INTELLIGENT LOGIC ACTIVITY RESOLUTION

BACKGROUND

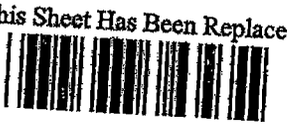
The present invention concerns automated process control and pertains particularly to use of intelligent logic activity resolution within an automated process control system.

Current methods of automated manufacturing process control are limited by the use of static connected state transition models to impose a predefined behavioral expectation on the process.

In a standard state transition model, each state can only transition to a limited number of other states based on fixed transition events. For example, a manufacturing process in a first state can only transition to a second state upon the occurrence of a first transition event or into a third state upon the occurrence of a second transition event. All other states and transition events are considered invalid. Under this static state transition process model, if, while in the first state, a third transition event occurs, the model will not be able to determine a next valid state. Instead, the system would, for example, issue an alarm and halt the manufacturing process.

An additional limitation to automated manufacturing process control systems that function in accordance with a standard state transition model is the static nature of the state model. There is no provision for the system to adapt itself to new states and transition events without significant manual intervention.

This Sheet Has Been Replaced



SUMMARY OF THE INVENTION

In accordance with the preferred embodiment of the present invention, intelligent logic activity resolution (ILAR) is provided for a process. Upon a first trigger event being received, a first subset of conditions of the process is detected. The first subset of conditions is selected based on the first trigger event. When the first subset of conditions match predetermined values, a first state is entered. When the first subset conditions do not match the predetermined values, additional conditions are checked. On the basis of the checked additional conditions, a new state is selected for entry.

In the preferred embodiment, the detection of the first subset of conditions is done by running a separate metaprocedure to detect each condition in the first subset of conditions. A metaprocedure is a software function that performs an automated check for a condition. A metaprocedure can also perform non-state change related functions such as sending messages to external systems, or storing and retrieving data variables relevant to the process. In the preferred embodiment, each condition is represented with a single metaprocedure.

In a preferred embodiment of the present invention, the subset of conditions are also selected based on a current state of the process.

Upon receiving a second trigger event, a second subset of conditions of the process are detected. The second subset of conditions are selected based on the second trigger event. When the second subset of conditions match predetermined values, a corresponding state is entered. When the second subset of conditions do not match the predetermined values, additional

conditions are checked. On the basis of the checked additional conditions a new state is selected for entry.

The ILAR system described herein removes the constraints of the static connected state transition model. The ILAR system disconnects the state and transition event relationship and treats each state and transition event as a unique and independent occurrence. Using a dynamic, real time logic engine, rather than a static connected state model, the ILAR heuristically determines the next state based on an analysis of all detectable manufacturing process conditions rather than the transition event detected. With this method, all states and transitions are always valid, unless specifically declared invalid by the user. This is a significant advance over prior art state machines.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates operation of a standard state transition model in accordance with the prior art.

Figure 2 illustrates intelligent logic activity resolution in accordance with the preferred embodiment of the present invention.

Figure 3 is a simplified block diagram of an intelligent logic activity resolution system in accordance with the preferred embodiment of the present invention.

Figure 4 illustrates logic relationship between trigger events, metaprocedures and elapsed timers in accordance with the preferred embodiment of the present invention.

Figure 5 is a flowchart that describes operation of an intelligent logic activity resolution system in accordance with the preferred embodiment of the present invention.

DESCRIPTION OF THE PRIOR ART

Figure 1 illustrates operation of a standard state transition model in accordance with the prior art. In the shown standard state transition model, a manufacturing process currently in a state 101 can transition to a state 102 upon the occurrence of a transition event 106. When the manufacturing process currently is in state 101, the manufacturing process can also transition to a state 104 upon the occurrence of a transition event 109. When the manufacturing process currently is in state 102, the manufacturing process can transition to a state 103 upon the occurrence of a transition event 107. All other states and transition events are considered invalid. Under this static state transition process model, if, while in state 101, transition event 107 occurs, the model will not be able to determine a next valid state. Instead, the system would, for example, issue an alarm and halt the manufacturing process.

DESCRIPTION OF THE PREFERRED EMBODIMENT

Figure 2 illustrates intelligent logic activity resolution (ILAR) in accordance with the preferred embodiment of the present invention. The ILAR system described herein removes the constraints of the static connected state transition model. An ILAR system disconnects the state and transition event relationship and treats each state and transition event

as a unique and independent occurrence. Using a dynamic, real time logic engine, rather than a static connected state model, the ILAR system heuristically determines the next state based on an analysis of all detectable manufacturing process conditions. This allows significant flexibility over the standard approach of basing state transitions just on a detected transition event.

In Figure 2, an ILAR logic decision engine 110 is made up of one or more independent decision processes that query all possible process conditions to determine a process state. The query processes are, for example, linked in Boolean logic trees to allow decisions to be made based upon the process queries.

In the preferred embodiment of the present invention, all states and transitions are always valid, unless specifically declared invalid by the user that programs the ILAR system. If ILAR logic decision engine 110 detects an event specifically declared invalid, ILAR logic decision engine 110 will behave much like the static state model by issuing an alarm and halting the process. In the case of a transition event that causes a state that is not declared invalid and not previously detected, ILAR logic decision engine 110 will run a user defined set of default metaprocedures known as differentia in order to re-establish the correct process state. Using this method, ILAR logic decision engine 110 can detect and report in real time new manufacturing process conditions.

Thus, as illustrated by Figure 2, a state 111, a state 112 and a state 113 are independent state locations, linked by a dynamically allocated event transition 116, a dynamically allocated event transition 117, a dynamically

allocated event transition 118 and a dynamically allocated event transition 119. This organization allows ILAR logic decision engine 110 to be easily adaptable and modified to provide real time behavioral determination of process execution.

The organization shown in Figure 2 allows transitions and states from different processes to be combined into higher order logic decision engines within ILAR logic decision engine 110. This allows ILAR logic decision engine 110 to be able to monitor and control a larger portion of the process. The dynamic nature of the manufacturing process is represented by the dynamic configuration capability of ILAR logic decision engine 110.

Additionally the organization shown in Figure 2 allows each of the states and transitions to be assigned a time parameter. If the state or transition triggers the time parameter, ILAR logic decision engine 110 can execute without the need for a specific process transition. In this way, the lack of process activity can also be detected and monitored.

Figure 3 is a simplified block diagram that shows a specific implementation of an intelligent logic activity resolution (ILAR) system within a manufacturing process.

As illustrated by Figure 3, a process is made up of the operation of one or more pieces of equipment. In Figure 3, this is represented by process equipment 121. Process equipment 121 may be, for example, an Agilent 83000 test system available from Agilent Technologies, Inc. Alternatively, process equipment 121 may be a materials handler or any other type of processing equipment manageable by a workstation.

Process equipment 121 is managed by a process equipment workstation 122. Within process equipment workstation 122 a workstation operating system (O/S) 126 runs. Workstation operating system 126 is, for example, the UNIX operating system, Windows NT operating system, Linux operating system, or another available operating system.

The ILAR system uses scripts 125 on the local station to execute required conditional check functions. A local ILAR process 124 coordinates execution of scripts 125 in accordance with instructions from an ILAR server workstation 128. Local ILAR process 124 also coordinates the transfer of process trigger events to ILAR server workstation 128 via a connection 127. Connection 127 is implemented, for example as part of a TCP/IP network.

