ABSTRACT

SIMULATION AND MATHEMATICAL NOTATION OF ALARMS UNIT FOR COMPUTER ASSISTED RESUSCITATION ALGORITHM

by
Swaroop Malangi

The Computer Assisted Resuscitation Algorithm [CARA] is a system that is used to drive a high output infusion pump used for infusing saline into patients suffering from conditions that lead to hypotension. The infusion pump infuses saline at a particular rate into the patient depending on the blood pressure of the patient.

The alarms unit of CARA was simulated for the infusion pump in which the occurrence of alarms depends on the various criteria the infusion pump encounters when saline is being infused into patients. Various criteria may vary from an air bubble in the line to varying high and low blood pressure. Using the alarms finite state machine already provided simulation of the alarms unit was done. The alarms finite state machine was constructed by using the requirements [2] provided by WRAIR [Walter Reed Army Institute of Research].

A mathematical specification was written which relates the English language description of the alarms unit and the alarms finite state machine. The Design Oriented Verification and Evaluation [DOVE] tool [5] was used to prove that the extended finite state machine satisfies the mathematical specification.

The simulation of the alarms unit was done as per the requirements [2] and extended finite state machines were created according to the code of the simulation. Safety properties and linear temporal logic for these safety properties were also written.

SIMULATION AND MATHEMATICAL NOTATION OF ALARMS UNIT FOR COMPUTER ASSISTED RESUSCITATION ALGORITHM

by
Swaroop Malangi

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

January 2004

APPROVAL PAGE

SIMULATION AND MATHEMATICAL NOTATION OF ALARMS UNIT FOR COMPUTER ASSISTED RESUSCITATION ALGORITHM

Swaroop Malangi

[illegible] [illegible]

Dr. Elsa L. Gunter, Thesis Advisor Date
Associate Professor of Computer Science, NJIT

[illegible] [illegible]

Dr. Ali Mili, Committee Member Date
Professor of Computer Science, NJIT

[illegible] [illegible]

Dr. Teunis Ott, Committee Member Date
Professor of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author:	Swaroop Malangi
Degree:	Master of Science
Date:	January 2004

Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2004
- Bachelor of Engineering in Computer Science,
  Bangalore University, Bangalore, India, 2001

Major: Computer Science

To my sister-in-law and brother Parvathi and Girish Malangi,
my friends
and my parents

ACKNOWLEDGEMENT

I owe my deepest gratitude to Dr. Elsa Gunter, who gave me this opportunity to work on this very interesting topic and in turn served as my Thesis advisor. She provided constant help and support by adjusting to my schedule and provided technical help so that I could work on my Thesis remotely. Special thanks are given to Dr. Ali Mili and Dr. Teunis Ott for actively participating in my committee.

I would also like to thank my fellow graduate students and my family for providing me with their help and support.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF TERMS

Manual Mode:	Manual mode is when the pump is plugged in and is infusing the fluid into a patient using the hardware flow setting on the pump.
Auto Control:	Auto Control is when the algorithm is controlling the pump.
Back EMF:	A rotating electric motor also develops an induced voltage; this is called a "back emf". The current through a rotating electric motor is greatly reduced because of its "back emf". The current through a non-rotating electric motor may be great enough to damage the motor.
CARA:	Computer Assisted Resuscitation Algorithm.
WRAIR:	Walter Reed Army Institute of Research.
Impedance:	Impedance is the degree to which an electronic component impedes the flow of current. In this case, the impedance of the infused liquid is checked to assure it is saline and not water.
Occlusion:	Occlusion is the inability of the fluid to flow through the pump.
DOVE:	Design Oriented Verification and Evaluation.
EFSM:	Extended Finite State Machine.
AFSM:	Alarms Finite State Machine.
BP:	Blood Pressure
LTL:	Linear Temporal Logic
A/C:	Alternating Current

CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Background

Walter Reed Army Institute of Research [WRAIR] is making medical equipments that are used during war and other critical situations. One of them is the development of an infusion pump that infuses saline into patients. This infusion pump works in both auto control and manual mode. During both the modes a caregiver or a paramedic is assumed to be present to operate the pump. Computer Assisted Resuscitation Algorithm [CARA] runs the infusion pump when it is in auto control mode. CARA consists of logical units like pump monitor, blood pressure monitor, algorithm and the display/alarm unit. WRAIR has constituted a committee from New Jersey Institute of Technology, University of Pennsylvania and State University of New York at Stony Brook to develop the software to run the infusion pump. The focus of this thesis is to develop the software to run the alarms unit and use verification tools to prove that the software developed and the requirements [2] are consistent. The verification is necessary because of the critical situations in which the infusion pump is used and inconsistency may lead to loss of life.

1.2 Objective

First, to simulate the alarms unit in the pump depending on various criteria the pump encounters when it is plugged in. Simulation was done using the Finite State Machine already provided in the requirements [2].

Secondly, to write a mathematical specification which relates the English language description and the alarms finite state machine. Such a mathematical specification can then be used with model checking tools or theorem proving tools such as DOVE to prove that the Finite State Machine satisfies the mathematical specification.

1.3 About CARA

Computer Assisted Resuscitation Algorithm is a software system used to drive the pump responsible for fluid resuscitation of patients suffering from conditions that lead to hypo-tension. CARA consists of the following components

- CARA [algorithm]
- Pump
- Environment
- Propaq
- Patient
- Interfaces

1.4 Interfaces

Computer Assisted Resuscitation Algorithm communicates with the Environment (caregiver, usually a medic or a nurse) via alarm signals, messages that appear on a display screen and user input via soft buttons. The pump raises alarm signals when it encounters problems during infusion (e.g., air bubbles). CARA further interfaces with the environment by setting the drive voltage for the infusion pump which is pumping saline into the patient and by reading the blood pressure of the patient from a physiological monitoring device.

1.5 Previous Work

Computer Assisted Resuscitation Algorithm system is developed as per the requirements specified by WRAIR [2]. WRAIR has constituted a group to work on this system. The requirements specified have been studied for their formal properties by researchers at several universities including New Jersey Institute of Technology, University of Pennsylvania [8] and SUNY – Stony Brook.

1.6 Methods

The background for the first part of the thesis, namely to simulate alarms unit, is the alarm finite state machine. K. L. Henniger [1] specifies the technique of representing software requirements as finite state machines. A finite state machine can be described as a machine that has a finite number of states, an initial state, one or more final states, grammar for the machine and at least (n-1) edges between states where n is the number of finite states.

The alarms state machine from the formal specification of CARA [8] is as shown in Figure 1.1. PluggedIn is a boolean value and PumpUnplugged is an alarm. ActiveAlarms is the set of alarms that are active. AddAlarms, ClearAlarms, SilenceAlarms, ResetAlarms and DisplayAlarms are all functions to be performed. Initially the pump is not plugged in and when it is plugged in the set of active alarms is empty. After pump is plugged in the functions mentioned above take place. If pump is unplugged all active alarms are displayed.

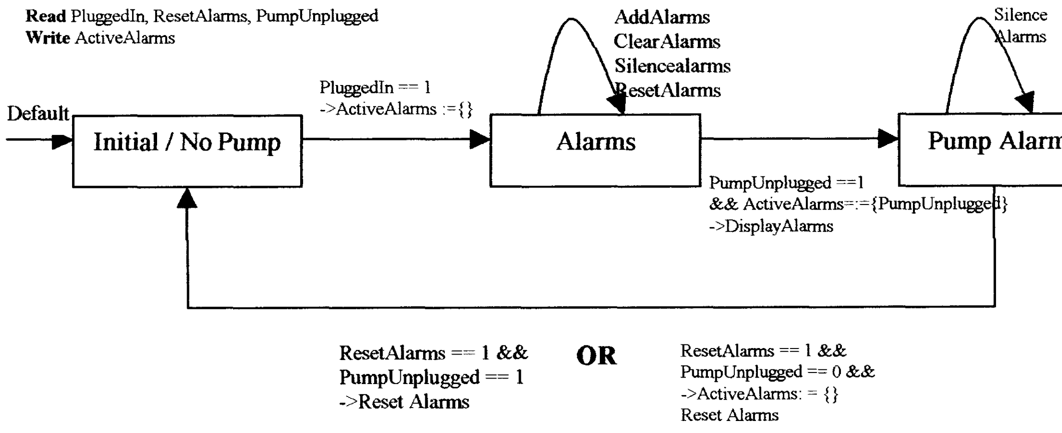

Figure 1.1 Alarms State Machine.

The CARA system might be seen as having four logical components.

