The ASTRAL Specifications of 8 Real-Time Systems

Paul Z. Kolano

University of California, Santa Barbara

Abstract

This report gives the ASTRAL specifications of 8 real-time systems. These systems include a bakery algorithm specification, a cruise control system, an elevator control system, an Olympic boxing scoring system, a phone system, a production cell, a railroad crossing system, and a stoplight control system.

1. Introduction

In order to determine the classification schemes that are most useful during analysis, a set of testbed systems was developed. These systems consist of a variety of different process and property types. The proofs of each system were performed to determine the proof patterns that occurred most often, which could then be reused in the proofs of other systems based on the classification schemes. The specifications that comprised the testbed varied from the specification of a distributed mutual exclusion protocol to a phone switching system to a production facility. More specifically, the specifications include a number of standard benchmark systems: the bakery specification describes the distributed mutual exclusion algorithm of [Lam 74], a cruise control system based on the description in [WM 85], an elevator control system adapted from [FF 84], the production cell specification based on the description in [LL 95], the railroad crossing system based on the description in [HL94], and a stoplight specification adapted from a stoplight control system described in [FF 84]. The testbed also includes the specification of an electronic scoring system for Olympic boxing based on a description of the system taken from the official 1996 Olympic web site [Oly 96]. Finally, the testbed includes a long distance telephony specification taken from [CGK 97]. In addition to their usefulness during the development of the systematic analysis methodology, as a side result, these systems demonstrate the flexibility and expressiveness of ASTRAL.

2. Bakery Algorithm

```
SPECIFICATION Bakery_Algorithm
  GLOBAL SPECIFICATION Bakery_Algorithm
    PROCESSES
              procs: array [ n_procs ]  of Proc
    TYPE
              procs_int: TYPEDEF i: integer ( 1 <= i
                                              & i <= n_procs ) ,
              nonneg_int: TYPEDEF i: integer ( i >= 0 ) ,
              pos_int: TYPEDEF i: integer ( i > 0 ) ,
              nonneg_real: TYPEDEF r: real ( r >= 0 ) ,
              pos_real: TYPEDEF r: real ( r > 0 )
    CONSTANT
              n_procs: pos_int,
              exec_time: pos_real
    SCHEDULE
      /* only a single proc can be in its critical region at any given time */
               ( FORALL i, j: procs_int
                          ( procs [ i ] .in_critical
                          & procs [ j ] .in_critical
                       ->   i = j )  )
  END Bakery_Algorithm
```

```
PROCESS SPECIFICATION Proc
  LEVEL Top_Level
    IMPORT
              pos_int, nonneg_int, nonneg_real, n_procs, exec_time, procs_int, procs, procs.choosing,
              procs.number, procs.in_critical
    EXPORT
              choosing, number, in_critical
    VARIABLE
              next_i: pos_int,
              choosing: boolean,
              number: nonneg_int,
              in_critical: boolean,
              delay: nonneg_real
    INITIAL
              next_i = 1
            & ~choosing
            & number = 0
            & ~in_critical
    INVARIANT
      /* must have finished loop to be in critical */
               ( in_critical
            ->   next_i > n_procs )
      /* when in loop, have already chosen a non-zero number */
            &  ( next_i > 1
            ->   ~choosing
               & number ~= 0 )
            &
      /* proc must have chosen a non-zero number to enter its critical region */
               ( in_critical
            ->   ~choosing
               & number ~= 0 )
      /* when number is changed to a non-zero value, the new value must be greater than or equal to all
         other numbers at the time that number "started changing" */
            &  ( Change ( number, now )
               & number ~= 0
            ->   FORALL i: procs_int
                          ( number >= past ( procs [ i ] .number + 1, now - exec_time )  )  )
      /* number can only change to zero when the proc has been in its critical region */
            &  ( Change ( number, now )
               & number = 0
            ->   ~in_critical
               & EXISTS t: time
                          ( Change [ 2 ]  ( number )  < t
                          & t < now
                          & past ( Change ( in_critical, t ) , t )
                          & past ( in_critical, t )  )  )
    SCHEDULE
      /* when a proc is in its critical region, its number and id must be the lowest of all procs with
         non-zero numbers */
               ( in_critical
            ->   FORALL i, j: procs_int
                          ( procs [ j ]  = self
                       ->   procs [ i ] .number = 0
                         |  number < procs [ i ] .number
                         |  number = procs [ i ] .number
                          & j < i )  )
      /* inductive loop invariant used to prove above */
            &  ( Start ( for_loop, now )
            ->   FORALL i, j: procs_int
                          ( procs [ j ]  = self
                          & i < next_i
                       ->   procs [ i ] .number = 0
                         |  number < procs [ i ] .number
                         |  number = procs [ i ] .number
                          & j <= i )  )
    IMPORTED VARIABLE
      /* proc must have chosen a non-zero number to enter its critical region */
               ( FORALL i: procs_int
                          ( procs [ i ] .in_critical
                       ->   ~procs [ i ] .choosing
                          & procs [ i ] .number ~= 0 )  )
      /* when number is changed to a non-zero value, the new value must be greater than or equal to all
         other numbers at the time that number "started changing" */
            &  ( FORALL i, j: procs_int
                          ( Change ( procs [ i ] .number, now )
                          & procs [ i ] .number ~= 0
                       ->   procs [ i ] .number >= past ( procs [ j ] .number + 1, now - exec_time )  )  )
      /* number can only change to zero when the proc has been in its critical region */
            &  ( FORALL i: procs_int
                          ( Change ( procs [ i ] .number, now )
                          & procs [ i ] .number = 0
                       ->   ~procs [ i ] .in_critical
                          & EXISTS t: time
                                     ( Change [ 2 ]  ( procs [ i ] .number )  < t
                                     & t < now
                                     & past ( Change ( procs [ i ] .in_critical, t ) , t )
                                     & past ( procs [ i ] .in_critical, t )  )  )  )
    TRANSITION set_choose
      ENTRY          [ TIME : exec_time ]
                now >= delay
              & ~choosing
              & number = 0
      EXIT
                choosing
```

```
      TRANSITION set_number
        ENTRY           [ TIME : exec_time ]
                  choosing
                & FORALL t: time
                           ( Change ( number, t )
                        ->   t < Change ( choosing )  )
        EXIT
                  FORALL i: procs_int
                           ( number >= procs [ i ] .number + 1 )
                & EXISTS i: procs_int
                           ( number = procs [ i ] .number + 1 )
      TRANSITION reset_choose
        ENTRY           [ TIME : exec_time ]
                  choosing
                & FORALL t: time
                           ( Change ( number, t )
                        ->   t > Change ( choosing )  )
        EXIT
                  ~choosing
      TRANSITION for_loop
        ENTRY           [ TIME : exec_time ]
                  next_i <= n_procs
                & ~choosing
                & number ~= 0
                & ~procs [ next_i ] .choosing
                &  ( procs [ next_i ] .number = 0
                  |  number < procs [ next_i ] .number
                  |  number = procs [ next_i ] .number
                   & FORALL j: procs_int
                              ( procs [ j ]  = self
                           ->   j <= next_i )  )
        EXIT
                  next_i = next_i' + 1
      TRANSITION start_critical
        ENTRY           [ TIME : exec_time ]
                  next_i > n_procs
                & ~in_critical
        EXIT
                  in_critical
      TRANSITION end_critical
        ENTRY           [ TIME : exec_time ]
                  in_critical
        EXIT
                  ~in_critical
                & next_i = 1
                & number = 0
                & delay >= now
    END Top_Level
  END Proc
END Bakery_Algorithm
```

3. Cruise Control

```
SPECIFICATION Cruise_Control
  GLOBAL SPECIFICATION Cruise_Control
    PROCESSES
              the_speed_control: Speed_Control,
              the_accelerometer: Accelerometer,
              the_speedometer: Speedometer,
              the_tire_sensor: Tire_Sensor
    TYPE
              nonneg_real: TYPEDEF r: real ( r >= 0 ) ,
              pos_real: TYPEDEF r: real ( r > 0 ) ,
              nonneg_int: TYPEDEF i: integer ( i >= 0 )
    CONSTANT
              tire_circumference: pos_real,
              sample_time: pos_real
  END Cruise_Control
  PROCESS SPECIFICATION Speed_Control
    LEVEL Top_Level
      IMPORT
                nonneg_real, pos_real, the_speedometer, the_speedometer.speed, the_accelerometer,
                the_accelerometer.acceleration
      EXPORT
                set_gas_pedal, set_brake_pedal, enable_cruise, disable_cruise, maintain_speed, resume_speed,
                begin_speed_increase, end_speed_increase
      CONSTANT
                input_dur, control_dur: pos_real,
                desired_acceleration: pos_real,
                full_throttle: pos_real,
                throttle_step: pos_real,
                speed_step: pos_real,
                increase_delay: pos_real
      VARIABLE
                throttle: nonneg_real,
                brake: nonneg_real,
                desired_speed: nonneg_real,
                cruise_on: boolean,
                maintaining_speed: boolean,
                increasing_speed: boolean,
                cruise_throttle: nonneg_real,
                foot_throttle: nonneg_real
```

```
INITIAL
          ~cruise_on
        & ~maintaining_speed
        & ~increasing_speed
        & throttle = 0
        & brake = 0
        & foot_throttle = 0
INVARIANT
  /* cruise control must be on to maintain speed */
           ( maintaining_speed
        ->   cruise_on )
        &
  /* must be maintaining speed to increase desired speed */
           ( increasing_speed
        ->   maintaining_speed )
        &
  /* when maintaining speed, throttle will be the higher of the pedal and the cruise throttle */
           ( maintaining_speed
           & foot_throttle < cruise_throttle
        ->   throttle = cruise_throttle )
        &  ( maintaining_speed
           & foot_throttle > cruise_throttle
        ->   throttle = foot_throttle )
        &
  /* when not maintaining speed, throttle is equal to foot throttle */
           ( ~maintaining_speed
        ->   throttle = foot_throttle )
SCHEDULE
  /* cruise control will stop maintaining speed as quickly as possible when the brake is applied */
           ( control_dur <= input_dur
           & now >= input_dur + input_dur
           & past ( maintaining_speed, now - input_dur - input_dur )
           & Call ( set_brake_pedal, now - input_dur - input_dur )
        ->   EXISTS t: time
                      ( now - input_dur - input_dur <= t
                      & t <= now
                      & ~past ( maintaining_speed, t )  )  )
        &  ( input_dur <= control_dur
           & now >= input_dur + control_dur
           & past ( maintaining_speed, now - input_dur - control_dur )
           & Call ( set_brake_pedal, now - input_dur - control_dur )
        ->   EXISTS t: time
                      ( now - input_dur - control_dur <= t
                      & t <= now
                      & ~past ( maintaining_speed, t )  )  )
FURTHER ASSUMPTIONS #1
  FURTHER PROCESS ASSUMPTIONS
    TRANSITION SELECTION
              enabled_transitions CONTAINS  { set_brake_pedal }
            & TRUE
         ->   elgible_transitions =  { set_brake_pedal }
TRANSITION set_gas_pedal ( v: pos_real )
  ENTRY           [ TIME : input_dur ]
            TRUE
  EXIT
            foot_throttle = v
          & IF
                    maintaining_speed'
                  & cruise_throttle' > foot_throttle
            THEN
                    throttle = cruise_throttle'
            ELSE
                    throttle = foot_throttle
            FI
TRANSITION set_brake_pedal ( v: pos_real )
  ENTRY           [ TIME : input_dur ]
            TRUE
  EXIT
            ~maintaining_speed
          & ~increasing_speed
          & throttle = foot_throttle'
          & brake = v
TRANSITION enable_cruise
  ENTRY           [ TIME : input_dur ]
            ~cruise_on
  EXIT
            cruise_on
TRANSITION disable_cruise
  ENTRY           [ TIME : input_dur ]
            cruise_on
  EXIT
            ~cruise_on
          & ~maintaining_speed
          & ~increasing_speed
          & throttle = foot_throttle'
TRANSITION maintain_speed
  ENTRY           [ TIME : input_dur ]
            cruise_on
          & ~maintaining_speed
  EXIT
            cruise_throttle = throttle'
          & desired_speed = the_speedometer.speed
          & maintaining_speed
```

```
    TRANSITION resume_speed
      ENTRY           [ TIME : input_dur ]
              cruise_on
            & ~maintaining_speed
      EXIT
              maintaining_speed
            & IF
                      cruise_throttle’ > foot_throttle’
              THEN
                      throttle = cruise_throttle’
              ELSE
                      throttle = foot_throttle’
              FI
    TRANSITION begin_speed_increase
      ENTRY           [ TIME : input_dur ]
              maintaining_speed
            & ~increasing_speed
      EXIT
              increasing_speed
            & desired_speed = desired_speed’ + speed_step
    TRANSITION end_speed_increase
      ENTRY           [ TIME : input_dur ]
              increasing_speed
      EXIT
              ~increasing_speed
    TRANSITION increase_speed
      ENTRY           [ TIME : control_dur ]
              increasing_speed
            & now - Change ( desired_speed )  >= increase_delay
      EXIT
              desired_speed = desired_speed’ + speed_step
    TRANSITION increase_throttle
      ENTRY           [ TIME : control_dur ]
              maintaining_speed
            &  ( desired_speed > the_speedometer.speed
               & the_accelerometer.acceleration < desired_acceleration
              |  desired_speed < the_speedometer.speed
               & the_accelerometer.acceleration <  - desired_acceleration )
      EXIT
              IF
                      cruise_throttle’ + throttle_step > full_throttle
              THEN
                      cruise_throttle = full_throttle
              ELSE
                      cruise_throttle = cruise_throttle’ + throttle_step
              FI
            & IF
                      cruise_throttle > foot_throttle’
              THEN
                      throttle = cruise_throttle
              ELSE
                      throttle = foot_throttle’
              FI
    TRANSITION decrease_throttle
      ENTRY           [ TIME : control_dur ]
              maintaining_speed
            &  ( desired_speed > the_speedometer.speed
               & the_accelerometer.acceleration > desired_acceleration
              |  desired_speed < the_speedometer.speed
               & the_accelerometer.acceleration >  - desired_acceleration )
      EXIT
              IF
                      cruise_throttle’ - throttle_step < 0
              THEN
                      cruise_throttle = 0
              ELSE
                      cruise_throttle = cruise_throttle’ - throttle_step
              FI
            & IF
                      cruise_throttle > foot_throttle’
              THEN
                      throttle = cruise_throttle
              ELSE
                      throttle = foot_throttle’
              FI
  END Top_Level
END Speed_Control
PROCESS SPECIFICATION Accelerometer
  LEVEL Top_Level
    IMPORT
            sample_time, the_speedometer, the_speedometer.speed
    EXPORT
            acceleration
    VARIABLE
            acceleration: real
    INITIAL
            acceleration = 0
    TRANSITION sample_acceleration
      ENTRY           [ TIME : sample_time ]
              TRUE
      EXIT
              IF
                      now < 2 * sample_time
              THEN
                      acceleration = the_speedometer.speed / sample_time
```

```
                 ELSE
                         acceleration =  ( the_speedometer.speed - Past ( the_speedometer.speed, now - 2 *
                                                                    sample_time )  )  / sample_time
                 FI
    END Top_Level
  END Accelerometer
  PROCESS SPECIFICATION Speedometer
    LEVEL Top_Level
      IMPORT
                nonneg_real, sample_time, tire_circumference, the_tire_sensor, the_tire_sensor.rotations
      EXPORT
                speed
      VARIABLE
                speed: nonneg_real
      INITIAL
                speed = 0
      TRANSITION sample_speed
        ENTRY           [ TIME : sample_time ]
                  TRUE
        EXIT
                  IF
                          now < 2 * sample_time
                  THEN
                          speed = the_tire_sensor.rotations * tire_circumference / sample_time
                  ELSE
                          speed =  ( the_tire_sensor.rotations - tire_circumference *
                            Past ( the_tire_sensor.rotations, now - 2 * sample_time )  )  * sample_time
                  FI
    END Top_Level
  END Speedometer
  PROCESS SPECIFICATION Tire_Sensor
    LEVEL Top_Level
      IMPORT
                nonneg_int, pos_real
      EXPORT
                rotate, rotations
      CONSTANT
                sense_time: pos_real
      VARIABLE
                rotations: nonneg_int
      INITIAL
                rotations = 0
      TRANSITION rotate
        ENTRY           [ TIME : sense_time ]
                  TRUE
        EXIT
                  rotations = rotations' + 1
    END Top_Level
  END Tire_Sensor
END Cruise_Control
```

