constraints

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 ${}^{C}\mathsf{POS\_TOL}$ mm from the maze wall, the pen cannot be within ${}^{C}\mathsf{POS\_TOL}$ mm of both the home position and either of

${}^{m}\mathsf{mazeStart}$ or ${}^{m}\mathsf{mazeEnd}$, i.e.,

$$ {}^{mc}\mathsf{penPos} \in \mathrm{tol}(\{({}^{C}\mathsf{HOME\_X}, {}^{C}\mathsf{HOME\_Y})\}) \Rightarrow \left(\begin{array}{l} {}^{mc}\mathsf{penPos} \notin \mathrm{tol}(\{{}^{m}\mathsf{mazeStart}\}) \wedge \\ {}^{mc}\mathsf{penPos} \notin \mathrm{tol}(\{{}^{m}\mathsf{mazeEnd}\}) \end{array} \right) $$

The pen tip can move at a maximum of 2.0 mm/s, i.e., $\left|\frac{d}{dt}{}^{mc}\mathsf{penPos}\right| \leq 2.0\frac{\mathrm{mm}}{\mathrm{s}}$

A.4.6 Dictionary

Types

pathT = sequence of tuples of $(s, f : \mathbf{Posn})$

Posn = tuple of $\left(x : \left[{}^{C}\mathsf{MIN\_X}, {}^{C}\mathsf{MAX\_X}\right], y : \left[{}^{C}\mathsf{MIN\_Y}, {}^{C}\mathsf{MAX\_Y}\right]\right)$

buttonT = $\{\ {}^{s}\mathsf{Up}, {}^{s}\mathsf{Down}\ \}$

Functions

connected : **Posn** × **Posn** → **Boolean**
connected(b, e)

$$ \stackrel{\mathrm{df}}{=} (\exists p \in \mathbf{pathT}) \left(\begin{array}{l} b = p\,[0]\,.s \wedge e = p\,[|p| - 1]\,.f \wedge \\ (\forall i \in [0 \ldots |p| - 1]) \\ \left(\begin{array}{l} (i < |p| - 1) \Rightarrow (p\,[i]\,.f = p\,[i + 1]\,.s) \wedge \\ (p\,[i]\,.s.x = p\,[i]\,.f.x \vee p\,[i]\,.s.y = p\,[i]\,.f.y) \wedge \\ \neg\,(\exists q \in \mathbf{Posn}) \\ \left(\begin{array}{l} \mathrm{wall}(q) \wedge \\ \left(\left(\begin{array}{l} p\,[i]\,.s.x \leq q.x \leq p\,[i]\,.f.x \wedge \\ p\,[i]\,.s.y \leq q.y \leq p\,[i]\,.f.y \end{array} \right) \vee \left(\begin{array}{l} p\,[i]\,.f.x \leq q.x \leq p\,[i]\,.s.x \wedge \\ p\,[i]\,.f.y \leq q.y \leq p\,[i]\,.s.y \end{array} \right) \right) \end{array} \right) \end{array} \right) \end{array} \right) $$

dist : **Posn** × **Posn** → **Real**
dist(p, q)

$$ \stackrel{\mathrm{df}}{=} \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2} $$

forward : **Posn** → **Boolean**

forward(p)

$p \notin \text{tol}($

$\text{p}_\text{T} : H_1$ $\text{r}_\text{T} : G$ Normal	
$\stackrel{\text{df}}{=}$ $Prev(@\mathbf{T}\left({}^{Md}\mathsf{Reverse}\right)) = \oslash$	$\text{path}\left(First(@\mathbf{T}\left({}^{Md}\mathsf{Forward}\right)), \text{lagTime}\right)$
$Prev(@\mathbf{T}\left({}^{Md}\mathsf{Reverse}\right)) \neq \oslash$	$\left(\begin{array}{l}\text{path}\left(\begin{array}{l}First(@\mathbf{T}\left({}^{Md}\mathsf{Forward}\right)),\\ Last(@\mathbf{T}\left({}^{Md}\mathsf{Reverse}\right))\end{array}\right) \setminus \\ \text{tol}\left(\text{path}\left(\begin{array}{l}Last(@\mathbf{T}\left({}^{Md}\mathsf{Reverse}\right)),\\ Last(@\mathbf{T}\left({}^{Md}\mathsf{Forward}\right))\end{array}\right)\right)\end{array}\right)$ $\cup \text{path}\left(Last(@\mathbf{T}\left({}^{Md}\mathsf{Forward}\right)), \text{lagTime}\right)$

holding :→ **Boolean**

holding

$\text{p}_\text{T} : H_1 \wedge H_2$ $\text{r}_\text{T} : G$ Normal	${}^{c}\mathsf{message} =$ "Stop button pressed, holding."	
	$\underline{true}$	$\underline{false}$
$\stackrel{\text{df}}{=}$ ${}^{m}\mathsf{stopButton} = {}^{s}\mathsf{Down}$	$\underline{true}$	${}^{m}\mathsf{t} - Last\left(@\mathbf{T}\left(\begin{array}{l}{}^{m}\mathsf{stopButton}\\ = {}^{s}\mathsf{Down}\end{array}\right)\right) > {}^{C}\mathsf{RESPONSE\_TIME}$
${}^{m}\mathsf{stopButton} = {}^{s}\mathsf{Up}$	${}^{m}\mathsf{t} - Last\left(@\mathbf{F}\left(\begin{array}{l}{}^{m}\mathsf{stopButton}\\ = {}^{s}\mathsf{Down}\end{array}\right)\right) < {}^{C}\mathsf{RESPONSE\_TIME}$	$\underline{false}$

inMaze : **Posn** → **Boolean**

inMaze(p)

$$\stackrel{\text{df}}{=} \left(\begin{array}{c} -{}^{C}\mathsf{M\_X\_OFFSET} \leq p.x \leq -{}^{C}\mathsf{M\_X\_OFFSET} + {}^{C}\mathsf{M\_WIDTH} \wedge \\ {}^{C}\mathsf{M\_Y\_OFFSET} \leq p.y \leq {}^{C}\mathsf{M\_Y\_OFFSET} + {}^{C}\mathsf{M\_HEIGHT} \end{array}\right)$$

lagTime :→ **Real**

lagTime

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1 \wedge G$ $\mathrm{r_T} : H_2$ Inverted	$Last(@\mathbf{T}\,(\text{holding}))$ $-{}^{C}\mathsf{RESPONSE\_TIME}$	${}^{m}\mathsf{t} - {}^{C}\mathsf{RESPONSE\_TIME}$
¬holding	$Last(@\mathbf{F}\,(\text{holding})) >$ ${}^{m}\mathsf{t} - {}^{C}\mathsf{RESPONSE\_TIME}$	$Last(@\mathbf{F}\,(\text{holding})) \leq$ ${}^{m}\mathsf{t} - {}^{C}\mathsf{RESPONSE\_TIME}$
holding	$\underline{\underline{true}}$	$\underline{\underline{false}}$

movingTime :→ **Real**

movingTime

$\stackrel{\text{df}}{=} \big(First(@\mathbf{T}\,\big({}^{Md}\mathsf{Done}\big)) - First(@\mathbf{T}\,\big({}^{Md}\mathsf{Forward}\big))\big)$
$\quad -totalDrtn\,\big(\text{holding}, First(@\mathbf{T}\,\big({}^{Md}\mathsf{Forward}\big)), First(@\mathbf{T}\,\big({}^{Md}\mathsf{Done}\big))\big)$

path : **Real** × **Real** → set of **Posn**

path(t_i, t_f)

$\stackrel{\text{df}}{=} \{q \in \mathbf{Posn} \,|\, (\exists t)\,(t_i \leq t \leq t_f \wedge q = {}^{mc}\mathsf{penPos}(t))\}$

reverse : **Posn** → **Boolean**

reverse(p)

$$\stackrel{\text{df}}{=} p \in \text{tol}\left(\begin{array}{l}\left(\begin{array}{l}\text{path}(First(@\mathbf{T}\,\big({}^{Md}\mathsf{Forward}\big)), Last(@\mathbf{T}\,\big({}^{Md}\mathsf{Reverse}\big)))\backslash\\ \text{tol}(\text{path}(Last(@\mathbf{T}\,\big({}^{Md}\mathsf{Reverse}\big)), \text{lagTime}))\end{array}\right)\\ \cup\{{}^{m}\mathsf{mazeStart}\}\end{array}\right)$$

tol : set of **Posn** → set of **Posn**

tol(p)

$\stackrel{\text{df}}{=} \big\{q \in \mathbf{Posn} \,\big|\, (\exists r \in p)\,\big(\text{dist}(q, r) \leq {}^{C}\mathsf{POS\_TOL}\big)\big\}$

wall : **Posn** → **Boolean**

wall(p)