Within ILAR workstation 128 a workstation operating system (O/S) 129 runs. Workstation operating system 129 is, for example, the UNIX operating system, Windows NT operating system, Linux operating system, or another available operating system. Figure 3 shows that within workstation O/S 129, various software processes run. These software processes along with local ILAR process 124 compose the ILAR system.

A distributed message hub 130 provides interprocess communication such as the process trigger events from process equipment workstation 126. Distributed message hub 130 is implemented, for example, using a Hume Integration Services Distributed Message Hub mailbox server, available from Hume Integration Services, having a business address of 35 Sundown Parkway, Austin, Texas 78746. The Hume Distributed Message Hub is a

high-level software facility providing interprocess communication for applications executing within a TCP/IP network.

Any source-connection process (equipment, driver, process, user, etc.) such as local ILAR process 124 or any other source-connection process existing on process equipment workstation 122 or other workstations used to control process equipment, request and present data from a particular external source via mailbox messages in the Hume mailbox (mbx) system within distributed message hub 130.

An ILAR execution engine 131 makes decisions about the current state of process equipment 121 and potentially other pieces of equipment. ILAR execution engine 131 takes action when the perceived state changes. The actions performed include the updating of the ILAR system's internal representation of the state of the processing equipment and may include the generation of messages to process equipment workstation 122, the requesting of information from a user and/or the display of messages to the user.

A user utilizes an ILAR graphical user interface 133 to specify within a database 137 process equipment types and process trigger events. From equipment database 136, a data distributor 136 prepares a runtime equipment database 135 utilized by ILAR execution engine 131 during runtime. The user also utilizes graphical user interface 133 to define and save process analysis metaprocedures and process state differentia, as represented in Figure 3 by a block 132. The decision logic that the Execution Engine follows is user defined, and also resides in a SQL database 135. The decision logic is composed of logical units and the links between them.

ILAR execution engine 131 stores run time process status variables and other run time information in SQL database 134.

Graphical user interface 133 is JAVA based and uses the Hume Data Hub tool to create and store centralized equipment database 137. In the preferred embodiment, ILAR execution engine 131 is written in tcl, using the Hume Data Hub mbx and database extensions. Each source-connection process may be written in whatever language is convenient to the communication method of the source and the requirement of writing to the Hume mbx system.

The total ILAR system takes advantage of the inherent logical structure of a process. As discussed above, a process is made up of the operation of one or more pieces of equipment. Each piece of equipment is defined by its type.

Over linear sequential time, a process moves through a series of states. For example, one state is loading material. Another state is testing material. A third state is calibration the process system. And so on.

Each process has one or more identifiable process conditions. For example, one process condition is that an operator is logged into the process system. Another process condition is that a test program is loaded. Another process condition is that a test program is running. Another process condition is that a materials handler is moving material into a chamber for processing. And so on.

Each state can be identified by a unique set of one or more process conditions. For example, in one set of processing conditions, the state "testing" can be identified when all the following conditions are true: an

operator is logged into the process system, a test program is loaded and a test program is running.

Transitioning from one state to another is considered a process event. The occurrence of this process event is called a process trigger event. For example, a process trigger event occurs when material is loaded. A process trigger event occurs when a test has begun. A process trigger event occurs when a test has finished. A process trigger event occurs when material has been unloaded. And so on.

The transition from one state to another is detected by monitoring the process trigger events.

In order to implement and execute an ILAR system, the user defines five different types of process information. The five different types of process information are: (1) equipment types; (2) equipment states; (3) process trigger events; (4) metaprocedures; and, (5) differentia.

A metaprocedure is a software function that performs an automated check for a condition. Each condition is represented with a single metaprocedure.

What is meant by differentia is a defined process condition set that indicates a process state. Each differentia represents a unique set of process conditions that define a unique process state.

Using ILAR graphical user interface 133, the user associates the five process information components to define a specific instance of a process execution. ILAR works by implementing the generated process logic.

Figure 4 illustrates logic relationship between trigger events, metaprocedures and elapsed timers as programmed by a user. Seven

trigger events 141 are shown: an trigger event T1, an trigger event T2, an trigger event T3, an trigger event T4, an trigger event T5, an trigger event T6 and an trigger event T7. Five process condition metaprocedures 142 are shown: a metaprocedure M1, a metaprocedure M2, a metaprocedure M3, a metaprocedure M4 and a metaprocedure M5. Five process event elapsed timers 143 are shown: an elapsed timer ET1, an elapsed timer ET2, an elapsed timer ET3, an elapsed timer ET4 and an elapsed timer ET5.

Table 1 below describes process execution logic for the logic relationships shown in Figure 4. The ILAR system can provide a unique process execution logic relationship and table for each state the process can be in.

Table 1

Trigger event	Run MP	Start Timers	P.S. on Pass	MP fails;	Timer Expires
T1	M1,M4	ET1,ET5	A	Run all MPs	Run all MPs
T2	M3	ET4	B	Run MPs M1,M4,M5	Run all MPs
T3	M5	ET2	C	Run all MPs	Send Trigger T2
T4	M2	ET2	D	Run MPs M3,M5	Run all MPs
T5	M3,M4	ET3,ET4	E	Run all MPs	Send Trigger T7
T6	M2,M4	ET1,ET5	F	Run MPs M1,M3	Run all MPs
T7	M1	ET3	G	Run all MPs	Run all MPs

In table 1, the first column sets out the trigger events. The second column shows which metaprocedures (MPs) are run for each trigger event set out in the first column. The third column shows which timers are started for each trigger event set out in the first column. The fourth column

shows which process state (P.S.) is set if the metaprocedures set out in the second column pass when they are run as a result of the trigger event set out in the first column. The fifth column shows which metaprocedures are run e.g., all the metaprocedures) when the metaprocedures set out in the second column do not pass when they are run as a result of the trigger event set out in the first column. The sixth column shows which metaprocedures are run (i.e., all the metaprocedures) when the timers set out in the third column expire after being started as a result of the trigger event set out in the first column. The expiration of a timer can also result in the generation of a new trigger to the input of ILAR execution engine 131.

When, as a result of a metaprocedure failure or the expiration of a timer, all the metaprocedures are run, the state is set as a result of which metaprocedures pass. Table 2 below shows which state is set depending upon which metaprocedure(s) pass(es).

Table 2

M1	M2	M3	M4	M5	Set State
Pass			Pass		A
		Pass			B
				Pass	C
	Pass				D
		Pass	Pass		E
	Pass		Pass		F
Pass					G

Figure 5 is a flowchart that describes operation of the ILAR system. In a step 151 the ILAR system starts up. In a step 152, all metaprocedures are run in order to query the process for all possible conditions. As discussed above, a metaprocedure is a software function that performs an

automated check for a condition. Each condition is represented with a single metaprocedure.

In a step 153, ILAR execution engine 131 compares the resulting map of conditions that return logical true and logical false results against all of the defined process condition sets. These condition sets are known in the ILAR system as differentia. As discussed above, each differentia represents a unique set of process conditions that define a unique process state. ILAR execution engine 131 returns as a result, the unique process state it has identified.

In a step 154, the ILAR system now waits for a process trigger event. The absence of a trigger event over a user determined period of time could itself represent an internally generated trigger event (i.e., the expiration of a timer).