4. Elevator Control System

```
SPECIFICATION Elevator_System
  GLOBAL SPECIFICATION Elevator_System
    PROCESSES
              the_elevator: Elevator,
              the_elevator_buttons: Elevator_Button_Panel,
              the_floor_buttons: array [ 1..n_floors ]  of Floor_Button_Panel
    TYPE
              pos_integer: TYPEDEF i: integer ( i > 0 ) ,
              pos_real: TYPEDEF r: real ( r > 0 ) ,
              floor: TYPEDEF i: pos_integer ( i <= n_floors )
    CONSTANT
              n_floors: pos_integer,
              request_dur, clear_dur: pos_real,
              t_service_request, t_move, t_stop, t_move_door: pos_real
    AXIOM
      /* clear_request must be able to fire no matter how many requests are made while the elevator door
         is opening */
               ( clear_dur + n_floors * request_dur < t_move_door )
      /* must be at least 2 floors in the building */
            &  ( n_floors >= 2 )
    SCHEDULE
      /* any request must be serviced within time t_service_request */
              FORALL f: floor
                       ( the_elevator_buttons.Call ( request_floor ( f ) , now - t_service_request )
                    ->   EXISTS t: time
                                  ( now - t_service_request < t
                                  & t <= now
                                  & the_elevator.position = f
                                  & Change ( the_elevator.door_open, t )
                                  & past ( the_elevator.door_open, t )  )  )
            & FORALL f: floor
                       ( f ~= n_floors
                       & the_floor_buttons [ f ] .Call ( request_up, now - t_service_request )
                    ->   EXISTS t: time
                                  ( now - t_service_request < t
                                  & t <= now
```

```
                              & the_elevator.position = f
                              & Change ( the_elevator.door_open, t )
                              & past ( the_elevator.door_open, t )
                              & past ( the_elevator.going_up, t )  )  )
          & FORALL f: floor
                     ( f ~= 1
                     & the_floor_buttons [ f ] .Call ( request_down, now - t_service_request )
                  ->   EXISTS t: time
                              ( now - t_service_request < t
                              & t <= now
                              & the_elevator.position = f
                              & Change ( the_elevator.door_open, t )
                              & past ( the_elevator.door_open, t )
                              & ~past ( the_elevator.going_up, t )  )  )
END Elevator_System
PROCESS SPECIFICATION Elevator
  LEVEL Top_Level
    IMPORT
              pos_real, floor, request_dur, the_elevator_buttons, the_floor_buttons, n_floors,
              the_elevator_buttons.floor_requested, the_floor_buttons.up_requested,
              the_floor_buttons.down_requested, t_stop, t_move, t_move_door t_service_request
    EXPORT
              position, going_up, door_open, moving, door_moving
    CONSTANT
              move_dur, arrive_dur, open_dur, close_dur, door_stop_dur: pos_real
    VARIABLE
              position: floor,
              going_up, door_open, moving, door_moving: boolean
    AXIOM
      /* t_service_request must be big enough to handle the worst case. One instance of the worst case
         is when the elevator is moving up from floor 1 to 2 and 2 has not been requested on the elevator
         panel nor has any request been made on 2's button panel. Let t_arrive be the next time such that
         End(arrive, t_arrive). up_request and down_request are simultaneously called on floor 2 an
         "instant" after t_arrive - 2 * request_dur and down_request fires first. In addition, every floor
         in the building (besides 2) has up_requested (except the top floor) and down_requested (except
         the bottom floor). Thus, the up request is not posted in time for the elevator to service it and
         the elevator must stop and open the door at every floor up to the top, back down to the bottom,
         and back up to 2. */
               ( t_service_request >= 2 * request_dur + move_dur + t_move + arrive_dur +
                    ( open_dur + t_move_door + door_stop_dur + t_stop + close_dur + t_move_door +
                        door_stop_dur + request_dur + move_dur + t_move + arrive_dur )  + open_dur +
                        t_move_door + door_stop_dur ) *  ( 2 * n_floors - 3 )
    DEFINE
              request_above ( f0: floor ) : boolean ==
                      EXISTS f: floor
                               ( f > f0
                               &  ( the_elevator_buttons.floor_requested ( f )
                                 |  the_floor_buttons [ f ] .up_requested
                                 |  the_floor_buttons [ f ] .down_requested )  ) ,
              request_below ( f0: floor ) : boolean ==
                      EXISTS f: floor
                               ( f < f0
                               &  ( the_elevator_buttons.floor_requested ( f )
                                 |  the_floor_buttons [ f ] .up_requested
                                 |  the_floor_buttons [ f ] .down_requested )  )
    INITIAL
              position = 1
            & going_up
            & ~door_open
            & ~moving
            & ~door_moving
    INVARIANT

      /* the elevator door must stay closed while the elevator is moving */
               ( moving
            ->   ~door_open
               & ~door_moving )
    CONSTRAINT
      /* if the elevator changes direction, there cannot be an outstanding request in the old direction
         */
               ( going_up
               & ~going_up'
            ->   ~request_below' ( position' )  )
            &  ( ~going_up
               & going_up'
            ->   ~request_above' ( position' )  )
    SCHEDULE
      /* if the elevator is moving in some direction, there must be an outstanding request in that
         direction */
               ( moving
               & going_up
            ->   request_above ( position )  )
            &  ( moving
               & ~going_up
            ->   request_below ( position )  )
      /* any request must be serviced within time t_service_request */
            &  ( FORALL f: floor
                          ( Change ( the_elevator_buttons.floor_requested ( f ), now - t_service_request +
                                       2 * request_dur )
                          & past ( the_elevator_buttons.floor_requested ( f ) , now - t_service_request +
                                     2 * request_dur )
                       ->   EXISTS t: time
                                     ( now - t_service_request + 2 * request_dur < t
                                     & t <= now
                                     & position = f
```

```
                                  & Change ( door_open, t )
                                  & past ( door_open, t )  )  )  )
        &  ( FORALL f: floor
                       ( f ~= n_floors
                       & Change ( the_floor_buttons [ f ] .up_requested, now - t_service_request +
                                    2 * request_dur )
                       & past ( the_floor_buttons [ f ] .up_requested, now - t_service_request + 2 *
                                  request_dur )
                    ->   EXISTS t: time
                                  ( now - t_service_request + 2 * request_dur < t
                                  & t <= now
                                  & position = f
                                  & Change ( door_open, t )
                                  & past ( door_open, t )
                                  & past ( going_up, t )  )  )
           & FORALL f: floor
                       ( f ~= 1
                       & Change ( the_floor_buttons [ f ] .down_requested, now - t_service_request +
                                    2 * request_dur )
                       & past ( the_floor_buttons [ f ] .down_requested, now - t_service_request +
                                  2 * request_dur )
                    ->   EXISTS t: time
                                  ( now - t_service_request + 2 * request_dur < t
                                  & t <= now
                                  & position = f
                                  & Change ( door_open, t )
                                  & past ( door_open, t )
                                  & ~past ( going_up, t )  )  )  )
IMPORTED VARIABLE
  /* buttons only clear after elevator has arrived and started opening the doors */
           ( FORALL f: floor
                       ( Change ( the_elevator_buttons.floor_requested ( f ) , now )
                       & ~the_elevator_buttons.floor_requested ( f )
                    ->   EXISTS t: time
                                  ( Change [ 2 ]  ( the_elevator_buttons.floor_requested ( f )  )  <
                                      t
                                  & t <= now
                                  & past ( position, t )  = f
                                  & ~past ( door_open, t )
                                  & past ( door_moving, t )  )  )  )
        &  ( FORALL f: floor
                       ( f ~= n_floors
                       & Change ( the_floor_buttons [ f ] .up_requested, now )
                       & ~the_floor_buttons [ f ] .up_requested
                    ->   EXISTS t: time
                                  ( Change [ 2 ]  ( the_floor_buttons [ f ] .up_requested )  < t
                                  & t <= now
                                  & past ( position, t )  = f
                                  & ~past ( door_open, t )
                                  & past ( door_moving, t )
                                  & past ( going_up, t )  )  )  )
        &  ( FORALL f: floor
                       ( f ~= 1
                       & Change ( the_floor_buttons [ f ] .down_requested, now )
                       & ~the_floor_buttons [ f ] .down_requested
                    ->   EXISTS t: time
                                  ( Change [ 2 ]  ( the_floor_buttons [ f ] .down_requested )  < t
                                  & t <= now
                                  & past ( position, t )  = f
                                  & ~past ( door_open, t )
                                  & past ( door_moving, t )
                                  & ~past ( going_up, t )  )  )  )
  /* the top floor never has an up request and the bottom floor never has a down request */
        &  ( ~the_floor_buttons [ n_floors ] .up_requested )
        &  ( ~the_floor_buttons [ 1 ] .down_requested )
TRANSITION move_up
  ENTRY           [ TIME : move_dur ]
            ~door_open
          & ~door_moving
          & request_above ( position )
          &  ( going_up
            |  ~going_up
             & ~request_below ( position )
             & ~the_floor_buttons [ position ] .up_requested )
          &  ( End ( arrive, now )
             & ~the_elevator_buttons.floor_requested ( position )
             & ~the_floor_buttons [ position ] .up_requested
            |  FORALL t, t1: time
                        ( Change ( moving, t )
                        & Change ( door_open, t1 )
                      ->   t < t1
                        & now >= t1 + request_dur )  )
  EXIT
            moving
          & going_up
TRANSITION move_down
  ENTRY           [ TIME : move_dur ]
            ~door_open
          & ~door_moving
          & request_below ( position )
          &  ( ~going_up
            |  going_up
             & ~request_above ( position )
             & ~the_floor_buttons [ position ] .down_requested )
          &  ( End ( arrive, now )
```

```
                 & ~the_elevator_buttons.floor_requested ( position )
                 & ~the_floor_buttons [ position ] .down_requested
                |  FORALL t, t1: time
                            ( Change ( moving, t )
                            & Change ( door_open, t1 )
                         ->   t < t1
                            & now >= t1 + request_dur )  )
      EXIT
                moving
              & ~going_up
    TRANSITION arrive
      ENTRY           [ TIME : arrive_dur ]
                moving
              & FORALL t: time
                         ( t <= now
                         &  ( End ( move_down, t )
                           |  End ( move_up, t )  )
                      ->   now - t_move >= t )
              & FORALL t, t1: time
                         ( t <= now
                         & End ( arrive, t )
                         &  ( End ( move_up, t1 )
                           |  End ( move_down, t1 )  )
                      ->   t < t1 )
      EXIT
                IF
                        going_up'
                THEN
                        position = position' + 1
                ELSE
                        position = position' - 1
                FI
    TRANSITION open_door
      ENTRY           [ TIME : open_dur ]
                ~door_open
              & ~door_moving
              &  ( ~moving
                |  moving
                 & EXISTS t: time
                            ( Change ( position, t )
                            & t > Change ( moving )  )  )
              &  ( the_elevator_buttons.floor_requested ( position )
                |  going_up
                 &  ( the_floor_buttons [ position ] .up_requested
                   |  ~request_above ( position )
                    & the_floor_buttons [ position ] .down_requested )
                |  ~going_up
                 &  ( the_floor_buttons [ position ] .down_requested
                   |  ~request_below ( position )
                    & the_floor_buttons [ position ] .up_requested )  )
      EXIT
                ~moving
              & door_moving
              & going_up =  ( going_up'
                            &  ( request_above' ( position' )
                              |  the_floor_buttons [ position' ] .up_requested' )
                           |  ~request_below' ( position' )
                            & ~the_floor_buttons [ position' ] .down_requested' )
    TRANSITION close_door
      ENTRY           [ TIME : close_dur ]
                door_open
              & ~door_moving
              & now - t_stop >= Change ( door_open )
      EXIT
                door_moving
    TRANSITION door_stop
      ENTRY           [ TIME : door_stop_dur ]
                door_moving
              & now - t_move_door >= Change ( door_moving )
      EXIT
                ~door_moving
              & door_open = ~door_open'
  END Top_Level
END Elevator
PROCESS SPECIFICATION Elevator_Button_Panel
  LEVEL Top_Level
    IMPORT
              floor, request_dur, clear_dur, the_elevator, the_elevator.position, the_elevator.door_open,
              the_elevator.door_moving
    EXPORT
              floor_requested, request_floor
    VARIABLE
              floor_requested ( floor ) : boolean
    ENVIRONMENT
      /* multiple button pushes should have no effect */
               ( FORALL f: floor
                          ( Change ( floor_requested ( f ) , now )
                          & ~floor_requested ( f )
                       ->   FORALL t: time
                                     ( Start ( request_floor ( f )  )  <= t
                                     & t <= now
                                  ->   ~Call ( request_floor ( f ) , t )  )  )  )
    INITIAL
              FORALL f: floor
                       ( ~floor_requested ( f )  )
```

```
    INVARIANT
      /* buttons only clear after elevator has arrived and started opening the doors */
               ( FORALL f: floor
                          ( Change ( floor_requested ( f ) , now )
                          & ~floor_requested ( f )
                       ->   EXISTS t: time
                                     ( Change [ 2 ]  ( floor_requested ( f )  )  < t
                                     & t <= now
                                     & past ( the_elevator.position, t )  = f
                                     & ~past ( the_elevator.door_open, t )
                                     & past ( the_elevator.door_moving, t )  )  )  )
    TRANSITION request_floor ( f: floor )
      ENTRY           [ TIME : request_dur ]
                ~floor_requested ( f )
      EXIT
                floor_requested ( f )
    TRANSITION clear_floor_request
      ENTRY           [ TIME : clear_dur ]
                floor_requested ( the_elevator.position )
              & ~the_elevator.door_open
              & the_elevator.door_moving
      EXIT
                ~floor_requested ( the_elevator.position )
  END Top_Level
END Elevator_Button_Panel
PROCESS SPECIFICATION Floor_Button_Panel
  LEVEL Top_Level
    IMPORT
              request_dur, clear_dur, n_floors, the_floor_buttons, the_elevator, the_elevator.position,
              the_elevator.door_open, the_elevator.going_up, the_elevator.door_moving
    EXPORT
              up_requested, down_requested, request_up, request_down
    VARIABLE
              up_requested, down_requested: boolean
    ENVIRONMENT
      /* multiple button pushes should have no effect */
               ( Change ( up_requested, now )
               & ~up_requested
            ->   FORALL t: time
                          ( Start ( request_up )  <= t
                          & t <= now
                       ->   ~Call ( request_up, t )  )  )
            &  ( Change ( down_requested, now )
               & ~down_requested
            ->   FORALL t: time
                          ( Start ( request_down )  <= t
                          & t <= now
                       ->   ~Call ( request_down, t )  )  )
      /* requests cannot be made of the elevator to stop at a floor between when the door starts opening
         on that floor until when it starts closing */
            &  ( Change ( the_elevator.door_moving, now )
               & the_elevator.door_moving
               & ~the_elevator.door_open
               & the_floor_buttons [ the_elevator.position ]  = Self
            ->   FORALL t: time
                          ( t >= Change [ 2 ]  ( the_elevator.door_moving )
                       ->    ( past ( the_elevator.going_up, t )
                          ->   ~Call ( request_up, t )  )
                          &  ( past ( ~the_elevator.going_up, t )
                          ->   ~Call ( request_down, t )  )  )  )
      /* assume that down arrow on first floor and up arrow on top floor have been physically covered
         and short circuited so that they are not available to the environment */
            &  (  ( the_floor_buttons [ 1 ]  = Self
               ->   ~Call ( request_down, now )  )
               &  ( the_floor_buttons [ n_floors ]  = Self
               ->   ~Call ( request_up, now )  )  )
    INITIAL
              ~up_requested
            & ~down_requested
    INVARIANT
      /* buttons only clear after elevator has arrived and started opening the doors */
               ( Change ( up_requested, now )
               & ~up_requested
            ->   EXISTS t: time
                          ( Change [ 2 ]  ( up_requested )  < t
                          & t <= now
                          & the_floor_buttons [ past ( the_elevator.position, t )  ]  = Self
                          & ~past ( the_elevator.door_open, t )
                          & past ( the_elevator.door_moving, t )
                          & past ( the_elevator.going_up, t )  )  )
            &  ( Change ( down_requested, now )
               & ~down_requested
            ->   EXISTS t: time
                          ( Change [ 2 ]  ( down_requested )  < t
                          & t <= now
                          & the_floor_buttons [ past ( the_elevator.position, t )  ]  = Self
                          & ~past ( the_elevator.door_open, t )
                          & past ( the_elevator.door_moving, t )
                          & ~past ( the_elevator.going_up, t )  )  )
      /* the top floor never has an up request and the bottom floor never has a down request */
            &  ( the_floor_buttons [ n_floors ]  = Self
            ->   ~up_requested )
            &  ( the_floor_buttons [ 1 ]  = Self
            ->   ~down_requested )
```

```
      SCHEDULE
        /* calls will be posted within 2 * request_dur time */
                 ( Call ( request_up, now - 2 * request_dur )
              ->   EXISTS t: time
                            ( now - 2 * request_dur < t
                            & t <= now
                            & past ( Change ( up_requested, t ) , t )
                            & past ( up_requested, t )  )  )
              &  ( Call ( request_down, now - 2 * request_dur )
              ->   EXISTS t: time
                            ( now - 2 * request_dur < t
                            & t <= now
                            & past ( Change ( down_requested, t ) , t )
                            & past ( down_requested, t )  )  )
      TRANSITION request_up
        ENTRY           [ TIME : request_dur ]
                  ~up_requested
                & the_floor_buttons [ n_floors ]  ~= Self
        EXIT
                  up_requested
      TRANSITION request_down
        ENTRY           [ TIME : request_dur ]
                  ~down_requested
                & the_floor_buttons [ 1 ]  ~= Self
        EXIT
                  down_requested
      TRANSITION clear_up_request
        ENTRY           [ TIME : clear_dur ]
                  up_requested
                & the_floor_buttons [ the_elevator.position ]  = Self
                & the_elevator.going_up
                & ~the_elevator.door_open
                & the_elevator.door_moving
        EXIT
                  ~up_requested
      TRANSITION clear_down_request
        ENTRY           [ TIME : clear_dur ]
                  down_requested
                & the_floor_buttons [ the_elevator.position ]  = Self
                & ~the_elevator.going_up
                & ~the_elevator.door_open
                & the_elevator.door_moving
        EXIT
                  ~down_requested
    END Top_Level
  END Floor_Button_Panel
END Elevator_System
```