$\stackrel{\text{df}}{=} (\exists q \in {}^{m}\mathsf{mazeWalls})\,\big(\text{dist}(q, p) \leq {}^{C}\mathsf{WALL\_SPACE}\big)$

Constants

Table A.6: Constants

Name	Possible Values	Interpretation
^{C}HOME_X	$[^{C}\text{MIN\_X} \ldots {}^{C}\text{MAX\_X}]$	x location of pen 'home' position, millimetres.
^{C}HOME_Y	$[^{C}\text{MIN\_Y} \ldots {}^{C}\text{MAX\_Y}]$	y location of pen 'home' position, millimetres.
^{C}MAX_TIME	$[60 \ldots 300]$	Maximum time allowed to trace the maze, seconds.
^{C}MAX_X	$[0 \ldots 500]$	Maximum valid x co-ordinate, millimetres.
^{C}MAX_Y	$[0 \ldots 500]$	Maximum valid y co-ordinate, millimetres.
^{C}M_CELL_SIZE	$[4 \ldots 25]$	Width/Height of a maze cell, millimetres.
^{C}M_HEIGHT	$[^{C}\text{M\_CELL\_SIZE} \ldots {}^{C}\text{MAX\_Y}]$	Height of maze, millimetres.
^{C}MIN_X	$[-500 \ldots 0]$	Minimum valid x co-ordinate, millimetres.
^{C}MIN_Y	$[-500 \ldots 0]$	Minimum valid y co-ordinate, millimetres.
^{C}M_WIDTH	$[^{C}\text{M\_CELL\_SIZE} \ldots {}^{C}\text{MAX\_X}]$	Width of maze, millimetres.
^{C}M_X_OFFSET	$[1 \ldots {}^{C}\text{MAX\_X} - {}^{C}\text{M\_WIDTH}]$	x distance of maze from origin, millimetres.
^{C}M_Y_OFFSET	$[1 \ldots {}^{C}\text{MAX\_Y} - {}^{C}\text{M\_HEIGHT}]$	y distance of maze from origin, millimetres.
^{C}POS_TOL	$[1 \ldots \frac{{}^{C}\text{M\_CELL\_SIZE}}{2}]$	Maximum tolerance on locating the start and end positions, millimetres.
^{C}RESPONSE_TIME	$[2 \ldots 15]$	Maximum delay before responding to a button, seconds.
^{C}WALL_SPACE	$[1 \ldots \frac{{}^{C}\text{M\_CELL\_SIZE}}{2}]$	Minimum distance between the pen and walls, millimetres.

Appendix B

Monitor Generator Users' Guide

The Monitor Generator is implemented as a component tool of the Table Tool System (TTS) as is described in the on-line version of [96]. This appendix describes how the Monitor Generator can be used to generate a monitor from a System Requirements Document, represented as expressions in a TTS context file. The interface to the generated code is described in Section 6.3.2.

B.1 SRD Format

The SRD is represented by a TTS context file, which contains an ordered list of named expressions and related symbol information. The order of the expressions in the context determines the order in which expressions are evaluated by `MON_update`, if necessary, so, for example, if a condition depends on a mode class, then it should be placed after that mode class definition. Table B.1 lists the contents of the TTS context file representing the SRD in Section A.4.

B.1.1 Environmental Variables

An expression is required to identify each variable that represents an environmental quantity in the SRD. The name of each such expression is of the form "`Env` *name*", where *name* is the name of the environmental variable. The expression is simply the variable.

For each variable (environmental or not) the following symbol information is required (numbers in brackets are the information module "info class" numbers):

Name (1) The variable name, which MG translates into the C++ variable name used in the monitor implementation.

Tag (4) `VarTag` (= 2)

CType (11) The variable type, formated such that `printf(` *ctype*, *name*`)` will print a valid C++ declaration.

The variable "`m_t`" is assumed to represent the current time and must always be present.

B.1.2 Conditions

Each condition used in the SRD must be defined in an expression, the name of which is of the form "`Cond` *name*", where *name* is the condition variable name. The expression is the predicate expression that defines the condition. It is evaluated to update the condition value in `MON_update`.

For each condition variable the following symbol information is required:

Name (1) The condition name, which MG translates into the C++ condition variable name used in the monitor implementation.

Tag (4) `VarTag` (= 2)

CType (11) `Condition %s`

The value of a condition is referenced in other expressions using the condition variable. The current value of condition, ^{p}`c`, is denoted "default(^{p}`c`)".

B.1.3 Histories

A history expression is required for each variable of which previous (i.e., historical) values are needed in the SRD. The name of the expression is of the form "`Hist` *name*`Hist`", where *name* is an variable name. The expression is the variable.

Histories are accessed in other expressions implicitly when a history function (e.g., `dBYdt`) is invoked with the variable as its argument.

B.1.4 Mode Classes

A mode class definition is required for each mode class in the SRD. The name of the expression is of the form "`ModeClass` *clname*", where *clname* is the mode class name. The expression is simply a tuple containing the modes in the class, each of which is a constant with name of the form "*clname*`::`*mdname*", where *clname* is the mode class name and *mdname* is the mode.

For each mode class, *clname*, the following functions must be defined in the dictionary:

- *clname*`::init` Initial mode function.
- *clname*`::trans` Ideal mode transition function.
- *clname*`::delay` Transition delay tolerance function.
- *clname*`::isSysMode` Mode apparency function.

There must also be a variable symbol defined for the mode variable, that has name "`mc_`*clname*". This variable is used to reference the current mode in this class using the "default" function, as for conditions, above. The CType of the mode variable symbol is "*clname*`::mode %s`".

B.1.5 Behaviour

An expression with name "`Behaviour`" is required. The expression is the characteristic predicate of acceptable system behaviour (typically the conjunction of controlled value relations, defined in the dictionary).

B.1.6 Dictionary

Auxiliary function and predicate definitions, as well as the controlled value relations, merit function, and mode class functions discussed above are passed directly to TOG for implementation as 'auxiliary functions', and so must conform to the form required by that tool:

- The 'root' of the expression is the special symbol `defined`.

- The first sub-expression gives the signature: function name and formal arguments.
- The second sub-expression gives the function definition in terms of the formal arguments.

The following symbol information is required for each function or predicate name:

Name (1) The function name, which MG translates into the C++ function name used in the monitor implementation.

Tag (4) `FATag` (= 3) or `LETag` (= 5)

Arity (6)

CType (11) The C++ data type of the function (**not** in `printf` format as for variables).

Expressions with name beginning with "`Std` " are ignored by the generator. This allows functions to be implemented by hand where appropriate. The function name symbol must have CForm (12) information defined, which is used to form the function invocation. It consists of an array of arity + 1 strings, which precede each argument expression, in turn, and follow the final argument. (e.g., "`f(` x, y`)`" is represented by the strings "`f(` ", "`,` ", "`)`".)

Table B.1: Maze-Tracer SRD TTS Context File

Name	Expression
`Env m_t`	^{m}t
`Env m_mazeWalls`	^{m}mazeWalls
`Env m_mazeStart`	^{m}mazeStart
`Env m_mazeEnd`	^{m}mazeEnd
`Env m_stopButton`	^{m}stopButton
`Env m_homeButton`	^{m}homeButton
`Env m_backButton`	^{m}backButton
`Env mc_penPos`	^{mc}penPos
`Env mc_penDown`	^{mc}penDown

Name	Expression
`Env c_message`	^{c}message
`Env c_powerOn`	^{c}powerOn
`Cond stopPressed`	^{m}stopButton = ^{s}Down
`Cond backPressed`	^{m}backButton = ^{s}Down
`Cond homePressed`	^{m}homeButton = ^{s}Down
`Cond penDown`	^{mc}penDown
`Cond atHome`	^{mc}penPos $\in$ tol $(\{({}^{C}$HOME_X, ^{C}HOME_Y$)\})$
`Cond atEnd`	^{mc}penPos $\in$ tol $(\{{}^{m}$mazeEnd$\})$
`Cond atStart`	^{mc}penPos $\in$ tol $(\{{}^{m}$mazeStart$\})$
`Cond holding`	see Holding on page 115
`Hist mc_penPosHist`	^{mc}penPos
`mc_penPos_cvf`	see ^{mc}penPos controlled value function on page 112
`mc_penDown_cvf`	see ^{mc}penDown controlled value function on page 112
`c_powerOn_cvf`	see ^{c}powerOn controlled value function on page 112
`c_message_cvf`	see ^{c}message controlled value function on page 113
`Behaviour`	mc_penPos_cvf(default(mc_direction)) $\wedge$ mc_penDown_cvf(default(mc_direction)) $\wedge$ c_powerOn_cvf(default(mc_direction)) $\wedge$ c_message_cvf(default(mc_direction))
`Merit`	see definition on page 113
`ModeClass direction`	$({}^{Md}$Starting, ^{Md}Forward, ^{Md}Reverse, ^{Md}Home, ^{Md}Done$)$
`direction::init`	see definition on page 111
`direction::trans`	see definition on page 111
`direction::delay`	see definition on page 112
`direction::isSysMode`	c_message_cvf(m)
`Cond forwardMode`	default(mc_direction) = ^{Md}Forward
`Cond reverseMode`	default(mc_direction) = ^{Md}Reverse
`Cond doneMode`	default(mc_direction) = ^{Md}Done
`forward`	see definition on page 115
`inMaze`	see definition on page 115

Name	Expression
`reverse`	see definition on page 116
`Std tol`	see definition on page 116
`lagTime`	see definition on page 116
`movingTime`	see definition on page 116
`Std wall`	see definition on page 116
`Std connected`	see definition on page 114
`Std path`	see definition on page 116

B.2 Users' Interface

The 'user' input required by the MG is to specify file names for:

- the System Interface Declarations code,
- the SRD context file, and
- the output files.