A step 155 is entered upon the occurrence of a trigger event. In this step, the ILAR system runs a subset of user defined metaprocedures to determine if the process is in the state that the user would expect based on the previous state and the trigger event.

In a step 156, a check is made to see if process is in the state that the user would expect based on the previous state and the trigger event (pass). If so, in a step 157, the correct state is set and in step 154, the next trigger event is awaited.

If in step 156, the ILAR check fails, i.e., in a step 158 the process is not in the expected state based on the previous state and trigger event, the ILAR begins to run more metaprocedures until it can once again establish a

passing differentia and determine the correct state. Then, in step 154, the next trigger event is awaited.

Table 2 and Table 3 below show two example metaprocedures used in a test installation.

Table 2

```
##-------------------------------------------------------------
# $Date: 2000/02/24 05:35:44 $          $ $
#-------------------------------------------------------------
#
#
#                MODULE  : smartest_metaproc.tcl
#
#-------------------------------------------------------------
#
#       (c) Copyright Agilent Technologies COMPANY, 2000
#
#-------------------------------------------------------------
#
#  AUTHOR  : Pierre Gauthier, Agilent Technologies Co.
#
# DATE    : Feb 24  09:18:29 PDT 2000
#
#-------------------------------------------------------------
#
#
#-------------------------------------------------------------
#
# Development Information:
#
#                   This module defines the metaprocedures
#                   and differentiae called by the ILAR
#                   Execution Engine for an equipment of type
#                   smartest. These are tcl procedures
#                   attached to the states and triggers created
#                   by the ILAR Maintenance GUI.
#
#
##############################################
        #
#-------------------------------------------------------------
#     $Log: smartest_metaproc.tcl,v $
#
```

```
##############################################
        #

# ***************************************************
# Description:
#
#            this function prints a debug message to stdout
#            based on a global flag set in the
#             smartest_custom_init procedure.
#
#
# INPUTS:  debug message
#
# OUTPUTS: none
#
# ***************************************************
proc smartest_debug { message } {
  global glob_smartest_debug
  if {$glob_smartest_debug == 1} {
    puts $message
  }
}
# ***************************************************
# Description:
#             this function responds to triggers associated to
#             smartest's operator interface such as ABORT,
#            PAUSE etc...
#
#
#
# INPUTS:  equipment_id (usually hostname)
#         message from trigger
#         param : special parameter defined in the
#          ILAR Maintenance GUI.
#        param can be ABORT PAUSE RESET STOP
#
# OUTPUTS: 0 = success 1 = failure
#
# ***************************************************

proc smartest_buttonstate { equip_id message param } {
  global $equip_id
  set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
  switch $param {      ABORT    {
              smartest_debug "changing state smartest_idle for
        $equip_id"
                                equip_log $equip_id $eq_timestamp 0
        smartest_abort_button_pressed \
```

```
                    [set ${equip_id}(state)] smartest_idle
smartest_buttonstate:ABORT $message

    updateStateTime $equip_id [set ${equip_id}(state)] 0
$eq_timestamp
     ilar_change_state $equip_id smartest_idle
    updateStateTime $equip_id smartest_idle
$eq_timestamp 0
          }  PAUSE  {
     smartest_debug "changing state pause for $equip_id"
                    equip_log $equip_id $eq_timestamp 0
smartest_pause_button_pressed \
                    [set ${equip_id}(state)] pause
smartest_buttonstate:PAUSE $message

    updateStateTime $equip_id [set ${equip_id}(state)] 0
$eq_timestamp
     ilar_change_state $equip_id pause
    updateStateTime $equip_id pause $eq_timestamp 0
  }  RESET  {
     smartest_debug "changing state smartest_idle for
$equip_id"
    equip_log $equip_id $eq_timestamp 0
smartest_reset_button_pressed \
                    [set ${equip_id}(state)] smartest_idle
smartest_buttonstate:RESET $message
   updateStateTime $equip_id [set ${equip_id}(state)] 0
$eq_timestamp
     ilar_change_state $equip_id smartest_idle
    updateStateTime $equip_id smartest_idle
$eq_timestamp 0
  }  STOP  {
     smartest_debug "changing state smartest_idle for
$equip_id"
    equip_log $equip_id $eq_timestamp 0
smartest_stop_button_pressed \
                    [set ${equip_id}(state)] smartest_idle
smartest_buttonstate:STOP $message
   updateStateTime $equip_id [set ${equip_id}(state)] 0
$eq_timestamp
     ilar_change_state $equip_id smartest_idle
    updateStateTime $equip_id smartest_idle
$eq_timestamp 0
  }  CONTINUE {
     smartest_debug "changing state testing for
$equip_id"
    equip_log $equip_id $eq_timestamp 0
smartest_continue_button_pressed \
```

```
                                    [set ${equip_id}(state)] testing
            smartest_buttonstate:CONTINUE $message
               updateStateTime $equip_id [set ${equip_id}(state)]  0
            $eq_timestamp
                ilar_change_state $equip_id testing
               updateStateTime $equip_id testing $eq_timestamp  0
             }   } return 0}
# ****************************************************
# Description:
#            this function responds to the
#             smartest_calibration_start
#             & smartest_calibration_end triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
#          message from trigger
#          param : special parameter defined in the
#          ILAR Maintenance GUI.
#         param can be START END
#
# OUTPUTS: 0 = success 1 = failure
#
# ****************************************************
 proc smartest_calibration { equip_id message param } {

  global $equip_id
  set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
        switch $param {      START   {
            smartest_debug "changing state calibrating for
        $equip_id"
           equip_log $equip_id $eq_timestamp 0
        smartest_calibration_start  \
                                    [set ${equip_id}(state)] calibrating
        smartest_calibration:START  $message
          updateStateTime $equip_id [set ${equip_id}(state)]  0
        $eq_timestamp
            ilar_change_state $equip_id calibrating
           updateStateTime $equip_id calibrating $eq_timestamp
        0
        }   END   {
            smartest_debug "changing state smartest_idle for
        $equip_id"
           equip_log $equip_id $eq_timestamp 0
        smartest_calibration_end  \
                                    [set ${equip_id}(state)] smartest_idle
        smartest_calibration:END  $message
          updateStateTime $equip_id [set ${equip_id}(state)]  0
        $eq_timestamp
```

```
            ilar_change_state $equip_id smartest_idle
            updateStateTime $equip_id smartest_idle
        $eq_timestamp   0
        }     }   return 0}
# ******************************************************
# Description:
#           this function responds to the
#            smartest_diagnostic_start
#            & smartest_diagnostic_end triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
#         message from trigger
#         param : special parameter defined in the
#         ILAR Maintenance GUI.
#        param can be START END
#
# OUTPUTS: 0 = success 1 = failure
#
# ******************************************************
proc smartest_diagnostic { equip_id message param } {
 global $equip_id
 set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
      switch $param {      START   {
          smartest_debug "changing state diagnosing for
      $equip_id"
                      equip_log $equip_id $eq_timestamp 0
      smartest_diagnostic_start  \
                          [set ${equip_id}(state)] diagnosing
      smartest_diagnostic:START  $message
        updateStateTime $equip_id [set ${equip_id}(state)]  0
      $eq_timestamp
          ilar_change_state $equip_id diagnosing
         updateStateTime $equip_id diagnosing $eq_timestamp
      0
      }   END   {
          smartest_debug "changing state smartest_idle for
      $equip_id"
                      equip_log $equip_id $eq_timestamp 0
      smartest_diagnostic_end  \
                          [set ${equip_id}(state)] smartest_idle
      smartest_diagnostic:END  $message
        updateStateTime $equip_id [set ${equip_id}(state)]  0
      $eq_timestamp
          ilar_change_state $equip_id smartest_idle
         updateStateTime $equip_id smartest_idle
      $eq_timestamp   0
      }     }   return 0}
```

```
# ******************************************************
# Description:
#           this function responds to the smartest_lot_start
#           & smartest_lot_end triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
#        message from trigger
#        param : special parameter defined in the
#        ILAR Maintenance GUI.
#       param can be START END
#
# OUTPUTS: 0 = success 1 = failure
#
# ******************************************************
proc smartest_lot { equip_id message param } {
 global $equip_id
 set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
   switch $param {     START   {
          smartest_debug "changing state setup for $equip_id"
                    equip_log $equip_id $eq_timestamp 0
       smartest_on_lot_start  \
                              [set ${equip_id}(state)] setup
       smartest_lot:START $message
         updateStateTime $equip_id [set ${equip_id}(state)]  0
       $eq_timestamp
          ilar_change_state $equip_id setup