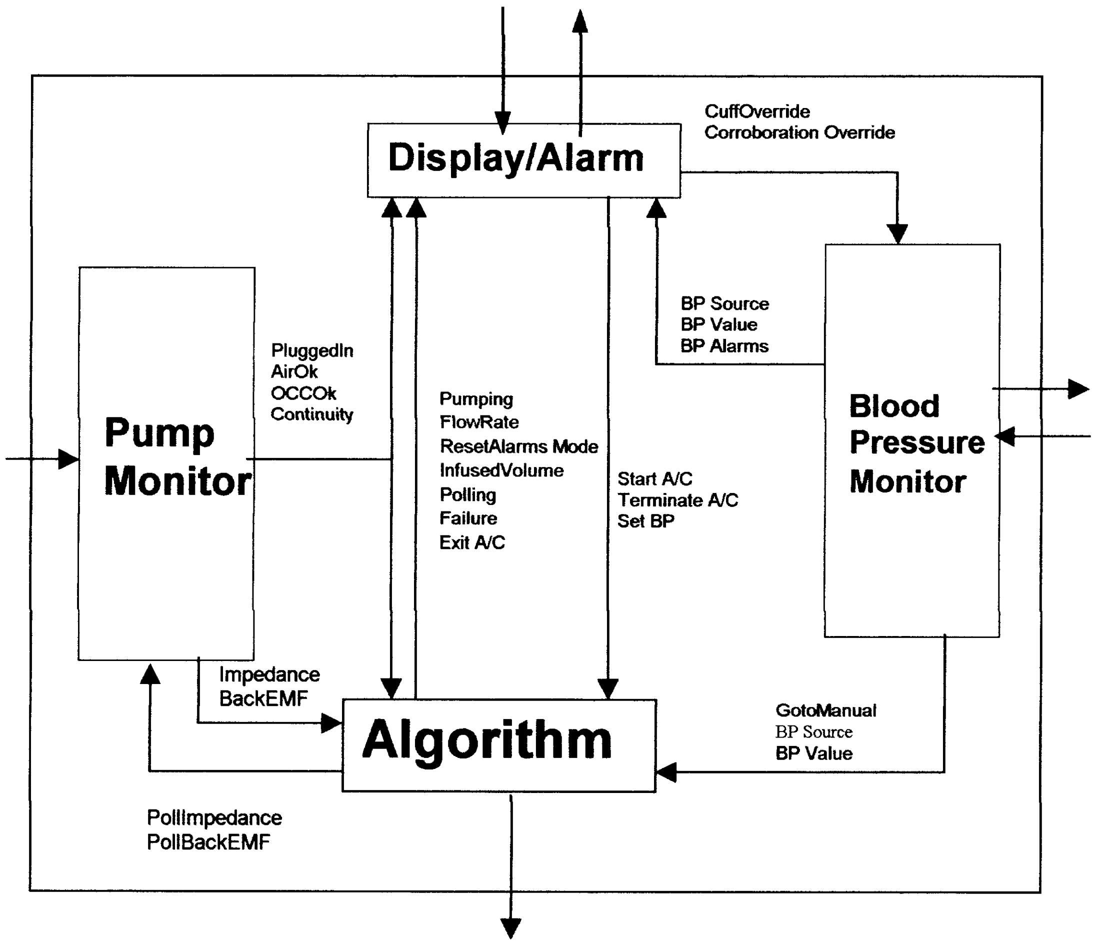

Figure 1.2 Logical components of CARA.

To write the mathematical specification Linear Temporal Logic is used. Temporal logic presents a formal approach to the specification, verification, and development of reactive systems. Reactive systems are (usually) concurrent systems whose role is to maintain an ongoing interaction with their environment, such as real-time process, which in this case is the infusion pump. The reliability and correctness of the infusion pump is a primary and critical condition for their applicability.

CHAPTER 2

DESCRIPTION

2.1 Functional Specification

2.1.1 Functions Performed for Alarms Unit

The purpose of alarms unit is to communicate with the environment. Alarms go off if something undesirable occurs like air bubbles in the pipe etc. Alarms will go off if the following occur. Though it is not the complete list, it gives a general idea.

- Pump Unplugged
- Continuity Fault
- Impedance
- Occlusion
- Falling Patient BP
- Loss of control BP Source
- Failure to reach 60 mmHg in 5 minutes or failure to reach setpoint in 20 minutes
- No BP
- Cuff pressures not available
- Cuff pressure is invalid
- Can not measure BP after 3 minutes
- BP source can not be corroborated
- Loss of non-control BP source

The alarms unit takes care of faults that may occur during infusion. If an error signal arrives from a monitor or algorithm, then an error message is displayed on the screen and an alarm is sounded. The message persists until the fault has been removed and the user resets the alarm.

Alarms may be silenced for a given period of time by pressing a soft "silence alarm" button on the screen. All active alarms are reset if there is a "pump unplugged" alarm. Alarms have priorities and silence times and they are shown in the table below

Table 2.1 Alarms Table

Alarm	**Priority**	**Silence Time**
Pump unplugged (during manual or auto control mode)	1	2
Continuity Fault (in. no Back EMF)	2	5
Impedance	3	2
Occlusion	4	2
Falling Patient BP	5	2
Failure to reach 60 mmHg in % minutes or failure to reach set point in 20 minutes	6	5
Loss of control BP Source	7	-
No BP (polling failure / no data stream)	8	-
Cuff pressures not available	9	-
Cuff pressure is invalid (out of range)	10	-
Can not measure BP after 3 minutes	11	-
BP source can not be corroborated	12	-
Loss of non-control BP source	13	-

2.1.2 Functions Performed in Mathematical Specification

The mathematical specification is based on Linear Temporal Logic. Using Linear Temporal Logic we map the hardware specifications expressed in English into formulas. DOVE is used to model the alarm system and to verify that the model satisfies the specification.

Design Oriented Verification and Evaluation is a tool built to simulate and reason about extended finite state machines (EFSMs). With DOVE it is possible to define systems in a modular and hierarchical fashion. It is also possible to reason by narrowing the proof to just those components that are actually relevant to a desired property.

2.1.3 Limitations and Restrictions

Some of the important limitations and restrictions are, even when the system is in auto-control, it always assumes that there is a caregiver who will be near the system. The algorithm to run needs an M100 (or equivalent) pump. Also we assume the pump is working properly.

Since mathematical specification is written for the part of a big system LTL can only approximate the English Description. When the negation of the mathematical specification is done, the model checker may not recognize the necessary path the English Description specifies. There may be more than one path and model checker may choose the path that is not specified in the English Description.

2.1.4 User Interface Design

The user display consists of a screen that gives the warning messages. As this is a subset of the larger system, the display and interfaces are restricted to user inputs, which simulate the real values possible for the lines. Also the output is made simple so that they just give warning messages, because in the real system there is a separate module that takes care of displaying alarms. Appropriate values are placed in variables that would be shared with the other modules in the total system.

2.1.5 Other User Inputs

The other input is a data file. This file is read by the simulated pump for giving the various states possible for it to work on. The data file may be randomly generated from any source.

2.2 Design Specification

2.2.1 System Data Flow Diagrams

The following diagram shows the different components of the system that relate to the alarms unit. It also shows the data flow between the components. The alarms unit interacts with the Algorithm, Blood Pressure Monitor and the Pump Monitor.

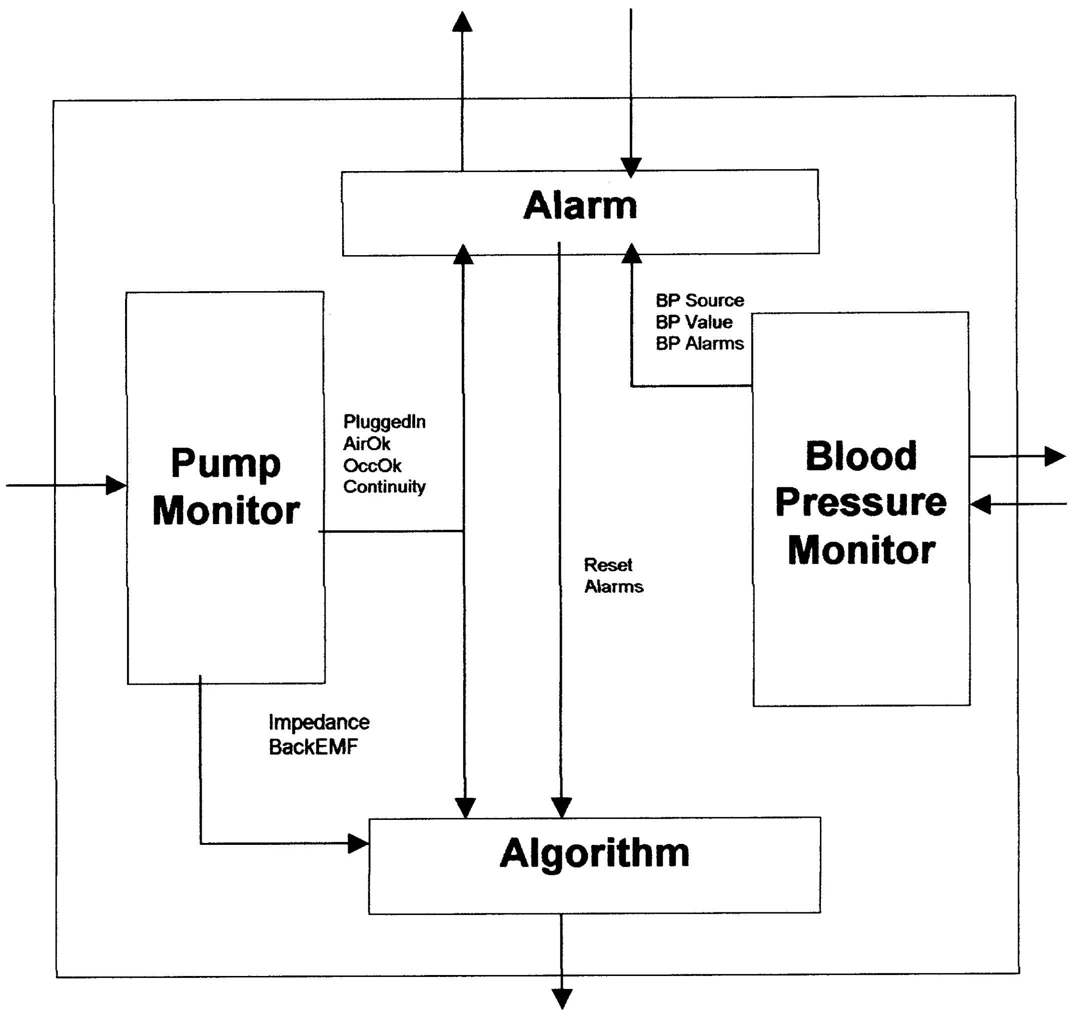

Figure 2.1 Components relating to alarms unit.