The prototype tool described in this thesis uses a simple command-line user interface, and assumes `behaviour.def` for the system interface declarations and `behaviour.cc` and `behaviour.h` for the output files. The SRD context file must be given on the command line.

For integration with the TTS a graphical user interface should be developed. The required interface is virtually identical to the existing TOG Users' interface, so that should be adapted for use by the monitor generator. The only changes required will be to text labels on the interface and help text.

Appendix C

Monitor Generator Design Documentation

This appendix contains design documentation for the Monitor Generator and Monitor Library.

C.1 Monitor Generator Interface

The relationships between monitor generator classes is given, using the Unified Modelling Language (UML) class diagram notation[84], in Figure C.1. The interface between the MG and other TTS tools is through the Requirements and Monitor Constructor modules, which are described in the following sub-sections.

C.1.1 Requirements

The Requirements module is responsible for interpreting a TTS context (`CHandle`) as a requirements document, determining if each expression is a specification object, the behaviour expression, or an auxiliary definition. The `MG_Req` constructor method constructs the following from the context, as illustrated in Figure C.2,

- a list of `MG_SpecObj` objects,
- a TTS context containing the auxiliary definitions, and

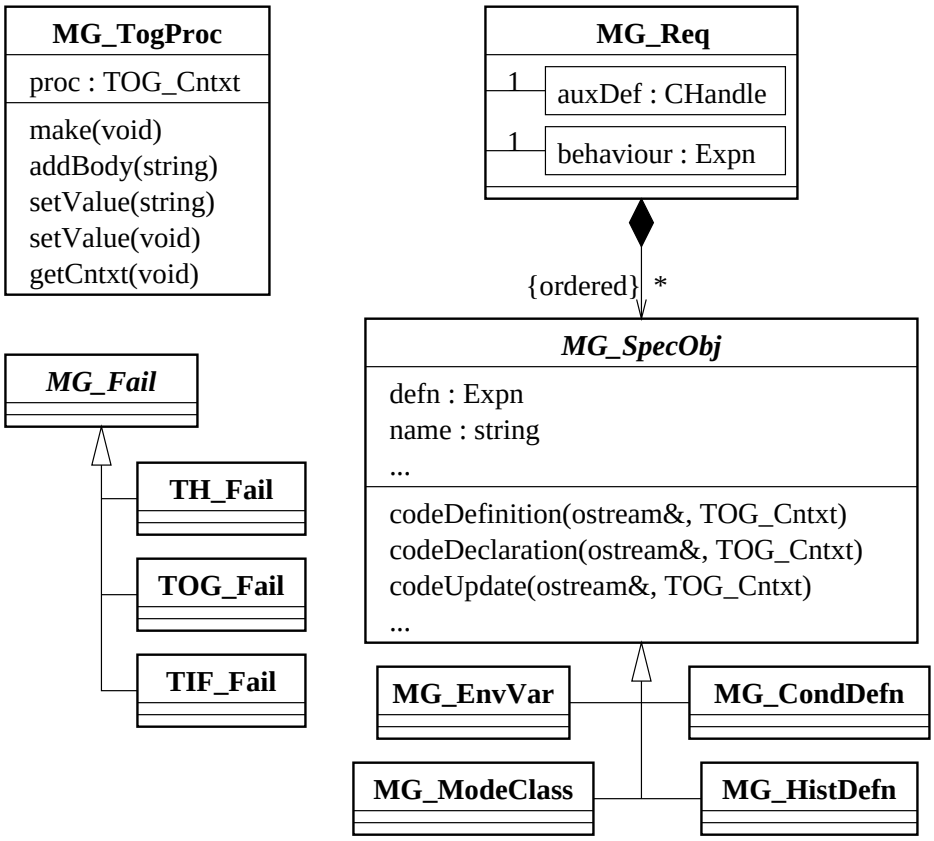

Figure C.1: Monitor Generator Class Diagram

- the behaviour expression.

C.1.2 Monitor Constructor

The Monitor Constructor module consists of one access program, `MG_monBuild(MG_Req& req, string& codefilename, string& userdef)`, which invokes methods in the appropriate order to generate the monitor implementation as illustrated in Figure C.3.

```
defs = empty list
for all e expression in ch do
  if name of e begins with "Env " then
    o = new MG_EnvVar object
    add o to defs
  else if name of e begins with "Cond " then
    o = new MG_CondDefn object
    add o to defs
  else if name of e begins with "Hist " then
    o = new MG_HistDefn object
    add o to defs
  else if name of e begins with "ModeClass " then
    o = new MG_ModeClass object
    add o to defs
  else if name of e = "Behaviour" then
    behaviour = e
  else if name of e begins with "Std " then
    // ignore this expression
  else // it's an auxiliary definition
    add e to auxDef
  end if
end for
```

Figure C.2: Requirements Object Constructor (`MG_Req(ch)`) Algorithm

C.2 Monitor Library Interface

Figure C.4 is a UML class diagram illustrating the relationships between the monitor library classes. The remainder of this section gives semi-formal interface specifications for the monitor library classes.

initialize TOG, passing `userdef`
for all *o* in `req.defs` **do**
  o.`codeDeclaration` into "`codefilename`.h" file.
end for
`TOG_oracle(req.auxDef)` `//` *generate auxiliary definition implementations*
update = `MG_TogProc`("MON_update")
for all *o* in `req.defs` **do**
  o.`codeDefinition` into "`codefilename`.cc" file.
  o.`codeUpdate` into *update*
end for
behave = `MG_TogProc`("MON_behaviour")
implement `req.behaviour` into *behave* using `TOG_expn`
close all files.

Figure C.3: Monitor Generation (`MG_monBuild`) Algorithm

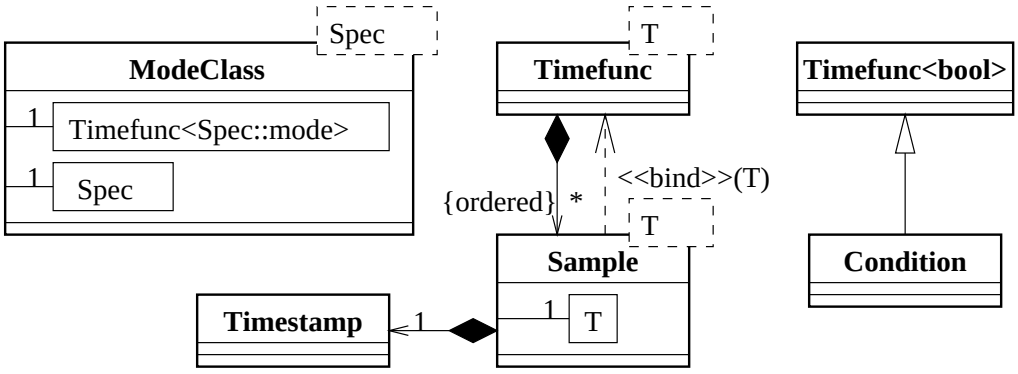

Figure C.4: Monitor Library Class Diagram

C.2.1 Timestamp

Exported Types

```
class Timestamp
```

Imported Types

Type	Defined in
ostream	iostream.h
istream	iostream.h
timeval	sys/time.h

Member Functions

Name	Value Type	Arg. 1 Type	Arg. 2 Type
Timestamp	Timestamp&	int	int
Timestamp	Timestamp&	timeval	
Timestamp	Timestamp&	double	
set	Timestamp&	int	int
operator+=	Timestamp&	Timestamp	
operator+=	Timestamp&	double	
operator-=	Timestamp&	Timestamp	
operator-=	Timestamp&	double	
seconds	int		
useconds	int		
operator double	double		

Non-member Functions

Name	Value Type	Arg. 1 Type	Arg. 2 Type
operator <<	ostream&	ostream&	Timestamp
operator >>	istream&	istream&	Timestamp
operator +	Timestamp	Timestamp	Timestamp
operator +	Timestamp	double	Timestamp