         updateStateTime $equip_id setup $eq_timestamp   0
        }   END    {
            set current_state [set ${equip_id}(state)]
            if { $current_state != "no_wip" } {
                    set new_state setup
                  } else {
                    set new_state no_wip
                  }
          smartest_debug "changing state $new_state for
       $equip_id"
         equip_log $equip_id $eq_timestamp 0
       smartest_on_lot_end  \
                              [set ${equip_id}(state)] $new_state
       smartest_lot:END $message
         updateStateTime $equip_id [set ${equip_id}(state)]  0
       $eq_timestamp
          ilar_change_state $equip_id $new_state
         updateStateTime $equip_id $new_state
       $eq_timestamp   0
```

```
        }    }  return 0}#
        ************************************************************
        *
# Description:
#           this function responds to the
#            smartest_operator_login
#            & smartest_operator_logout triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
#        message from trigger
#        param : special parameter defined in the
#        ILAR Maintenance GUI.
#        param can be LOGIN LOGOUT
#
# OUTPUTS: 0 = success 1 = failure
#
# ************************************************************
 proc smartest_operator { equip_id message param } {
  global $equip_id
  set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
        switch $param {      LOGIN   {
            smartest_debug "changing state user_idle for
        $equip_id"
                          equip_log $equip_id $eq_timestamp 0
        smartest_operator_login  \
                               [set ${equip_id}(state)] user_idle
        smartest_operator:LOGIN $message
          updateStateTime $equip_id [set ${equip_id}(state)]  0
        $eq_timestamp
            ilar_change_state $equip_id user_idle
           updateStateTime $equip_id user_idle $eq_timestamp
        0
         }   LOGOUT {
            smartest_debug "changing state os_idle for
        $equip_id"
           equip_log $equip_id $eq_timestamp 0
        smartest_operator_logout  \
                               [set ${equip_id}(state)] os_idle
        smartest_operator:LOGOUT $message
          updateStateTime $equip_id [set ${equip_id}(state)] 0
        $eq_timestamp
            ilar_change_state $equip_id os_idle
           updateStateTime $equip_id os_idle $eq_timestamp    0
         }    }  return 0}
# ************************************************************
# Description:
#           this function responds to the
```

```
#             smartest_smartest_start
#             & smartest_smartest_end triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
#          message from trigger
#          param : special parameter defined in the
#          ILAR Maintenance GUI.
#         param can be START END
#
# OUTPUTS: 0 = success 1 = failure
#
# ******************************************************
proc smartest_smartest { equip_id message param } {
 global $equip_id
 set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
   switch $param {      START   { smartest_debug "changing
      state smartest_idle for $equip_id"
              equip_log $equip_id $eq_timestamp 0
      smartest_smartest_start \
                              [set ${equip_id}(state)] smartest_idle
      smartest_smartest:START $message
        updateStateTime $equip_id [set ${equip_id}(state)] 0
      $eq_timestamp
         ilar_change_state $equip_id smartest_idle
        updateStateTime $equip_id smartest_idle
      $eq_timestamp   0
       }    EXIT    { smartest_debug "changing state user_idle
      for $equip_id"
        equip_log $equip_id $eq_timestamp 0
      smartest_smartest_exit \
                              [set ${equip_id}(state)] user_idle
      smartest_smartest:EXIT $message
        updateStateTime $equip_id [set ${equip_id}(state)] 0
      $eq_timestamp
         ilar_change_state $equip_id user_idle
        updateStateTime $equip_id user_idle $eq_timestamp
      0
       }      } return 0}
# ******************************************************
# Description:
#             this function responds to the smartest_testflow_start
#             & smartest_testflow_end triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
#          message from trigger
```

```
#       param : special parameter defined in the
#       ILAR Maintenance GUI.
#      param can be START END
#
# OUTPUTS: 0 = success 1 = failure
#
# ******************************************************
proc smartest_testflow { equip_id message param } {
 global $equip_id

 set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
 switch $param {     START   {
          smartest_debug "changing state testing for $equip_id"
                    equip_log $equip_id $eq_timestamp 0
       smartest_on_test_flow_start  \
                    [set ${equip_id}(state)] testing
       smartest_testflow:START $message
         updateStateTime $equip_id [set ${equip_id}(state)]  0
       $eq_timestamp
          ilar_change_state $equip_id testing
         updateStateTime $equip_id testing $eq_timestamp   0
       }   END   {
            set current_state [set ${equip_id}(state)]
            if { $current_state != "no_wip" } {
                    set new_state indexing
                  } else {
                    set new_state no_wip
                  }
          smartest_debug "changing state $new_state for
       $equip_id"
                    equip_log $equip_id $eq_timestamp 0
       smartest_on_test_flow_end  \
                    [set ${equip_id}(state)] $new_state
       smartest_testflow:END $message
         updateStateTime $equip_id [set ${equip_id}(state)]  0
       $eq_timestamp
          ilar_change_state $equip_id $new_state
         updateStateTime $equip_id $new_state
       $eq_timestamp   0
        }     } return 0}
# ******************************************************
# Description:
#           this function responds to the
#            smartest_testprogram_start
#            & smartest_testprogram_end triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
```

```
#         message from trigger
#         param : special parameter defined in the
#         ILAR Maintenance GUI.
#        param can be START END
#
# OUTPUTS: 0 = success 1 = failure
#
# ******************************************************
 proc smartest_testprogram { equip_id message param } {
  global $equip_id
  set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
    switch $param {      START   {
            smartest_debug "changing state setup for $equip_id"
                        equip_log $equip_id $eq_timestamp 0
        smartest_test_program_start  \
                                [set ${equip_id}(state)] setup
        smartest_testprogram:START  $message
          updateStateTime $equip_id [set ${equip_id}(state)]  0
        $eq_timestamp
            ilar_change_state $equip_id setup
          updateStateTime $equip_id setup $eq_timestamp   0
         }   END   {
              set current_state [set ${equip_id}(state)]
                        if { $current_state != "no_wip" } {
                          set new_state smartest_idle
                        } else {
                          set new_state no_wip
                        }
            smartest_debug "changing state $new_state for
        $equip_id"
                        equip_log $equip_id $eq_timestamp 0
        smartest_test_program_end  \
                                [set ${equip_id}(state)] $new_state
        smartest_testprogram:END  $message
          updateStateTime $equip_id [set ${equip_id}(state)]  0
        $eq_timestamp
            ilar_change_state $equip_id $new_state
          updateStateTime $equip_id $new_state
        $eq_timestamp   0
         }      }   return 0}
# ******************************************************
# Description:
#             this function responds to the smartest_no_wip
#            trigger
#
#
#
# INPUTS:  equipment_id (usually hostname)
#         message from trigger
```

```
#
#
# OUTPUTS: 0 = success 1 = failure
#
# ******************************************************

proc smartest_no_wip { equip_id message } {
 global $equip_id
 set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]

 smartest_debug "changing state no_wip for $equip_id"
 equip_log $equip_id $eq_timestamp 0 smartest_no_wip \
                              [set ${equip_id}(state)] no_wip
      smartest_no_wip:  $message
 updateStateTime $equip_id [set ${equip_id}(state)]  0
      $eq_timestamp
 ilar_change_state $equip_id no_wip
 updateStateTime $equip_id no_wip $eq_timestamp   0

 return 0
}
#************************************************************
# Description:
#            this function responds to the smartest_wafer_start
#            & smartest_wafer_end triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
#         message from trigger
#         param : special parameter defined in the
#         ILAR Maintenance GUI.
#        param can be START END
#
# OUTPUTS: 0 = success 1 = failure
#
# ******************************************************
 proc smartest_wafer { equip_id message param } {
 global $equip_id
 set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
 switch $param {      START   {
           smartest_debug "changing state setup"
                        equip_log $equip_id $eq_timestamp 0
      smartest_on_wafer_start  \
                              [set ${equip_id}(state)] setup
      smartest_wafer:START  $message
        updateStateTime $equip_id [set ${equip_id}(state)]  0
      $eq_timestamp
          ilar_change_state $equip_id setup
```

```
            updateStateTime $equip_id setup $eq_timestamp  0
          }  END  {
            smartest_debug "changing state setup"
                        equip_log $equip_id $eq_timestamp 0
        smartest_on_wafer_end  \
                              [set ${equip_id}(state)] setup