2.2.2 Specification of Display/Alarms Component

The alarms unit works in conjunction with the Display Unit. This is how the system communicates with the environment.

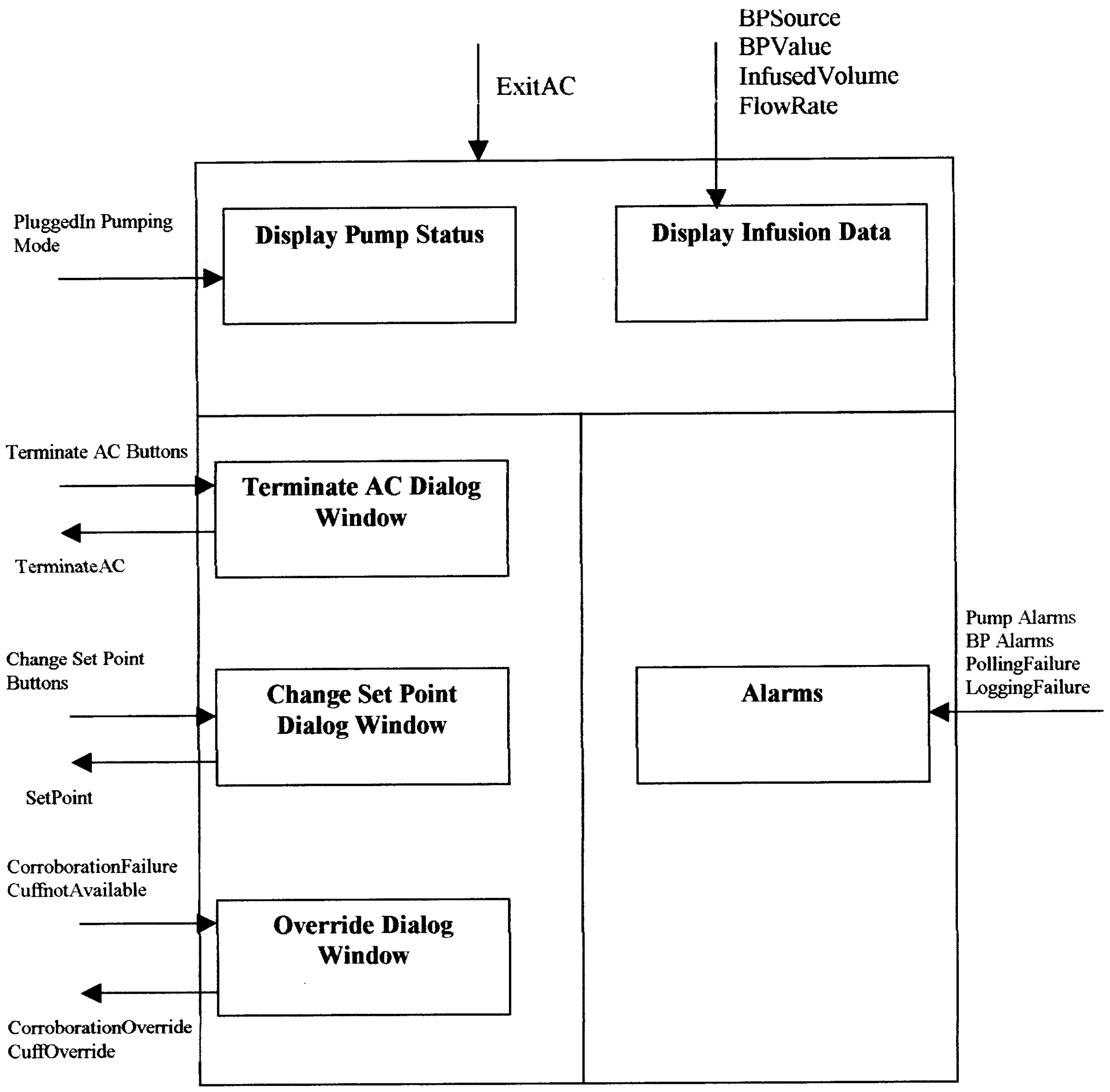

Figure 2.2 Display unit and alarms unit.

2.2.3 System Data Dictionary

For realizing the different alarms, there has to be some well defined variables used. These variables give the state of the system at any point of time. The different variables used are shown in the table.

Table 2.2 Variables Used

Name	Scope (G/L)	Type	Description
Alarm	Local to AFSM	Record	Below
Active alarms	Local to AFSM	Set of alarm	Set of active alarms
Polled Alarms	Local to AFSM	Set of Alarm	Set of polling alarms
All Alarms	Local to AFSM		Set of Boolean alarms
New Alarm	Local to AFSM		Metavariable for alarms
Fixed alarm	Local to AFSM		Metavariable for alarms
Silence Alarm	Global	boolean	User silences audible alarm
SilenceButton	Local to AFSM	boolean	Silence button enabled
ResetButton	Local to AFSM	boolean	Reset button enabled
AudibleAlarm	Local to AFSM	boolean	Sets audible alarm false
ResetAlarms	Global	boolean	User rests active alarms

The variable Alarm is of type record, with the following items: alarm name, alarm priority, alarm message, audible level and temporary silence time.

2.2.4 Procedures Used

The procedures are written in pseudo code. There are two expressions, AddAlarms and ClearAlarms, which capture the fact that new alarms may become active and some active alarms may be fixed.

$$AddAlarms = \quad \bigvee_{NewAlarm \in AllAlarms} NewAlarm == 1 \rightarrow AddAlarm \ \ (NewAlarm)$$

$$ClearAlarms = \quad \bigvee_{FixedAlarm \in AllAlarms} FixedAlarm == 0 \rightarrow ClearAlarm \ \ (FixedAlarm)$$

The procedures that implement each individual action are first described informally.

- Procedure *Add Alarms*
  Adds a new alarm that has become active into the set of active alarms. Activates the silence button and the reset button.

- Procedure *Clear Alarms*
  Will remove any fixed alarm from the set of active alarms.

- Procedure *Display Alarms*
  Given a set of active alarms, it will (re) display all currently active alarms in the order determined by their priorities.

- Procedure *Sound Alarms*
  Will sound audible alarm, according to alarm level. A level 1 alarm will be continuous and requires user intervention to stop. A level 2 alarm is sounded 5 times and requires no user intervention to stop.

- Procedure *Silence Alarms*
  Will silence active alarms for a period of time.

- Procedure *Reset Alarms*
  Will reset active alarms, clear the screen and stop the audible part. However, if the conditions have not been actually fixed, this will not change the value of active alarms and the alarms will be redisplayed and sounded again.

The pseudo code for the above procedures mentioned is found in Appendix A.

2.2.5 Mathematical Specification

As mentioned above DOVE (Design Oriented Verification and Evaluation) will be used to verify the model. It contains a graphical user interface to construct extended finite state machines. Extended finite state machines are finite state machines that have the ability to model the manipulation of variables. Safety properties and linear temporal logic are written for the system from the requirements [2]. Safety properties and linear temporal logic is explained in detail in chapter 4. DOVE can be used to verify the model by checking the linear temporal logic against the extended finite state machines.

Linear Temporal Logic has non-temporal logical connectives And, Or, Not, Implies and temporal connectives Next, Until, WeakUntil, Always, and Eventually. These will be used to give the mathematical specification of the Airline Ok Monitor as an example.

The connectives And, Or, Not and Implies are basic logical connectives. The connective Next describes the next step in the process. The term A Next B means that B immediately holds after A.

The connective Until means one process holds till the other is done. The term A Until B means either B holds at the beginning or A does and A keeps holding until B finally holds and B must hold at some point.

The connective WeakUntil means one process holds till the other is done. This is different from Until because the second process need not hold at some point like Until. The term A WeakUntil B means either B holds at the beginning or A holds at the beginning and A keeps on holding. Here B need not hold at some point.

These logical connectives are used to show how an EFSM works.

Consider the Air Line Monitor,

The english language description of this in the CARA Requirements is as follows,

8 The CARA will Monitor the Air OK Line whenever the pump is plugged in.

8.1 If the Air OK signal remains low for 10 seconds.

8.1.1 An appropriate error message should be issued.

8.1.2 A level 1 Alarm should be issued

Mathematical statements for the Air Ok Alarm is as follows,

```
Always(AirOk >limit and
               Next (t = 0 and
                      (AirOk ≤ limit until t = 10) )
implies
( ( Eventually
        Error_Message = "Air Ok Fault")
and
(Eventually
        Alarm_Level = 1) ) )
```

If the AirOk variable becomes false, then a timer is started with t = 0. If the timer reaches ten seconds, then the AirOk alarm will go off. It will remain so till the alarm AirOk variable becomes true and AirOk alarm also becomes false.

To verify if the mathematical specification is consistent with the English description, mathematical specification is negated and compared with the description. Consistency between the extended finite state machines and the actual alarms unit simulator is checked. Necessary changes are made to the simulator and the mathematical specification if there is any inconsistency.

CHAPTER 3

SIMULATION OF ALARMS UNIT OF CARA

3.1 Introduction

Simulation was done using the Java programming language. The entire code of the simulation is shown in Appendix B. This chapter gives a description of definitions and declarations used in the program. It also gives an overview of the functions individually. The working of the program is also described in detail.