Name	Value Type	Arg. 1 Type	Arg. 2 Type
`operator +`	`Timestamp`	`Timestamp`	`double`
`operator -`	`Timestamp`	`Timestamp`	`Timestamp`
`operator -`	`Timestamp`	`double`	`Timestamp`
`operator -`	`Timestamp`	`Timestamp`	`double`
`now`	`Timestamp *`	`Timestamp *`	
`operator <`	`bool`	`Timestamp`	`Timestamp`
`operator ==`	`bool`	`Timestamp`	`Timestamp`

Representation

$R \stackrel{\text{df}}{=} (\mathsf{sec}, \mathsf{usec} : \texttt{int})$

Behaviour

$\texttt{Timestamp}(\texttt{int}\ s, \texttt{int}\ u)$
$\quad \stackrel{\text{df}}{=} R' = (s, u) \wedge \mathsf{value} = R'$

$\texttt{Timestamp}(\texttt{timeval}\ t)$
$\quad \stackrel{\text{df}}{=} R' = (t.\texttt{tv\_sec}, t.\texttt{tv\_usec}) \wedge \mathsf{value} = R'$

$\texttt{Timestamp}(\texttt{double}\ d)$
$\quad \stackrel{\text{df}}{=} R' = (\texttt{int}(d), (d - \texttt{int}(d)) * 10^6) \wedge \mathsf{value} = R'$

$R.\texttt{set}(\texttt{int}\ s, \texttt{int}\ u)$
$\quad \stackrel{\text{df}}{=} R' = (s, u) \wedge \mathsf{value} = R'$

$R\texttt{+=}(\texttt{Timestamp}\ t)$
$\quad \stackrel{\text{df}}{=} R' = (\text{`}R.\mathsf{sec} + t.\mathsf{sec}, \text{`}R.\mathsf{usec} + t.\mathsf{usec}) \wedge \mathsf{value} = R'$

$R\texttt{+=}(\texttt{double}\ d)$
$\quad \stackrel{\text{df}}{=} R' = \text{`}R\texttt{+=}\ \texttt{Timestamp}(d) \wedge \mathsf{value} = R'$

$R\texttt{-=}(\texttt{Timestamp}\ t)$
$\quad \stackrel{\text{df}}{=} R' = (\text{`}R.\mathsf{sec} - t.\mathsf{sec}, \text{`}R.\mathsf{usec} - t.\mathsf{usec}) \wedge \mathsf{value} = R'$

R−=(double d)
$\stackrel{\text{df}}{=} R' = \text{`}R-= \texttt{Timestamp}(d) \wedge \mathsf{value} = R'$

R.seconds()
$\stackrel{\text{df}}{=} R' = \text{`}R \wedge \mathsf{value} = \text{`}R.\mathsf{sec}$

R.useconds()
$\stackrel{\text{df}}{=} R' = \text{`}R \wedge \mathsf{value} = \text{`}R.\mathsf{usec}$

double(R)
$\stackrel{\text{df}}{=} R' = \text{`}R \wedge \mathsf{value} = \text{`}R.\mathsf{sec} + \text{`}R.\mathsf{usec}/10^6$

ostream& s << Timestamp t
$\stackrel{\text{df}}{=} s' = \text{`}s << t.\mathsf{sec} << \text{“.”} << t.\mathsf{usec}$ (fixed precision, 6 decimal points)$\wedge$
$\mathsf{value} = s'$

istream& s >> Timestamp t
$\stackrel{\text{df}}{=} s' = \text{`}s >> t.\mathsf{sec} >> \text{“.”} >> t.\mathsf{usec}$

Timestamp l + Timestamp r
$\stackrel{\text{df}}{=} \mathsf{value} = \texttt{Timestamp}(l.\mathsf{sec} + r.\mathsf{sec}, l.\mathsf{usec} + r.\mathsf{usec})$

Timestamp l + double r
$\stackrel{\text{df}}{=} \mathsf{value} = l + \texttt{Timestamp}(r)$

double l + Timestamp r
$\stackrel{\text{df}}{=} \mathsf{value} = \texttt{Timestamp}(l) + r$

Timestamp l − Timestamp r
$\stackrel{\text{df}}{=} \mathsf{value} = \texttt{Timestamp}(l.\mathsf{sec} - r.\mathsf{sec}, l.\mathsf{usec} - r.\mathsf{usec})$

Timestamp l − double r
$\stackrel{\text{df}}{=} \mathsf{value} = l - \texttt{Timestamp}(r)$

double l − Timestamp r
$\stackrel{\text{df}}{=} \mathsf{value} = \texttt{Timestamp}(l) - r$

$\texttt{now}(t)$
$\quad \stackrel{\text{df}}{=} *t' = \text{ current system time } \wedge \mathsf{value} = t'$

$\texttt{Timestamp}\ l < \texttt{Timestamp}\ r$
$\quad \stackrel{\text{df}}{=} \mathsf{value} = l.\mathsf{sec} < r.\mathsf{sec} \vee l.\mathsf{sec} = r.\mathsf{sec} \wedge l.\mathsf{usec} < r.\mathsf{usec}$

$\texttt{Timestamp}\ l == \texttt{Timestamp}\ r$
$\quad \stackrel{\text{df}}{=} \mathsf{value} = l.\mathsf{sec} = r.\mathsf{sec} \wedge l.\mathsf{usec} = r.\mathsf{usec}$

Implementation Files

`timestamp.h`, `timestamp.cc`

C.2.2 Time Function

Exported Types

`template class Timefunc<T>`

Imported Types

Type	Defined in
`ostream`	`iostream.h`
`istream`	`iostream.h`
`template class list<T>`	`list.h` (STL)
`class Timestamp`	`timestamp.h`

Member Types

`template class Timefunc<T>::iterator`

`template class Timefunc<T>::const_iterator`

`template class Timefunc<T>::Sample`

`template class Timefunc<T>::UnknownVal`

Member Functions

Name	Value Type	Arg. 1 Type	Arg. 2 Type	Arg. 3 Type
`Timefunc<T>`	`Timefunc<T>&`	`int`	`bool`	
`Timefunc<T>`	`Timefunc<T>&`	`double`	`bool`	
`Timefunc<T>`	`Timefunc<T>&`	`Timefunc<T>&`	`Timestamp&`	`Timestamp&`
`addPoint`	`Timefunc<T>&`	`Timestamp&`	`T`	
`operator()`	`T`			
`operator()`	`T`	`Timestamp&`		
`after`	`T`	`Timestamp&`		
`before`	`T`	`Timestamp&`		
`oldest`	`T`			
`dBYdt`	`T`			
`begin`	`Timefunc<T>::iterator`			

Name	Value Type	Arg. 1 Type	Arg. 2 Type	Arg. 3 Type
`begin`	`Timefunc<T>::const_iterator`			
`end`	`Timefunc<T>::iterator`			
`end`	`Timefunc<T>::const_iterator`			
`empty`	`bool`			

Non-member Functions

Name	Value Type	Arg. 1 Type	Arg. 2 Type	Arg. 3 Type
`operator <<`	`ostream&`	`ostream&`	`Timefunc<T>&`	
`operator <<`	`ostream&`	`ostream&`	`Timefunc<T>::Sample&`	
`template <class T, class P> findPrev`	`Timestamp&`	`Timefunc<T>&`	`Pred&`	`int`
`template <class T, class P> findFirst`	`Timestamp&`	`Timefunc<T>&`	`Pred&`	`int`

Representation

$\mathsf{Sample} \stackrel{\text{df}}{=} (\mathsf{time} : \texttt{Timestamp}; \mathsf{val} : \texttt{T})$

$$R \stackrel{\text{df}}{=} \begin{pmatrix} \mathsf{hist} : \text{ sequence of } \mathsf{Sample}; \\ \mathsf{maxLen} : \texttt{int}; \\ \mathsf{maxDur} : \texttt{double}; \\ \mathsf{keepInit} : \texttt{bool} \end{pmatrix}$$

Behaviour

`Timefunc`(`int` l, `bool` k)

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	$\underline{true}$

$\stackrel{\text{df}}{=}$

R.hist′	empty list
R.maxLen′	l
R.maxDur′	0
R.keepInit′	k
value	R'

`Timefunc`(`double` d, `bool` k)

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	$\underline{true}$

$\stackrel{\text{df}}{=}$

R.hist′	empty list
R.maxLen′	0
R.maxDur′	d
R.keepInit′	k
value	R'

`Timefunc`(`Timefunc&` f, `Timestamp&` t_1, `Timestamp&` t_2)

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	$\underline{true}$

$\stackrel{\text{df}}{=}$

R.hist′	copy of $\text{`}f$.hist$[i] \mid t_1 \leq \text{`}f$.hist$[i]$.time $\leq t_2$
R.maxLen′	$\text{`}f$.maxLen
R.maxDur′	$\text{`}f$.maxDur
R.keepInit′	$\text{`}f$.keepInit
value	R'

