```
    p !heat !no ;   Control_System [p]
    []
    i; p !heat !yes ;
    ( ( p !gas !on ; exit   |||   p !ign !on ; exit  )
      >>
      ( Igniting [p]
        [>
          ( p !flame !no ;
            ( ( p !gas !off ; exit  |||  p !ign !off ; exit  )
              >>
                Control_System [p]
            )
            []
            p !flame !yes ;   p !ign !off ;
            ( Burning [p]
              [>
                ( p !flame !no ;   p !gas !off ;  Control_System [p]
                  []
                  p !heat !no ;   p !gas !off ;  Control_System [p]
    ) ) ) )  )
  endproc (* Control_System *)

(* loops on igniting *)
  process Igniting [p] :noexit :=
    p !igniting ;   Igniting [p]
  endproc (* Igniting *)

(* loops on burning *)
  process Burning [p] :noexit:=
    p !burning ;   Burning [p]
  endproc (* Burning *)
endspec (* Gas_Burner *)
```

The step-by-step and symbolic  executions of this specification are similar to the ones presented for the previous specification. This is, because the first specification is an expansion of the second one, and therefore both specifications behave in the same way.

```
    yes <> yes = false; yes <> no = true;
endtype (* is_needed *)
behaviour

(* compose in parallel all the devices of the Gas Burner. *)
(* Initialize Gas to on, Ignition to on, and no heat is being sent. *)
  ( Gas [pipe] (on)  |||  Ignition [pipe] (on)  |||  Flame [pipe]  |||  Thermostat [pipe] (no)  )
  ||
(* devices synchronize with the control system *)
  Control_System [pipe]

  where
(* the gas turned on/off. *)
  process Gas [p] (K:switch) :noexit :=
    p !gas !K ;   Gas [p] (not(K))
  endproc (* Gas *)

(* ignition is on, but while igniting it may be disabled by an off. *)
  process Ignition [p] (K:switch) :noexit :=
    p !ign !K ;
    ( Igniting [p]
      [>
        Ignition [p] (not(K))  )
  endproc (* Ignition *)

(* if no flame, process Flame loops on itself. If there is flame (W <> no), it is burning and may
    be disabled at any time by a lack of flame. *)
  process Flame [p] :noexit :=
    p !flame ?W:is_needed ;
    ( [ W == no ] -> Flame [p]
      []
      [ W <> no ] ->  ( Burning [p]
                         [>
                           Flame [p]  )  )
  endproc (* Flame *)

(* if heat is needed (no), process Thermostat is instantiated with no heat. If it is needed,
    the process is instantiated by heat (yes). *)
  process Thermostat [p] (W:is_needed) :noexit :=
    p !heat !W ;   Thermostat [p] (neg(W))
    []
    p !heat !neg(W) ;   Thermostat [p] (W)
  endproc (* Thermostat *)

(* this process controls the ordering of events from different devices of the system. *)
  process Control_System [p] :noexit :=
```

## *V . The Constaint-Oriented LOTOS Specification*

In this second LOTOS specification, the devices of the Gas Burner are composed in parallel using the "interleave |||" operator. The resulting composition is composed in parallel with the Control System. Both behaviours synchronize with each other on all events "operator ||".

This is an example of Constraint-Oriented LOTOS style, where all the devices of the Gas Burner are visible as separate processes. We do not see any advantage to this style for this particular example, however we show it for the sake of the interested reader.

```
specification Gas_Burner [pipe] :noexit
(* use of the standard Boolean type defined in the LOTOS library. *)
library Boolean endlib

type  material is
  sorts state
  opns
    heat,ign,gas,flame,igniting,burning : -> state
endtype (* material *)

type switchs is
  sorts switch
  opns
    on, off : -> switch
    not : switch -> switch
  eqns
   ofsort switch
     not(on) = off ;
     not(off) = on
endtype (* switchs *)

type is_needed is
  sorts is_needed
  opns
    no, yes : -> is_needed
    neg : is_needed -> is_needed
    _==_,_<>_ :is_needed,is_needed -> Bool
  eqns
   ofsort is_needed
     neg(no)   = yes ;
     neg(yes) = no
   ofsort Bool
     no ==  no = true;  no ==  yes = false;
     no <> no = false; no <> yes = true;
     yes ==  yes = true;  yes ==  no = false;
```