3.2 Definitions

Function Process: This function is called when the alarm becomes active. Depending on the alarm, it starts an active thread and, depending on the user input, it calls the respective function, which might be Silencealarms or Soundalarams. It checks if there is an alarm that is already active. If there is one, it compares the priority of already existing alarm and the alarm that recently occurred. If the recently occurred alarm is of higher priority, it creates a new thread for the recently occurred alarm and suspends the active alarm. If the recently occurred alarm is of lower priority than the already existing alarm, then it just adds the alarms to the list of active alarms. If no alarm exists, it creates a new thread for the alarm and adds it to the list of active alarms. If an alarm that is suspended gets control after another alarm is cleared, then its thread is resumed. If the alarm condition becomes active (if the function *polling* returns a change in Boolean value) for an alarm that is already suspended, its thread is not suspended but killed.

Function polling: This function checks all the Boolean values of all the alarms starting from the highest priority. It returns the value of the alarm that is of highest priority and does not continue to check for the rest of the lower priority alarms. If no alarm condition exists it returns a very large number greater than fifteen.

Function addalarm: This function adds any new alarm that has become active. If the alarm being added is a suspended alarm then the suspended thread for that alarm is killed.

Function clearalarm: This function checks if the fixed alarm exists in the active alarms list. Removes the fixed alarm from the set of active alarms. It checks if the alarm is a polled alarm, and if it is a polled alarm, it will check the value of these alarms, compares them with the acceptable value and then finally clears the alarm. This part is commented out from the program because the value needed to check the existence of the polled alarm is outside the scope of the program. After clearing the alarm, it goes through the list of active alarms checking for alarm conditions and calls function p*rocess* when an alarm condition is detected. If a suspended alarm was cleared then it kills the thread of the suspended alarm.

Function displayalarms: This function when the pump is unplugged, will display all currently active alarms in the order determined by their priorities. If the pump is plugged in , then it will display the current active alarm.

Function soundalarms: This function sounds an audible sound, according to alarm level. The level of the alarm depends on each individual alarm as listed in the alarms table.

Function silencealarms: This function silences the alarm when caregiver chooses to silence it. It calls the R*eminder* in which a timer is set for the silence time of that particular alarm.

Class Reminder: This class belongs to the timer class. Depending on the silence time of the alarm sets a timer. If the alarm timer expires, then it displays a visual message respective of the alarm. Once the alarm condition does not exist anymore, it clears the timer.

Class Pollingschedule: This class belongs to the timer class. It gets the value from function *polling*, which in turn returns the alarm with highest priority.

Class Test: This class belongs to the timer class. It changes the Boolean values of the alarms at regular intervals. A random number from one to fifteen is generated. Boolean value of that random number is changed using random Boolean values. Also three is added and subtracted to this random number and the Boolean value of these numbers are also changed using random Boolean values to make sure more than one alarm value is changed each time.

Class NewThread: This class belongs to the thread class. New thread is created when a new alarm occurs. The thread is alive while the alarm condition exists. Once the alarm condition does not exist anymore it calls the function *clearalarms* in which the alarm is eventually cleared.

3.3 Declarations

alarmsarray is the array of Boolean values of all the alarms which is used by class *test.*

alarmslist is the vector with the list of currently active alarms.

alarmssuspend is the array of Boolean values of all the suspended alarms.

siltime is the array of integer values that contains silence times of all the alarms.

3.4 Working

The program for simulating the alarms unit does the following,

- The program uses *test* scheduler to change the Boolean values of all the alarm conditions at regular intervals. It uses a random number generator to choose the alarm whose condition is to be changed and uses random Boolean generator to change its value.

- The program uses *pollingschedule* to check for alarm conditions at regular intervals. It gets a value from the *polling* function and that in turn checks the Boolean values of all the alarms and returns the alarm number of the highest priority.

- The priorities of alarms are as listed in the alarm table.

- The program issues a visual message when an alarm becomes active.

- The program gives caregiver option for silencing or sounding or acknowledge/resetting the alarm. Acknowledge/reset option is only for polled alarms.

- An alarm is added to list of active alarms when alarm condition occurs.

- A thread is created for that alarm when an alarm becomes active. This thread is alive until the alarm condition ceases to exist.
- Each alarm occurrence is logged in to a file as and when it occurs.
- An alarm will be silenced for a period of time specified in the alarm table if the alarm silence button is pushed. It uses *Reminder* as a timer for silencing the alarm.
- The silence times of all the alarms are as listed in the alarms table.
- If an alarm is silenced and an alarm of higher priority becomes active, an option to silence the new alarm is displayed. The thread of the lower priority alarm is suspended and the thread of higher the priority alarm becomes active.
- If an alarm is not silenced, it will be sounded. The level of the alarm will depend on each individual alarm as listed in the alarms table.
- If a higher priority alarm is active and a lower priority alarm also becomes active, then the latter is just added to the active alarms list.
- If all the current alarms are silenced and a new alarm condition occurs, a new message will be displayed for that alarm.
- If an alarm that is suspended becomes active then the thread of that alarm is resumed again. If the alarm condition for that alarm changes it is not resumed.
- The alarms will be automatically reset when the alarm condition ceases to exist.
- When the alarm condition does not exist, the alarm is cleared.
- Control is automatically given to the next priority alarm when an alarm is cleared.
- As the alarm is cleared, its thread is killed and a new thread for the next priority alarm is created if it is not suspended earlier. If the thread was suspended earlier then it will be resumed.
- If an alarm reset button is pressed and a polling device was the source of the alarm, the program will immediately check if the alarm condition exists before clearing the alarm. This is not implemented in the program because the value needed to check the condition of the alarm is outside the scope of the program.
- The program automatically resets all the alarms when the pump is unplugged.
- The program displays the list of active alarms when the pump is unplugged from the highest priority to the lowest priority.

CHAPTER 4

MATHEMATICAL SPECIFICATION

4.1 Extended Finite State Machines

Extended finite state machines are finite state machines that have the ability to model the manipulation of variables. The components of the EFSM as encoded in DOVE [5] are a number of states, one start state, transitions between states, heap variables and input variables. Heap variables are local to the machine and they can be changed in transitions. Input variables cannot be modified in transitions and they change when the environment changes. Each transition can be looked at as an if-then statement. If a condition (*Guard* in DOVE) of the heap and input variables is satisfied, then (*Act* in DOVE) some heap variables may or may not be modified in the action part.

Extended finite state machines were constructed with the help of the DOVE tool. Each EFSM was constructed with respect to the functions used in the implementation. The combination or the product of all EFSMs is not shown because it contains hundreds of states.

Type declarations and data type decalarations used in all the EFSMs are as follows.

- list_alarms is a list of alarms.
- alarm is a data type that can be assigned one of four values, namely Active, NotActive, Suspended and Dormant.
- namesslist is a list of names of alarms that are of data type alarm.
- array is a list of natural numbers.

The descriptions of all the EFSMs are as follows.

Polling EFSM: It starts when the timer in the polling schedule gets timed out, i.e. when it is time to check the states of all the alarms. The EFSM checks each alarm one by one from the highest priority alarm and stops once an alarm condition is found. It starts checking from the alarm of highest priority once again when the time in the polling schedule times out.

The program code for polling EFSM is as follows,

```
public static int polling()
{
//Alarms checking
for(int i=1;i<16;i++)
{
if(globals.alarmsarray[i]) return i;
}
return 99;
}
```

From the above program code we can construct the extended finite state machine.

The machine starts when a variable do_polling is set in pollingschedule EFSM as shown in the GetAlarm transition. It goes to the CheckAlarm state and as shown in the Check transition checks for an alarm condition and if it does not find one it increments the index and checks again. If it finds an alarm condition, it goes to the DonePoll? state and stops. It sets variable so that pollingschedule EFSM can continue further. If it does not find an alarm condition and the value of I reaches 16 it still goes to DonePoll? and returns a large number for the pollingschedule EFSM. The same method is used to for constructing the other machines from the program code.

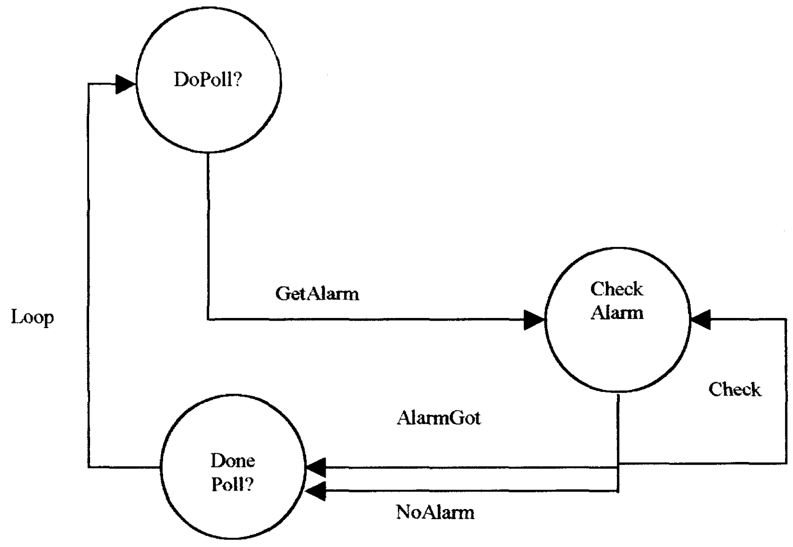

Figure 4.1 Polling EFSM.

Heap Variables used in Polling EFSM

poll_alarm of type natural numbers.

i of type natural numbers.

Input Variables used in Polling EFSM

do_poll of type Boolean.

list_alarms of type alarm.