        smartest_wafer:END  $message
          updateStateTime $equip_id [set ${equip_id}(state)]  0
        $eq_timestamp
            ilar_change_state $equip_id setup
          updateStateTime $equip_id setup $eq_timestamp  0
         }     }  return 0}
#*************************************************************
        # Description:
#           this function responds to the
#            smartest_workstation_up
#            & smartest_workstation_down triggers
#
#
#
# INPUTS:  equipment_id (usually hostname)
#         message from trigger
#         param : special parameter defined in the
#         ILAR Maintenance GUI.
#        param can be UP DOWN
#
# OUTPUTS: 0 = success 1 = failure
#
# *******************************************************
proc smartest_workstation { equip_id message param } {
 global $equip_id
 set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
 switch $param {     UP  {
          smartest_debug "changing state up for $equip_id"
         equip_log $equip_id $eq_timestamp 0
        smartest_workstation_up  \
                              [set ${equip_id}(state)] os_idle
        smartest_workstation:UP  $message
         updateStateTime $equip_id [set ${equip_id}(state)]  0
        $eq_timestamp
          ilar_change_state $equip_id os_idle
         updateStateTime $equip_id os_idle $eq_timestamp  0
       }  DOWN {
         smartest_debug "changing state down for $equip_id"
                    equip_log $equip_id $eq_timestamp 0
        smartest_workstation_down  \
                              [set ${equip_id}(state)] ws_off
        smartest_workstation:DOWN  $message
```

```
            updateStateTime $equip_id [set ${equip_id}(state)]  0
           $eq_timestamp
             ilar_change_state $equip_id ws_off
            updateStateTime $equip_id ws_off $eq_timestamp   0
        }
    }
    return 0}
    #*************************************************
    # Description:
    #            this function responds to the smartest_default
    #           trigger.
    #            the default trigger is used to act on to unexpected
    #            triggers in a given state. If the trigger "default"
    #            is not specified for a given state, ilar will re-run
    #            the differentiae for the equipment in order to
    #            determine the current state.
    #
    #
    # INPUTS:  equipment_id (usually hostname)
    #         message from trigger
    #         param : special parameter defined in the
    #          ILAR Maintenance GUI.
    #         param not defined at this point
    #
    # OUTPUTS: 0 = success 1 = failure
    #
    # *******************************************************
    proc smartest_default { equip_id message } {
     global $equip_id
     set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]
       smartest_debug "executing default code for $equip_id"
     set current_state [set ${equip_id}(state)]
     #this code is to make sure that we update the current
     #state time information before switching to another state
     updateStateTime $equip_id [set ${equip_id}(state)]  0
          $eq_timestamp
     ilar_execute_differentiae $equip_id smartest
     equip_log $equip_id $eq_timestamp 0 default \
                                  $current_state [set
          ${equip_id}(state)] default: $message
     return 0 }
    # *******************************************************
    # Description:
    #            this function responds to the smartest_ph_error
    #            & smartest_ph_resume triggers. It is used to
    #            detect errors with prober/handler equipments.
    #
    #            smartest_ph_error  : carries an error code or
    #                        a specific condition that
```

```
#                       has halted the handler or prober
#                       like a jam, input cover open, timeout etc.
#            smartest_ph-resume : trigger sent to ilar when the
#                       error condition has been cleared. The
#                       prober orhandler will resume its
#                       operation and the program will continue.
#
#
#
# INPUTS:  equipment_id (usually hostname)
#      message from trigger (will contain an error code and
#       some text that describes the prober/handler condition)
#       param : special parameter defined in the
#       ILAR Maintenance GUI.
#      param can be ERROR RESUME
#
# OUTPUTS: 0 = success 1 = failure
#
# ******************************************************
proc smartest_ph { equip_id message param } {

 global $equip_id
 set eq_timestamp [lindex [lindex [lindex $message 0] 1] 0]

 switch $param {

  ERROR  {
         smartest_debug "changing state waiting_ph for
     $equip_id"
        equip_log $equip_id $eq_timestamp 0
     smartest_ph_error  \
                        [set ${equip_id}(state)] waiting_ph
     smartest_ph:ERROR $message
       updateStateTime $equip_id [set ${equip_id}(state)]  0
     $eq_timestamp
         ilar_change_state $equip_id waiting_ph
        updateStateTime $equip_id waiting_ph
     $eq_timestamp   0
      }
  RESUME  {
         smartest_debug "changing state testing for $equip_id"
        equip_log $equip_id $eq_timestamp 0
     smartest_ph_resume  \
                        [set ${equip_id}(state)] testing
     smartest_ph:RESUME $message
       updateStateTime $equip_id [set ${equip_id}(state)]  0
     $eq_timestamp
         ilar_change_state $equip_id testing
        updateStateTime $equip_id testing $eq_timestamp   0
```

```
        }

    }

    return 0
}
#############################################
        #
# helper functions for the differentia procedures for SmarTest
# testers
#############################################
        #
# returns 0 if success
# returns 1 if failure
#*************************************************************
# Description:
#             this function executes a remote command on the
#             equipment specified by equip_id. The procedure
#             drops a message in a mail box called
#               ILAR_COMMAND_name_of_equipment. The
#             equipmentruns a process called local_ilar (a dmh82
#             shell) which is programmed to respond to such
#             messages and put the reply in the reply mail box.
#             This function is at the heart of most tcl differentiae
#             that are used to determine the equipment state.
#              Remote commands can be commands such at "ps" to
#             verify if a given process is in memory.
#
#
# INPUTS:  equip_id (usually hostname)
#
#         local_command : any valid unix command or custom
        program
#
# OUTPUTS: value returned by the command. 1 if failure.
#
# *****************************************************
proc equipment_exec {equip_id local_command} {
 global $equip_id

 mbx flush "ILAR_COMMAND_$equip_id"
 mbx flush "ILAR_COMMAND_REPLY_$equip_id"
 if { [catch { mbx putr "ILAR_COMMAND_$equip_id"
        "ILAR_COMMAND_REPLY_$equip_id"
        "$local_command" } catch_error ]} {
   global errorInfo
     mbx flush "ILAR_COMMAND_$equip_id"
     mbx flush "ILAR_COMMAND_REPLY_$equip_id"
    #logErrors ....
```

```
  return 1
 }
  if { [catch { set reply [mbx_timed_get
        "ILAR_COMMAND_REPLY_$equip_id" 10000] }
        catch_error ] } {
   global errorInfo
     mbx flush "ILAR_COMMAND_$equip_id"
     mbx flush "ILAR_COMMAND_REPLY_$equip_id"
   #logErrors ....
  return 1
 }
  mbx flush "ILAR_COMMAND_$equip_id"
  mbx flush "ILAR_COMMAND_REPLY_$equip_id"
 return $reply
}
#*************************************************************
# Description:
#            this function uses the ping command to verify if
#            at least the equipment is turned on and connected
#            to the network.
#
#
# INPUTS:  equip_id (usually hostname)
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ***********************************************************
proc isEquipmentUp { equip_id } {
 global $equip_id
 global tcl_platform
 #use the ping command for now.  Need to improve this code
 #since we read the string result from the command, which
       varies from
 #one operating system to another (and probably one version of
       an operating
 #system to another version!
 if {$tcl_platform(platform) == "unix"} {
   set ping_command "/usr/sbin/ping"
 } else {
   set ping_command "ping"
 }

 if { [catch {set reply [exec $ping_command $equip_id -n 1]}
       catch_error] } {
   global errorInfo
   #logErrors ...
  return 1
 }
```

```
 if { ([string first "Request timed out"  $reply] > -1) || \
    ([string first "Bad IP address"  $reply] > -1)    || \
        ([string first "unknown host"  $reply] > -1)      || \
        ([string first "0 packets received"  $reply] > -1)} {
  return 1
 } else {
  return 0
 }
}
#*********************************************************
# Description:
#            this function returns the name of the user currently
#            logged on. The local_ilar process on the equipment
#            is present when the user has logged on but is
#            removed from memory when the user logs off. If the
#             communication times out with local_ilar (assuming
#            it did not go down), we conclude that the user has
#           logged off.
#
#
# INPUTS:  equip_id (usually hostname)
#
#
# OUTPUTS: user name. 1 if failure.
#
# *****************************************************
proc isUserLoggedIn {equip_id } {
 global $equip_id
 set reply [equipment_exec $equip_id "exec whoami"]
 return $reply
}
#*********************************************************
# Description:
#            this function verifies if smartest is running on
#           the equipment.
#
#
# INPUTS:  equip_id (usually hostname)
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# *****************************************************
proc isSmarTestRunning  {equip_id} {
 global $equip_id
 #check for /hp83000/fw/bin/mcd