The symbolic execution tree generated for this specification uses a depth of 10. This depth is chosen to include all the events involved in burning the gas and going back to the idle state. The number of a behaviour is given on the left as ***bh#***. The behaviour whose number is bh0 is the root of the tree. Alternative behaviours are shown on the same column. The fact that a behaviour has reached its last action (stop), it is represented by DEADLOCK. If a behaviour contains in its sequence another predescribed behaviour, the symbolic tree of the former is represented as the initial sequence of its actions followed by an ***again*** to the included behaviour.
A sample sequence of events is shown in the tree using the bold style.

bh0 * 1 i (specified explicitly) [27]
bh1 * | 1 ***pipe !heat:state !yes:is_needed*** [27]
bh2 * | | 1 ***pipe !gas:state !on:switch*** [28]
bh3 * | | | 1 ***pipe !ign:state !on:switch*** [30]
bh4 * | | | | 1 pipe !igniting:state [57]
bh5 * | | | | | 1 pipe !igniting:state [57] ==> again bh5
 * | | | | | 2 pipe !flame:state !no:is_needed [35]
bh6 * | | | | | | 1 pipe !gas:state !off:switch [36]
bh7 * | | | | | | | 1 pipe !ign:state !off:switch [38]
bh8 * | | | | | | | | 1 i (specified explicitly) [27] ==> again bh1
 * | | | | | | | | 2 pipe !heat:state !no:is_needed [24] DEADLOCK
 * | | | | | | 2 pipe !ign:state !off:switch [38]
bh10* | | | | | | | 1 pipe !gas:state !off:switch [36] ==> again bh8
 * | | | | | 3 pipe !flame:state !yes:is_needed [44]
bh11* | | | | | | 1 ***pipe !ign:state !off:switch*** [45]
bh12* | | | | | | | 1 pipe !burning:state [62] ==> again bh12
 * | | | | | | | 2 ***pipe !flame:state !no:is_needed*** [48]
bh13* | | | | | | | | 1 ***pipe !gas:state !off:switch*** [49] ==> again bh0
 * | | | | | | | 3 pipe !heat:state !no:is_needed [51] ==> again bh13
 * | | | | 2 pipe !flame:state !no:is_needed [35] ==> again bh6
 * | | | | 3 ***pipe !flame:state !yes:is_needed*** [44] ==> again bh11
 * | | 2 pipe !ign:state !on:switch [30]
bh14* | | | 1 pipe !gas:state !on:switch [28] ==> again bh4
 *2 pipe !heat:state !no:is_needed [24] DEADLOCK

An informal description of this sample sequence would be: *heat* required (***action 1 of bh1***), turn *gas on* (***action 1 of bh2***), turn *ignition on* (***action 1 of bh3***), there is *flame* (***action 3 of bh4, then go to bh11***), turn *ignition off* (***action 1 of bh11***), there is *no flame* (***action 2 of bh12***), turn *gas off* (***action 1 of bh13, then go back to idle state (bh0)***).

Whenever the execution tree of a specification is finite, as in this case, it represents the behaviour of the specification in the form of a finite state automaton. Such an automaton represents explicitly all the execution paths of the specification and can be thought to constitute a sort of "verification" of the specification.

Note also that this tree could be easily written in the form of a "monolithic" LOTOS specification, i.e. using only the **[]** operator. This specification would be observationally equivalent to the first one.

<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
==> go 1
***(* go back to the check point set above. Try path 3. *)***
<1>- pipe !igniting:state ---> bh1 [61]
<2>- pipe !flame:state !no:is_needed ---> bh2 [35]
<3>- pipe !flame:state !yes:is_needed ---> bh3 [44]
==> 3
***(* after igniting, there is flame. Therefore turn ignition off. *)***
<1>- pipe !ign:state !off:switch ---> bh1 [45]
==> 1
***(* it is now burning. *)***
<1>- pipe !burning:state ---> bh1 [66]
<2>- pipe !flame:state !no:is_needed ---> bh2 [48]
<3>- pipe !heat:state !no:is_needed ---> bh3 [51]
==> 1
<1>- pipe !burning:state ---> bh1 [66]
<2>- pipe !flame:state !no:is_needed ---> bh2 [48]
<3>- pipe !heat:state !no:is_needed ---> bh3 [51]
***(* set a check point, to be able to try both actions 2 & 3. *)***
==> cpset
Ok!!!
==> 2
***(* with action #2, there is no flame. Must turn the gas off. *)***
<1>- pipe !gas:state !off:switch ---> bh1 [49]
==> 1
***(* back to the initial state (idle) of the system. *)***
<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
***(* back to the second check point in this execution. *)***
==> go 2
<1>- pipe !burning:state ---> bh1 [66]
<2>- pipe !flame:state !no:is_needed ---> bh2 [48]
<3>- pipe !heat:state !no:is_needed ---> bh3 [51]
==> 3
***(* with action #3, there is no need for heat. Must turn the gas off. *)***
<1>- pipe !gas:state !off:switch ---> bh1 [52]
==> 1
***(* back to the initial state (idle) of the system. *)***
<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
***(* exit from ISLA interpreter. *)***
==> exit
EXIT ISLA ? (n/y) => y