5. Olympic Boxing Scoring System

```
SPECIFICATION Olympic_Boxing
  GLOBAL SPECIFICATION Olympic_Boxing
    PROCESSES
              Time_Keeper: Timer,
              Judges: array [ 1..5 ]  of Judge,
              Scorer: Tabulate
    TYPE
              Pos_Integer: TYPEDEF i: Integer ( i > 0 ) ,
              Non_Negative: TYPEDEF i: Integer ( i >= 0 ) ,
              Pos_Real: TYPEDEF r: Real ( r > 0 ) ,
              Name:  ( Fighter1, Fighter2, None ) ,
              Boxer: TYPEDEF n: Name ( n ~= None ) ,
              Judge_ID: TYPEDEF i: Pos_Integer ( i <= 5 ) ,
              Set_Of_Judge_ID: SET OF Judge_ID,
              Decision:  ( In_Progress, Win, Draw )
    CONSTANT
              Num_Rounds: Pos_Integer, /* 3 */
              Window: Pos_Real /* 1 second */
    ENVIRONMENT
              FORALL t: Time, j: Judge_ID, B: Boxer
                       ( t <= Now - Window
                       & past ( Judges [ j ] .Call ( Score ( B ) , t ) , t )
                       & FORALL t1: Time, i: Judge_ID
                                  ( t1 >= t - Window
                                  & t1 < t
                               ->   ~past ( Judges [ i ] .Call ( Score, t1 ) , t1 )  )
                    ->   EXISTS S: Set_Of_Judge_ID
                                  ( SET_SIZE  ( S )  >= 3
                                  & FORALL i: Judge_ID
                                             ( i ISIN S
                                        <->    EXISTS t1: Time
                                                        ( t1 >= t
                                                        & t1 < t + Window
                                                        & past ( Judges [ i ] .Call ( Score ( B ) , t1 ) ,
                                                                   t1 )  )  )  )
                       & FORALL t1: Time, i: Judge_ID
                                  ( t1 >= t + Window
                                  & t1 < t + 2 * Window
                               ->   ~past ( Judges [ i ] .Call ( Score, t1 ) , t1 )  )  )
```

```
  SCHEDULE
             ( Scorer.Outcome ~= In_Progress
          ->    ( Time_Keeper.Round_Number = Num_Rounds
                & ~Time_Keeper.In_Round )  )
          &  ( Scorer.Outcome = Win
          ->   EXISTS S: Set_Of_Judge_ID
                        ( SET_SIZE  ( S )  >= 3
                        & FORALL j: Judge_ID
                                   ( j ISIN S
                                ->   FORALL B: Boxer
                                              ( Judges [ j ] .Score_Card ( Scorer.Winner )  >=
                                                  Judges [ j ] .Score_Card ( B )  )  )  )  )
END Olympic_Boxing
PROCESS SPECIFICATION Timer
  LEVEL Top_Level
    IMPORT
              Pos_Real, Non_Negative, Num_Rounds
    EXPORT
              In_Round, Round_Number
    CONSTANT
              Begin_Dur, End_Dur: Pos_Real,
              Round_Length: Pos_Real, /* 3 minutes */
              Between_Rounds: Pos_Real /* 1 minute */
    VARIABLE
              Round_Number: Non_Negative,
              In_Round: Boolean
    INITIAL
              Round_Number = 0
            & ~In_Round
    INVARIANT
              FORALL t: Time
                       ( t >= 0
                       & t <= Now
                       & past ( Round_Number, t )  = Num_Rounds
                       & ~past ( In_Round, t )
                    ->   FORALL t1: Time
                                  ( t1 > t
                                  & t1 <= Now
                               ->   past ( Round_Number, t1 )  = past ( Round_Number, t )  )
                       & FORALL t1: Time
                                  ( t1 > t
                                  & t1 <= Now
                               ->   past ( In_Round, t1 )  = past ( In_Round, t )  )  )
    TRANSITION Begin_Round
      ENTRY           [ TIME : Begin_Dur ]
                 ( Round_Number = 0
                |  Now - Start ( End_Round )  >= Between_Rounds )
              & Round_Number < Num_Rounds
              & ~In_Round
      EXIT
                Round_Number = Round_Number’ + 1
              & In_Round
    TRANSITION End_Round
      ENTRY           [ TIME : End_Dur ]
                Now - Start ( Begin_Round )  >= Round_Length
              & In_Round
      EXIT
                ~In_Round
  END Top_Level
END Timer
PROCESS SPECIFICATION Tabulate
  LEVEL Top_Level
    IMPORT
              Pos_Real, Name, Boxer, Judges, Judge_ID, Set_Of_Judge_ID, Non_Negative, Num_Rounds,
              Window, Time_Keeper.Round_Number, Judges.Score, Time_Keeper, Time_Keeper.In_Round, Decision
    EXPORT
              Winner, Outcome
    CONSTANT
              Final_Dur: Pos_Real,
              Update_Dur: Pos_Real
    VARIABLE
              Points ( Boxer ) : Non_Negative,
              Outcome: Decision,
              Winner: Name
    INITIAL
              FORALL B: Boxer
                       ( Points ( B )  = 0 )
            & Outcome = In_Progress
            & Winner = None
    INVARIANT
               ( Outcome = Win
            ->   Winner ~= None )
            &  ( Outcome ~= In_Progress
            ->   EXISTS t: Time
                          ( t >= 0
                          & t <= Now
                          & End ( Final_Decision, t )  )  )
    CONSTRAINT
              Winner ~= Winner’
         ->   Outcome ~= In_Progress
    SCHEDULE
              Outcome ~= In_Progress
         ->    ( Time_Keeper.Round_Number = Num_Rounds
               & ~Time_Keeper.In_Round )
```

```
      IMPORTED VARIABLE
                FORALL t: Time
                         ( t >= 0
                         & t <= Now
                         & past ( Time_Keeper.Round_Number, t )  = Num_Rounds
                         & ~past ( Time_Keeper.In_Round, t )
                      ->   FORALL t1: Time
                                    ( t1 > t
                                    & t1 <= Now
                                 ->   past ( Time_Keeper.Round_Number, t1 )  =
                                        past ( Time_Keeper.Round_Number, t )  )
                         & FORALL t1: Time
                                    ( t1 > t
                                    & t1 <= Now
                                 ->   past ( Time_Keeper.In_Round, t1 )  = past ( Time_Keeper.In_Round,
                                                                                  t )  )  )
      TRANSITION Update ( B: Boxer )
        ENTRY           [ TIME : Update_Dur ]
                  EXISTS S: Set_Of_Judge_ID
                           ( SET_SIZE  ( S )  >= 3
                           & FORALL j: Judge_ID
                                      ( j ISIN S
                                 <->    Now - Judges [ j ] .Start ( Score ( B )  )  <= Window )  )
                & Now - Start ( Update )  >= Window
                & Outcome = In_Progress
        EXIT
                  Points ( B )  BECOMES Points' ( B )  + 1
      TRANSITION Final_Decision
        ENTRY           [ TIME : Final_Dur ]
                  Time_Keeper.Round_Number = Num_Rounds
                & ~Time_Keeper.In_Round
                & Points ( Fighter1 )  ~= Points ( Fighter2 )
                & Outcome = In_Progress
        EXIT
                  EXISTS B: Boxer
                           ( FORALL B1: Boxer
                                      ( Points' ( B )  > Points' ( B1 )
                                     |  B = B1 )
                           & Winner = B
                           & Outcome = Win )
        EXCEPT          [ TIME : Final_Dur ]
                  Time_Keeper.Round_Number = Num_Rounds
                & ~Time_Keeper.In_Round
                & Points ( Fighter1 )  = Points ( Fighter2 )
                & Outcome = In_Progress
        EXIT
                  Outcome = Draw
    END Top_Level
  END Tabulate
  PROCESS SPECIFICATION Judge
    LEVEL Top_Level
      IMPORT
                Pos_Real, Boxer, Time_Keeper.In_Round, Non_Negative, Window, Time_Keeper
      EXPORT
                Score, Score_Card
      CONSTANT
                Score_Dur: Pos_Real
      VARIABLE
                Score_Card ( Boxer ) : Non_Negative
      AXIOM
                Score_Dur < Window
      ENVIRONMENT
                 ( FORALL t: Time
                            ( t >= 0
                            & t <= Now
                            & Call [ 2 ]  ( Score, t )
                         ->   Call ( Score )  - t >= 2 * Window )  )
              &  ( FORALL t: Time
                            ( t >= 0
                            & t <= Now
                            & Call ( Score, t )
                         ->   past ( Time_Keeper.In_Round, t )  )  )
      INITIAL
                FORALL B: Boxer
                         ( Score_Card ( B )  = 0 )
      SCHEDULE
                FORALL t: Time
                         ( t >= 0
                         & t <= Now
                         & Call ( Score, t )
                      ->   Start ( Score, t )  )
      TRANSITION Score ( B: Boxer )
        ENTRY           [ TIME : Score_Dur ]
                  Time_Keeper.In_Round
                & FORALL t: Time
                           ( t < Now
                           & past ( Start ( Score, t ) , t )
                        ->   Now - t > Window )
        EXIT
                  Score_Card ( B )  BECOMES Score_Card' ( B )  + 1
    END Top_Level
  END Judge
END Olympic_Boxing
```