Transition Definitions for Polling EFSM

transdef "GetAlarm" "
Guard: do_poll=true
Act:i=1;
poll_ready<-- false;"

transdef "Check" "
Guard:list_alarms!i=false and i<16
Act:i=i+1;"

transdef "AlarmGot" "
Guard:list_alarms!i=true and i<16
Act:poll_ready<--true;
poll_alarm<--i;"

trandef "NoAlarm" "
Guard:i=16
Act:poll_ready<--true;
poll_alarm<--#99;"

transdef "Loop" "
Guard poll_ready=false and i < #15
Act:Skip;"

PollingSchedule EFSM: It starts when the time is more than 5 ms after the previous polling. It sets a Boolean value do _poll so that the polling EFSM starts. It waits till the polling EFSM returns an alarm condition or does not return any alarm condition. If there is an alarm condition, it sets the variable handle to true to stat the handle EFSM. It waits till the next schedule and starts up again.

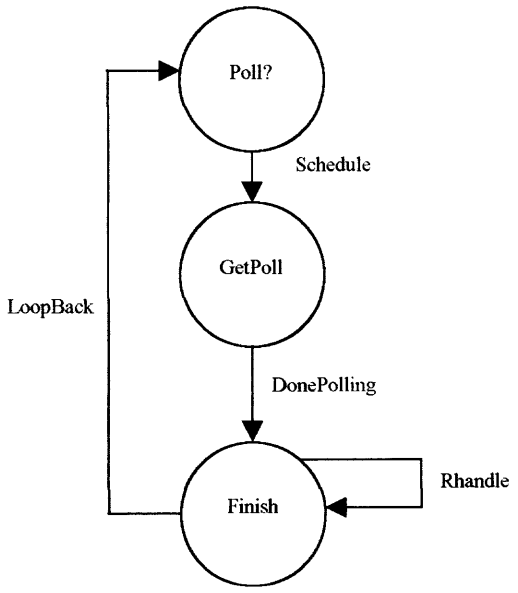

Figure 4.2 Polling Schedule EFSM.

Heap Variables used in Polling Schedule EFSM

curr_alarm of type natural numbers.

do_poll of type Boolean.

handle of type Boolean.

last_time of type natural numbers.

Input Variables used in Polling Schedule EFSM

poll_alarm of type natural numbers.

poll_ready of type Boolean.

resethandle of type Boolean.

timer of type natural numbers.

Transition Definitions for PollingSchedule EFSM

```
transdef "Schedule" "
Guard:Not(timer-last_time<#100)
Act:last_time<-- timer;
do_poll<-- true;

transdef "DonePolling" "
Guard:poll_ready=true
Act:curr_alarm <-- poll_alarm;
do_poll<-- false;
handle<-- true;"

transdef "Rhandle" "
Guard:resethandle=true;
Act:handle<--false;

transdef "LoopBack" "
Guard:do_poll=false;
Act:Skip;
```

Handler EFSM: It starts when an alarm condition occurs, either when pollingschedule EFSM sets the handle variable or clear EFSM sets the nextal variable. As it enters the machine the name and number of the alarms are stored in heap variables alarmname and alarmno. It goes into pump unplugged state when the pump is unplugged. It stays there till the pump is plugged back in. If the alarm occurring is the first alarm, then it goes into the first alarm state. The alarm is added, and silencesound EFSM and thread EFSM are started by setting the silencesound and threadstart variables. It stays there till a new alarm occurs, and if a new alarm occurs then it goes back to the check state.

If it is not the first alarm, then the priority of the current alarm is checked against the priority of the previous alarm in the NotFirst transition. If it is of higher priority, it goes to the HigherPriority state otherwise it goes to the LowerPriority state. In the HigherPriority the alarm is added, silencesound EFSM and thread EFSM are started. In the LowerPriority state the alarm is just added. The machine remains in these states till a new alarm occurs. When the new alarm condition occurs it goes back to the Check state.

If the alarm condition is found after clearing an alarm then it goes to the Resume state after the check state. Here the current alarm is checked if it was suspended before. If it was suspended before, it is resumed. If it was not suspended and is a new alarm, then the alarm is added, and silencesound EFSM and thread EFSM are started. It remains in that state till a new alarm occurs after which it will go to the Check state through the AfterHigher and AfterLower transitions.

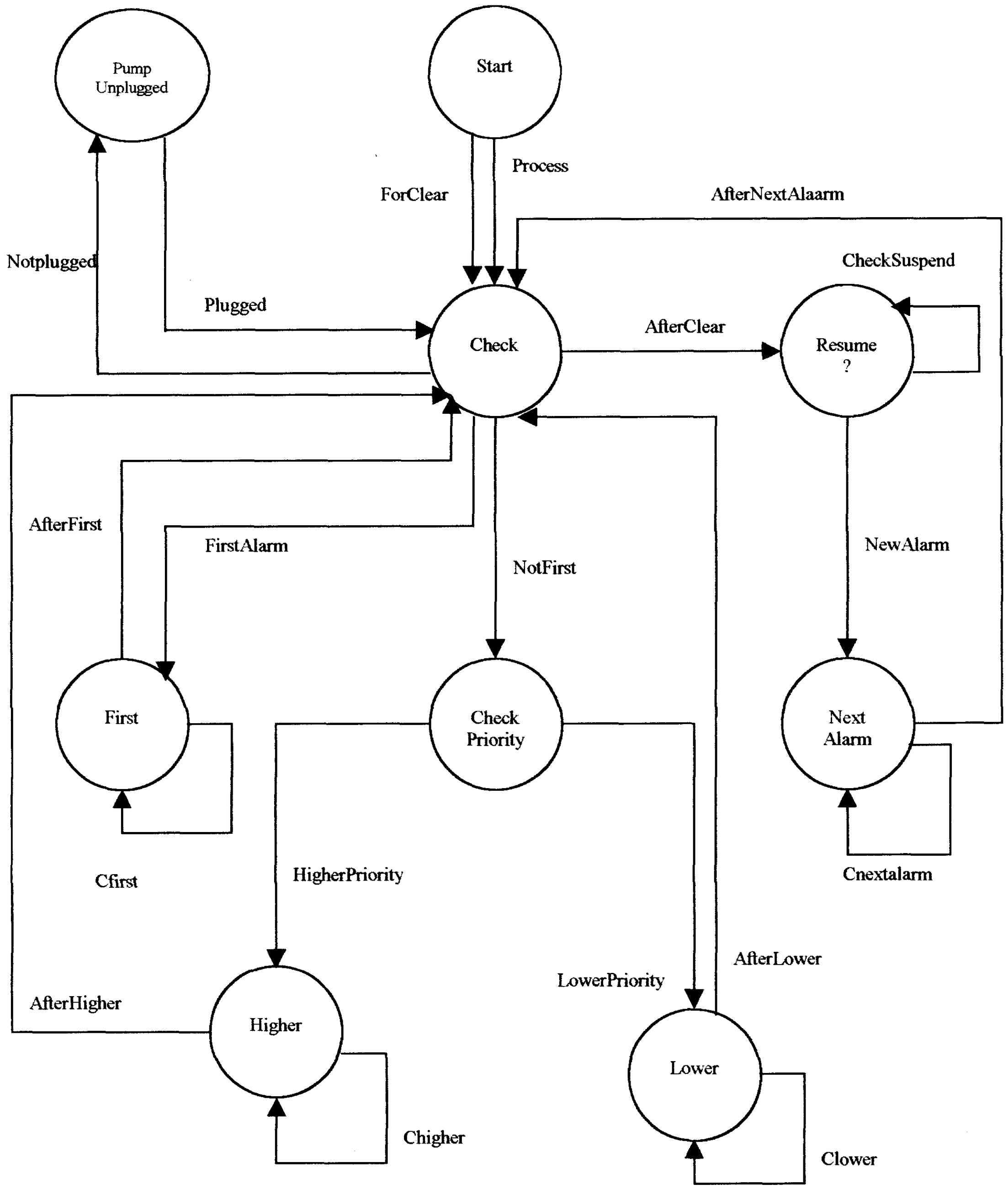

Figure 4.3 Handler EFSM.

Heap Variables used in Handler EFSM

Cafterclear of type Boolean.

Nalarm of type Boolean.

add of type Boolean.

Cafterclear of type Boolean.

alarmname of type alarm.

alarmno of type natural numbers.

alarmonthread of type Boolean.

display of type Boolean.

namess is of type namesslist.

pump_unplugged of type Boolean.

soundsilence of type Boolean.

threadstart of type Boolean.

Input Variables used in Handler EFSM

Cthreadstart of type Boolean.

afterclear of type Boolean.

curr_alarm of type natural numbers.

handle of type Boolean.

nextal of type Boolean.

silencesound of type Boolean.

i of type natural numbers.