`addPoint`(`Timestamp`& t, `T` v)

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	$\mathrm{tooLong}(\text{`}R, t)$	$\neg\mathrm{tooLong}(\text{`}R, t)$
$R.\mathsf{hist}'$	$\mathrm{dropIt}(\text{`}R, t) + (t, v)$	$\text{`}R.\mathsf{hist} + (t, v)$
$R.\mathsf{maxLen}'$	$\text{`}R.\mathsf{maxLen}$	
$R.\mathsf{maxDur}'$	$\text{`}R.\mathsf{maxDur}$	
$R.\mathsf{keepInit}'$	$\text{`}R.\mathsf{keepInit}$	
value	R'	

$\stackrel{\text{df}}{=}$

$R()$

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	$\lvert\text{`}R.\mathsf{hist}\rvert > 0$	$\lvert\text{`}R.\mathsf{hist}\rvert = 0$
value	$\text{`}R.\mathsf{hist}[\lvert\text{`}R.\mathsf{hist}\rvert - 1].\mathsf{val}$	throw `UnknownVal`

$\stackrel{\text{df}}{=}$

$\wedge R' = \text{`}R$

R(`Timestamp` t)

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 = G$ Vector	value
$\lvert\text{`}R.\mathsf{hist}\rvert > 0 \wedge$ $\text{`}R.\mathsf{hist}[0].\mathsf{time} \leq t$	$\text{`}R.\mathsf{hist}[i].\mathsf{val} \;\middle\vert\; \text{`}R.\mathsf{hist}[i].\mathsf{time} = max\left(\left\{ t_s \;\middle\vert\; \exists j, t_s = \text{`}R.\mathsf{hist}[j].\mathsf{time} \wedge t_s \leq t \right\}\right)$
$\lvert\text{`}R.\mathsf{hist}\rvert = 0 \vee$ $\text{`}R.\mathsf{hist}[0].\mathsf{time} > t$	throw `UnknownVal`

$\stackrel{\text{df}}{=}$

$\wedge R' = \text{`}R$

R.`after`(`Timestamp` t)

$\stackrel{\text{df}}{=} R(t)$

R.`before`(`Timestamp` t)

$p_T : H_1$ $r_T : H_2 = G$ Vector	value
$\|\text{`}R.\mathsf{hist}\| > 0 \wedge$ $\text{`}R.\mathsf{hist}[0].\mathsf{time} \leq t$	$\text{`}R.\mathsf{hist}[i].\mathsf{val} \mid \text{`}R.\mathsf{hist}[i].\mathsf{time} = max\left(\left\{ t_s \mid \exists j, t_s = \text{`}R.\mathsf{hist}[j].\mathsf{time} \wedge t_s < t \right\}\right)$
$\|\text{`}R.\mathsf{hist}\| = 0 \vee$ $\text{`}R.\mathsf{hist}[0].\mathsf{time} > t$	throw `UnknownVal`

$\stackrel{\text{df}}{=}$ $\wedge R' = \text{`}R$

R.`oldest`()

$p_T : H_1$ $r_T : H_2 = G$ Vector	value
$\|\text{`}R.\mathsf{hist}\| > 0$	$\text{`}R.\mathsf{hist}[0].\mathsf{val}$
$\|\text{`}R.\mathsf{hist}\| = 0$	throw `UnknownVal`

$\stackrel{\text{df}}{=}$ $\wedge R' = \text{`}R$

R.`dBYdt`()

$p_T : H_1$ $r_T : H_2 = G$ Vector	value
$\|\text{`}R.\mathsf{hist}\| > 1$	$\frac{\text{`}R.\mathsf{hist}[\|\text{`}R.\mathsf{hist}\|-1].\mathsf{val} - \text{`}R.\mathsf{hist}[\|\text{`}R.\mathsf{hist}\|-2].\mathsf{val}}{\text{`}R.\mathsf{hist}[\|\text{`}R.\mathsf{hist}\|-1].\mathsf{time} - \text{`}R.\mathsf{hist}[\|\text{`}R.\mathsf{hist}\|-2].\mathsf{time}}$
$\|\text{`}R.\mathsf{hist}\| \leq 1$	throw `UnknownVal`

$\stackrel{\text{df}}{=}$ $\wedge R' = \text{`}R$

R.`begin`()

$\stackrel{\text{df}}{=}$ $\mathsf{value} = \&(\text{`}R.\mathsf{hist}[0]) \wedge R' = \text{`}R$

R.`end`()

$\stackrel{\text{df}}{=}$ $\mathsf{value} = \&(\text{`}R.\mathsf{hist}[|\text{`}R.\mathsf{hist}|]) \wedge R' = \text{`}R$

R.`empty`()

$\stackrel{\text{df}}{=} \mathsf{value} = (|\backprime R.\mathsf{hist}| = 0) \wedge R' = \backprime R$

`ostream`& s << `Timefunc` f

$\stackrel{\text{df}}{=} s' = \backprime s << \ (\text{all samples}) \ \wedge \mathsf{value} = s'$

`ostream`& s << `Sample` f

$\stackrel{\text{df}}{=} s' = \backprime s << f.\mathsf{time} << \text{“}\backslash t\text{”} << f.\mathsf{val} \wedge$
$\mathsf{value} = s'$

`findPrev`(`Timefunc` f, `Pred` p, `int` n)

	$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 = G$ Vector	value
$\stackrel{\text{df}}{=}$	$card\ (\{i \mid p(\backprime f.\mathsf{hist}[i].\mathsf{val})\}) \geq n$	$\backprime f.\mathsf{hist}[i].\mathsf{time} \mid card\left(\left\{ j \mid \begin{array}{l} j \geq i \wedge \\ p(\backprime f.\mathsf{hist}[j].\mathsf{val}) \end{array} \right\}\right) = n \wedge p(\backprime f.\mathsf{hist}[i].\mathsf{val})$
	$card\ (\{i \mid p(\backprime f.\mathsf{hist}[i].\mathsf{val})\}) < n$	throw `UnknownVal`

`findFirst`(`Timefunc` f, `Pred` p, `int` n)

	$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 = G$ Vector	value
$\stackrel{\text{df}}{=}$	$card\ (\{i \mid p(\backprime f.\mathsf{hist}[i].\mathsf{val})\}) \geq n$	$\backprime f.\mathsf{hist}[i].\mathsf{time} \mid card\left(\left\{ j \mid \begin{array}{l} j \leq i \wedge \\ p(\backprime f.\mathsf{hist}[j].\mathsf{val}) \end{array} \right\}\right) = n \wedge p(\backprime f.\mathsf{hist}[i].\mathsf{val})$
	$card\ (\{i \mid p(\backprime f.\mathsf{hist}[i].\mathsf{val})\}) < n$	throw `UnknownVal`

Implementation Files

`timefunc.h`

Auxiliary Definitions

tooLong : `Timefunc` $\times$ `Timestamp` $\rightarrow$ **Boolean**

tooLong(f, t)

$\mathrm{p_T} : H_1 \wedge H_2$ $\mathrm{r_T} : G$ Normal	f.keepInit	$\neg f$.keepInit
f.maxLen > 0	$\lvert f$.hist$\rvert \geq f$.maxLen	$\lvert f$.hist$\rvert \geq f$.maxLen
f.maxLen $= 0 \wedge$ f.maxDur > 0	f.hist$[1]$.time $<$ $t - f$.maxDur	f.hist$[0]$.time $<$ $t - f$.maxDur
f.maxLen $= 0 \wedge$ f.maxDur $= 0$	$\underline{false}$	$\underline{false}$

dropIt : `Timefunc` $\times$ `Timestamp` $\rightarrow$ sequence of `Sample`

dropIt(f, t)

$\mathrm{p_T} : H_1 \wedge H_2$ $\mathrm{r_T} : G$ Normal	f.keepInit	$\neg f$.keepInit
f.maxLen > 0	f.hist$[0] +$ f.hist$[2 \ldots \lvert f$.hist$\rvert - 1]$	f.hist$[1 \ldots \lvert f$.hist$\rvert - 1]$
f.maxLen $= 0 \wedge$ f.maxDur > 0	f.hist$[0]+$ f.hist$[i] \mid i > 0 \wedge f$.hist$[i]$.time $\geq t - f$.maxDur	f.hist$[i] \mid f$.hist$[i]$.time $\geq t - f$.maxDur

C.2.3 Condition

Exported Types

`class Condition`

Member Types

`enum Change`

Imported Types

Type	Defined in
`Timestamp`	`timestamp.h`
`template Timefunc<T>`	`timefunc.h`

Inherits: `Timefunc<bool>`.