6. Production Cell

```
SPECIFICATION Production_Cell
  GLOBAL SPECIFICATION Production_Cell
    PROCESSES
              robot: P_Robot,
              press: P_Press,
              feed: P_Feed,
              deposit: P_Deposit,
              the_table: P_Table,
              crane: P_Crane,
              feed_sensor: P_Feed_Sensor,
              deposit_sensor: P_Deposit_Sensor
    TYPE
              pos_real: TYPEDEF r: real ( r > 0 ) ,
              table_statuses:  ( at_belt, at_robot, moving_to_belt, moving_to_robot ) ,
              level_statuses:  ( at_lower, at_middle, at_upper, moving_to_lower, moving_to_middle,
                                   moving_to_upper ) ,
              arm_statuses:  ( extended, retracted, extending, retracting ) ,
              crane_statuses:  ( at_deposit, at_stockpile, moving_to_deposit, moving_to_stockpile )
    CONSTANT
              feed_speed, deposit_speed: pos_real,
              feed_length, deposit_length: pos_real,
              feed_response, deposit_response, table_response, deposit_sensor_response: pos_real,
              blank_length: pos_real
    AXIOM
      /* feed belt length must be big enough to accommodate 2 blanks plus the distance it takes the feed
         sensor to detect a blank */
              feed_length > blank_length + blank_length + feed_response * feed_speed
      /* deposit belt length must be big enough to accommodate 2 blanks plus the distance it takes the
         deposit sensor to detect a blank */
            & deposit_length > blank_length + blank_length + deposit_response * deposit_speed
  END Production_Cell
  PROCESS SPECIFICATION P_Robot
    LEVEL Top_Level
      IMPORT
                pos_real, the_table, the_table.h_status, the_table.v_status, table_statuses, press,
                press.press_status, arm_statuses, level_statuses, deposit_sensor, deposit_sensor.is_object,
                table_response
      EXPORT
                arm1_status, arm2_status, arm1_has_object, arm2_has_object
      TYPE
                robot_statuses:  ( arm1_at_table, arm1_at_press, arm2_at_press, arm2_at_deposit,
                                     moving_arm1_to_table, moving_arm1_to_press, moving_arm2_to_press,
                                     moving_arm2_to_deposit )
      CONSTANT
                t_move_arm1_to_table, t_move_arm2_to_press, t_move_arm2_to_deposit: pos_real
                t_move_arm1_to_press, t_move_arm: pos_real,
                rotate_arm_dur, arm_arrive_dur, move_arm_dur, arm_moved_dur, arm_object_dur: pos_real
      VARIABLE
                robot_status: robot_statuses,
                arm1_status, arm2_status: arm_statuses,
                arm1_has_object, arm2_has_object: boolean
      AXIOM
                table_response <= rotate_arm_dur
      INITIAL
                robot_status = arm2_at_deposit
              & arm1_status = retracted
              & arm2_status = retracted
              & arm1_has_object
              & ~arm2_has_object
      INVARIANT
        /* arms are retracted when robot is moving */
                 ( robot_status = moving_arm1_to_press
                |  robot_status = moving_arm2_to_deposit
                |  robot_status = moving_arm1_to_table
                |  robot_status = moving_arm2_to_press
              ->   arm1_status = retracted
                 & arm2_status = retracted )
        /* arms have and don't have objects at right times */
              &  ( robot_status = moving_arm1_to_press
              ->   arm1_has_object )
              &  ( robot_status = moving_arm2_to_deposit
              ->   arm2_has_object )
              &  ( robot_status = moving_arm1_to_table
              ->   ~arm1_has_object )
              &  ( robot_status = moving_arm2_to_press
              ->   ~arm2_has_object )
        /* only drop at right place */
              &  ( ~arm2_has_object
                 & Change ( arm2_has_object, now )
              ->   robot_status = arm2_at_deposit
                 & arm2_status = extended )
        /* only pickup when something there to pickup */
              &  ( Start ( Arm2_Pickup, now )
              ->   EXISTS t: time
                           ( t < now
                           & End ( Arm1_Drop, t )
                           & FORALL t1: time
                                      ( End ( Arm2_Pickup, t1 )
                                  ->   t1 < t )  )  )
```

```
  /* blanks don't collide */
        &  ( Start ( Arm1_Drop, now )
        ->   FORALL t: time
                       ( End ( Arm1_Drop, t )
                    ->   EXISTS t1: time
                                  ( t < t1
                                  & t1 < now
                                  & End ( Arm2_Pickup, t1 )  )  )  )
        &  ( Start ( Arm2_Drop, now )
        ->   FORALL t: time
                       ( End ( Arm2_Drop, t )
                    ->   EXISTS t1: time
                                  ( t < t1
                                  & t1 <= now
                                  & past ( Change ( deposit_sensor.is_object, t1 ) , t1 )
                                  & past ( deposit_sensor.is_object, t1 )  )  )  )
SCHEDULE
  /* don't collide with press */
           ( robot_status = arm1_at_press
        ->    ( press.press_status ~= at_upper
              & press.press_status ~= moving_to_upper
             |  arm1_status = retracted )  )
        &  ( robot_status = arm2_at_press
        ->    ( press.press_status ~= at_middle
              & press.press_status ~= moving_to_middle
             |  arm2_status = retracted )  )
  /* only pickup at right place */
        &  ( arm1_has_object
           & Change ( arm1_has_object, now )
        ->   robot_status = arm1_at_table
           & arm1_status = extended
           & the_table.h_status = at_robot
           & the_table.v_status = at_robot )
        &  ( arm2_has_object
           & Change ( arm2_has_object, now )
        ->   robot_status = arm2_at_press
           & arm2_status = extended
           & press.press_status = at_lower )
  /* only drop at right place */
        &  ( ~arm1_has_object
           & Change ( arm1_has_object, now )
        ->   robot_status = arm1_at_press
           & arm1_status = extended
           & press.press_status = at_middle )
  /* only pickup when something there to pickup */
        &  ( Start ( Arm1_Pickup, now )
        ->   EXISTS t: time
                       ( t <= now
                       & past ( Change ( the_table.v_status = at_robot
                                       & the_table.h_status = at_robot, t ) , t )
                       & past ( the_table.v_status = at_robot
                              & the_table.h_status = at_robot, t )
                       &  ( FORALL t1: time
                                     ( Start [ 2 ]  ( Arm1_Pickup, t1 )
                                  ->   t1 < t )  )  )  )
IMPORTED VARIABLE
  /* press only moves from middle after arm1 drops blank */
           ( Change ( press.press_status, now )
           & press.press_status ~= at_middle
        ->   EXISTS t1, t2: time
                       ( FORALL t: time
                                  ( Change [ 2 ]  ( press.press_status, t )
                                  & past ( press.press_status, t )  = at_middle
                               ->   t < t1 )
                       & t1 < t2
                       & t2 <= now
                       & past ( Change ( arm1_has_object, t1 ) , t1 )
                       & ~past ( arm1_has_object, t1 )
                       & past ( arm1_status, t2 )  = retracted )  )
  /* press only moves from lower after arm2 picks up blank */
        &  ( Change ( press.press_status, now )
           & press.press_status ~= at_lower
        ->   FORALL t: time
                       ( Change [ 2 ]  ( press.press_status, t )
                       & past ( press.press_status, t )  = at_lower
                    ->   EXISTS t1, t2: time
                                  ( t < t1
                                  & t1 < t2
                                  & t2 <= now
                                  & past ( Change ( arm2_has_object, t1 ) , t1 )
                                  & past ( arm2_has_object, t1 )
                                  & past ( arm2_status, t2 )  = retracted )  )  )
  /* table only moves from robot after arm1 picks up blank */
        &  ( Change ( the_table.v_status = at_robot
                    & the_table.h_status = at_robot, now )
           & ~ ( the_table.v_status = at_robot
               & the_table.h_status = at_robot )
        ->   FORALL t: time
                       ( Change [ 2 ]  ( the_table.v_status = at_robot
                                       & the_table.h_status = at_robot, t )
                    ->   EXISTS t1: time
                                  ( t <= t1
                                  & t1 < now
                                  & past ( Change ( arm1_has_object, t1 ) , t1 )
                                  & past ( arm1_has_object, t1 )  )  )  )
```

```
  /* table moves within time table_response of arm1 picking up a blank */
        &  ( Change ( arm1_has_object, now - table_response )
           & past ( arm1_has_object, now - table_response )
           & past ( the_table.v_status = at_robot
                  & the_table.h_status = at_robot, now - table_response )
        ->   EXISTS t: time
                      ( now - table_response < t
                      & t <= now
                      & past ( Change ( the_table.v_status = at_robot
                                      & the_table.h_status = at_robot, t ) , t )
                      & ~past ( the_table.v_status = at_robot
                              & the_table.h_status = at_robot, t )  )  )
TRANSITION Rot_Arm1_CCW_To_Press
  ENTRY           [ TIME : rotate_arm_dur ]
            robot_status = arm2_at_deposit
          & ~arm2_has_object
          & arm2_status = retracted
  EXIT
            robot_status = moving_arm1_to_press
TRANSITION Arm1_Arrived_At_Press
  ENTRY           [ TIME : arm_arrive_dur ]
            robot_status = moving_arm1_to_press
          & now - Change ( robot_status )  >= t_move_arm1_to_press
  EXIT
            robot_status = arm1_at_press
TRANSITION Rot_Arm1_CW_To_Table
  ENTRY           [ TIME : rotate_arm_dur ]
            robot_status = arm1_at_press
          & ~arm1_has_object
          & arm1_status = retracted
  EXIT
            robot_status = moving_arm1_to_table
TRANSITION Arm1_Arrived_At_Table
  ENTRY           [ TIME : arm_arrive_dur ]
            robot_status = moving_arm1_to_table
          & now - Change ( robot_status )  >= t_move_arm1_to_table
  EXIT
            robot_status = arm1_at_table
TRANSITION Rot_Arm2_CCW_To_Press
  ENTRY           [ TIME : rotate_arm_dur ]
            robot_status = arm1_at_table
          & arm1_has_object
          & arm1_status = retracted
  EXIT
            robot_status = moving_arm2_to_press
TRANSITION Arm2_Arrived_At_Press
  ENTRY           [ TIME : arm_arrive_dur ]
            robot_status = moving_arm2_to_press
          & now - Change ( robot_status )  >= t_move_arm2_to_press
  EXIT
            robot_status = arm2_at_press
TRANSITION Rot_Arm2_CCW_To_Deposit
  ENTRY           [ TIME : rotate_arm_dur ]
            robot_status = arm2_at_press
          & arm2_has_object
          & arm2_status = retracted
  EXIT
            robot_status = moving_arm2_to_deposit
TRANSITION Arm2_Arrived_At_Deposit
  ENTRY           [ TIME : arm_arrive_dur ]
            robot_status = moving_arm2_to_deposit
          & now - Change ( robot_status )  >= t_move_arm2_to_deposit
  EXIT
            robot_status = arm2_at_deposit
TRANSITION Extend_Arm1
  ENTRY           [ TIME : move_arm_dur ]
             ( robot_status = arm1_at_table
             & ~arm1_has_object
             & the_table.h_status = at_robot
             & the_table.v_status = at_robot
            |  robot_status = arm1_at_press
             & press.press_status = at_middle
             & arm1_has_object )
          & arm1_status = retracted
  EXIT
            arm1_status = extending
TRANSITION Arm1_Extended
  ENTRY           [ TIME : arm_moved_dur ]
            arm1_status = extending
          & now - Change ( arm1_status )  >= t_move_arm
  EXIT
            arm1_status = extended
TRANSITION Extend_Arm2
  ENTRY           [ TIME : move_arm_dur ]
             ( robot_status = arm2_at_press
             & ~arm2_has_object
             & press.press_status = at_lower
            |  robot_status = arm2_at_deposit
             & arm2_has_object )
          & arm2_status = retracted
  EXIT
            arm2_status = extending
```

```
    TRANSITION Arm2_Extended
      ENTRY          [ TIME : arm_moved_dur ]
                arm2_status = extending
              & now - Change ( arm2_status )  >= t_move_arm
      EXIT
                arm2_status = extended
    TRANSITION Retract_Arm1
      ENTRY          [ TIME : move_arm_dur ]
                 ( robot_status = arm1_at_table
                 & arm1_has_object
                |  robot_status = arm1_at_press
                 & ~arm1_has_object )
              & arm1_status = extended
      EXIT
                arm1_status = retracting
    TRANSITION Arm1_Retracted
      ENTRY          [ TIME : arm_moved_dur ]
                arm1_status = retracting
              & now - Change ( arm1_status )  >= t_move_arm
      EXIT
                arm1_status = retracted
    TRANSITION Retract_Arm2
      ENTRY          [ TIME : move_arm_dur ]
                 ( robot_status = arm2_at_press
                 & arm2_has_object
                |  robot_status = arm2_at_deposit
                 & ~arm2_has_object )
              & arm2_status = extended
      EXIT
                arm2_status = retracting
    TRANSITION Arm2_Retracted
      ENTRY          [ TIME : arm_moved_dur ]
                arm2_status = retracting
              & now - Change ( arm2_status )  >= t_move_arm
      EXIT
                arm2_status = retracted
    TRANSITION Arm1_Pickup
      ENTRY          [ TIME : arm_object_dur ]
                ~arm1_has_object
              & robot_status = arm1_at_table
              & arm1_status = extended
              & the_table.h_status = at_robot
              & the_table.v_status = at_robot
      EXIT
                arm1_has_object
    TRANSITION Arm1_Drop
      ENTRY          [ TIME : arm_object_dur ]
                arm1_has_object
              & robot_status = arm1_at_press
              & arm1_status = extended
              & press.press_status = at_middle
      EXIT
                ~arm1_has_object
    TRANSITION Arm2_Pickup
      ENTRY          [ TIME : arm_object_dur ]
                ~arm2_has_object
              & robot_status = arm2_at_press
              & arm2_status = extended
              & press.press_status = at_lower
      EXIT
                arm2_has_object
    TRANSITION Arm2_Drop
      ENTRY          [ TIME : arm_object_dur ]
                arm2_has_object
              & robot_status = arm2_at_deposit
              & arm2_status = extended
              & FORALL t: time
                         ( End ( Arm2_Drop, t )
                      ->   EXISTS t1: time
                                   ( t < t1
                                   & t1 <= now
                                   & past ( Change ( deposit_sensor.is_object, t1 ) , t1 )
                                   & past ( deposit_sensor.is_object, t1 )  )  )
      EXIT
                ~arm2_has_object
  END Top_Level
END P_Robot
PROCESS SPECIFICATION P_Press
  LEVEL Top_Level
    IMPORT
              pos_real, level_statuses, robot, robot.arm1_has_object, robot.arm2_has_object,
              robot.arm1_status, robot.arm2_status, arm_statuses
    EXPORT
              press_status
    CONSTANT
              t_move_press_level: pos_real,
              move_press_dur, press_arrive_dur: pos_real
    VARIABLE
              press_status: level_statuses
    INITIAL
              press_status = at_middle
```

```
    INVARIANT
      /* press only moves from middle after arm1 drops blank */
               ( Change ( press_status, now )
               & press_status ~= at_middle
            ->   EXISTS t1, t2: time
                        ( FORALL t: time
                                   ( Change [ 2 ]  ( press_status, t )
                                   & past ( press_status, t )  = at_middle
                                ->   t < t1 )
                        & t1 < t2
                        & t2 <= now
                        & past ( Change ( robot.arm1_has_object, t1 ) , t1 )
                        & ~past ( robot.arm1_has_object, t1 )
                        & past ( robot.arm1_status, t2 )  = retracted )  )
      /* press only moves from lower after arm2 picks up blank */
            &  ( Change ( press_status, now )
               & press_status ~= at_lower
            ->   FORALL t: time
                        ( Change [ 2 ]  ( press_status, t )
                        & past ( press_status, t )  = at_lower
                     ->   EXISTS t1, t2: time
                                   ( t < t1
                                   & t1 < t2
                                   & t2 <= now
                                   & past ( Change ( robot.arm2_has_object, t1 ) , t1 )
                                   & past ( robot.arm2_has_object, t1 )
                                   & past ( robot.arm2_status, t2 )  = retracted )  )  )
    TRANSITION Move_To_Upper
      ENTRY           [ TIME : move_press_dur ]
                press_status = at_middle
              & EXISTS t1, t2: time
                       ( t1 < t2
                       & t2 <= now
                       & past ( Change ( robot.arm1_has_object, t1 ) , t1 )
                       & ~past ( robot.arm1_has_object, t1 )
                       & past ( robot.arm1_status, t2 )  = retracted
                       & FORALL t3: time
                                  ( Change ( press_status, t3 )
                               ->   t3 < t1 )  )
      EXIT
                press_status = moving_to_upper
    TRANSITION Arrived_At_Upper
      ENTRY           [ TIME : press_arrive_dur ]
                press_status = moving_to_upper
              & now - Change ( press_status )  >= t_move_press_level
      EXIT
                press_status = at_upper
    TRANSITION Move_To_Lower
      ENTRY           [ TIME : move_press_dur ]
                press_status = at_upper
      EXIT
                press_status = moving_to_lower
    TRANSITION Arrived_At_Lower
      ENTRY           [ TIME : press_arrive_dur ]
                press_status = moving_to_lower
              & now - Change ( press_status )  >= t_move_press_level + t_move_press_level
      EXIT
                press_status = at_lower
    TRANSITION Move_To_Middle
      ENTRY           [ TIME : move_press_dur ]
                press_status = at_lower
              & EXISTS t1, t2: time
                       ( Change ( press_status )  < t1
                       & t1 < t2
                       & t2 <= now
                       & past ( Change ( robot.arm2_has_object, t1 ) , t1 )
                       & past ( robot.arm2_has_object, t1 )
                       & past ( robot.arm2_status, t2 )  = retracted )
      EXIT
                press_status = moving_to_middle
    TRANSITION Arrived_At_Middle
      ENTRY           [ TIME : press_arrive_dur ]
                press_status = moving_to_middle
              & now - Change ( press_status )  >= t_move_press_level