Transition Definitions for Handler EFSM

transdef "process" "
Guard:handle=true
Act:alarmno<--curr_alarm;
alarmname<--namess!curr_alarm;
resethandle<--false;"

transdef "ForClear" "
Guard:nextal
Act:alarmno<--i;
alarmname<--namess!i;
Nalarm<--true;
afterclear<--true;

transdef "NotPlugged" "
Guard:alarmno=1
Act:display<--true;
pump_unplugged<--true;

transdef "Plugged" "
Guard:handle And Not(pump_unplugged)
Act:Skip;"

transdef "FirstAlarm" "
Guard:alarmonthread=false
Act:alarmonthread<--true;
add<--true;
threadstart<--true;
soundsilence<--true;
alarmname<--Active;"

transdef "Notfirst" "
Guard:alarmonthread
Act:Skip;"

transdef "AfterClear" "
Guard:afterclear And alarmonthread
Act:Cafterclear<--false;"

transdef "HigherPriority" "
Guard:alarmno<curr_alarm
Act:add<--true;
threadstart<--true;
soundsilence<--true;
alarmname<--Suspended;"

transdef "LowerPriority" "
Guard:curr_alarm<alarmno
add<--true;
alarmname<--Dormant;"
transdef "CheckSuspended" "
Guard:alarmname=Suspended
Act:afterclear<--false;

alarmonthread<--false;
add<--true;
soundsilence<--true;
alarmname<--Active;"

transdef "NewAlarm" "
Guard:alarmname=Active Or alarmname=Dormant
Act:afterclear<--false;
alarmonthread<--false;
add<--true;
transdef "Cfirst" "
Guard:Cthreadstart
Act:threadstart<--false;

transdef "Chigher" "
Guard:Cthreadstart
Act:threadstart<--false;

transdef "Clower" "
Guard:Cthreadstart
Act:threadstart<--false;

transdef "Cnextalarm" "
Guard:Cthreadstart
Act:threadstart<--false;
threadstart<--true;
soundsilence<--true;"

transdef "AfterFirst" "
Guard:handle
Act:Skip;"

transdef "AfterHigher" "
Guard:handle
Act:Skip;"

transdef "AfterLower" "
Guard:handle
Act:Skip;"

transdef "AfterNextAlarm" "
Guard:handle
Act:Skip;"

Thread EFSM: It starts when an alarm condition occurs and the handler EFSM sets input variable threadstart. If the alarm is to be suspended, then it goes to the SuspendAlarm state after checking the variable suspend. It remains there until it is resumed. While in that state it is constantly checked to see if the variable alarmname is Suspended. Once alarm is resumed it goes to the Run state. If an alarm was not suspended it directly goes to the Run state when the machine was started. The alarm condition is constantly checked in the Run state. It remains in the Run state till the alarm condition is removed. Once the alarm condition is removed it goes to the ClearAlarms state. It remains in ClearAlarms state till a new alarm condition occurs and a thread is started by the handler EFSM. When such a thing happens, it goes back into the Run state. It may also go to the SuspendAlarm state if another alarm is suspended.

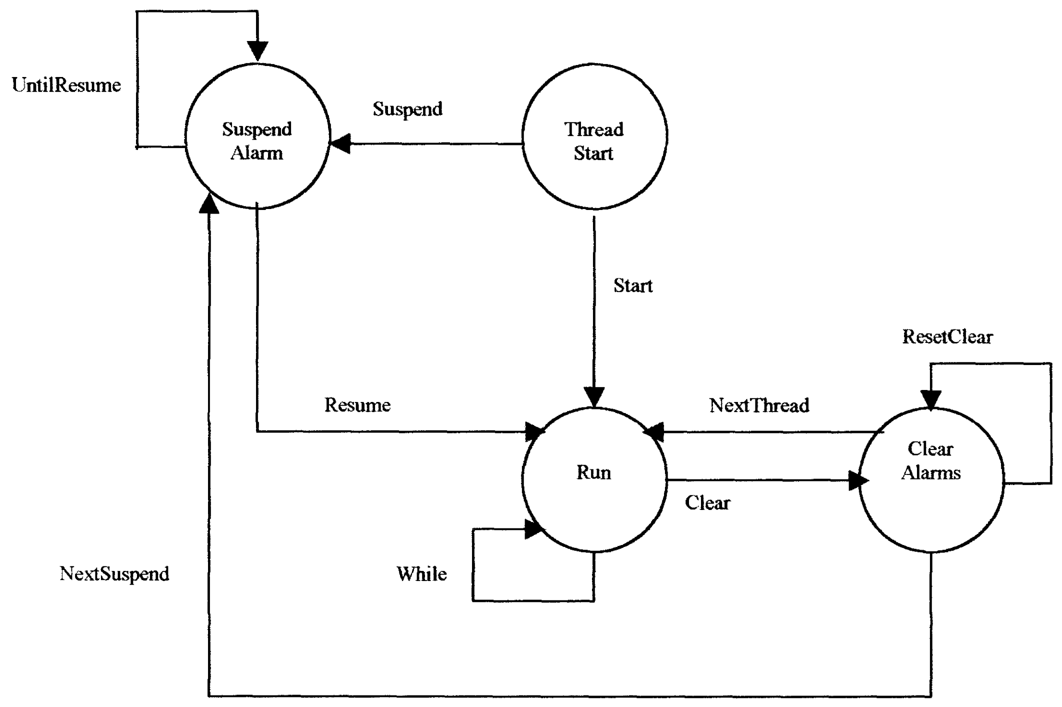

Figure 4.4 Thread EFSM.

Heap Variables used in Thread EFSM

Cthreadstart of type Boolean.

clear of type Boolean.

Input Variables used in Thread EFSM

threadstart of type Boolean.

resclear of type Boolean.

suspend of type Boolean.

alarmname of type alarm.

alarmno of type natural numbers.

Transition Definitions for Thread EFSM

transdef "Start " "
Guard:threadstart
Act:Skip;"

transdef "Suspend " "
Guard:suspend
Act:Skip;"

transdef "UntilResume " "
Guard:alarmname=Suspended
Act:Skip;"

transdef "Resume " "
Guard:Not(alarmname=Suspended)
Act:Skip;"

transdef "Clear " "
Guard:Not(alarmname=Active)
Act:clear<--true;
Cthreadstart<--false;"

transdef "While " "
Guard:alarmname=Active
Act:Cthreadstart<--false"

transdef "NextThread" "
Guard:threadstart
Act:Skip;"

transdef "NextSuspend " "
Guard:suspend
Act:Skip;"

Clear EFSM: It starts when an alarm is cleared and the variable clear is set in the thread EFSM. It goes to the Check state. Here it is checked for the next higher priority alarm starting from the highest priority. It goes through all the alarm conditions to find the next alarm. If there is an alarm it goes to the ResetClear state where it sets a variable nextal so that the variable clear is set to false in the handler EFSM. After this it goes to the NextAlarm state and starts the Handler EFSM. It will remain in that state till some other alarm is cleared and on such a situation it goes to the Check state again. If no alarm condition exists after the Check state it goes to the NoAlarm state. It remains in that state till a new alarm is cleared and on such a situation it goes to the Check state again.

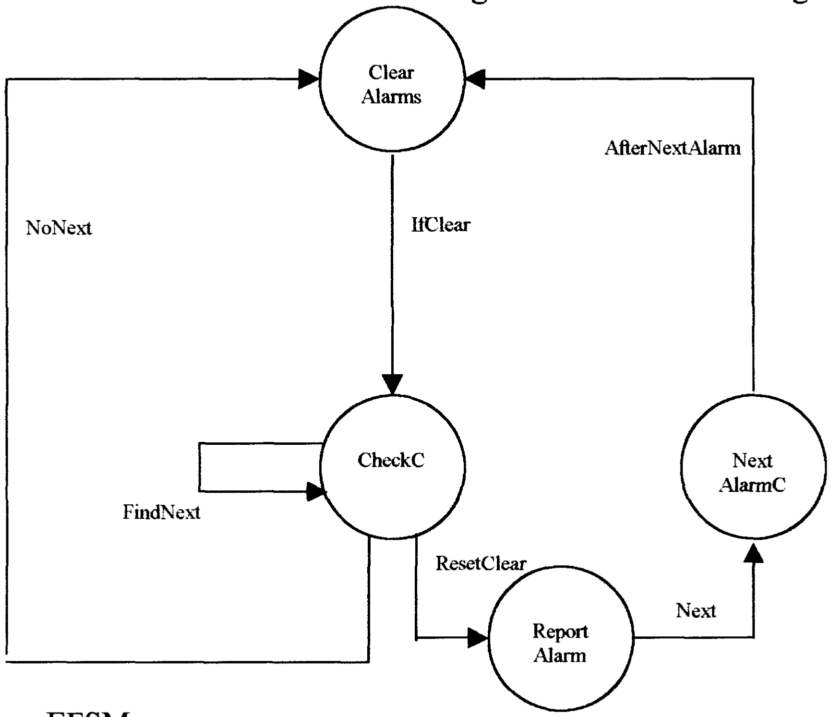

Figure 4.5 Clear EFSM.

Heap Variables used in Clear EFSM

afterclear of type Boolean.

resclear of type Boolean.

nextal of type Boolean.

i of type natural numbers.

Input Variables used in Clear EFSM

Cafterclear of type Boolean.

Nalarm of type Boolean.

clear of type Boolean.

names of type namesslist.

Transition Definitions for Clear EFSM

transdef "IfClear " "
Guard:clear
Act:Skip;"

transdef "FindNext " "
Guard: (names!i=NotActive)
Act:i<--i+1;"

transdef "NoNext " "
Guard:Not(i<#15)
Act:resclear<--true;"

transdef "ResetClear " "
Guard: (names!i=Active)
Act:nextal<--true;
afterclear<--true;"

transdef "Next" "
Guard: Nalarm
Act:nextal<--false;"

transdef "AfterNextAlarm " "
Guard:Cafterclear
Act:afterclear<--false;"

SilenceSound EFSM: It starts when a variable silencesound is set in the Handler EFSM. The function of this machine is to silence or sound the alarm. If an alarm is to be silenced it will go to the Silence state if the variable silence is set or it will go to the Sound state if the variable sound is set. It will remain in the Silence state till the silence time of the current alarm is not expired. If silence time of the current alarm expires then it goes into the Sound state and sound variable is set. It may go back to the SilenceSound state if a new alarm condition occurs. After the alarm is sounded it remains in the Sound state till a new alarm condition occurs and on such a case it goes back to the SilenceSound state.