### ***b. Generation of the Symbolic Execution Tree***

**The available processes are:**
**[1] Gas_Burner[pipe]()**
**[2] Burner[p]()**
**[3] Igniting[p]()**
**[4] Burning[p]()**

PROC: lp[rocesses] | ? | h[elp] | <process number> | <command>
==> 1
Enter actual process gates for execution
==> Gas_Burner
Applying inferences rules !
***(* initial state (idle). To begin execute action # 2. *)***
<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
==> 1
***(* while heat is not needed, the system is idle. *)***
<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
==> 1
<1>- pipe !heat:state !yes:is_needed ---> bh1 [27]
==> 1
***(* heat is needed, turn gas and ignition on. *)***
<1>- pipe !gas:state !on:switch ---> bh1 [28]
<2>- pipe !ign:state !on:switch ---> bh2 [30]
==> 1
<1>- pipe !ign:state !on:switch ---> bh1 [30]
==> 1
***(* it is igniting. Wait for flame. *)***
<1>- pipe !igniting:state ---> bh1 [61]
<2>- pipe !flame:state !no:is_needed ---> bh2 [35]
<3>- pipe !flame:state !yes:is_needed ---> bh3 [44]
==> 1
<1>- pipe !igniting:state ---> bh1 [61]
<2>- pipe !flame:state !no:is_needed ---> bh2 [35]
<3>- pipe !flame:state !yes:is_needed ---> bh3 [44]
==> cpset
Ok!!!
***(* a check point is set here to execute the two paths (2 & 3). First path 2 is chosen. *)***
==> 2
***(* after igniting, no flame. Therefore turn gas and ignition off. *)***
<1>- pipe !gas:state !off:switch ---> bh1 [36]
<2>- pipe !ign:state !off:switch ---> bh2 [38]
==> 1
<1>- pipe !ign:state !off:switch ---> bh1 [38]
==> 1
***(* the system is now idle. *)***

```
        ( p !flame !no ;
    (* if no flame, turn gas and ignition off. *)
         ( ( p !gas !off ; exit   |||   p !ign !off ; exit )
           >>
         (* go back to idle state. *)
            Burner [p]
         )
         []
         p !flame !yes ;   p !ign !off ;
    (* if there is flame, turn ignition off. The device is burning. *)
         ( Burning [p]
           [>
         (* burning is disabled by lack of flame or heat is not needed. *)
         (* in either case, turn the gas off. *)
            ( p !flame !no ;   p !gas !off ; Burner [p]
              []
              p !heat !no ;   p !gas !off ; Burner [p]
 )  ) ) )  )

   where
  (* loops on Igniting. *)
    process Igniting [p] :noexit :=
      p !igniting ;  Igniting [p]
    endproc (* Igniting  *)

  (* loops on Burning. *)
    process Burning [p] :noexit:=
      p !burning ;  Burning [p]
    endproc (* Burning  *)
 endproc (* Burner *)
endspec (* Gas_Burner *)
```

## ***IV . Executing LOTOS Specifications Using an Interpreter.***

Because of the fact that the specification is written in executable LOTOS style, it is possible to test it with an appropriate tool, such as the University of Ottawa LOTOS interpreter. Two execution modes were used: step-by-step and symbolic tree generation.

### ***a. Step-By-Step Execution***

0

The step-by-step execution of the specification is given below. At each level, the user may choose any action among the offered ones. To choose an action, type in the number shown on its right.

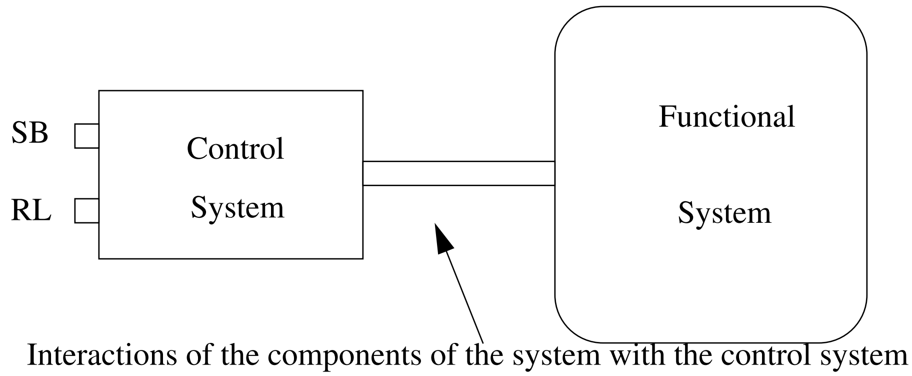
```
                ISLA
     Interactive System for LOTOS Applications
```

The **Abstract Data Types** used in both specifications use three type definitions; type of material, type for switching a device (on/off) , and a type to check whether some material is needed or not.