 set reply [equipment_exec $equip_id "isProcessRunning
      mcd"]
```

```
 return $reply
}
#*********************************************************
# Description:
#           this function verifies if calibration is in progress
#
#
# INPUTS:  equip_id (usually hostname)
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# *******************************************************
proc isCalibrationInProgress  {equip_id} {
 global $equip_id
 #checks for /hp83000/pws/bin/ka

 set reply [equipment_exec $equip_id "isProcessRunning ka"]

 return $reply
}
#*********************************************************
# Description:
#           this function verifies if diagnostic is in progress
#
#
# INPUTS:  equip_id (usually hostname)
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# *******************************************************
proc isDiagnosticInProgress  {equip_id} {
 global $equip_id
 #checks for /hp83000/pws/bin/di

 set reply [equipment_exec $equip_id "isProcessRunning di"]

 return $reply
}
#*********************************************************
# Description:
#           this function verifies if test is in progress
#
#
# INPUTS:  equip_id (usually hostname)
#
#
```

```
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc isTestInProgress { equip_id } {
  global $equip_id
  set reply [equipment_exec $equip_id "isProcessRunning
        ftest"]

  return $reply
}
#*************************************************************
# Description:
#             this function verifies if the equipment is paused
#
#
# INPUTS:  equip_id (usually hostname)
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc isEquipPaused { equip_id } {
  global $equip_id
  set reply [equipment_exec $equip_id "exec
        /opt/ilar/f330/smartest_queries/get_flag"]

  if {$reply == 1} {
    return 0
  } else {
    return 1
  }
}

##############################################
        #
# differentiae
#
# The differentiae are used to determine the
# current equipment state. Above in this module,
# you will find a number of helper functions that
# called by the differentiae to determine the state.
##############################################
        ####
#*************************************************************
# Description:
#             this function runs the calibration differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
```

```
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc smartest_diff_calibrating { equip_id param1 } {
 global $equip_id
      smartest_debug "running diff_calibrating for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                              TBD TBD
       smartest_diff_calibrating:$param1 NONE

 set status [isCalibrationInProgress $equip_id]
 if {$status == 1} {
  updateStateTime $equip_id calibrating [clock second] 0
  return 0
 } else {
  return 1
 } }
#************************************************************
# Description:
#            this function runs the diagnostic differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc smartest_diff_diagnosing { equip_id param1 } {
 global $equip_id
      smartest_debug "running diff_diagnosing for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                              TBD TBD
       smartest_diff_diagnosing:$param1 NONE
 set status [isDiagnosticInProgress $equip_id]
 if {$status == 1} {
  updateStateTime $equip_id diagnosing [clock second] 0
  return 0
 } else {
  return 1
 }
    }
#************************************************************
# Description:
#            this function runs the indexing differentia.
```

```
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc smartest_diff_indexing { equip_id param1 } {
 global $equip_id
     smartest_debug "running diff_indexing for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                                TBD TBD
       smartest_diff_indexing:$param1  NONE
 return 0  }
#*************************************************************
# Description:
#            this function runs the pause differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc smartest_diff_pause { equip_id param1 } {
 global $equip_id
     smartest_debug "running diff_pause for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                                TBD TBD
       smartest_diff_pause:$param1  NONE
 #at this point, we assume a user has logged in
 #now check if smartest is up
 set status [isEquipPaused $equip_id]
 if {$status == 0} {
  updateStateTime $equip_id pause [clock second]  0
  return 0
 } else {
  #OK at this point we know SmarTest is testing but we need
  #to determine if it's in a pause mode  or setup
  #or calibrating or diagnosing!

  return 1
 }
 return 0  }
#*************************************************************
```

```
# Description:
#             this function runs the setup differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc smartest_diff_setup { equip_id param1 } {
 global $equip_id
      smartest_debug "running diff_setup for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                              TBD TBD
       smartest_diff_setup:$param1 NONE

 return 1  }
#*************************************************************
# Description:
#             this function runs the smartest idle differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc smartest_diff_smartest_idle { equip_id param1 } {
 global $equip_id

 smartest_debug "running diff_smartest_idle for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                              TBD TBD
       smartest_diff_smartest_idle:$param1 NONE

 #check if calibration is in progress
 set status_cal [isCalibrationInProgress $equip_id]
 if {$status_cal == 0} {
       smartest_debug "***changing state to calibrating"
  updateStateTime $equip_id calibrating [clock second] 0
  return calibrating
 }

 #check if diagnostic is in progress
 set status_diag [isDiagnosticInProgress $equip_id]
```

```
if {$status_cal == 0} {
      smartest_debug "***changing state to diagnosing"
  updateStateTime $equip_id diagnosing [clock second] 0
  return diagnosing
}
#now check if smartest doing something
set status [isTestInProgress $equip_id]
if {$status == 1} {

  #otherwise smartest is really not doing anything!
  updateStateTime $equip_id smartest_idle [clock second] 0
  return 0

} else {
  #OK at this point we know SmarTest is in setup or testing
  #further down the list of states, the engine will
  #try to determine what exactly it's doing

  return 1
}
}
#*********************************************************
# Description:
#           this function runs the test differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# *******************************************************
proc smartest_diff_testing { equip_id param1 } {
 global $equip_id
    smartest_debug "running diff_testing for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                          TBD TBD
      smartest_diff_testing:$param1 NONE

 updateStateTime $equip_id testing [clock second] 0    return 0
      }
#*******************************************************
# Description:
#           this function runs the user idle differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
```

```
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ********************************************************
proc smartest_diff_user_idle { equip_id param1 } {
 global $equip_id

  smartest_debug "running diff_user_idle for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                          TBD TBD
       smartest_diff_user_idle:$param1 NONE
 #at this point, we assume a user has logged in
 #now check if smartest is up
 set status [isSmarTestRunning $equip_id]
 if {$status == 1} {
  updateStateTime $equip_id user_idle [clock second] 0
  return 0
 } else {
  return 1
 }
 }
#*******************************************************
# Description:
#            this function runs the waiting for prober/handler
#           differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ********************************************************
proc smartest_diff_waiting_ph { equip_id param1 } {

 global $equip_id
  smartest_debug "running diff_waiting_ph for $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                          TBD TBD
       smartest_diff_waiting_ph:$param1 NONE
 return 0  }
#*************************************************************
# Description:
#            this function runs the workstation down differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
```

```
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc smartest_diff_workstation_down { equip_id param1 } {

 global $equip_id

 smartest_debug "running diff_workstation_down for
      $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                    TBD TBD
      smartest_diff_workstation_down:$param1 NONE
 set status [isEquipmentUp $equip_id]
 #0 = equipment is up, so it's not down!
 if {$status == 0} {
  return 1
 } else {
  updateStateTime $equip_id ws_off [clock second] 0
  return 0
 } }
#************************************************************
# Description:
#          this function runs the workstation up differentia.
#
#
# INPUTS:  equip_id (usually hostname) param1 not used for
# now.
#
#
# OUTPUTS: 0 if success. 1 if failure.
#
# ******************************************************
proc smartest_diff_workstation_up { equip_id param1 } {

 global $equip_id
 smartest_debug "running diff_workstation_up for
      $equip_id"
 equip_log $equip_id [clock second] 0 diffentiae \
                    TBD TBD
      smartest_diff_workstation_up:$param1 NONE
 set user [isUserLoggedIn $equip_id]
 #if different than 1 or TIMEOUT, it means that the
      local_ILAR client
 #has been started, then a user has logged in.
 if {($user == 1)| | ($user == "TIMEOUT") } {
  updateStateTime $equip_id os_idle [clock second] 0
```

```
    return 0
  } else {
    return 1
  }
    }
```