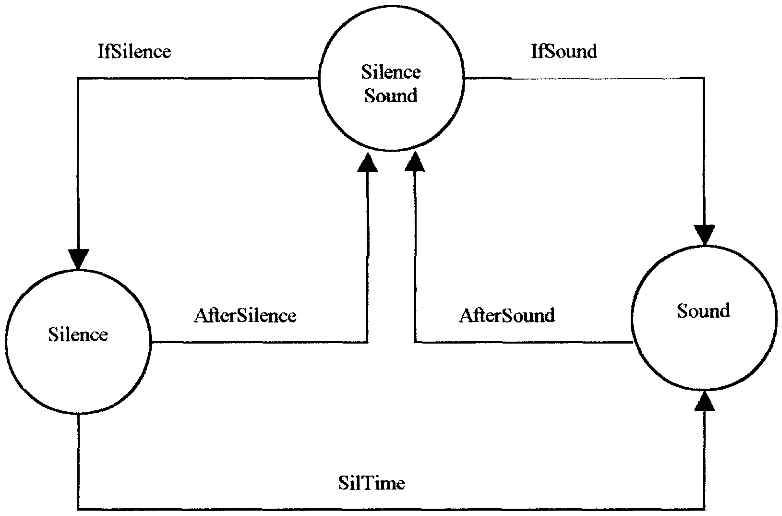

Figure 4.6 SilenceSound EFSM.

Heap Variables used in SilenceSound EFSM

currtime of type natural numbers.

silence of type Boolean.

sound of type Boolean.

siltimes is of type array.

Input Variables used in SilenceSound EFSM

alarmno of type natural numbers.

systime of type natural numbers.

soundsilence of type Boolean.

Transition Definitions for SilenceSound EFSM

```
transdef "IfSilence" "
Guard:silence
Act:currtime<--systime;"
transdef "IfSound" "
Guard:sound
Act:Skip;"

transdef "AfterSilence" "
Guard:soundsilence
Act:sound<--false;
silence<--false;"

transdef "AfterSound" "
Guard:soundsilence
Act:sound<--false;
silence<--false;"

transdef "SilTime" "
Guard:Not((systime-currtime) < (siltimes!alarmno))
Act:sound<--true;"
```

4.2 Safety Properties

Safety properties are those properties that hold true at any state of the finite state machine. In other words at all finite points in time nothing bad has happened to the system. Liveness properties also exist in a system like safety properties. Informally it can be stated as "something good will eventually happen". For example the second safety property in the following list of safety properties might be a Liveness property.

Safety properties are identified from the original specifications given and written down in English.

The Safety Properties identified from the original specifications are as follows.

1. Always the alarm that is sounding is the highest priority alarm that is active.
2. If pump is unplugged eventually only pump_unplugged alarm will remain or become active.
3. Alarm silence/reset button is active only if the alarm is triggered.
4. Alarm sounds only if it becomes active.

One of the widely used specification languages to do formal specification of concurrent programs is Linear Temporal Logic (LTL).

So the safety properties is written in reverse Linear Temporal Logic (LTL) and is as follows

1. Never has the alarm been sounding for an alarm condition other than the one of highest priority.
2. Never is any other alarm active if pump is unplugged.
3. Never has there been an option for silence/sound if an alarm was not active.
4. Never has an alarm sounded if it is not active.

Now the safety properties are written in Linear Temporal Logic as follows

1.

Alarm Sounding ➔

((Priority(current alarm)) > (priority(other active alarm)))

2.

unplugged and unplugged alarm

from then on

((sometime pluggedin) V Not(other alarm))

3.

Always (silence sound option)

➔sometime active

4.

Always (sounded)

➔sometime active

Future Work: The safety properties defined above in LTL are to be defined as properties in the product of all the EFSM machines described in the previous section. Using the DOVE tool the properties are to be verified. In other words the product must be proved or disproved using the safety properties.

APPENDIX A

PROCEDURES USED

This section gives the pseudo code for the procedures used, that was provided in the requirements [2].

```
procedure Add Alarm(NewAlarm : BooleanAlarms);
begin
    ActivateAlarms : = ActiveAlarms + { NewAlarm };
    SilenceButton : = true;
    ResetButton : = true;
    Display Alarms;
end;

procedure Clear Alarm(FixedAlarm : BooleanAlarms);
begin
    if FixedAlarm in ActiveAlarms then do
        if  FixedAlarm in PolledAlarms then do
            wait until ResetAlarms = = true; % user must reset for polled values
        od;
        ActiveAlarms : =  ActiveAlarms – { FixedAlarm };
        if ActiveAlarms  < > { } then DisplayAlarms else do
            begin
                clear screen;
```

```
                        SilenceButton : = false;

                        ResetButton : = false;

                    end;

                od;

            od;

        end;


        procedure Display Alarms( );

        begin

            ListAlarms : list of alarm;

            TopAlarm : Alarm;


            clear screen;

            AudibleAlarm : = false;

            ListAlarms : = sort (ActiveAlarms) % sort alarms in order of priorities and

                                              % display them in this order


            if ListAlarms = = TopAlarm : : ListAlarms then do

                begin

                    display(TopAlarm); % prints out alarm type and how to fix it

                    Sound Alarm(TopAlarm); % activates audible alarm

                    while ListAlarms = / = { } do

                        begin
                        ListAlarms = = TopAlarm : : ListAlarms;
                        display(TopAlarm);
                        sound(TopAlarm);
                    end;
                od;
            end;
        od;
    end;

    procedure Sound Alarm(NewAlarm : Alarm);
    begin
        if NewAlarm.level = = 2 then sound level2 else % 5 high – pitch "bings"
            if AudibleAlarm = = false then do
                begin
                    AudibleAlarm : = true;
                    sound level1;
                end;
            od;
    end;

    procedure Silence Alarms( );
    begin
        timer : clock;
        sil : int;
        timer : = 0;
        sil : = min(ActivateAlarms.silence); % find minimum temp. silence period
        until timer = = sil
        do
                AudibleAlarm : = false; % signal to stop audible alarms
        od;
        AudibleAlarm : = true;
        SilenceButton : = false;
end;
procedure Reset Alarms ( );
begin
        if ActiveAlarms = { } then do
                begin
                        clear screen;
                        AudibleAlarm : = false;
                end;
        od;
        else do
                begin
                        Display Alarms;
                        SilenceButton : = false;
                end;
        od;

end;
```