      EXIT
                press_status = at_middle
  END Top_Level
END P_Press
PROCESS SPECIFICATION P_Feed
  LEVEL Top_Level
    IMPORT
              pos_real, feed_sensor, feed_sensor.is_object, the_table, the_table.h_status,
              the_table.v_status, table_statuses, feed_response
    EXPORT
              Add_Blank, moving
    CONSTANT
              move_feed_dur, add_blank_dur: pos_real
    VARIABLE
              moving: boolean
    AXIOM
      /* the time it takes to start/stop the feed belt must be less than the required feed response time
         */
              move_feed_dur <= feed_response
    INITIAL
              ~moving
```

```
    SCHEDULE
      /* blank won't be moved off belt unless table is in right place */
               ( moving
               & feed_sensor.is_object
               & Change ( moving )  > Change ( feed_sensor.is_object )
            ->   the_table.h_status = at_belt
               & the_table.v_status = at_belt )
      /* table stops between when a blank arrives and when it departs */
            &  ( Change ( feed_sensor.is_object, now )
               & ~feed_sensor.is_object
            ->   EXISTS t: time
                          ( Change [ 2 ]  ( feed_sensor.is_object )  < t
                          & t < now
                          & ~past ( moving, t )  )  )
    IMPORTED VARIABLE
      /* table only moves from feed belt after blank loaded */
               ( Change ( the_table.v_status = at_belt
                        & the_table.h_status = at_belt, now )
               & ~ ( the_table.v_status = at_belt
                   & the_table.h_status = at_belt )
            ->   EXISTS t1: time
                          ( FORALL t: time
                                     ( Change [ 2 ]  ( the_table.v_status = at_belt
                                                     & the_table.h_status = at_belt, t )
                                  ->   t < t1 )
                          & t1 <= now
                          & past ( Change ( feed_sensor.is_object, t1 ) , t1 )
                          & ~past ( feed_sensor.is_object, t1 )  )  )
      /* feed sensor asserts is_object for at least feed_response time */
            &  ( Change ( feed_sensor.is_object, now )
               & ~feed_sensor.is_object
            ->   now - Change [ 2 ]  ( feed_sensor.is_object )  > feed_response )
    TRANSITION Start_Move
      ENTRY           [ TIME : move_feed_dur ]
                ~moving
              & ~feed_sensor.is_object
      EXIT
                moving
    TRANSITION Stop_Move
      ENTRY           [ TIME : move_feed_dur ]
                moving
              & feed_sensor.is_object
              & Change ( feed_sensor.is_object )  >= Change ( moving )
      EXIT
                ~moving
    TRANSITION Move_Object_Onto_Table
      ENTRY           [ TIME : move_feed_dur ]
                ~moving
              & feed_sensor.is_object
              & the_table.h_status = at_belt
              & the_table.v_status = at_belt
      EXIT
                moving
    TRANSITION Add_Blank
      ENTRY           [ TIME : add_blank_dur ]
                FORALL t: time
                         ( End ( Add_Blank, t )
                      ->   EXISTS t1: time
                                    ( t < t1
                                    & t1 < now
                                    & past ( Change ( feed_sensor.is_object, t1 ) , t1 )
                                    & past ( feed_sensor.is_object, t1 )  )  )
      EXIT
                TRUE
  END Top_Level
END P_Feed
PROCESS SPECIFICATION P_Deposit
  LEVEL Top_Level
    IMPORT
              pos_real, deposit_sensor, deposit_sensor.is_object, deposit_response
    EXPORT
              moving
    CONSTANT
              move_deposit_dur: pos_real
    VARIABLE
              moving: boolean
    AXIOM
      /* the time it takes to start/stop the deposit belt must be less than the required deposit response
         time */
              move_deposit_dur <= deposit_response
    INITIAL
              ~moving
    TRANSITION Start_Move
      ENTRY           [ TIME : move_deposit_dur ]
                ~moving
              & ~deposit_sensor.is_object
      EXIT
                moving
    TRANSITION Stop_Move
      ENTRY           [ TIME : move_deposit_dur ]
                moving
              & deposit_sensor.is_object
      EXIT
                ~moving
  END Top_Level
```

```
END P_Deposit
PROCESS SPECIFICATION P_Table
  LEVEL Top_Level
    IMPORT
              pos_real, table_statuses, feed_sensor, feed_sensor.is_object, robot, robot.arm1_has_object,
              table_response
    EXPORT
              h_status, v_status
    CONSTANT
              t_move_table_level, t_rotate_table: pos_real,
              move_table_dur, rotate_table_dur, table_arrive_dur: pos_real
    VARIABLE
              h_status: table_statuses,
              v_status: table_statuses
    AXIOM
              table_response >= rotate_table_dur
    INITIAL
              h_status = at_belt
            & v_status = at_belt
    INVARIANT
      /* table only moves from robot after arm1 picks up blank */
               ( Change ( v_status = at_robot
                        & h_status = at_robot, now )
               & ~ ( v_status = at_robot
                   & h_status = at_robot )
            ->   FORALL t: time
                          ( Change [ 2 ]  ( v_status = at_robot
                                          & h_status = at_robot, t )
                       ->   EXISTS t1: time
                                     ( t <= t1
                                     & t1 < now
                                     & past ( Change ( robot.arm1_has_object, t1 ) , t1 )
                                     & past ( robot.arm1_has_object, t1 )  )  )  )
      /* table only moves from feed belt after blank loaded */
            &  ( Change ( v_status = at_belt
                        & h_status = at_belt, now )
               & ~ ( v_status = at_belt
                   & h_status = at_belt )
            ->   EXISTS t1: time
                          ( FORALL t: time
                                     ( Change [ 2 ]  ( v_status = at_belt
                                                     & h_status = at_belt, t )
                                  ->   t <= t1 )
                          & t1 < now
                          & past ( Change ( feed_sensor.is_object, t1 ) , t1 )
                          & ~past ( feed_sensor.is_object, t1 )  )  )
      /* table moves within time table_response of arm1 picking up a blank */
            &  ( Change ( robot.arm1_has_object, now - table_response )
               & past ( robot.arm1_has_object, now - table_response )
               & past ( v_status = at_robot
                      & h_status = at_robot, now - table_response )
            ->   EXISTS t: time
                          ( now - table_response < t
                          & t <= now
                          & past ( Change ( v_status = at_robot
                                          & h_status = at_robot, t ) , t )
                          & ~past ( v_status = at_robot
                                  & h_status = at_robot, t )  )  )
    TRANSITION Move_To_Upper
      ENTRY           [ TIME : move_table_dur ]
                h_status = at_belt
              & v_status = at_belt
              & EXISTS t: time
                         ( t <= now
                         & past ( Change ( feed_sensor.is_object, t ) , t )
                         & ~past ( feed_sensor.is_object, t )
                         & FORALL t1: time
                                    ( Change ( v_status, t1 )
                                 ->   t1 <= t )  )
      EXIT
                v_status = moving_to_robot
    TRANSITION Arrived_At_Upper
      ENTRY           [ TIME : table_arrive_dur ]
                v_status = moving_to_robot
              & now - Change ( v_status )  >= t_move_table_level
      EXIT
                v_status = at_robot
    TRANSITION Rot_CW_To_Robot
      ENTRY           [ TIME : rotate_table_dur ]
                h_status = at_belt
              & v_status = at_robot
      EXIT
                h_status = moving_to_robot
    TRANSITION Arrived_At_Robot
      ENTRY           [ TIME : table_arrive_dur ]
                h_status = moving_to_robot
              & now - Change ( h_status )  >= t_rotate_table
      EXIT
                h_status = at_robot
    TRANSITION Rot_CCW_To_Feed
      ENTRY           [ TIME : rotate_table_dur ]
                h_status = at_robot
              & v_status = at_robot
```

```
              & EXISTS t: time
                         ( Change ( v_status )  <= t
                         & t <= now
                         & past ( Change ( robot.arm1_has_object, t ) , t )
                         & past ( robot.arm1_has_object, t )  )
      EXIT
               h_status = moving_to_belt
    TRANSITION Arrived_At_Feed
      ENTRY          [ TIME : table_arrive_dur ]
               h_status = moving_to_belt
             & now - Change ( h_status )  >= t_rotate_table
      EXIT
               h_status = at_belt
    TRANSITION Move_To_Lower
      ENTRY          [ TIME : move_table_dur ]
               h_status = at_belt
             & v_status = at_robot
      EXIT
               v_status = moving_to_belt
    TRANSITION Arrived_At_Lower
      ENTRY          [ TIME : table_arrive_dur ]
               v_status = moving_to_belt
             & now - Change ( v_status )  >= t_move_table_level
      EXIT
               v_status = at_belt
  END Top_Level
END P_Table
PROCESS SPECIFICATION P_Crane
  LEVEL Top_Level
    IMPORT
             pos_real, deposit_sensor, deposit_sensor.is_object, level_statuses, crane_statuses,
             deposit_sensor_response
    EXPORT
             gripper_has_object
    CONSTANT
             t_move_crane, t_move_gripper: pos_real,
             move_crane_dur, crane_arrive_dur: pos_real,
             move_gripper_dur, gripper_arrive_dur, gripper_object_dur: pos_real
    VARIABLE
             crane_status: crane_statuses,
             gripper_status: level_statuses,
             gripper_has_object: boolean
    AXIOM
             deposit_sensor_response < t_move_gripper
    INITIAL
             crane_status = at_deposit
           & gripper_status = at_upper
           & ~gripper_has_object
    INVARIANT
      /* gripper has and doesn't have object at right times */
              ( crane_status = moving_to_stockpile
           ->   gripper_has_object )
           &  ( crane_status = moving_to_deposit
           ->   ~gripper_has_object )
      /* only drop at right place */
           &  ( ~gripper_has_object
              & Change ( gripper_has_object, now )
           ->   crane_status = at_stockpile
              & gripper_status = at_lower )
    SCHEDULE
      /* only pickup when something there to pickup */
              ( Start ( Gripper_Pickup, now )
           ->   EXISTS t: time
                          ( t <= now
                          & past ( Change ( deposit_sensor.is_object, t ) , t )
                          & past ( deposit_sensor.is_object, t )
                          & FORALL t1: time
                                     ( End ( Gripper_Pickup, t1 )
                                  ->   t1 <= t )  )  )
    IMPORTED VARIABLE
      /* deposit sensor will detect object departure within time deposit_sensor_response */
              ( Change ( gripper_has_object, now - deposit_sensor_response )
              & past ( gripper_has_object, now - deposit_sensor_response )
           ->   EXISTS t: time
                          ( now - deposit_sensor_response <= t
                          & t <= now
                          & ~past ( deposit_sensor.is_object, t )  )  )
    TRANSITION Move_To_Stockpile
      ENTRY          [ TIME : move_crane_dur ]
               gripper_has_object
             & crane_status = at_deposit
             & gripper_status = at_upper
      EXIT
               crane_status = moving_to_stockpile
    TRANSITION Arrived_At_Stockpile
      ENTRY          [ TIME : crane_arrive_dur ]
               crane_status = moving_to_stockpile
             & now - Change ( crane_status )  >= t_move_crane
      EXIT
               crane_status = at_stockpile
    TRANSITION Move_To_Deposit
      ENTRY          [ TIME : move_crane_dur ]
               ~gripper_has_object
             & crane_status = at_stockpile
             & gripper_status = at_upper
```

```
      EXIT
                 crane_status = moving_to_deposit
    TRANSITION Arrived_At_Deposit
      ENTRY            [ TIME : crane_arrive_dur ]
                 crane_status = moving_to_deposit
               & now - Change ( crane_status )  >= t_move_crane
      EXIT
                 crane_status = at_deposit
    TRANSITION Move_To_Lower
      ENTRY            [ TIME : move_gripper_dur ]
                 crane_status = at_deposit
               & gripper_status = at_upper
      EXIT
                 gripper_status = moving_to_lower
    TRANSITION Arrived_At_Lower
      ENTRY            [ TIME : gripper_arrive_dur ]
                 gripper_status = moving_to_lower
               & now - Change ( gripper_status )  >= t_move_gripper
      EXIT
                 gripper_status = at_lower
    TRANSITION Move_To_Upper
      ENTRY            [ TIME : move_gripper_dur ]
                 crane_status = at_stockpile
               & ~gripper_has_object
               & gripper_status = at_middle
              |  crane_status = at_deposit
               & gripper_has_object
               & gripper_status = at_lower
      EXIT
                 gripper_status = moving_to_upper
    TRANSITION Arrived_At_Upper
      ENTRY            [ TIME : gripper_arrive_dur ]
                 gripper_status = moving_to_upper
               & now - Change ( gripper_status )  >= t_move_gripper
      EXIT
                 gripper_status = at_upper
    TRANSITION Move_To_Middle
      ENTRY            [ TIME : move_gripper_dur ]
                 crane_status = at_stockpile
               & gripper_status = at_upper
      EXIT
                 gripper_status = moving_to_middle
    TRANSITION Arrived_At_Middle
      ENTRY            [ TIME : gripper_arrive_dur ]
                 gripper_status = moving_to_middle
               & now - Change ( gripper_status )  >= t_move_gripper
      EXIT
                 gripper_status = at_middle
    TRANSITION Gripper_Pickup
      ENTRY            [ TIME : gripper_object_dur ]
                 ~gripper_has_object
               & crane_status = at_deposit
               & gripper_status = at_lower
               & deposit_sensor.is_object
      EXIT
                 gripper_has_object
    TRANSITION Gripper_Drop
      ENTRY            [ TIME : gripper_object_dur ]
                 gripper_has_object
               & crane_status = at_stockpile
               & gripper_status = at_middle
      EXIT
                 ~gripper_has_object
  END Top_Level
END P_Crane
PROCESS SPECIFICATION P_Feed_Sensor
  LEVEL Top_Level
    IMPORT
               pos_real, feed_length, feed_speed, blank_length, feed, feed.moving, feed.Add_Blank, crane,
               crane.gripper_has_object, feed_response
    EXPORT
               is_object
    CONSTANT
               sensor_dur: pos_real
    VARIABLE
               is_object: boolean
    INITIAL
               ~is_object
    INVARIANT
      /* feed sensor asserts is_object for at least feed_response time */
                ( Change ( is_object, now )
                & ~is_object
             ->   now - Change [ 2 ]  ( is_object )  > feed_response )
    TRANSITION Object_Arrive
      ENTRY            [ TIME : sensor_dur ]
                 ~is_object
               & feed.moving
               & EXISTS t: time
                          (  ( Change ( crane.gripper_has_object, t )
                             & ~past ( crane.gripper_has_object, t )
                            |  feed.End ( Add_Blank, t )  )
                          & FORALL t1: time
                                     ( Change ( is_object, t1 )
                                     & past ( is_object, t1 )
                                 ->   t1 < t )
```

```
                           &  ( t < Change ( feed.moving )
                           ->   now - Change ( feed.moving )  >=  ( feed_length - blank_length )  /
                                  feed_speed - sensor_dur - feed_response )
                           &  ( t >= Change ( feed.moving )
                           ->   now - t >=  ( feed_length - blank_length )  / feed_speed - sensor_dur -
                                  feed_response )  )
        EXIT
                  is_object
      TRANSITION Object_Depart
        ENTRY           [ TIME : sensor_dur ]
                  is_object
                & feed.moving
                & Change ( feed.moving )  > Change ( is_object )
                & now - Change ( feed.moving )  >= blank_length / feed_speed + feed_response
        EXIT
                  ~is_object
    END Top_Level
  END P_Feed_Sensor
  PROCESS SPECIFICATION P_Deposit_Sensor
    LEVEL Top_Level
      IMPORT
                pos_real, deposit_length, deposit_speed, deposit, deposit.moving, robot,
                robot.arm2_has_object, crane, crane.gripper_has_object, deposit_sensor_response,
                deposit_response, blank_length
      EXPORT
                is_object
      CONSTANT
                sensor_dur: pos_real
      VARIABLE
                is_object: boolean
      AXIOM
                sensor_dur + sensor_dur <= deposit_sensor_response
      INITIAL
                ~is_object
      INVARIANT
        /* deposit sensor will detect object departure within time deposit_sensor_response */
                 ( Change ( crane.gripper_has_object, now - deposit_sensor_response )
                 & past ( crane.gripper_has_object, now - deposit_sensor_response )
              ->   EXISTS t: time
                            ( now - deposit_sensor_response <= t
                            & t <= now
                            & ~past ( is_object, t )  )  )
      TRANSITION Object_Arrive
        ENTRY           [ TIME : sensor_dur ]
                  ~is_object
                & deposit.moving
                & EXISTS t: time
                           ( Change ( robot.arm2_has_object, t )
                           & ~past ( robot.arm2_has_object, t )
                           & FORALL t1: time
                                      ( Change ( is_object, t1 )
                                      & past ( is_object, t1 )
                                   ->   t1 < t )
                           &  ( t < Change ( deposit.moving )
                           ->   now - Change ( deposit.moving )  >=  ( deposit_length - blank_length )  /
                                  deposit_speed - sensor_dur - deposit_response )
                           &  ( t >= Change ( deposit.moving )
                           ->   now - t >= deposit_length / deposit_speed )  )
        EXIT
                  is_object
      TRANSITION Object_Depart
        ENTRY           [ TIME : sensor_dur ]
                  is_object
                & EXISTS t: time
                           ( Change ( is_object )  <= t
                           & t <= now
                           & past ( Change ( crane.gripper_has_object, t ) , t )
                           & past ( crane.gripper_has_object, t )  )
        EXIT
                  ~is_object
    END Top_Level
  END P_Deposit_Sensor
END Production_Cell
```