## *III. An Unstructured LOTOS Specification*

In this LOTOS specification, the sequence of events involved in the process of burning the gas is given in the same behaviour expression. No additional system is needed to control this behaviour.

```
Specification Gas_Burner [pipe] :noexit

type  material is
  sorts state
  opns
    heat,ign,gas,flame,igniting,burning : -> state
endtype (* material *)

type switchs is
  sorts switch
  opns
    on, off : -> switch
endtype (* switchs *)

type is_needed is
  sorts is_needed
  opns
    no, yes : -> is_needed
endtype (* is_needed *)

behaviour
  Burner [pipe]

  where
(* the whole system described as one process. *)
  process Burner [p] :noexit :=
(* if no heat, the system stays idle. *)
    p !heat !no ;   Burner [p]
    []
(* after some internal events ('i'), heat is needed. *)
    i; p !heat !yes ;
(* turn gas and ignition on. *)
    ( ( p !gas !on ; exit   |||   p !ign !on ; exit )
      >>
      ( Igniting [p]
        [>
```




SB : Start Button
RL : Red Lamp

Figure 1. Top Level of the Structure of a Gas Burner

The Functional System (Figure 1.) consists of :

1. a Gas Valve (**GAS**) that may be turned on or off,
2. an Ignition Device (**IGN**) that may also be turned on or off,
3. a Flame Sensor (**FS**) that indicates if the gas is burning, and
4. a Thermostat (**TH**) that indicates whether heating is required or not.

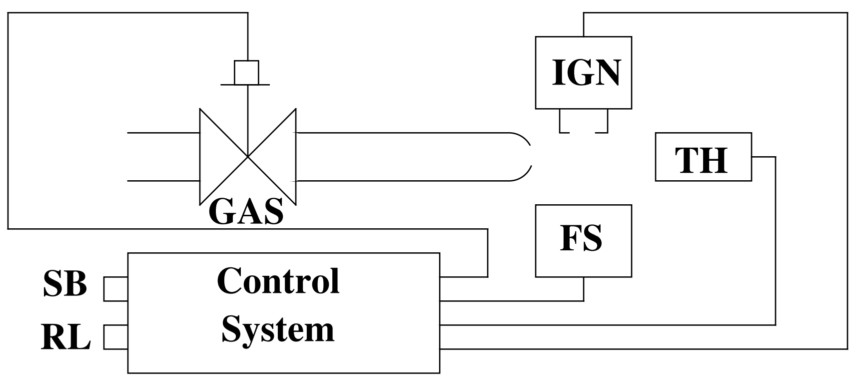



Figure 2. Functional Structure of a Gas Burner

The behaviour of this Gas Burner may be specified in LOTOS in two ways:

1. the behaviour of the system is given as a sequence of a single process,
2. the behaviour of the system reflects its structure. Using a constraint-oriented style of LOTOS specifications, the system may be represented as a parallel composition of the components of the system with the Control System.

*Using LOTOS*
*To*
# *Specify a Gas Burner :*
***An Example for Beginners***

*by*
*Rezki Boumezbeur*
*Computer Science Department*
*University of Ottawa*

## *Abstract*

In this paper, we use LOTOS (Language Of Temporal Ordering Specification) to formally describe the behaviour of a Gas Burner. The LOTOS specification is presented in two different styles. We present also some sample executions using tools provided by an interpreter. The graphical representation is also presented.

## ***I . Introduction***

The use of LOTOS in the formal description of systems is receiving an increasing interest from researchers. In this paper, we demonstrate its use in the formal description of a system taken from real life, where any would be familiar with and can easily understand a LOTOS specification. For beginners to LOTOS, this is a good example to start with, and threfore can easily undrstand the behaviour defined in LOTOS.
We give two types of descriptions of the system. First, the system is described using any operator of LOTOS, then we restrict our use to some specific operators to write a constraint-oriented specification. To execute the specification, we have used tools provided by the University of Ottawa LOTOS interpreter.