Table 3

```
#!/bin/sh
# restart using dmh82 \
exec dmh82_client -f "$0" "$@"
global reconnect_to_ilar
# return  0 if calibration is running
# returns 1 if not
proc isCalibrationRunning { } {
   # checks for /hp83000/pws/bin/ka
   return [isProcessRunning ka]
}
# return  0 if diagnostic is running
# returns 1 if not
proc isDiagnosticRunning { } {
   # checks for /hp83000/pws/bin/di
   return [isProcessRunning di]
}
# return  0 if SmarTest is running
# returns 1 if not
proc isSmarTestRunning { } {
  #check for /hp83000/fw/bin/mcd
  return [isProcessRunning mcd]
}
# return  0 if the ftest process is running
# if ftest is running, this means that the test program has
      started
# and is , at a minimum, loading the setup files before getting
      in the
# application model and running the parts.
# returns 1 if not
proc is_FTEST_Running { } {
   return [isProcessRunning ftest]
}
proc sendSmarTestExitMsg { } {
  global reconnect_to_ilar
 if { [isSmarTestRunning] == 1 } {
   if { $reconnect_to_ilar == 1 } {
     after 10000 sendSmarTestStartedMsg
     return
   }
   set hostname [lindex [split [info host] .] 0]
```

```
      mbx put $hostname "$hostname smartest_smartest_exit
          {{smartestexit} {[clock second] [clock clicks]}}"
     after 10000 sendSmarTestStartedMsg
    } else {
     after 10000 sendSmarTestExitMsg
    }

}
proc sendSmarTestStartedMsg { } {
  global reconnect_to_ilar
 if { [isSmarTestRunning] == 0 } {
  if { $reconnect_to_ilar == 1 } {
   after 10000 sendSmarTestExitMsg
   return
  }
  set hostname [lindex [split [info host] .] 0]
   mbx put $hostname "$hostname smartest_smartest_start
       {{smarteststarted} {[clock second] [clock clicks]}}"
  after 10000 sendSmarTestExitMsg
 } else {
  after 10000 sendSmarTestStartedMsg
 }

}
proc sendTestProgramEndMsg { } {
  global reconnect_to_ilar
 if { [is_FTEST_Running] == 1 } {
  if { $reconnect_to_ilar == 1 } {
   after 10000 sendTestProgramStartedMsg
   return
  }
  set hostname [lindex [split [info host] .] 0]
  mbx put $hostname "$hostname
      smartest_test_program_end {{testprogramend} {[clock
      second] [clock clicks]}}"
  after 10000 sendTestProgramStartedMsg
 } else {
  after 10000 sendTestProgramEndMsg
 }

}
proc sendTestProgramStartedMsg { } {
  global reconnect_to_ilar
 if { [is_FTEST_Running] == 0 } {
  if { $reconnect_to_ilar == 1 } {
   after 10000 sendTestProgramEndMsg
   return
  }
  set hostname [lindex [split [info host] .] 0]
```

```
    mbx put $hostname "$hostname
        smartest_test_program_start {{testprogramstarted}
        {[clock second] [clock clicks]}}"
    after 10000 sendTestProgramEndMsg
  } else {
    after 10000 sendTestProgramStartedMsg
  }

}
proc sendCalibrationEndMsg { } {
  global reconnect_to_ilar
  if { [isCalibrationRunning] == 1 } {
    if { $reconnect_to_ilar == 1 } {
      after 10000 sendCalibrationStartedMsg
      return
    }
    set hostname [lindex [split [info host] .] 0]
     mbx put $hostname "$hostname smartest_calibration_end
        {{calibrationend} {[clock second] [clock clicks]}}"
    after 10000 sendCalibrationStartedMsg
  } else {
    after 10000 sendCalibrationEndMsg
  }

}
proc sendCalibrationStartedMsg { } {
  global reconnect_to_ilar
  if { [isCalibrationRunning] == 0 } {
    if { $reconnect_to_ilar == 1 } {
      after 10000 sendCalibrationEndMsg
      return
    }
    set hostname [lindex [split [info host] .] 0]
     mbx put $hostname "$hostname smartest_calibration_start
        {{calibrationstarted} {[clock second] [clock clicks]}}"
    after 10000 sendCalibrationEndMsg
  } else {
    after 10000 sendCalibrationStartedMsg
  }

}
proc sendDiagnosticEndMsg { } {
  global reconnect_to_ilar
  if { [isDiagnosticRunning] == 1 } {
    if { $reconnect_to_ilar == 1 } {
      after 10000 sendDiagnosticStartedMsg
      return
    }
    set hostname [lindex [split [info host] .] 0]
```

```
        mbx put $hostname "$hostname smartest_diagnostic_end
            {{diagnosticend} {[clock second] [clock clicks]}}"
       after 10000 sendDiagnosticStartedMsg
      } else {
       after 10000 sendDiagnosticEndMsg
      }

     }
     proc sendDiagnosticStartedMsg { } {
      global reconnect_to_ilar
      if { [isDiagnosticRunning] == 0 } {
       if { $reconnect_to_ilar == 1 } {
        after 10000 sendDiagnosticEndMsg
        return
       }
       set hostname [lindex [split [info host] .] 0]
        mbx put $hostname "$hostname smartest_diagnostic_start
            {{diagnosticstarted} {[clock second] [clock clicks]}}"
       after 10000 sendDiagnosticEndMsg
      } else {
       after 10000 sendDiagnosticStartedMsg
      }

     }
     proc remote_command {message replybox} {

      puts "$message $replybox"
      if [catch {set reply [eval $message]} catch_reply] {
       set reply "ERROR: $catch_reply"
      }

      mbx put $replybox $reply

     }
     proc connect_to_ilar { } {
      global ILAR_MBX_SERVER
      global reconnect_to_ilar
      set equip_id [lindex [split [info host] .] 0]
      if { [catch { mbx init
            $ILAR_MBX_SERVER(hostname):$ILAR_MBX_SERV
            ER(group) } catch_error] } {
       global error_info
       set reconnect_to_ilar 1
       after 10000 connect_to_ilar
       return
      }
      set reconnect_to_ilar 0

      mbx flush "ILAR_COMMAND_$equip_id"
```

```
    mbx whenmsg "ILAR_COMMAND_$equip_id"
        {remote_command $mbxmsg $mbxreply ; mbx
        whenmsg again}
}
# try to reconnect if the server connection is lost
proc mh_app_lostserver {group connID} {
 #try to reconnect
  connect_to_ilar
}
load "/opt/ilar/local_ILAR/libutils.sl"
source /opt/ilar/config/ilar_server.tcl
set reconnect_to_ilar 0
after 10000 sendSmarTestStartedMsg
after 10000 sendTestProgramStartedMsg
after 10000 sendCalibrationStartedMsg
after 10000 sendDiagnosticStartedMsg
connect_to_ilar
```