Member Functions

Name	Value Type	Arg. 1 Type	Arg. 2 Type
`Condition`	`Condition&`	`int`	`bool`
`Condition`	`Condition&`	`double`	`bool`
`addPoint`	`Condition&`	`Timestamp`	`bool`

Non-member Functions

Name	Value Type	Arg. 1 Type	Arg. 2 Type	Arg. 3 Type
`Last`	`Timestamp`	`Change`	`Condition&`	
`First`	`Timestamp`	`Change`	`Condition&`	
`Since`	`double`	`Change`	`Condition&`	`Timestamp&`
`Drtn`	`double`	`Condition&`	`Timestamp&`	
`totalDrtn`	`double`	`Condition&`	`Timestamp&`	`Timestamp&`
`AtTrue`	`double`	`Condition&`	`Timestamp&`	
`AtFalse`	`double`	`Condition&`	`Timestamp&`	
`When`	`double`	`Condition&`	`Timestamp&`	

Name	Value Type	Arg. 1 Type	Arg. 2 Type	Arg. 3 Type
`While`	`double`	`Condition&`	`Timestamp&`	
`WhenF`	`double`	`Condition&`	`Timestamp&`	
`WhileF`	`double`	`Condition&`	`Timestamp&`	

Representation

$\texttt{Sample} \stackrel{\text{df}}{=} (\mathsf{time} : \texttt{Timestamp}; \mathsf{val} : \texttt{bool})$

$$R \stackrel{\text{df}}{=} \begin{pmatrix} \mathsf{hist} : \text{ sequence of } \texttt{Sample}; \\ \mathsf{maxLen} : \texttt{int}; \\ \mathsf{maxDur} : \texttt{double}; \\ \mathsf{keepInit} : \texttt{bool} \end{pmatrix}$$

Behaviour

`Condition`(`int` l, `bool` k)

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	$\underline{true}$

$\stackrel{\text{df}}{=}$

$R.\mathsf{hist}'$	empty list
$R.\mathsf{maxLen}'$	l
$R.\mathsf{maxDur}'$	0
$R.\mathsf{keepInit}'$	k
value	R'

`Condition`(`double` d, `bool` k)

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	$\underline{true}$

$\stackrel{\text{df}}{=}$

$R.\mathsf{hist}'$	empty list
$R.\mathsf{maxLen}'$	0
$R.\mathsf{maxDur}'$	d
$R.\mathsf{keepInit}'$	k
value	R'

R.addPoint(Timestamp& t, bool v)

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	$\backprime R.\mathsf{hist}[\lvert\backprime R.\mathsf{hist}\rvert - 1].\mathsf{val} = v$	
	false	*true*
R'	Timefunc<bool>(R).addPoint(t, v)	$\backprime R$

Last(Change d, Condition& f)

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 = G$ Vector	value
$\exists i, \mathrm{IsChange}(d, \backprime f, i)$	$\backprime f.\mathsf{hist}[i].\mathsf{time} \mid i = max\,(\{j \mid \mathrm{IsChange}(d, \backprime f, j)\})$
$\neg\exists i, \mathrm{IsChange}(d, \backprime f, i)$	throw UnknownVal

First(Change d, Condition& f)

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 = G$ Vector	value
$\exists i, \mathrm{IsChange}(d, \backprime f, i)$	$\backprime f.\mathsf{hist}[i].\mathsf{time} \mid i = min\,(\{j \mid \mathrm{IsChange}(d, \backprime f, j)\})$
$\neg\exists i, \mathrm{IsChange}(d, \backprime f, i)$	throw UnknownVal

Since(Change d, Condition& f, Timestamp t)

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 = G$ Vector	value
$\exists i, \mathrm{IsChange}(d, \backprime f, i)$	$t - \mathtt{Last}(d, f)$
$\neg\exists i, \mathrm{IsChange}(d, \backprime f, i)$	0

Drtn(Condition& f, Timestamp t)

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 = G$ Vector	value
$\backprime f.\mathsf{hist}[\lvert\backprime f.\mathsf{hist}\rvert - 1].\mathsf{val}$	$t - \mathtt{Last}(\mathtt{AtT}, f)$
$\neg\backprime f.\mathsf{hist}[\lvert\backprime f.\mathsf{hist}\rvert - 1].\mathsf{val}$	0

`totalDrtn`(`Condition`& f, `Timestamp` t_1, `Timestamp` t_2)
$\stackrel{\text{df}}{=}$ value $= \sum_{I \in \text{YesTimes}(\backprime f, t_1, t_2)} (I.e - I.b)$

`AtTrue`(`Condition`& f, `Timestamp` t)
$\stackrel{\text{df}}{=}$ value $= (\exists i, \backprime f.\text{hist}[i].\text{time} = t \wedge \backprime f.\text{hist}[i].\text{val} = \texttt{true})$

`AtFalse`(`Condition`& f, `Timestamp` t)
$\stackrel{\text{df}}{=}$ value $= (\exists i, \backprime f.\text{hist}[i].\text{time} = t \wedge \backprime f.\text{hist}[i].\text{val} = \texttt{false})$

`When`(`Condition`& f, `Timestamp` t)
$\stackrel{\text{df}}{=}$ value $= \backprime f.\texttt{before}(t)$

`While`(`Condition`& f, `Timestamp` t)
$\stackrel{\text{df}}{=}$ value $= (\backprime f.\texttt{before}(t) \wedge \backprime f.\texttt{after}(t))$

`WhenF`(`Condition`& f, `Timestamp` t)
$\stackrel{\text{df}}{=}$ value $= \neg \backprime f.\texttt{before}(t)$

`WhileF`(`Condition`& f, `Timestamp` t)
$\stackrel{\text{df}}{=}$ value $= (\neg \backprime f.\texttt{before}(t) \wedge \neg \backprime f.\texttt{after}(t))$

Implementation Files

`condition.h`, `condition.cc`

Auxiliary Definitions

IsChange : `Change` $\times$ `Condition` $\times$ **integer** $\rightarrow$ **Boolean**
IsChange(c, f, i)

$\stackrel{\text{df}}{=}$

$\text{p}_\text{T} : H_1$ $\text{r}_\text{T} : G$ Normal	
$c =$ `AtT`	f.hist$[i]$.val
$c =$ `AtF`	$\neg f$.hist$[i]$.val

$\text{YesTimes} : \texttt{Condition} \times \texttt{Timestamp} \times \texttt{Timestamp} \rightarrow \text{ set of } (b, e : \texttt{Timestamp})$

$\text{YesTimes}(f, t_1, t_2)$

$$\stackrel{\text{df}}{=} \left\{ (b, e) \left| \begin{array}{l} f.\mathsf{hist}[i].\mathsf{val} \wedge \\ \left(\begin{array}{l} (b = f.\mathsf{hist}[i].\mathsf{time} \wedge t_1 < b) \ \vee \\ (b = t_1 \wedge f.\mathsf{hist}[i].\mathsf{time} \leq t_1) \end{array} \right) \ \wedge \\ \left(\begin{array}{l} (e = f.\mathsf{hist}[i+1].\mathsf{time} \wedge e < t_2) \ \vee \\ (e = t_2 \wedge f.\mathsf{hist}[i+1].\mathsf{time} \geq t_2) \end{array} \right) \end{array} \right. \right\}$$

C.2.4 Mode Class

Exported Types

```
template <class Spec> class ModeClass
enum ModeChange { Enter, Leave }
```

Imported Types

Type	Defined in
`class Timefunc`	`timefunc.h`
`class Timestamp`	`timestamp.h`

Member Types

`Spec::mode M`

Member Functions

Name	Value Type	Arg. 1 Type	Arg. 2 Type	Arg. 3 Type
`ModeClass`	`ModeClass&`	`M`		
`ModeClass`	`ModeClass&`			
`init`	`M`			
`update`	`M`			
`operator ()`	`M`			
`operator int()`	`int`			
`Since`	`double`	`ModeChange`	`M`	`Timestamp&`
`Drtn`	`double`	`M`	`Timestamp&`	

Representation

$$\textbf{Spec} \stackrel{\text{df}}{=} \begin{pmatrix} \text{init} :\rightarrow \texttt{M}; \\ \text{trans} : \texttt{M} \rightarrow \texttt{M}; \\ \text{delay} : \texttt{M} \times \texttt{M} \rightarrow \texttt{double}; \\ \text{isSysMode} : \texttt{M} \rightarrow \textbf{Boolean}; \end{pmatrix}$$

$R \stackrel{\text{df}}{=} (\mathsf{cur} : \texttt{Timefunc <M>}; \mathsf{spec} : \textbf{Spec})$

Behaviour

`ModeClass` $<S>$ (`M` m)

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	*true*
R.cur′	‘cur.`addPoint`$(0.0, m)$
R.spec′	S
value	R'

`ModeClass` $<S>$ ()

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	*true*
R.cur′	empty list
R.spec′	S
value	R'

R.`init`()

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	*true*
R.cur′	‘R.cur.`addPoint`$(0.0,$ spec.`init`$())$
R.spec′	‘R.spec
value	R.cur′()

R.`update`()