APPENDIX B

PROGRAM CODE FOR SIMULATION OF ALARMS UNIT

This section gives the program code used in the implementation for the simulation of alarms unit.

```
import java.io.*;
import java.util.Random;
import java.util.Vector;
import java.util.Timer;
import java.util.TimerTask;

public class alarm
{
        public static Vector alarmslist=new Vector(15); // contains list of alarms
        public static int[] siltime=new int[15];
        public static boolean pump_plugged=false;
        public static File outputFile = new File("log1.doc");
        public static NewThread curr=null;
        public static NewThread t1=null;
        public static void main(String args[ ])
        {
                int alarmno;
                for(int i=0;i<16;i++) // initialization of alarms list
        {
                alarmslist.insertElementAt("none",i);
        }
        for(int i=0;i<16;i++) // initialization of alarms array
        {
                globals.alarmsarray[i]=false;
        }
        for(int i=0;i<16;i++) // initialization of alarms array
        {
                globals.alarmsuspend[i]=false;
        }
        new test(0,500);
        new pollingschedule(0,10);
}
public static void process(int alno, boolean resume)
{
        globals.silenceall=false;
        globals.pronce=alno;
        globals.alarmno=alno;
        siltime[0]=2;
        siltime[1]=5;
        siltime[2]=2;
        siltime[3]=2;
```

```
                siltime[4]=2;

                siltime[5]=5;

                siltime[6]=7;

                siltime[7]=7;

                siltime[8]=7;

                siltime[9]=7;

                siltime[10]=7;

                siltime[11]=7;

                siltime[12]=7;

                String input="S";

                        if(globals.alarmno==1)

                        {

                                globals.display=true;

                                displayalarms();

                        }


if(globals.alarmno==2) {globals.alarmname="Continuity Fault";globals.priority=2;}

if(globals.alarmno==3) {globals.alarmname="Impedence fault";globals.priority=3;}

if(globals.alarmno==4) {globals.alarmname="Occulusion fault";globals.priority=4;}

if(globals.alarmno==5) {globals.alarmname="FallingBP fault";globals.priority=5;}

if(globals.alarmno==6) {globals.alarmname="fail fault";globals.priority=6;}

if(globals.alarmno==7) {globals.alarmname="LossBP fault";globals.priority=7;}

if(globals.alarmno==8) {globals.alarmname="NoBP fault";globals.priority=8;}
if(globals.alarmno==9) {globals.alarmname="No Cuff pressure
fault";globals.priority=9;}
if(globals.alarmno==10) {globals.alarmname="Invalid cuff fault";globals.priority=10;}
if(globals.alarmno==11) {globals.alarmname="Not able to measure BP
fault";globals.priority=11;}
if(globals.alarmno==12) {globals.alarmname="No BP source fault";globals.priority=12;}
if(globals.alarmno==13) {globals.alarmname="Non Control BP
fault";globals.priority=13;}
if(!globals.alarmonthread)
{
        NewThread t1=new NewThread(globals.alarmname,globals.alarmno);
        System.out.println("First occurence Alarm priority is : " +globals.priority);
try
{
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Do you want to silence alarm Y/N ");
input = stdin.readLine();
}
catch ( IOException e ) {System.out.println (e);}
addalarm(globals.alarmname,globals.priority);
globals.alarmonthread=true;
globals.curralarm=globals.alarmno;
curr=t1;
```

```
t1.start();

if(input.equals("Y") || input.equals("y"))

{

silencealarm(globals.alarmno,globals.alarmname);

}

else

{

soundalarm(globals.alarmno);

}

}

else if(globals.alarmresume && globals.alarmonthread)

{

if(globals.alarmsuspend[globals.priority])

{

System.out.println("Current thread is " +curr.name );

curr.resume();

}

else

{

NewThread t1=new NewThread(globals.alarmname,globals.alarmno);

curr=t1;

t1.start();

}
```

```
globals.alarmresume=false;

globals.alarmonthread=false;

System.out.println("After clear Alarm priority is : " +globals.priority);

try

{

BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

System.out.println("Do you want to silence alarm Y/N ");

input = stdin.readLine();

}

catch ( IOException e ) {System.out.println (e);}

addalarm(globals.alarmname,globals.priority);

globals.alarmonthread=true;

globals.curralarm=globals.alarmno;

if(input.equals("Y") || input.equals("y"))

{

silencealarm(globals.alarmno,globals.alarmname);

}

else

{

soundalarm(globals.alarmno);

}

}

else
```

```
{
NewThread t1=new NewThread(globals.alarmname,globals.alarmno);
if(globals.priority<globals.curralarm)
{
globals.alarmsuspend[globals.curralarm]=true;
curr.suspend();
addalarm(globals.alarmname,globals.priority);
globals.curralarm=globals.alarmno;
System.out.println("Higher Priority Alarm priority is : " +globals.priority);
try
{
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Do you want to silence alarm Y/N ");
input = stdin.readLine();
}
catch ( IOException e ) {System.out.println (e);}
curr=t1;
t1.start();
if(input.equals("Y") || input.equals("y"))
{
silencealarm(globals.alarmno,globals.alarmname);
}
else
{
soundalarm(globals.alarmno);
}
}
else
{
addalarm(globals.alarmname,globals.priority);
}
}
}
public static int polling()
{
//Alarms checking
for(int i=1;i<16;i++)
{
if(globals.alarmsarray[i]) return i;
}
return 99;
}
static void addalarm(String newalarm, int position)
{
alarmslist.remove(position);
if(position!=globals.once)
```

```
{
try
{
FileWriter out = new FileWriter(outputFile,true);
out.write("Alarm with priority "+position+" being added \n");
out.flush();
}
catch ( IOException e ) {System.out.println (e);}
}
alarmslist.insertElementAt(newalarm,position);
globals.silencebutton=true;
globals.resetbutton=true;
globals.once=position;
}
static void silencealarm(int num,String name)
{
System.out.println("Alarm to be silenced");
globals.alarmsarray[num]=false;
new Reminder(siltime[num-1],name,num);
try
{
FileWriter out = new FileWriter(outputFile,true);
out.write("Alarm with priority "+num+" being silenced \n");
out.flush();

}

catch ( IOException e ) {System.out.println (e);}

globals.audiblealarm=true;

globals.silencebutton=false;

}

static void soundalarm(int num)

{

System.out.println("Alarm to be sounded");

try

{

FileWriter out = new FileWriter(outputFile,true);

out.write("Alarm with priority "+num+" being silenced \n");

out.flush();

}

catch ( IOException e ) {System.out.println (e);}

if (globals.audiblealarm == false )

{

globals.audiblealarm=true;

System.out.println("Level 2 alarm");

}

}

static void clearalarms(String fixedalarm,int fixedno)
```

```
{

System.out.println("Alarm to be cleared " +fixedno);

globals.alarmsuspend[fixedno]=false;

try

{

FileWriter out = new FileWriter(outputFile,true);

out.write("Alarm with priority "+fixedno+" being cleared \n");

out.flush();

}

catch ( IOException e ) {System.out.println (e);}

boolean empty=true;

// for polled alarms

/*if(fixedno==2 || fixedno==3)

{

// check if the alarm is really cleared

// get value from pump monitor

}*/

if(alarmslist.contains(fixedalarm))

{

alarmslist.remove(fixedalarm);

alarmslist.insertElementAt("none",fixedno);

for(int i=1;i<16;i++)

{
```

```
if(globals.alarmsarray[i]) {empty=false;globals.val=i;break;}

}

if(empty==false)

{

globals.alarmresume=true;

if(globals.val==globals.pronce)

globals.pronce1=false;

else

globals.pronce1=true;

if(globals.pronce1)

process(globals.val,globals.alarmresume);

}

else

{

globals.silencebutton=false;

globals.resetbutton=false;

globals.alarmonthread=false;

}

}

}

static void displayalarms()

{

System.out.println("Alarm to be displayed");
```

```
try

{

FileWriter out = new FileWriter(outputFile,true);

out.write("Alarms to be displayed\n");

out.flush();

}

catch ( IOException e ) {System.out.println (e);}

int alarmsLen = alarmslist.size();

for (int i=0; i < alarmsLen; i++)

{

if(alarmslist.elementAt (i)!="none")

System.out.println (alarmslist.elementAt (i));

}

for(int i=0;i<16;i++) // reinitialization of alarms list

{

alarmslist.insertElementAt("none",i);

}

for(int i=0;i<16;i++) // reinitialization of alarms array

{

globals.alarmsarray[i]=false;

}

for(int i=0;i<16;i++) // reinitialization of alarms array

{
```

```
globals.alarmsuspend[i]=false;
}
System.out.println("Pump is unplugged");
System.exit(0);
}
}
class globals
{
public static int alarmno=0;
public static String alarmname;
public static int priority;
public static boolean alarmonthread=false;
public static int curralarm;
public static NewThread t1;
public static boolean hold=true;
public static boolean x=true;
public static boolean silencebutton=true;
public static boolean resetbutton=true;
public static boolean audiblealarm=false;
public static boolean[] alarmsarray=new boolean[17];
public static boolean[] alarmsuspend=new boolean[17];
public static int temp=0;
public static boolean display=false;
public static int once=0;

public static int val=0;

public static boolean alarmresume=false;

public static int pronce=0;

public static boolean pronce1=true;

public static boolean silenceall=true;

}

class NewThread extends Thread

{

public String name;

Thread t;

public int threadno;

public String alarmname;

NewThread(String threadname, int runno)

{

name = threadname;

t=new Thread(this,name);

threadno=runno;

alarmname=threadname;

}

public void run()

{

while(globals.alarmsarray[threadno])
```

```
{
yield();
}
alarm.clearalarms(alarmname,threadno);
}
}
class Reminder
{
Timer timer;
public String aname;//alarm name to be used for printing.
public int alno;// alarm number to be used for interrupting printing.
public Reminder(int seconds, String name,int no)
{
timer = new Timer();
aname=name;
alno=no;
timer.schedule(new RemindTask(), seconds*1000);
}
class RemindTask extends TimerTask
{
public void run()
{
globals.alarmsarray[alno]=true;
```

```
timer.cancel(); //Terminate the timer thread

}

}

}

class pollingschedule

{

Timer timer;

public pollingschedule(int seconds,int delay)

{

timer = new Timer();

timer.schedule(new RemindTask(), seconds*1000,delay*100);

}

class RemindTask extends TimerTask

{

public void run()

{

int value=alarm.polling();

if(value == globals.pronce)

globals.pronce1=false;

else

globals.pronce1=true;

if(value != 99)

globals.alarmsuspend[value]=false;
```

```
if(value!=0 && value!=99 && globals.pronce1)

{

alarm.process(value,globals.alarmresume);

}

}

}

class test

{

Timer timer;

public test(int seconds, int delay)

{

timer = new Timer();

timer.schedule(new TestTask(), seconds*1000, delay*10);

}

class TestTask extends TimerTask

{

public void run()

{

Random r=new Random();

Random rn = new Random();

int rand=Math.abs(r.nextInt())%15;

int rand1=(rand+3);

int rand2=(rand-3);
if(rand!=0 && rand!=1) globals.alarmsarray[rand]=rn.nextBoolean();

if(rand1>1 && rand1<16 && rand!=1) globals.alarmsarray[rand1]=rn.nextBoolean();

if(rand2>1 && rand2<16 && rand!=1) globals.alarmsarray[rand2]=rn.nextBoolean();


}

}

}
```

REFERENCES

The first is for SCR (and using finite state machine to specify systems), and the rest are for CARA.

[1] Heninger, K. L. (1980). Software Engineering. In Specifying Software Requirements for Complex Systems: New Techniques and Their Application (pp. 2-13).

[2] WRAIR Dept. of Resuscitative Medicine. (2001). CARA Questions, Proprietary Document. WRAIR, Silver Spring, MD.

[3] WRAIR Dept. of Resuscitative Medicine. (2001). CARA Tagged Requirements, Proprietary Document. WRAIR, Silver Spring, MD.

[4] Manna & Pnueli. (1991). The Temporal Logic of Reactive and Concurrent Systems: Specification. New York: Springer-Verlag.

[5] Ozols, M. A., Cant, A., & Eastaughffe, K. A. DOVE: A Tool for Design, Modelling, and Verification in Safety Critical Systems.

[6] Elsa Gunter & Doron Peled. (1999). Path Exploration Tool.
In Rance Cleaveland editor, Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, volume 1579 of Lecture Notes in Computer Science pages 405-419, Amsterdam, The Netherlands, 1999. Springer.

[7] Elsa Gunter & Doron Peled. Temporal Debugging for Concurrent Systems.
In Joost-Pieter Katoen and Perdita Stevens, editors, Tools and Algorithms for Construction and Analysis of Systems, 8th International Conference, TACAS '02, to appear., Grenoble, France, 2002.

[8] Rajeev Alur, David Arney, Elsa L. Gunter, Insup Lee, Jaime Lee, Wonhong Nam, Frederick Pearce, Steve Van Albert, Jiaxiang Zhou. Formal Specifications and Analysis of the Computer Assisted Resuscitation Algorithm (CARA) Infusion Pump Control System. International Journal on Software Tools for Technology Transfer, to appear.