7. Phone System

```
SPECIFICATION Phone_System
  GLOBAL SPECIFICATION Phone_System
    PROCESSES
              Phones: array [ 1 .. Num_Phone ] of Phone,
              Centrals: array [ 1 .. Num_Area ] of Central_Control
    TYPE
              Positive_Integer IS TYPEDEF p: Integer ( p > 0 ) ,
              Digit IS TYPEDEF d: Integer ( d >= 0
                                          & d <= 9 ) ,
              Digit_List IS LIST OF Digit,
              Connection IS STRUCTURE OF  ( From_Area, From_Number, To_Area, To_Number: Digit_List ) ,
              Phone_ID IS TYPEDEF pid: ID ( IDTYPE  ( pid )  = Phone ) ,
              Central_Control_ID IS TYPEDEF pid: ID ( IDTYPE  ( pid )  = Central_Control ) ,
              Enabled_State IS  ( Idle, Ready_To_Dial, Dialing, Ringing, Waiting, Talk, Calling,
                                   Disconnecting, Busy, Alarm ) ,
              Connection_Status IS  ( Available, In_Progress, Disconnect, Connect, Talking )
```

```
  CONSTANT
            In_Area ( Phone_ID ) : Central_Control_ID,
            Max_Cust, Num_Phone, Num_Area: Positive_Integer,
            LD_Timeout: Time
  DEFINE
            Plug ( L1, L2: Connection ) : Boolean ==
                    L1 [ From_Area ]  = L2 [ To_Area ]
                  & L1 [ From_Number ]  = L2 [ To_Number ]
                  & L1 [ To_Area ]  = L2 [ From_Area ]
                  & L1 [ To_Number ]  = L2 [ From_Number ]
  ENVIRONMENT
            FORALL C: Central_Control_ID
                     ( SET_SIZE (  { SETDEF P: Phone_ID ( In_Area ( P )  = C
                                                        & Now - 2 <= P.Call ( Pickup )
                                                        & P.Call ( Pickup )  <= Now )  }  )  <= Max_Cust )
  SCHEDULE
            FORALL P: Phone_ID, t, t1, t2: Time
                     ( t <= t1
                     & t1 < t2
                     & Change [ 2 ]  ( In_Area ( P ) .Phone_State ( P ) , t )
                     & past ( In_Area ( P ) .Phone_State ( P ) , t )  = Idle
                     & P.End ( Pickup, t1 )
                     & P.Offhook
                     & Change ( In_Area ( P ) .Phone_State ( P ) , t2 )
                  ->   past ( In_Area ( P ) .Phone_State ( P ) , t2 )  = Ringing
                    |  past ( In_Area ( P ) .Phone_State ( P ) , t2 )  = Ready_To_Dial
                     & t2 <= t1 + 2 )
END Phone_System
PROCESS SPECIFICATION Phone
  LEVEL Top_Level
    IMPORT
              Digit, Phone_ID, Central_Control_ID, Enabled_State, In_Area, Centrals.Phone_State,
              Centrals.Enabled_Ring_Pulse, Centrals.Enabled_Ringback_Pulse
    EXPORT
              Offhook, Next_Digit, Pickup, Enter_Digit, Hangup
    CONSTANT
              T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11: Time
    VARIABLE
              Offhook, Dialtone, Ring, Ringback, Busytone: Boolean,
              Next_Digit: Digit
    DEFINE
              My_Central: Central_Control_ID ==
                      In_Area ( Self )
    ENVIRONMENT
              FORALL t: Time
                       ( Call ( Pickup, t )
                    ->   ~past ( Offhook, t )  )
            & FORALL t: Time
                       ( Call ( Hangup, t )
                    ->   past ( Offhook, t )  )
            & FORALL t: Time
                       ( Call ( Enter_Digit, t )
                    ->    ( past ( Dialtone, t )
                         |  EXISTS t1: Time, n: Integer, D: Digit
                                     ( 2 <= n
                                     & Call [ n ]  ( Enter_Digit ( D ) , t1 )
                                     & past ( Dialtone, t1 )
                                     &  ( n <= 7
                                        & D ~= 1
                                       |  n <= 11
                                        & D = 1 )
                                     & FORALL t2: Time
                                                ( t1 <= t2
                                                & t2 <= t
                                              ->   past ( Offhook, t2 )  )  )  )  )
            & FORALL t: Time
                       ( Call [ 2 ]  ( Pickup, t )
                    ->   Call ( Pickup )  - Call [ 2 ]  ( Pickup )  >= 1 )
    INITIAL
              ~Offhook
            & ~Dialtone
            & ~Busytone
            & ~Ring
            & ~Ringback
    INVARIANT
               ( Dialtone
            ->   Offhook )
            &  ( Ringback
            ->   Offhook )
            &  ( Busytone
            ->   Offhook )
            &  ( Ring
            ->   ~Offhook )
            &  ( Ring
            ->   ~Dialtone
               & ~Ringback
               & ~Busytone )
    SCHEDULE
               ( Dialtone
            ->   ~Ring
               & ~Ringback
               & ~Busytone )
            &  ( Ringback
            ->   ~Dialtone
               & ~Ring
```

```
           & ~Busytone )
        &  ( Busytone
        ->   ~Dialtone
           & ~Ring
           & ~Ringback )
IMPORTED VARIABLE
           ( My_Central.Phone_State ( Self )  = Busy
        ->   past ( My_Central.Phone_State ( Self ) ,
                     Change [ 2 ]  ( My_Central.Phone_State ( Self )  )  )  = Dialing
           & EXISTS t: time
                     ( Change [ 2 ]  ( My_Central.Phone_State ( Self )  )  < t
                     & past ( End ( Enter_Digit )  = Now, t )  )  )
        &  ( My_Central.Phone_State ( Self )  = Waiting
        ->   past ( My_Central.Phone_State ( Self ) ,
                     Change [ 2 ]  ( My_Central.Phone_State ( Self )  )  )  = Dialing
           & EXISTS t: time
                     ( Change [ 2 ]  ( My_Central.Phone_State ( Self )  )  < t
                     & past ( End ( Enter_Digit )  = Now, t )  )  )
        &  ( My_Central.Phone_State ( Self )  = Dialing
        ->   past ( My_Central.Phone_State ( Self ) ,
                     Change [ 2 ]  ( My_Central.Phone_State ( Self )  )  )  = Ready_To_Dial
           & EXISTS t: time
                     ( Change [ 2 ]  ( My_Central.Phone_State ( Self )  )  < t
                     & past ( End ( Enter_Digit )  = Now, t )  )  )
        &  ( My_Central.Phone_State ( Self )  = Ready_To_Dial
        ->   past ( My_Central.Phone_State ( Self ) ,
                     Change [ 2 ]  ( My_Central.Phone_State ( Self )  )  )  = Idle )
        &  ( EXISTS t: time
                     ( past ( My_Central.Phone_State ( Self ) , t )  = Idle
                     & FORALL t1: time
                                ( t <= t1
                                & t1 <= Now
                        ->   past ( My_Central.Phone_State ( Self ) , t1 )  ~= Waiting )  )
        ->   ~My_Central.Enabled_Ringback_Pulse ( Self )  )
FURTHER ASSUMPTIONS #1
  FURTHER PROCESS ASSUMPTIONS
    TRANSITION SELECTION
              enabled_transitions CONTAINS any_subset  (  { Stop_Ringback, Stop_Busytone }  )
            & TRUE
         ->   elgible_transitions =  { Stop_Ringback, Stop_Busytone }  INTERSECT enabled_transitions
TRANSITION Pickup
  ENTRY           [ TIME : T1 ]
            ~Offhook
  EXIT
            Offhook
          & ~Dialtone
          & ~Ring
          & ~Ringback
          & ~Busytone
TRANSITION Start_Tone
  ENTRY           [ TIME : T2 ]
            Offhook
          & My_Central.Phone_State ( Self )  = Ready_To_Dial
          & ~Dialtone
          & FORALL t: time
                     ( Change ( Dialtone, t )
                  ->   t < Change ( Offhook )  )
  EXIT
            Dialtone
TRANSITION Enter_Digit ( D: Digit )
  ENTRY           [ TIME : T4 ]
            Offhook
          &  ( My_Central.Phone_State ( Self )  = Ready_To_Dial
             & Dialtone
            |  My_Central.Phone_State ( Self )  = Dialing )
  EXIT
            Next_Digit = D
          & ~Dialtone
TRANSITION Start_Ring
  ENTRY           [ TIME : T5 ]
            ~Offhook
          & My_Central.Phone_State ( Self )  = Ringing
          & My_Central.Enabled_Ring_Pulse ( Self )
          & ~Ring
  EXIT
            Ring
TRANSITION Stop_Ring
  ENTRY           [ TIME : T6 ]
            Ring
          & ~My_Central.Enabled_Ring_Pulse ( Self )
  EXIT
            ~Ring
TRANSITION Start_Ringback
  ENTRY           [ TIME : T7 ]
            Offhook
          & ~Ringback
          & My_Central.Phone_State ( Self )  = Waiting
          & My_Central.Enabled_Ringback_Pulse ( Self )
  EXIT
            Ringback
TRANSITION Stop_Ringback
  ENTRY           [ TIME : T8 ]
            Ringback
          & ~My_Central.Enabled_Ringback_Pulse ( Self )
```

```
      EXIT
                ~Ringback
    TRANSITION Start_Busytone
      ENTRY           [ TIME : T9 ]
                Offhook
              & My_Central.Phone_State ( Self )  = Busy
              & ~Busytone
      EXIT
                Busytone
    TRANSITION Stop_Busytone
      ENTRY           [ TIME : T10 ]
                Busytone
              & My_Central.Phone_State ( Self )  ~= Busy
      EXIT
                ~Busytone
    TRANSITION Hangup
      ENTRY           [ TIME : T11 ]
                Offhook
      EXIT
                ~Offhook
              & ~Dialtone
              & ~Ring
              & ~Ringback
              & ~Busytone
  END Top_Level
END Phone
PROCESS SPECIFICATION Central_Control
  LEVEL Top_Level
    IMPORT
              LD_Timeout, Digit, Digit_List, Connection, Phone_ID, Central_Control_ID, Enabled_State,
              Connection_Status, In_Area, Max_Cust, Plug, Phones.Offhook, Phones.Next_Digit,
              Phones.Pickup, Phones.Enter_Digit
    EXPORT
              Phone_State, Enabled_Ring_Pulse, Enabled_Ringback_Pulse, LDOut_Line, LDOut_Status,
              Receive_LD, Start_LD, Start_Talk_2, Terminate_LD_2
    TYPE
              Area_Phone IS TYPEDEF p: Phone_ID ( In_Area ( p )  = Self )
    CONSTANT
              Uptime_Ring, Downtime_Ring, Uptime_Ringback, Downtime_Ringback, LD_Timeout, Delta: Time,
              Tim1, Tim2, Tim3, Tim4, Tim5, Tim6, Tim7, Tim8, Tim9, Tim10, Tim11, Tim12, Tim13, Tim14,
                Tim15, Tim16: Time,
              Get_ID ( Digit_List ) : Area_Phone,
              Get_Number ( Area_Phone ) , Get_Area ( Central_Control_ID ) , Pick_Area ( Digit_List ) ,
                Pick_Number ( Digit_list ) : Digit_List,
              MAX ( Time, Time, Time, Time, Time, Time, Time, Time, Time, Time, Time, Time, Time, Time,
                      Time, Time ) : Time
    VARIABLE
              Phone_State ( Area_Phone ) : Enabled_State,
              Long_Distance ( Area_Phone ) : Boolean,
              Enabled_Ring_Pulse ( Area_Phone ) , Enabled_Ringback_Pulse ( Area_Phone ) : Boolean,
              Connected_To ( Area_Phone ) : Area_Phone,
              Number ( Area_Phone ) : Digit_List,
              LDOut_Line ( Area_Phone ) : Connection,
              LDOut_Status ( Area_Phone ) : Connection_Status
    AXIOM
              FORALL d: Digit_List
                       ( LIST_LEN  ( Pick_Number ( d )  )  = 7
                       & LIST_LEN  ( Pick_Area ( d )  )  = 3 )
    DEFINE
              Count ( P: Area_Phone ) : Integer ==
                      LIST_LEN  ( Number ( P )  ) ,
              Calling_Out ( P: Area_Phone, L: Connection ) : Boolean ==
                      P.Offhook
                    & Long_Distance ( P )
                    & Get_Area ( Self )  = L [ To_Area ]
                    & Get_Number ( P )  = L [ To_Number ]
                    & Plug ( LDOut_Line ( P ) , L )
    ENVIRONMENT
              FORALL t: Time, L: Connection
                       ( Call ( Terminate_LD_2 ( L ) , t )
                    ->   EXISTS t1: Time
                                  ( t1 < t
                                  &  ( Call ( Receive_LD ( L ) , t1 )
                                    |  Call ( Start_LD ( L ) , t1 )  )  )  )
            & FORALL t: Time, L: Connection
                       ( Call ( Start_Talk_2 ( L ) , t )
                    ->   EXISTS t1: Time
                                  ( t1 < t
                                  & Call ( Start_LD ( L ) , t1 )  )  )
            & FORALL t: Time, L: Connection
                       ( Call ( Start_LD ( L ) , t )
                    ->   EXISTS t1: Time, P: Area_Phone
                                  ( t1 < t
                                  & past ( Phone_State ( P ) , t1 )  = Calling
                                  & past ( Plug ( LDOut_Line ( P ) , L ) , t )  )  )
            & FORALL t: Time
                       ( Call [ 2 ]  ( Receive_LD, t )
                    ->   Call ( Receive_LD )  - t > LD_Timeout )
    INITIAL
              FORALL P: Area_Phone ( Phone_State ( P )  = Idle
                                   & Number ( P )  = NIL
                                   & ~Enabled_Ring_Pulse ( P )
                                   & ~Enabled_Ringback_Pulse ( P )
                                   & ~Long_Distance ( P )
                                   & LDOut_Status ( P )  = Available )
```

```
INVARIANT
          FORALL P: Area_Phone
                   (  ( Long_Distance ( P )
                   ->   Count ( P )  >= 0
                      & Count ( P )  <= 11 )
                   &  ( ~Long_Distance ( P )
                   ->    ( Count ( P )  >= 0
                         & Count ( P )  <= 7
                         &  ( Phone_State ( P )  = Waiting
                         ->   Phone_State ( Connected_To ( P )  )  = Ringing )
                         &  ( Phone_State ( P )  = Ringing
                         ->   Phone_State ( Connected_To ( P )  )  = Waiting )
                         &  ( Phone_State ( P )  = Talk
                         ->   Phone_State ( Connected_To ( P )  )  = Talk )  )  )  )
CONSTRAINT
          FORALL P: Area_Phone (  ( Phone_State' ( P )  = Busy
                                  |  Phone_State' ( P )  = Alarm
                                  |  Phone_State' ( P )  = Disconnecting )
                                & Phone_State ( P )  ~= Phone_State' ( P )
                             ->   Phone_State ( P )  = Idle )
SCHEDULE
          FORALL P: Area_Phone, t, t1, t2: Time
                    ( t <= t1
                   & t1 < t2
                   & Change [ 2 ]  ( Phone_State ( P ) , t )
                   & past ( Phone_State ( P ) , t )  = Idle
                   & P.End ( Pickup, t1 )
                   & P.Offhook
                   & Change ( Phone_State ( P ) , t2 )
                ->   past ( Phone_State ( P ) , t2 )  = Ringing
                  |  past ( Phone_State ( P ) , t2 )  = Ready_To_Dial
                   & t2 <= t1 + 2 )
        & FORALL P: Area_Phone
                   ( Phone_State ( P )  = Ringing
                   & Now - Change ( Phone_State ( P )  )  >= Downtime_Ring
                ->   EXISTS n: Integer
                            ( End [ n ]  ( Enable_Ring ( P )  )  > Change ( Phone_State ( P )  )
                            & End [ n ]  ( Enable_Ring ( P )  )  <= Change ( Phone_State ( P )  )  +
                                Downtime_Ring )  )
        & FORALL P: Area_Phone
                   ( Phone_State ( P )  = Ringing
                   & End ( Enable_Ring ( P )  )  > Change ( Phone_State ( P )  )
                   & Now >= End ( Enable_Ring ( P )  )  + Uptime_Ring + Delta
                ->    ( End ( Disable_Ring_Pulse ( P )  )  >=
                          End ( Enable_Ring ( P )  )  + Uptime_Ring
                      & End ( Disable_Ring_Pulse ( P )  )  <=
                          End ( Enable_Ring ( P )  )  + Uptime_Ring + Delta )  )
        & FORALL P: Area_Phone
                   ( Phone_State ( P )  = Ringing
                   & End ( Disable_Ring_Pulse ( P )  )  > Change ( Phone_State ( P )  )
                   & Now >= End ( Disable_Ring_Pulse ( P )  )  + Downtime_Ring + Delta
                ->    ( End ( Enable_Ring ( P )  )  >=
                          End ( Disable_Ring_Pulse ( P )  )  + Downtime_Ring
                      & End ( Enable_Ring ( P )  )  <=
                          End ( Disable_Ring_Pulse ( P )  )  + Downtime_Ring + Delta )  )
        & FORALL P: Area_Phone
                   ( ~Long_Distance ( P )
                   & Phone_State ( P )  = Waiting
                   & Now - End ( Process_Local_Call ( P )  )  >= Downtime_Ring
                ->   EXISTS n, m: Integer
                            ( End [ n ]  ( Enable_Ring ( Connected_To ( P )  )  )  >
                                End ( Process_Local_Call ( P )  )
                            & End [ n ]  ( Enable_Ring ( Connected_To ( P )  )  )  <=
                                End ( Process_Local_Call ( P )  )  + Downtime_Ring
                            & End [ m ]  ( Enable_Ringback ( P )  )  >
                                End ( Process_Local_Call ( P )  )
                            & End [ m ]  ( Enable_Ringback ( P )  )  <=
                                End [ n ]  ( Enable_Ring ( Connected_To ( P )  )  )  + 0.5 )  )
        & FORALL P: Area_Phone
                   ( Phone_State ( P )  = Waiting
                   & End ( Enable_Ringback ( P )  )  > Change ( Phone_State ( P )  )
                   & Now >= End ( Enable_Ringback ( P )  )  + Uptime_Ringback + Delta
                ->    ( End ( Disable_Ringback_Pulse ( P )  )  >= End ( Enable_Ringback ( P )  )  +
                          Uptime_Ringback
                      & End ( Disable_Ringback_Pulse ( P )  )  <= End ( Enable_Ringback ( P )  )  +
                          Uptime_Ringback + Delta )  )
        & FORALL P: Area_Phone
                   ( Phone_State ( P )  = Waiting
                   & End ( Disable_Ringback_Pulse ( P )  )  > Change ( Phone_State ( P )  )
                   & Now >= End ( Disable_Ringback_Pulse ( P )  )  + Downtime_Ringback + Delta
                ->    ( End ( Enable_Ringback ( P )  )  >= End ( Disable_Ringback_Pulse ( P )  )  +
                          Downtime_Ringback
                      & End ( Enable_Ringback ( P )  )  <= End ( Disable_Ringback_Pulse ( P )  )  +
                          Downtime_Ringback + Delta )  )
IMPORTED VARIABLE
          SET_SIZE (  { SETDEF P: Area_Phone ( Now - 2 <= P.Start ( Pickup )
                                             & P.Start ( Pickup )  <= Now )  }  )  <= Max_Cust
FURTHER ASSUMPTIONS #1
  FURTHER PROCESS ASSUMPTIONS
    TRANSITION SELECTION
            enabled_transitions CONTAINS  { Give_Dial_Tone }
          & TRUE
         ->   elgible_transitions =  { Give_Dial_Tone }
```

```
    CONSTANT REFINEMENT
              2 > MAX ( Tim1, Tim2, Tim3, Tim4, Tim5, Tim6, Tim7, Tim8, Tim9, Tim10, Tim11, Tim12,
                          Tim13, Tim14, Tim15, Tim16 )  +  ( Max_Cust + 1 )  * Tim1
TRANSITION Give_Dial_Tone ( P: Area_Phone )
  ENTRY           [ TIME : Tim1 ]
          P.Offhook
        & Phone_State ( P )  = Idle
  EXIT
          Phone_State ( P )  BECOMES Ready_To_Dial
TRANSITION Process_Digit ( P: Area_Phone )
  ENTRY           [ TIME : Tim2 ]
          P.Offhook
        &  (  ( Long_Distance ( P )
              & Count ( P )  < 11 )
          |   ( ~Long_Distance ( P )
              & Count ( P )  < 7 )  )
        &  (  ( Phone_State ( P )  = Ready_To_Dial
              & P.End ( Enter_Digit )  > End ( Give_Dial_Tone ( P )  )  )
          |   ( Phone_State ( P )  = Dialing )
           & P.End ( Enter_Digit )  > End ( Process_Digit ( P )  )  )
  EXIT
          IF
                  Phone_State' ( P )  = Ready_To_Dial
          THEN
                  IF
                          P.Next_Digit' = 1
                  THEN
                          Long_Distance ( P )  BECOMES True
                  ELSE
                          Long_Distance ( P )  BECOMES False
                  FI
                & Phone_State ( P )  BECOMES Dialing
                & Number ( P )  BECOMES LISTDEF  ( P.Next_Digit' )
          ELSE
                  Number ( P )  BECOMES Number' ( P )  CONCAT LISTDEF  ( P.Next_Digit' )
          FI
TRANSITION Process_Local_Call ( P: Area_Phone )
  ENTRY           [ TIME : Tim3 ]
          P.Offhook
        & ~Long_Distance ( P )
        & Count ( P )  = 7
        & Phone_State ( P )  = Dialing
        & ~Get_ID ( Number ( P )  ) .Offhook
        & Phone_State ( Get_ID ( Number ( P )  )  )  = Idle
  EXIT
          Phone_State ( Get_ID ( Number' ( P )  )  )  = Ringing
        & Phone_State ( P )  = Waiting
        & ~Long_Distance ( Get_ID ( Number' ( P )  )  )
        & Connected_To ( P )  = Get_ID ( Number' ( P )  )
        & Connected_To ( Get_ID ( Number' ( P )  )  )  = P
        & FORALL P1: Area_Phone
                   ( P1 ~= P
                   & P1 ~= Get_ID ( Number' ( P )  )
                ->   NOCHANGE  ( Phone_State ( P1 )  )
                   & NOCHANGE  ( Connected_To ( P1 )  )  )
  EXCEPT          [ TIME : Tim3 ]
          P.Offhook
        & ~Long_Distance ( P )
        & Count ( P )  = 7
        & Phone_State ( P )  = Dialing
        &  ( Get_ID ( Number ( P )  ) .Offhook
          |  Phone_State ( Get_ID ( Number ( P )  )  )  ~= Idle )
  EXIT
          Phone_State ( P ) BECOMES Busy
TRANSITION Connect_Long_Distance ( P: Area_Phone )
  ENTRY           [ TIME : Tim4 ]
          P.Offhook
        & Count ( P )  = 11
        & Long_Distance ( P )
        & Phone_State ( P )  = Dialing
        & Pick_Area ( Number ( P )  )  ~= Get_Area ( Self )
  EXIT
          LDOut_Line ( P )  [ From_Area ]  = Get_Area ( Self )
        & LDOut_Line ( P )  [ From_Number ]  = Get_Number ( P )
        & LDOut_Line ( P )  [ To_Area ]  = Pick_Area ( Number' ( P )  )
        & LDOut_Line ( P )  [ To_Number ]  = Pick_Number ( Number' ( P )  )
        & LDOut_Status ( P ) BECOMES In_Progress
        & Phone_State ( P ) BECOMES Calling
        & FORALL P1: Area_Phone ( P1 ~= P
                              ->   NOCHANGE ( LDOut_Line ( P1 )  )  )
  EXCEPT          [ TIME : Tim4 ]
          P.Offhook
        & Count ( P )  = 11
        & Long_Distance ( P )
        & Phone_State ( P )  = Dialing
        & Pick_Area ( Number ( P )  )  = Get_Area ( Self )
  EXIT
          Long_Distance ( P ) BECOMES False
        & Number ( P ) BECOMES Pick_Number ( Number' ( P )  )
TRANSITION Enable_Ring ( P: Area_Phone )
  ENTRY           [ TIME : Tim5 ]
          ~P.Offhook
        & Phone_State ( P )  = Ringing
        & ~Enabled_Ring_Pulse ( P )
```

```
          & FORALL t: Time ( End ( Disable_Ring_Pulse ( P ) , t )
                           & FORALL t1: Time ( t <= t1
                                             & t1 <= Now
                                          ->   past ( Phone_State ( P ) , t1 )  = Ringing )
                        ->   Now - t >= Downtime_Ring )
  EXIT
            Enabled_Ring_Pulse ( P ) BECOMES True
TRANSITION Disable_Ring_Pulse ( P: Area_Phone )
  ENTRY           [ TIME : Tim6 ]
            Enabled_Ring_Pulse ( P )
          &  ( Now - End ( Enable_Ring ( P )  )  >= Uptime_Ring
            |  P.Offhook )
  EXIT
            Enabled_Ring_Pulse ( P ) BECOMES False
TRANSITION Enable_Ringback ( P: Area_Phone )
  ENTRY           [ TIME : Tim7 ]
            P.Offhook
          & Phone_State ( P )  = Waiting
          & ~Enabled_Ringback_Pulse ( P )
          & FORALL t: Time
                     ( End ( Disable_Ringback_Pulse ( P ) , t )
                     & FORALL t1: Time
                                ( t <= t1
                                & t1 <= Now
                             ->   past ( Phone_State ( P ) , t1 )  = Waiting )
                  ->   Now - t >= Downtime_Ringback )
  EXIT
            Enabled_Ringback_Pulse ( P ) BECOMES True
TRANSITION Disable_Ringback_Pulse ( P: Area_Phone )
  ENTRY           [ TIME : Tim8 ]
            Enabled_Ringback_Pulse ( P )
          &  ( Now - End ( Enable_Ringback ( P )  )  >= Uptime_Ringback
            |  ~P.Offhook )
  EXIT
            Enabled_Ringback_Pulse ( P ) BECOMES False
TRANSITION Receive_LD ( LDIn_Line: Connection )
  ENTRY           [ TIME : Tim9 ]
            LDIn_Line [ To_Area ]  = Get_Area ( Self )
          & Phone_State ( Get_ID ( LDIn_Line [ To_Number ]  )  )  = Idle
          & ~Get_ID ( LDIn_Line [ To_Number ]  ) .Offhook
  EXIT
            Phone_State ( Get_ID ( LDIn_Line [ To_Number ]  )  )  BECOMES Ringing
          & Long_Distance ( Get_ID ( LDIn_Line [ To_Number ]  )  )
          & LDOut_Status ( Get_ID ( LDIn_Line [ To_Number ]  )  )  BECOMES Connect
          & Plug ( LDOut_Line ( Get_ID ( LDIn_Line [ To_Number ]  )  ) , LDIn_Line )
          & FORALL P: Area_Phone
                     ( P ~= Get_ID ( LDIn_Line [ To_Number ]  )
                  ->   NOCHANGE  ( LDOut_Line ( P )  )  )
TRANSITION Start_Talk_1 ( P: Area_Phone )
  ENTRY           [ TIME : Tim10 ]
            P.Offhook
          & Phone_State ( P )  = Ringing
  EXIT
            Phone_State ( P )  = Talk
          & IF
                    ~Long_Distance’ ( P )
            THEN
                    Phone_State ( Connected_To’ ( P )  )  = Talk
                  & FORALL P1: Area_Phone
                            ( P1 ~= P
                            & P1 ~= Connected_To’ ( P )  )
               ->   NOCHANGE  ( Phone_State ( P )  )
            ELSE
                    LDOut_Status ( P )  BECOMES Talking
            FI
TRANSITION Start_Talk_2 ( LDIn_Line: Connection )
  ENTRY           [ TIME : Tim11 ]
            EXISTS P: Area_Phone
                     ( Calling_Out ( P, LDIn_Line )
                     & Phone_State ( P )  = Waiting
                     & LDOut_Status ( P )  = Connect )
  EXIT
            EXISTS P: Area_Phone
                     ( Calling_Out’ ( P, LDIn_Line )
                     & Phone_State’ ( P )  = Waiting
                     & LDOut_Status’ ( P )  = Connect
                     & LDOut_Status ( P )  BECOMES Talking
                     & Phone_State ( P )  BECOMES Talk )
TRANSITION Start_LD ( LDIn_Line: Connection, LDIn_Status: Connection_Status )
  ENTRY           [ TIME : Tim12 ]
            LDIn_Status = Connect
          & EXISTS P: Area_Phone
                     ( Calling_Out ( P, LDIn_Line )
                     & Phone_State ( P )  = Calling
                     & LDOut_Status ( P )  = In_Progress )
  EXIT
            EXISTS P: Area_Phone
                     ( Calling_Out’ ( P, LDIn_Line )
                     & Phone_State’ ( P )  = Calling
                     & LDOut_Status’ ( P )  = In_Progress
                     & LDOut_Status ( P )  BECOMES Connect
                     & Phone_State ( P )  BECOMES Waiting )
```

```
  EXCEPT          [ TIME : Tim12 ]
            LDIn_Status = Disconnect
          & EXISTS P: Area_Phone
                     ( Calling_Out ( P, LDIn_Line )
                     & Phone_State ( P )  = Calling
                     & LDOut_Status ( P )  = In_Progress )
  EXIT
            EXISTS P: Area_Phone
                     ( Calling_Out’ ( P, LDIn_Line )
                     & Phone_State’ ( P )  = Calling
                     & LDOut_Status’ ( P )  = In_Progress
                     & LDOut_Status ( P )  BECOMES Available
                     & Phone_State ( P )  BECOMES Busy )
TRANSITION Terminate_LD_1 ( P: Area_Phone )
  ENTRY           [ TIME : Tim13 ]
            ~P.Offhook
          & Long_Distance ( P )
          & Phone_State ( P )  ~= Idle
          & Phone_State ( P )  ~= Ringing
          & LDOut_Line ( P )  [ From_Area ]  = Get_Area ( Self )
          & LDOut_Line ( P )  [ From_Number ]  = Get_Number ( P )
          & LDOut_Status ( P )  ~= Available
  EXIT
            Phone_State ( P )  BECOMES Idle
          & ~Enabled_Ringback_Pulse ( P )
          & LDOut_Status ( P )  BECOMES Available
TRANSITION Generate_Alarm ( P: Area_Phone )
  ENTRY           [ TIME : Tim14 ]
            P.Offhook
          &  ( Phone_State ( P )  = Ready_To_Dial
            |  Phone_State ( P )  = Dialing
             & P.Call ( Enter_Digit )  < Start ( Process_Digit ( P )  )  )
          &  ( Count ( P )  = 0
             & Now - End ( Give_Dial_Tone ( P )  )  > 30
            |  Count ( P )  > 0
             & Count ( P )  < 7
             & Now - End ( Process_Digit ( P )  )  > 20
            |  ~Long_Distance ( P )
             & Count ( P )  < 7
             & Now - End ( Give_Dial_Tone ( P )  )  > 100
            |  Long_Distance ( P )
             & Count ( P )  < 11
             & Now - End ( Give_Dial_Tone ( P )  )  > 100 )
  EXIT
            Phone_State ( P ) BECOMES Alarm
TRANSITION Terminate_Local_Call ( P: Area_Phone )
  ENTRY           [ TIME : Tim15 ]
            ~P.Offhook
          & ~Long_Distance ( P )
          & Phone_State ( P )  ~= Idle
          & Phone_State ( P )  ~= Ringing
  EXIT
            Phone_State ( P )  = Idle
          & ~Enabled_Ringback_Pulse ( P )
          & IF
                    Phone_State’ ( P )  = Talk
                 |  Phone_State’ ( P )  = Waiting
            THEN
                    IF
                            Phone_State’ ( P )  = Talk
                    THEN
                            Phone_State ( Connected_To’ ( P )  )  = Disconnecting
                    ELSE
                            Phone_State ( Connected_To’ ( P )  )  = Idle
                          & ~Enabled_Ring_Pulse ( Connected_To’ ( P )  )
                    FI
                  & FORALL P1: Area_Phone
                             ( P1 ~= P
                             & P1 ~= Connected_To’ ( P )
                          ->   NOCHANGE  ( Phone_State ( P1 )  )  )
            ELSE
                    FORALL P1: Area_Phone
                             ( P1 ~= P
                          ->   NOCHANGE  ( Phone_State ( P1 )  )  )
            FI
TRANSITION Terminate_LD_2 ( LDIn_Line: Connection )
  ENTRY           [ TIME : Tim16 ]
            EXISTS P: Area_Phone
                     ( Calling_Out ( P, LDIn_Line )
                     & Phone_State ( P )  = Talk
                     & LDOut_Status ( P )  = Talking )
  EXIT
            EXISTS P: Area_Phone
                     ( Calling_Out’ ( P, LDIn_Line )
                     & Phone_State’ ( P )  = Talk
                     & LDOut_Status’ ( P )  = Talking
                     & LDOut_Status ( P )  BECOMES Disconnect
                     & Phone_State ( P )  BECOMES Disconnecting )
  EXCEPT          [ TIME : Tim16 ]
            EXISTS P: Area_Phone
                     ( Calling_Out ( P, LDIn_Line )
                     & Phone_State ( P )  = Ringing
                     & LDOut_Status ( P )  = Connect )
```

```
        EXIT
                  EXISTS P: Area_Phone
                           ( Calling_Out' ( P, LDIn_Line )
                           & Phone_State' ( P )  = Ringing
                           & LDOut_Status' ( P )  = Connect
                           & LDOut_Status ( P )  BECOMES Available
                           & Phone_State ( P )  BECOMES Idle
                           & ~Enabled_Ring_Pulse ( P )  )
    END Top_Level
  END Central_Control
END Phone_System
```