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	`m_t` $\in$ *domain*(R.spec.`trans`(R.cur()))	
	true	*false*
R.cur′	‘R.cur.`addPoint`$(0.0, R$.spec.`trans`$(R$.cur$()))$	‘R.cur
R.spec′	‘R.spec	
value	R.cur′()	

$R()$

$\stackrel{\text{df}}{=}$

$p_T : H_2$ $r_T : H_1 = G$ Vector	$`R.\mathsf{spec.isSysMode}(`R.\mathsf{cur}())$	
	true	*false*
value	$`R.\mathsf{cur}()$	$`R.\mathsf{cur.hist}[\lvert `R.\mathsf{cur.hist}\rvert - 2].\mathsf{val}$

$\wedge R' = `R$

$\texttt{int}(R)$

$\stackrel{\text{df}}{=} \mathsf{value} = \texttt{int}(`R.\mathsf{cur}()) \wedge R' = `R$

$R.\texttt{Since}(\texttt{ModeChange}\ c, \texttt{M}\ m, \texttt{Timestamp}\ t)$

$\stackrel{\text{df}}{=}$

$p_T : H_1$ $r_T : H_2 = G$ Vector	value
$\exists i, \mathrm{IsMdCh}(c, `R.\mathsf{cur}, m, i)$	$t - `R.\mathsf{cur.hist}[i].\mathsf{time}$ $\mid i = max(\{i \mid \mathrm{IsMdCh}(c, `R.\mathsf{cur}, m, i)\})$
$\neg\exists i, \mathrm{IsMdCh}(c, `R.\mathsf{cur}, m, i)$	0

$\wedge R' = `R$

$R.\texttt{Drtn}(\texttt{M}\ m, \texttt{Timestamp}\ t)$

$\stackrel{\text{df}}{=}$

$p_T : H_1$ $r_T : H_2 = G$ Vector	value
$`R.\mathsf{cur}() = m$	$t - `R.\mathsf{cur.hist}[\lvert `R.\mathsf{cur.hist}\rvert - 1].\mathsf{time}$
$`R.\mathsf{cur}() \neq m$	0

$\wedge R' = `R$

Implementation Files

`modeclass.h`

Auxiliary Definitions

IsMdCh : `ModeChange` $\times$ `Timefunc` <`M`> $\times$ `M` $\times$ **integer** $\rightarrow$ **Boolean**

IsMdCh(c, f, m, i)

$$\stackrel{\text{df}}{=} \left[\begin{array}{l} \mathrm{p_T} : H_1 \\ \mathrm{r_T} : G \\ \text{Normal} \end{array}\right]$$

$c =$ `Enter`	$f.\mathsf{hist}[i].\mathsf{val} = m$
$c =$ `Leave`	$f.\mathsf{hist}[i-1].\mathsf{val} = m \wedge f.\mathsf{hist}[i].\mathsf{val} \neq m$

Bibliography

[1] M. Abadi and L. Lamport, "Composing Specifications," in *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), no. 430 in Lecture Notes in Computer Science, (Berlin, Germany), pp. 1–41, Springer, June 1989. Proc. REX Workshop.

[2] R. F. Abraham, "Evaluating Generalized Tabular Expressions in Software Documentation," M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, Feb. 1997. Also printed as CRL Report No. 346, Telecommunications Research Institute of Ontario (TRIO).

[3] J.-R. Abrial, "Steam-Boiler Control Specification Problem," in Abrial *et al.* [4], pp. 500–509.

[4] J.-R. Abrial, E. E. Borger, and H. Langmaack, eds., *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control.* No. 1165 in Lecture Notes in Computer Science, Springer-Verlag, 1996.

[5] B. Alpern and F. B. Schneider, "Defining Liveness," *Information Processing Letters*, vol. 21, pp. 181–185, Oct. 1985.

[6] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore, "Software Requirements for the A-7E Aircraft," Tech. Rep. NRL/FR/5546-92-9194, Naval Research Lab., Washington DC, 1992.

[7] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho, "Hybrid Automata: an Algorithmic Approach to the Specification and Verification of Hybrid Systems," in *Hybrid Systems I*, no. 736 in Lecture Notes in Computer Science, pp. 209–229, Springer-Verlag, 1993.

[8] R. Alur and T. A. Henzinger, "Logics and Models of Real Time: A Survey," in *Proc. Real-Time: Theory in Practice (REX)* (J. W. de Bakker, C. Huizing,

W. P. de Roever, and G. Rozenberg, eds.), no. 600 in Lecture Notes in Computer Science, pp. 74–106, Berlin, Germany: Springer, June 1992.

[9] J. Armstrong and L. Barroca, "Specification and Verification of Reactive System Behaviour: The Railroad Crossing Example," *Real-Time Systems*, vol. 10, pp. 143–178, 1996.

[10] J. M. Atlee and A. Malton, "The Non-simultaneity of SCR Events," in SCR '96 [93].

[11] J. M. Atlee and J. Gannon, "State-Based Model Checking of Event-Driven System Requirements," *IEEE Trans. Software Engineering*, vol. 19, no. 1, pp. 24–40, Jan. 1993.

[12] 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.

[13] R. Bharadwaj and C. L. Heitmeyer, "Applying the SCR Requirements Specification Method to Practical systems: A Case Study," in *Proc. Software Engineering Workshop*, NASA Goddard Space Flight Center, 1996.

[14] G. B. Bochmann, R. Dssouli, and J. R. Zhao, "Trace Analysis for Conformance and Arbitration Testing," *IEEE Trans. Software Engineering*, vol. 15, no. 11, pp. 1347–1355, Nov. 1989.

[15] G. Booch, *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[16] 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.

[17] A. Coen-Porisini, C. Ghezzi, and R. A. Kemmerer, "Specificaiton of Realtime Systems Using ASTRAL," *IEEE Trans. Software Engineering*, vol. 23, no. 9, pp. 572–598, Sept. 1997.

[18] *Proc. Conf. Computer Assurance (COMPASS)*, (Gaithersburg, MD), National Institute of Standards and Technology, June 1995.

[19] A. M. Davis, *Software Requirements-Objects, Functions, and States.* Prentice-Hall, 1993.

[20] T. DeMarco, *Structured Analysis and System Specification*. Prentice Hall, 1979.

[21] 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.

[22] L. K. Dillon, G. Kutty, P. M. Melliar-Smith, L. E. Moser, and Y. S. Ramakrishna, "A Graphical Interval Logic for Specifying Concurrent Systems," *ACM Trans. Software Eng. and Methodology*, vol. 3, no. 2, pp. 131–165, Apr. 1994.

[23] 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.

[24] 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.

[25] P. Du Bois, *The Albert II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis*. PhD thesis, Computer Science Department, University of Namur, Belgium, Sept. 1995.

[26] P. Du Bois, *The Albert II Reference Manual: Language Constructs and Informal Semantics*, 2.0 ed., July 1997.

[27] P. Du Bois, E. Dubois, and J.-M. Zeippen, "On the Use of a Formal RE Language: The Generalized Railroad Crossing Problem," in RE '97 [86], pp. 128–139.

[28] E. A. Emerson, "Temporal and Modal Logic," in van Leeuwen [101], pp. 995–1072.

[29] S. R. Faulk, *State Determination in Hard-Embedded Systems*. PhD thesis, Dept. of Computer Science, University of North Carolina at Chapel Hill, June 1989.

[30] S. R. Faulk, "Specifying Concurrent Events in SCR Requirements," in SCR '96 [93].

[31] S. R. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr., "The CoRE Method for Real-Time Requirements," *IEEE Software*, vol. 9, no. 6, pp. 22–33, Sept. 1992.

[32] 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.

[33] C. Fidge, "Fundamentals of Distributed System Observation," *IEEE Software*, vol. 13, no. 6, pp. 77–83, Nov. 1996.

[34] S. J. Greenspan, A. Borgida, and J. Mylopoulos, "A Requirements Modeling Language," *Information Systems*, vol. 11, no. 1, pp. 9–23, 1986.

[35] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[36] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, "On the Formal Semantics of Statecharts," in *Proc. IEEE Symp. Logic in Computer Science*, (Washington, D.C., USA), pp. 54–64, IEEE Computer Society Press, 1987.

[37] M. P. E. Heimdahl and N. G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Trans. Software Engineering*, vol. 22, no. 6, pp. 363–377, June 1996. reprinted from `ICSE96`.

[38] C. L. Heitmeyer, "Requirements Specifications for Hybrid Systems," in *Hybrid systems III: verification and control* (R. Alur, T. A. Henzinger, and E. D. Sontag, eds.), no. 1066 in Lecture Notes in Computer Science, (New York, NY, USA), pp. 304–314, Springer-Verlag, 1996.

[39] C. L. Heitmeyer, A. Bull, C. Gasarch, and B. G. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," in COMPASS '95 [18], pp. 109–122.