The foregoing discussion discloses and describes merely exemplary methods and embodiments of the present invention. As will be understood by those familiar with the art, the invention may be embodied in other specific forms without departing from the spirit or essential characteristics thereof. Accordingly, the disclosure of the present invention is intended to be illustrative, but not limiting, of the scope of the invention, which is set forth in the following claims.



CLAIMS

We Claim:

1. A method for providing intelligent logic activity resolution for a process comprising the following step:

(a) performing the following substeps upon receiving a first trigger event (141):

(a.1) detecting a first subset of conditions of the process, the first subset of conditions being selected based on the first trigger event (141),

(a.2) when the first subset of conditions match predetermined values, entering a first state (111-113); and,

(a.3) when the first subset conditions do not match the predetermined values, performing the following substeps:

checking additional conditions, and

selecting a new state (111-113)to enter on the basis of the checked additional conditions.

2. A method as claimed in claim 1, wherein upon start-up performing the following:

checking conditions to select an initial state (111-113) to enter.

3. A method as claimed in claim 1 or claim 2, wherein in substep (a.1) the detection of the first subset of conditions is done by running a separate metaprocedure (142) to detect each condition in the first subset of conditions.

4. A method as claimed in claim 1, claim 2 or claim 3, additionally comprising the following condition:

(b) performing the following substeps upon receiving a second trigger event (141):

(b.1) detecting a second subset of conditions of the process, the second subset of conditions being selected based on the second trigger event (141),

(b.2) when the second subset of conditions match predetermined values, entering a corresponding state (111-113); and,

(b.3) when the second subset of conditions do not match the predetermined values, performing the following substeps:

checking additional conditions, and

selecting a new state (111-113) to enter on the basis of the checked additional conditions.

5. An intelligent logic activity resolution (ILAR) system for a process, the ILAR system comprising:

local ILAR (124) for monitoring process equipment to detect trigger

events (141);

ILAR execution engine (131), the ILAR execution engine (131), upon receiving notification of a first trigger event (141), detects a first subset of conditions of the process, the first subset of conditions being selected based on the first trigger event (141), when the first subset of conditions match predetermined values, the ILAR execution engine (131) enters a first state (111-113), and when the first subset conditions do not match the predetermined values, the ILAR execution engine (131) checks additional conditions, and on the basis of the checked additional conditions selects a new state (111-113) to enter.

6. An ILAR system as claimed in claim 5, wherein the ILAR execution engine (131) detects the first subset of conditions is done by running a separate metaprocedure (142) to detect each condition in the first subset of conditions.

7. An ILAR system as claimed in claim 5 or claim 6, wherein upon receiving a second trigger event (141) the ILAR execution engine (131) detects a second subset of conditions of the process, the second subset of conditions being selected based on the second trigger event (141), when the second subset of conditions match predetermined values, the ILAR execution engine (131) enters a first state (111-113), and when the second subset of conditions do not match the predetermined values, the ILAR

execution engine (131) checks additional conditions and on the basis of the checked additional conditions selects a new state (111-113) to enter.

8. An ILAR system as claimed in claim 5, claim 6 or claim 7, wherein the subset of conditions are also selected based on a current state (111-113) of the process.

9. An ILAR system as claimed in any of the claims 5 to 8, additionally comprising a graphics user interface (133) that allows a user to specify equipment types and trigger events (141) for the process, metaprocedures (142) and differentia.

10. An ILAR system as claimed in any of the claims 5 to 9, additionally comprising a distributed message hub (130) that facilitates communication between the ILAR execution engine (131) and the local ILAR (124).

11. A method for providing intelligent logic activity resolution for a process, as described herein with reference to Figures 2 to 5.

12. An intelligent logic activity resolution (ILAR) system for a process, as described herein with reference to Figures 2 to 5.