8. Railroad Crossing

```
SPECIFICATION Railroad_Crossing
  GLOBAL SPECIFICATION Railroad_Crossing
    PROCESSES
              the_gate: Gate,
              the_sensors: array [ 1..n_tracks ]  of Sensor
    TYPE
              pos_integer: TYPEDEF i: integer ( i > 0 ) ,
              pos_real: TYPEDEF i: real ( i > 0 ) ,
              gate_position:  ( raised, raising, lowered, lowering ) ,
              sensor_id: TYPEDEF i: id ( IDTYPE  ( i )  = Sensor )
    CONSTANT
              n_tracks: pos_integer,
              min_speed, max_speed: pos_real,
              dist_R_to_I, dist_I_to_out: pos_real,
              response_time, wait_time: pos_real
    AXIOM
              max_speed >= min_speed
            & response_time < dist_R_to_I / max_speed
    SCHEDULE
      /* gate will be down before fastest train reaches crossing */
               ( EXISTS s: sensor_id
                          ( s.train_in_R
                          & now - s.Call ( enter_R )  >= dist_R_to_I / max_speed )
            ->   the_gate.position = lowered )
      /* gate will be up after slowest train exits crossing and a reasonable wait time has elapsed */
            &  ( FORALL s: sensor_id
                          ( ~s.train_in_R
                          &  ( EXISTS t: time
                                        ( s.Call ( enter_R, t )  )
                          ->   now - s.Call ( enter_R )  >=  ( dist_R_to_I + dist_I_to_out )  / min_speed +
                                 wait_time )  )
            ->   the_gate.position = raised )
  END Railroad_Crossing
  PROCESS SPECIFICATION Sensor
    LEVEL Top_Level
      IMPORT
                pos_real, max_speed, min_speed, dist_R_to_I, dist_I_to_out, response_time
      EXPORT
                train_in_R, enter_R
      CONSTANT
                enter_dur, exit_dur: pos_real
      VARIABLE
                train_in_R: boolean
      AXIOM
                response_time >= enter_dur
              &  ( dist_R_to_I + dist_I_to_out )  / min_speed >= response_time + exit_dur
      ENVIRONMENT
        /* only one train will be in the region at the same time on the same track */
                Call ( enter_R, now )
              & EXISTS t: time
                         ( t >= 0
                         & t <= now
                         & Call [ 2 ]  ( enter_R, t )  )
           ->   Call ( enter_R )  - Call [ 2 ]  ( enter_R )  >  ( dist_R_to_I + dist_I_to_out )  / min_speed
      INITIAL
                ~train_in_R
      INVARIANT
        /* once a sensor reports a train, it will keep reporting a train at least as long as it takes the
           fastest train to exit the region */
                Change ( train_in_R, now )
              & ~train_in_R
           ->   0 <= now -  (  ( dist_R_to_I + dist_I_to_out )  / max_speed - response_time )
              & FORALL t: time
                         ( now -  (  ( dist_R_to_I + dist_I_to_out )  / max_speed - response_time )  <=
                             t
                         & t < now
                      ->   past ( train_in_R, t )  )
      SCHEDULE
        /* train will be sensed within enter_dur of call */
                 ( now >= response_time
                 & Call ( enter_R, now - response_time )
              ->   train_in_R )
        /* sensor will be reset when the slowest train is beyond the crossing */

              &  ( now >=  ( dist_R_to_I + dist_I_to_out )  / min_speed
                 & Call ( enter_R, now -  ( dist_R_to_I + dist_I_to_out )  / min_speed )
              ->   ~train_in_R )
```

```
      TRANSITION enter_R
        ENTRY           [ TIME : enter_dur ]
                  ~train_in_R
        EXIT
                  train_in_R
      TRANSITION exit_I
        ENTRY           [ TIME : exit_dur ]
                  train_in_R
                & now - Start ( enter_R )  >=  ( dist_R_to_I + dist_I_to_out )  / min_speed - exit_dur
        EXIT
                  ~train_in_R
    END Top_Level
  END Sensor
  PROCESS SPECIFICATION Gate
    LEVEL Top_Level
      IMPORT
                pos_real, gate_position, max_speed, dist_R_to_I, dist_I_to_out, wait_time, response_time,
                sensor_id, the_sensors.train_in_R
      EXPORT
                position
      CONSTANT
                lower_dur, raise_dur, up_dur, down_dur: pos_real,
                raise_time, lower_time: pos_real
      VARIABLE
                position: gate_position
      AXIOM
                wait_time >= raise_dur + raise_time + up_dur
              & dist_R_to_I / max_speed >= response_time + lower_dur + lower_time + down_dur + raise_dur
              & dist_R_to_I / max_speed >= response_time + lower_dur + lower_time + down_dur + up_dur
      INITIAL
                position = raised
      SCHEDULE
        /* gate will be down before fastest train reaches crossing */
                 ( EXISTS s: sensor_id
                            ( s.train_in_R
                            & now - Change ( s.train_in_R )  >= dist_R_to_I / max_speed - response_time )
              ->   position = lowered )
        /* gate will be up after slowest train exits crossing and enough time has elapsed for gate to be
           raised */
              &  ( FORALL s: sensor_id
                            ( FORALL t: time
                                       ( now - wait_time <= t
                                       & t <= now
                                   ->   ~past ( s.train_in_R, t )  )  )
              ->   position = raised )
      IMPORTED VARIABLE
        /* once a sensor reports a train, it will keep reporting a train at least as long as it takes the
           fastest train to exit the region */
                FORALL s: sensor_id
                         ( Change ( s.train_in_R, now )
                         & ~s.train_in_R
                      ->   0 <= now -  (  ( dist_R_to_I + dist_I_to_out )  / max_speed - response_time )
                         & FORALL t: time
                                    ( now -  (  ( dist_R_to_I + dist_I_to_out )  / max_speed -
                                                 response_time )  <= t
                                    & t < now
                                 ->   past ( s.train_in_R, t )  )  )
      TRANSITION lower
        ENTRY           [ TIME : lower_dur ]
                  ~ ( position = lowering
                   |  position = lowered )
                & EXISTS s: sensor_id
                           ( s.train_in_R )
        EXIT
                  position = lowering
      TRANSITION down
        ENTRY           [ TIME : down_dur ]
                  position = lowering
                & now - End ( lower )  >= lower_time
        EXIT
                  position = lowered
      TRANSITION raise
        ENTRY           [ TIME : raise_dur ]
                  ~ ( position = raising
                   |  position = raised )
                & FORALL s: sensor_id
                           ( ~s.train_in_R )
        EXIT
                  position = raising
      TRANSITION up
        ENTRY           [ TIME : up_dur ]
                  position = raising
                & now - End ( raise )  >= raise_time
        EXIT
                  position = raised
    END Top_Level
  END Gate
END Railroad_Crossing
```