[40] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "A Benchmark for Comparing Different Approaches for Specifying and Verifying Real-time Systems," in *Proc. Int'l Workshop Real-Time Operating Systems and Software*, May 1993.

[41] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 3, pp. 231–261, April-June 1996.

[42] C. L. Heitmeyer and N. Lynch, "The Generalized Railroad Crossing: A Case Study in Fromal Verificaiton of Real-Time Systems," in *Proc. IEEE Real-Time Systems Symposium*, Dec. 1994.

[43] C. L. Heitmeyer and D. Mandrioli, eds., *Formal Methods for Real-Time Computing*. No. 5 in Trends in Software, John Wiley and Sons, 1996.

[44] C. L. Heitmeyer and J. McLean, "Abstract Requirements Specification: A New Approach and Its Application," *IEEE Trans. Software Engineering*, vol. 9, no. 5, pp. 580–589, Sept. 1983.

[45] 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.

[46] 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.

[47] C. A. R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[48] D. N. Hoover and Z. Chen, "Tablewise, a Decision Table Tool," in COMPASS '95 [18], pp. 97–108.

[49] International Standards Organization, *LOTOS: A Formal Description Technique*, 1989. ISO Standard IS8807.

[50] International Telecommunication Union, Geneva, *Specification and Description Language, SDL*, 1992. Recommendation Z.100.

[51] F. Jahanian and A. K. Mok, "Modechart: A Specification Language for Real-Time Systems," *IEEE Trans. Software Engineering*, vol. 20, no. 12, pp. 933–947, Dec. 1994.

[52] F. Jahanian and A. K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Software Engineering*, vol. SE-12, no. 9, pp. 890–904, Sept. 1986.

[53] 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.

[54] R. Janicki, "On a Formal Semantics of Tabular Expressions," CRL Report 355, Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada, Oct. 1997.

[55] R. Janicki and P. E. Lauer, *Specification and Analysis of Concurrent Systems : the COSY Approach.* No. 26 in EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[56] R. Janicki, D. L. Parnas, and J. Zucker, "Tabular Representations in Relational Documents," in *Relational Methods in Computer Science—Advances in Computing Science* (C. Brink, W. Kahl, and G. Schmidt, eds.), pp. 184–196, New York: Springer Wien, 1997.

[57] J. Kirby, Jr., "Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System," Tech. Rep. TR-87-07, Wang Institute of Graduate Studies, July 1987.

[58] R. Koymans and W. P. de Roever, *Examples of a Realtime Temporal Logic Specification.* No. 207 in Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1985.

[59] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Trans. Software Engineering*, vol. 20, no. 9, Sept. 1994.

[60] N. Lynch and F. Vaandrager, "Forward and Backward Simulations-Part II: Timing-based Systems," *Information and Computation*, vol. 128, no. 1, pp. 1–25, July 1996.

[61] D. Mandrioli, A. Morzenti, M. Pezzè, P. S. Pietro, and S. Silva, "A Petri Net and Logic Approach to the Specification and Verification of Real-Time Systems," in Heitmeyer and Mandrioli [43], ch. 6, pp. 135–166.

[62] 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.

[63] 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.

[64] J. McLean, "A General Theory of Composition for a Class of 'Possibilistic' Properties," *IEEE Trans. Software Engineering*, vol. 22, no. 1, pp. 53–67, Jan. 1996.

[65] S. Meyers, *Effective C++—50 Ways to Improve Your Programs and Designs.* Addison Wesley, second ed., 1998.

[66] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

[67] R. Milner, "Operational and Algebraic Semantics of Concurrent Processes," in van Leeuwen [101], pp. 1202–1242.

[68] A. K. Mok and G. Liu, "Efficient Run-Time Monitoring of Timing Constraints," in RTAS '97 [90].

[69] F. Moller and C. Tofts, "A Temporal Calculus of Communicating Systems," in *Proc. Int'l Conf. Concurrency Theory (CONCUR)* (J. C. M. Baeten and J. W. Klop, eds.), no. 458 in Lecture Notes in Computer Science, pp. 401–415, Springer-Verlag, 1990.

[70] T. O. O'Malley, D. J. Richardson, and L. K. Dillon, "Efficient Specification-Based Test Oracles for Critical Systems," in *Proc. California Software Symp.*, Apr. 1996.

[71] T. Ostrand, ed., *International Symposium on Software Testing and Analysis (ISSTA)*, Special issue of ACM SIGSOFT Software Engineering Notes, Aug. 1994.

[72] J. S. Ostroff, *Temporal Logic for Real-Time Systems.* John Wiley & Sons Inc., 1989.

[73] D. L. Parnas and D. K. Peters, "An Easily Extensible Toolset for Tabular Mathematical Expressions," in *Proc. Fifth Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems*, no. 1579 in Lecture Notes in Computer Science, pp. 345–359, Springer-Verlag, Mar. 1999.

[74] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications ACM*, pp. 1053–1058, Dec. 1972.

[75] D. L. Parnas, "Tabular Representation of Relations," CRL Report 260, Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada, Nov. 1992.

[76] D. L. Parnas, "Predicate Logic for Software Engineering," *IEEE Trans. Software Engineering*, vol. 19, no. 9, pp. 856–862, Sept. 1993.

[77] D. L. Parnas and J. Madey, "Functional Documentation for Computer Systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, Oct. 1995.

[78] 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.

[79] 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.

[80] D. K. Peters and D. L. Parnas, "Generating a Test Oracle from Program Documentation—Work in Progress," in Ostrand [71], pp. 58–65.

[81] D. K. Peters and M. von Mohrenschildt, "Course handout for computer engineering 3VA3." URL `http://ece.eng.mcmaster.ca/faculty/mohrens/robot/robot_rs.html`, Sept. 1997.

[82] A. Pnueli and M. Shalev, "What is in a Step: On the Semantics of Statecharts," in *Proc. Int'l Conf. Theoretical Aspects of Computer Software (TACS)* (T. Ito and A. R. Meyer, eds.), no. 526 in Lecture Notes in Computer Science, pp. 244–265, Springer-Verlag, Sept. 1991.

[83] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing Principles, Algorithms and Applications*. Maxwell Macmillan, second ed., 1992.

[84] Rational Software Inc., *et al*, *UML Notation Guide*, version 1.1 ed., Sept. 1997.

[85] 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.

[86] *International Symposium on Requirements Engineering (RE '97)*, IEEE Computer Society Press, Jan. 1997.

[87] G. M. Reed and A. W. Roscoe, "A Timed Model for Communicating Sequential Processes," *Theoretical Computer Science*, vol. 58, no. 6, pp. 249–261, July 1988.

[88] D. J. Richardson, "TAOS: Testing with Analysis and Oracle Support," in Ostrand [71].

[89] 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.

[90] *Real-Time Technology and Applications Symposium*, June 1997.

[91] T. Savor and R. E. Seviora, "An Approach to Automatic Detection of Software Failures in Real-Time Systems," in RTAS '97 [90].

[92] U. Schmid, "Monitoring Distributed Real-Time Systems," *Real-Time Systems*, vol. 7, no. 1, pp. 33–56, July 1994.

[93] *International Software Cost Reduction Workshop*, (Ottawa, Ontario), Feb. 1996.

[94] Silicon Graphics Computer Systems, Inc., `http://www.sgi.com/Technology/STL/`, *Standard Template Library Programmer's Guide*, 1999.

[95] 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, Oct. 1996.

[96] Software Engineering Research Group, "Table Tool System Developer's Guide," CRL Report 339, Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada, Jan. 1997.

[97] A. Sowmya and S. Ramesh, "Extending Statecharts with Temporal Logic," *IEEE Trans. Software Engineering*, vol. 24, no. 3, pp. 216–231, Mar. 1998.

[98] B. Stroustrup, *The C++ Programming Language*. Addison Wesley, third ed., 1997.

[99] 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.

[100] J. J. Tsai and S. J. Yang, eds., *Monitoring and Debugging of Distributed Real-Time Systems*. IEEE Computer Society Press, 1995.

[101] J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, vol. B. Elsevier Science, 1990.

[102] 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).

[103] 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.

[104] M. von Mohrenschildt and D. K. Peters, "The Draw-Bot: A Project for Teaching Software Engineering," in *Proc. Frontiers in Education Conf. (FIE '98)*, pp. 1022–1027, American Society for Engineering Education, Nov. 1998.

[105] G. H. Weiss, R. Hohendorf, A. Wassyng, B.Quigley, and M. R. Borsch, "Verification of the Shutdown System Software At the Darlington Nuclear Generating Station," in *Int'l Conf. Control & Instrumentation in Nuclear Installations*, no. 4.3, (Glasgow, United Kingdom), Institution of Nuclear Engineers, May 1990.

[106] E. J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[107] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Trans. Software Engineering*, vol. SE-8, no. 3, pp. 250–269, May 1982.

[108] 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.