9. Stoplight Control System

```
SPECIFICATION Stoplight
  GLOBAL SPECIFICATION Stoplight
    PROCESSES
              the_controller: Controller,
              the_sensors: array [ 4 ]  of Sensor,
              the_LT_sensors: array [ 4 ]  of Sensor
    TYPE
              pos_real: TYPEDEF r: real ( r > 0 )
    CONSTANT
              min_green, min_yellow, max_wait: pos_real
  END Stoplight
  PROCESS SPECIFICATION Controller
    LEVEL Top_Level
      IMPORT
                pos_real, min_yellow, min_green, max_wait, the_sensors, the_LT_sensors,
                the_sensors.is_object, the_LT_sensors.is_object
      TYPE
                direction: TYPEDEF i: integer ( 1 <= i
                                              & i <= 4 ) ,
                signal:  ( green, yellow, red )
      CONSTANT
                main_dir: direction,
                change_dur: pos_real
      VARIABLE
                circle ( direction ) : signal,
                arrow ( direction ) : signal
      AXIOM
                max_wait >= 3 * min_green + 4 * min_yellow
              & min_green >= change_dur
              & min_yellow >= change_dur
      DEFINE
                adj1 ( d: direction ) : direction ==
                         ( d + 1 )  mod 4,
                opp ( d: direction ) : direction ==
                         ( d + 2 )  mod 4,
                adj2 ( d: direction ) : direction ==
                         ( d + 3 )  mod 4,
                car ( d: direction ) : boolean ==
                        the_sensors [ d ] .is_object,
                LT_car ( d: direction ) : boolean ==
                        the_LT_sensors [ d ] .is_object
      INITIAL
                circle ( main_dir )  = green
              & arrow ( main_dir )  = green
              & FORALL d: direction
                         ( d ~= main_dir
                      ->   circle ( d )  = red )
              & FORALL d: direction
                         ( d ~= main_dir
                      ->   arrow ( d )  = red )
      INVARIANT
        /* there is always some direction that is yellow or green */
                 ( EXISTS d: direction
                            ( circle ( d )  ~= red
                           |  arrow ( d )  ~= red )  )
        /* if a light is green, then all opposing lights are red */
              &  ( FORALL d: direction
                            ( circle ( d )  = green
                         ->   arrow ( opp ( d )  )  = red
                            & circle ( adj1 ( d )  )  = red
                            & circle ( adj2 ( d )  )  = red
                            & arrow ( adj1 ( d )  )  = red
                            & arrow ( adj2 ( d )  )  = red )  )
              &  ( FORALL d: direction
                            ( arrow ( d )  = green
                         ->   circle ( opp ( d )  )  = red
                            & circle ( adj1 ( d )  )  = red
                            & circle ( adj2 ( d )  )  = red
                            & arrow ( adj1 ( d )  )  = red
                            & arrow ( adj2 ( d )  )  = red )  )
        /* cars must only wait a fixed amount of time before the next green */
              &  ( FORALL d: direction
                            ( now >= max_wait
                            & FORALL t: time
                                       ( now - max_wait <= t
                                       & t <= now
                                    ->   past ( car ( d ) , t )  )
                         ->   EXISTS t: time
                                       ( now - max_wait <= t
                                       & t <= now
                                       & past ( circle ( d ) , t )  = green )  )  )
              &  ( FORALL d: direction
                            ( now >= max_wait
                            & FORALL t: time
                                       ( now - max_wait <= t
                                       & t <= now
                                    ->   past ( LT_car ( d ) , t )  )
                         ->   EXISTS t: time
                                       ( now - max_wait <= t
                                       & t <= now
                                       & past ( arrow ( d ) , t )  = green )  )  )
```

```
  /* light will stay green for at least min_green */
        &  ( FORALL d: direction
                      ( Change ( circle ( d ) , now )
                      & circle ( d )  = yellow
                   ->   FORALL t: time
                                 ( Change [ 2 ]  ( circle ( d ) , t )
                              ->   t <= now - min_green )  )  )
        &  ( FORALL d: direction
                      ( Change ( arrow ( d ) , now )
                      & arrow ( d )  = yellow
                   ->   FORALL t: time
                                 ( Change [ 2 ]  ( arrow ( d ) , t )
                              ->   t <= now - min_green )  )  )
  /* light will stay yellow for at least min_yellow */
        &  ( FORALL d: direction
                      ( Change ( circle ( d ) , now )
                      & circle ( d )  = red
                   ->   FORALL t: time
                                 ( Change [ 2 ]  ( circle ( d ) , t )
                              ->   t <= now - min_yellow )  )  )
        &  ( FORALL d: direction
                      ( Change ( arrow ( d ) , now )
                      & arrow ( d )  = red
                   ->   FORALL t: time
                                 ( Change [ 2 ]  ( arrow ( d ) , t )
                              ->   t <= now - min_yellow )  )  )
CONSTRAINT
  /* lights change from green to yellow to red to green */
           ( FORALL d: direction
                      (  ( circle ( d )  = yellow
                         & circle’ ( d )  ~= circle ( d )
                      ->   circle’ ( d )  = green )
                      &  ( circle ( d )  = red
                         & circle’ ( d )  ~= circle ( d )
                      ->   circle’ ( d )  = yellow )
                      &  ( circle ( d )  = green
                         & circle’ ( d )  ~= circle ( d )
                      ->   circle’ ( d )  = red )  )  )
        &  ( FORALL d: direction
                      (  ( arrow ( d )  = yellow
                         & arrow’ ( d )  ~= arrow ( d )
                      ->   arrow’ ( d )  = green )
                      &  ( arrow ( d )  = red
                         & arrow’ ( d )  ~= arrow ( d )
                      ->   arrow’ ( d )  = yellow )
                      &  ( arrow ( d )  = green
                         & arrow’ ( d )  ~= arrow ( d )
                      ->   arrow’ ( d )  = red )  )  )
TRANSITION Give_Green_Circle ( d: direction )
  ENTRY            [ TIME : change_dur ]
            car ( d )
          & circle ( d )  = red
          & circle ( opp ( d )  )  ~= yellow
          &  ( arrow ( d )  = yellow
             & now - Change ( arrow ( d )  )  >= min_yellow - change_dur
            |  arrow ( opp ( d )  )  = yellow
             & now - Change ( arrow ( opp ( d )  )  )  >= min_yellow - change_dur )
  EXIT
            circle ( d )  = green
          & Nochange  ( circle ( adj1 ( d )  )  )
          & Nochange  ( circle ( adj2 ( d )  )  )
          & IF
                    car’ ( opp ( d )  )
            THEN
                    circle ( opp ( d )  )  = green
            ELSE
                    Nochange  ( circle ( opp ( d )  )  )
            FI
          & IF
                     ( arrow’ ( d )  = yellow )
            THEN
                    arrow ( d )  = red
            ELSE
                    Nochange  ( arrow ( d )  )
            FI
          & arrow ( opp ( d )  )  = red
          & Nochange  ( arrow ( adj1 ( d )  )  )
          & Nochange  ( arrow ( adj2 ( d )  )  )
  EXCEPT           [ TIME : change_dur ]
            car ( d )
          & circle ( d )  = red
          &  ( circle ( adj1 ( d )  )  = yellow
             & circle ( adj2 ( d )  )  ~= green
             & arrow ( adj1 ( d )  )  ~= green
             & now - Change ( circle ( adj1 ( d )  )  )  >= min_yellow - change_dur
            |  circle ( adj2 ( d )  )  = yellow
             & circle ( adj1 ( d )  )  ~= green
             & arrow ( adj2 ( d )  )  ~= green
             & now - Change ( circle ( adj2 ( d )  )  )  >= min_yellow - change_dur )
          & ~LT_car ( d )
          & ~LT_car ( opp ( d )  )
  EXIT
            circle ( d )  = green
          & circle ( adj1 ( d )  )  = red
          & circle ( adj2 ( d )  )  = red
```

```
        & IF
                car’ ( opp ( d )  )
          THEN
                circle ( opp ( d )  )  = green
          ELSE
                Nochange  ( circle ( opp ( d )  )  )
          FI
        & arrow ( adj1 ( d )  )  = red
        & arrow ( adj2 ( d )  )  = red
        & Nochange  ( arrow ( d )  )
        & Nochange  ( arrow ( opp ( d )  )  )
EXCEPT        [ TIME : change_dur ]
          car ( d )
        & circle ( d )  = red
        &  ( arrow ( adj1 ( d )  )  = yellow
           & arrow ( adj2 ( d )  )  ~= green
           & circle ( adj1 ( d )  )  = red
           & now - Change ( arrow ( adj1 ( d )  )  )  >= min_yellow - change_dur
          |  arrow ( adj2 ( d )  )  = yellow
           & arrow ( adj1 ( d )  )  ~= green
           & circle ( adj2 ( d )  )  = red
           & now - Change ( arrow ( adj2 ( d )  )  )  >= min_yellow - change_dur )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
        & ~LT_car ( d )
        & ~LT_car ( opp ( d )  )
EXIT
          circle ( d )  = green
        & circle ( adj1 ( d )  )  = red
        & circle ( adj2 ( d )  )  = red
        & IF
                car’ ( opp ( d )  )
          THEN
                circle ( opp ( d )  )  = green
          ELSE
                Nochange  ( circle ( opp ( d )  )  )
          FI
        & arrow ( adj1 ( d )  )  = red
        & arrow ( adj2 ( d )  )  = red
        & Nochange  ( arrow ( d )  )
        & Nochange  ( arrow ( opp ( d )  )  )
EXCEPT        [ TIME : change_dur ]
          car ( d )
        & circle ( d )  = red
        & circle ( opp ( d )  )  = green
        & arrow ( opp ( d )  )  = red
        & FORALL d2: direction
                 ( ~LT_car ( d2 )  )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
EXIT
          circle ( d )  BECOMES green
EXCEPT        [ TIME : change_dur ]
          car ( d )
        & circle ( d )  = red
        & arrow ( d )  = green
        & arrow ( opp ( d )  )  = red
        & ~car ( opp ( d )  )
EXIT
          circle ( d )  BECOMES green
EXCEPT        [ TIME : change_dur ]
          d = main_dir
        & circle ( d )  ~= red
        & arrow ( d )  ~= red
        &  ( circle ( d )  = yellow
           & now - Change ( circle ( d )  )  >= min_yellow - change_dur
          |  arrow ( d )  = yellow
           & now - Change ( arrow ( d )  )  >= min_yellow - change_dur )
        & FORALL d2: direction
                 (  ( circle ( d2 )  ~= yellow
                 ->   ~car ( d2 )  )
                 &  ( arrow ( d2 )  ~= yellow
                 ->   ~LT_car ( d2 )  )  )
EXIT
          IF
                circle’ ( d )  = yellow
          THEN
                circle ( d )  = red
              & circle ( opp ( d )  )  = red
          ELSE
                Nochange  ( circle ( d )  )
              & circle ( opp ( d )  )  = green
          FI
        & IF
                circle’ ( d )  = yellow
              & arrow’ ( d )  = yellow
          THEN
                circle ( adj1 ( d )  )  = green
          ELSE
                circle ( adj1 ( d )  )  = red
          FI
        & circle ( adj2 ( d )  )  = red
```

```
          & IF
                  arrow’ ( d )  = yellow
            THEN
                  arrow ( d )  = red
                & arrow ( opp ( d )  )  = red
            ELSE
                  Nochange  ( arrow ( d )  )
                & arrow ( opp ( d )  )  = green
            FI
          & arrow ( adj1 ( d )  )  = red
          & arrow ( adj2 ( d )  )  = red
TRANSITION Give_Green_Arrow ( d: direction )
  ENTRY         [ TIME : change_dur ]
            LT_car ( d )
          & arrow ( d )  = red
          &  ( circle ( adj1 ( d )  )  = yellow
             & arrow ( adj1 ( d )  )  ~= green
             & circle ( adj2 ( d )  )  ~= green
             & now - Change ( circle ( adj1 ( d )  )  )  >= min_yellow - change_dur
            |  circle ( adj2 ( d )  )  = yellow
             & arrow ( adj2 ( d )  )  ~= green
             & circle ( adj1 ( d )  )  ~= green
             & now - Change ( circle ( adj2 ( d )  )  )  >= min_yellow - change_dur )
  EXIT
            arrow ( d )  = green
          & arrow ( adj1 ( d )  )  = red
          & arrow ( adj2 ( d )  )  = red
          & IF
                  LT_car’ ( opp ( d )  )
            THEN
                  arrow ( opp ( d )  )  = green
            ELSE
                  Nochange  ( arrow ( opp ( d )  )  )
            FI
          & circle ( adj1 ( d )  )  = red
          & circle ( adj2 ( d )  )  = red
          & Nochange  ( circle ( opp ( d )  )  )
          & IF
                  car’ ( d )
                & ~LT_car’ ( opp ( d )  )
            THEN
                  circle ( d )  = green
            ELSE
                  Nochange  ( circle ( d )  )
            FI
  EXCEPT        [ TIME : change_dur ]
            LT_car ( d )
          & arrow ( d )  = red
          &  ( arrow ( adj1 ( d )  )  = yellow
             & arrow ( adj2 ( d )  )  ~= green
             & circle ( adj1 ( d )  )  = red
             & now - Change ( arrow ( adj1 ( d )  )  )  >= min_yellow - change_dur
            |  arrow ( adj2 ( d )  )  = yellow
             & arrow ( adj1 ( d )  )  ~= green
             & circle ( adj2 ( d )  )  = red
             & now - Change ( arrow ( adj2 ( d )  )  )  >= min_yellow - change_dur )
          & ~car ( adj1 ( d )  )
          & ~car ( adj2 ( d )  )
  EXIT
            arrow ( d )  = green
          & arrow ( adj1 ( d )  )  = red
          & arrow ( adj2 ( d )  )  = red
          & IF
                  LT_car’ ( opp ( d )  )
            THEN
                  arrow ( opp ( d )  )  = green
            ELSE
                  Nochange  ( arrow ( opp ( d )  )  )
            FI
          & Nochange  ( circle ( adj1 ( d )  )  )
          & Nochange  ( circle ( adj2 ( d )  )  )
          & Nochange  ( circle ( opp ( d )  )  )
          & IF
                  car’ ( d )
                & ~LT_car’ ( opp ( d )  )
            THEN
                  circle ( d )  = green
            ELSE
                  Nochange  ( circle ( d )  )
            FI
  EXCEPT        [ TIME : change_dur ]
            LT_car ( d )
          & arrow ( d )  = red
          & arrow ( opp ( d )  )  ~= yellow
          &  ( circle ( d )  = yellow
             & now - Change ( circle ( d )  )  >= min_yellow - change_dur
            |  circle ( opp ( d )  )  = yellow
             & now - Change ( circle ( opp ( d )  )  )  >= min_yellow - change_dur )
          & ~car ( adj1 ( d )  )
          & ~car ( adj2 ( d )  )
          & ~LT_car ( adj1 ( d )  )
          & ~LT_car ( adj2 ( d )  )
```

```
  EXIT
          arrow ( d )  = green
        & Nochange  ( circle ( adj1 ( d )  )  )
        & Nochange  ( circle ( adj2 ( d )  )  )
        & IF
                  LT_car' ( opp ( d )  )
          THEN
                  arrow ( opp ( d )  )  = green
          ELSE
                  Nochange  ( arrow ( opp ( d )  )  )
          FI
        & IF
                  circle' ( d )  = yellow
          THEN
                  circle ( d )  = red
          ELSE
                  Nochange  ( circle ( d )  )
          FI
        & circle ( opp ( d )  )  = red
        & Nochange  ( arrow ( adj1 ( d )  )  )
        & Nochange  ( arrow ( adj2 ( d )  )  )
  EXCEPT          [ TIME : change_dur ]
          LT_car ( d )
        & arrow ( d )  = red
        & arrow ( opp ( d )  )  = green
        & circle ( opp ( d )  )  = red
        & FORALL d2: direction
                   ( ~car ( d2 )  )
        & ~LT_car ( adj1 ( d )  )
        & ~LT_car ( adj2 ( d )  )
  EXIT
          arrow ( d )  BECOMES green
  EXCEPT          [ TIME : change_dur ]
          LT_car ( d )
        & arrow ( d )  = red
        & circle ( d )  = green
        & circle ( opp ( d )  )  = red
        & ~car ( opp ( d )  )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
        & ~LT_car ( opp ( d )  )
        & ~LT_car ( adj1 ( d )  )
        & ~LT_car ( adj2 ( d )  )
  EXIT
          arrow ( d )  BECOMES green
  EXCEPT          [ TIME : change_dur ]
          d = main_dir
        &  ( circle ( d )  = red
          |  arrow ( d )  = red )
        & FORALL d2: direction
                   (  ( circle ( d2 )  ~= yellow
                   ->   ~car ( d2 )  )
                   &  ( arrow ( d2 )  ~= yellow
                   ->   ~LT_car ( d2 )  )  )
        & FORALL d2: direction
                   (  ( circle ( d2 )  = yellow
                   ->   now - Change ( circle ( d )  )  >= min_yellow - change_dur )
                   &  ( arrow ( d2 )  = yellow
                   ->   now - Change ( arrow ( d )  )  >= min_yellow - change_dur )  )
  EXIT
          IF
                   ( circle' ( main_dir )  = yellow )
          THEN
                  circle ( main_dir )  = red
          ELSE
                  circle ( main_dir )  = green
          FI
        & FORALL d2: direction
                   ( d2 ~= main_dir
                ->   circle ( d2 )  = red )
        & IF
                   ( arrow' ( main_dir )  = yellow )
          THEN
                  arrow ( main_dir )  = red
          ELSE
                  arrow ( main_dir )  = green
          FI
        & FORALL d2: direction
                   ( d2 ~= main_dir
                ->   arrow ( d2 )  = red )
TRANSITION Give_Yellow_Circle ( d: direction )
  ENTRY           [ TIME : change_dur ]
          circle ( d )  = green
        & arrow ( d )  = red
        &  ( car ( adj1 ( d )  )
          |  car ( adj2 ( d )  )
          |  LT_car ( adj1 ( d )  )
          |  LT_car ( adj2 ( d )  )  )
        & now - Change ( circle ( d )  )  >= min_green - change_dur
        &  ( circle ( opp ( d )  )  = green
        ->   now - Change ( circle ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
          circle ( d )  = yellow
        & Nochange  ( circle ( adj1 ( d )  )  )
        & Nochange  ( circle ( adj2 ( d )  )  )
```

```
        & IF
                circle' ( opp ( d )  )  = green
          THEN
                circle ( opp ( d )  )  = yellow
          ELSE
                Nochange  ( circle ( opp ( d )  )  )
          FI
  EXCEPT         [ TIME : change_dur ]
          circle ( d )  = green
        & arrow ( d )  = red
        & LT_car ( opp ( d )  )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
        & ~LT_car ( adj1 ( d )  )
        & ~LT_car ( adj2 ( d )  )
        & now - Change ( circle ( d )  )  >= min_green - change_dur
        &  ( circle ( opp ( d )  )  = green
           & LT_car ( d )
        ->   now - Change ( circle ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
          circle ( d )  = yellow
        & Nochange  ( circle ( adj1 ( d )  )  )
        & Nochange  ( circle ( adj2 ( d )  )  )
        & IF
                circle' ( opp ( d )  )  = green
              & LT_car' ( d )
          THEN
                circle ( opp ( d )  )  = yellow
          ELSE
                Nochange  ( circle ( opp ( d )  )  )
          FI
  EXCEPT         [ TIME : change_dur ]
          d ~= main_dir
        & circle ( d )  = green
        & arrow ( d )  = red
        & FORALL d2: direction
                 ( ~car ( d2 )
                 & ~LT_car ( d2 )  )
        & now - Change ( circle ( d )  )  >= min_green - change_dur
        &  ( opp ( d )  ~= main_dir
           & circle ( opp ( d )  )  = green
        ->   now - Change ( circle ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
          circle ( d )  = yellow
        & Nochange  ( circle ( adj1 ( d )  )  )
        & Nochange  ( circle ( adj2 ( d )  )  )
        & IF
                opp ( d )  ~= main_dir
              & circle' ( opp ( d )  )  = green
          THEN
                circle ( opp ( d )  )  = yellow
          ELSE
                Nochange  ( circle ( opp ( d )  )  )
          FI
  EXCEPT         [ TIME : change_dur ]
          TRUE
  EXIT
          TRUE
TRANSITION Give_Yellow_Arrow ( d: direction )
  ENTRY          [ TIME : change_dur ]
          arrow ( d )  = green
        & circle ( d )  = red
        &  ( car ( adj1 ( d )  )
          |  car ( adj2 ( d )  )
          |  LT_car ( adj1 ( d )  )
          |  LT_car ( adj2 ( d )  )  )
        & now - Change ( arrow ( d )  )  >= min_green - change_dur
        &  ( arrow ( opp ( d )  )  = green
        ->   now - Change ( arrow ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
          arrow ( d )  = yellow
        & Nochange  ( arrow ( adj1 ( d )  )  )
        & Nochange  ( arrow ( adj2 ( d )  )  )
        & IF
                arrow' ( opp ( d )  )  = green
          THEN
                arrow ( opp ( d )  )  = yellow
          ELSE
                Nochange  ( arrow ( opp ( d )  )  )
          FI
  EXCEPT         [ TIME : change_dur ]
          arrow ( d )  = green
        & circle ( d )  = red
        & car ( opp ( d )  )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
        & ~LT_car ( adj1 ( d )  )
        & ~LT_car ( adj2 ( d )  )
        & now - Change ( arrow ( d )  )  >= min_green - change_dur
        &  ( arrow ( opp ( d )  )  = green
           & car ( d )
        ->   now - Change ( arrow ( opp ( d )  )  )  >= min_green - change_dur )
```

```
      EXIT
                arrow ( d )  = yellow
              & Nochange  ( arrow ( adj1 ( d )  )  )
              & Nochange  ( arrow ( adj2 ( d )  )  )
              & IF
                        arrow’ ( opp ( d )  )  = green
                      & car’ ( d )
                THEN
                        arrow ( opp ( d )  )  = yellow
                ELSE
                        Nochange  ( arrow ( opp ( d )  )  )
                FI
      EXCEPT          [ TIME : change_dur ]
                arrow ( d )  = green
              & circle ( d )  = green
              & ~car ( opp ( d )  )
              &  ( car ( adj1 ( d )  )
                |  car ( adj2 ( d )  )
                |  LT_car ( adj1 ( d )  )
                |  LT_car ( adj2 ( d )  )  )
              & now - Change ( arrow ( d )  )  >= min_green - change_dur
              & now - Change ( circle ( d )  )  >= min_green - change_dur
      EXIT
                arrow ( d )  BECOMES yellow
              & circle ( d )  BECOMES yellow
      EXCEPT          [ TIME : change_dur ]
                d ~= main_dir
              & arrow ( d )  = green
              & circle ( d )  = red
              & FORALL d2: direction
                         ( ~car ( d2 )
                         & ~LT_car ( d2 )  )
              & now - Change ( arrow ( d )  )  >= min_green - change_dur
              &  ( opp ( d )  ~= main_dir
                 & arrow ( opp ( d )  )  = green
              ->   now - Change ( arrow ( opp ( d )  )  )  >= min_green - change_dur )
      EXIT
                arrow ( d )  = yellow
              & Nochange  ( arrow ( adj1 ( d )  )  )
              & Nochange  ( arrow ( adj2 ( d )  )  )
              & IF
                        opp ( d )  ~= main_dir
                      & arrow’ ( opp ( d )  )  = green
                THEN
                        arrow ( opp ( d )  )  = yellow
                ELSE
                        Nochange  ( arrow ( opp ( d )  )  )
                FI
      EXCEPT          [ TIME : change_dur ]
                d ~= main_dir
              & arrow ( d )  = green
              & circle ( d )  = green
              & FORALL d2: direction
                         ( ~car ( d2 )
                         & ~LT_car ( d2 )  )
      EXIT
                arrow ( d )  BECOMES yellow
              & circle ( d )  BECOMES yellow
    END Top_Level
  END Controller
  PROCESS SPECIFICATION Sensor
    LEVEL Top_Level
      IMPORT
              pos_real
      EXPORT
              Arrive, Depart, is_object
      CONSTANT
              sense_dur: pos_real
      VARIABLE
              is_object: boolean
      INITIAL
              ~is_object
      TRANSITION Arrive
        ENTRY           [ TIME : sense_dur ]
                ~is_object
        EXIT
                is_object
      TRANSITION Depart
        ENTRY           [ TIME : sense_dur ]
                is_object
        EXIT
                ~is_object
    END Top_Level
  END Sensor
END Stoplight
```

References

[CGK 97] Coen-Porisini, A., C. Ghezzi, and R.A. Kemmerer. "Specification of realtime systems using ASTRAL". *IEEE Transactions on Software Engineering*, Sept. 1997, vol. 23, (no. 9): 572-598.

[FF 84] Filman, R.E. and D.P. Friedman. Coordinated computing: tools and techniques for distributed software. New York, NY: McGraw-Hill, 1984.

[HL 94] Heitmeyer, C. and N. Lynch. "The generalized railroad crossing: a case study in formal verification of real-time systems". *Proceedings of the Real-Time Systems Symposium*. San Juan, Puerto Rico, Dec. 1994, pp. 120-131.

[Lam 74] Lamport, L. "A new solution of Dijkstra's concurrent programming problem". *Communications of the ACM*, Aug. 1974, vol. 17, (no. 8): 453-455.

[LL 95] Lewerentz, C. and T. Lindner (eds.). Formal development of reactive systems: case study production cell. New York, NY: Springer-Verlag, 1995.

[Oly 96] Official 1996 Olympic web site, <http://www.atlanta.olympic.org>.

[WM 85] Ward, P.T. and S.J. Mellor. Structured development for real-time systems. New York, NY: Yourdon Press, 1985.