## ***II . Structure of the Gas Burner:***

The Gas Burner consists of two parts; the **Functional System** and the **Control System**. The general structure is given by Figure

```
    p !heat !no ;   Control_System [p]
    []
    i; p !heat !yes ;
    ( ( p !gas !on ; exit   |||   p !ign !on ; exit  )
      >>
      ( Igniting [p]
        [>
          ( p !flame !no ;
            ( ( p !gas !off ; exit  |||  p !ign !off ; exit  )
              >>
                Control_System [p]
            )
            []
            p !flame !yes ;   p !ign !off ;
            ( Burning [p]
              [>
                ( p !flame !no ;   p !gas !off ;  Control_System [p]
                  []
                  p !heat !no ;   p !gas !off ;  Control_System [p]
    ) ) ) )  )
  endproc (* Control_System *)

(* loops on igniting *)
  process Igniting [p] :noexit :=
    p !igniting ;   Igniting [p]
  endproc (* Igniting *)

(* loops on burning *)
  process Burning [p] :noexit:=
    p !burning ;   Burning [p]
  endproc (* Burning *)
endspec (* Gas_Burner *)
```

The step-by-step and symbolic  executions of this specification are similar to the ones presented for the previous specification. This is, because the first specification is an expansion of the second one, and therefore both specifications behave in the same way.

```
    yes <> yes = false; yes <> no = true;
endtype (* is_needed *)
behaviour

(* compose in parallel all the devices of the Gas Burner. *)
(* Initialize Gas to on, Ignition to on, and no heat is being sent. *)
  ( Gas [pipe] (on)  |||  Ignition [pipe] (on)  |||  Flame [pipe]  |||  Thermostat [pipe] (no)  )
  ||
(* devices synchronize with the control system *)
  Control_System [pipe]

  where
(* the gas turned on/off. *)
  process Gas [p] (K:switch) :noexit :=
    p !gas !K ;   Gas [p] (not(K))
  endproc (* Gas *)

(* ignition is on, but while igniting it may be disabled by an off. *)
  process Ignition [p] (K:switch) :noexit :=
    p !ign !K ;
    ( Igniting [p]
      [>
        Ignition [p] (not(K))  )
  endproc (* Ignition *)

(* if no flame, process Flame loops on itself. If there is flame (W <> no), it is burning and may
    be disabled at any time by a lack of flame. *)
  process Flame [p] :noexit :=
    p !flame ?W:is_needed ;
    ( [ W == no ] -> Flame [p]
      []
      [ W <> no ] ->  ( Burning [p]
                        [>
                          Flame [p]  )  )
  endproc (* Flame *)

(* if heat is needed (no), process Thermostat is instantiated with no heat. If it is needed,
    the process is instantiated by heat (yes). *)
  process Thermostat [p] (W:is_needed) :noexit :=
    p !heat !W ;   Thermostat [p] (neg(W))
    []
    p !heat !neg(W) ;   Thermostat [p] (W)
  endproc (* Thermostat *)

(* this process controls the ordering of events from different devices of the system. *)
  process Control_System [p] :noexit :=
```

## *V . The Constaint-Oriented LOTOS Specification*

In this second LOTOS specification, the devices of the Gas Burner are composed in parallel using the "interleave |||" operator. The resulting composition is composed in parallel with the Control System. Both behaviours synchronize with each other on all events "operator ||".

This is an example of Constraint-Oriented LOTOS style, where all the devices of the Gas Burner are visible as separate processes. We do not see any advantage to this style for this particular example, however we show it for the sake of the interested reader.

```
specification Gas_Burner [pipe] :noexit
(* use of the standard Boolean type defined in the LOTOS library. *)
library Boolean endlib

type  material is
  sorts state
  opns
    heat,ign,gas,flame,igniting,burning : -> state
endtype (* material *)

type switchs is
  sorts switch
  opns
    on, off : -> switch
    not : switch -> switch
  eqns
   ofsort switch
     not(on) = off ;
     not(off) = on
endtype (* switchs *)

type is_needed is
  sorts is_needed
  opns
    no, yes : -> is_needed
    neg : is_needed -> is_needed
    _==_,_<>_ :is_needed,is_needed -> Bool
  eqns
   ofsort is_needed
     neg(no)   = yes ;
     neg(yes) = no
   ofsort Bool
     no ==  no = true;  no ==  yes = false;
     no <> no = false; no <> yes = true;
     yes ==  yes = true;  yes ==  no = false;
```

The symbolic execution tree generated for this specification uses a depth of 10. This depth is chosen to include all the events involved in burning the gas and going back to the idle state. The number of a behaviour is given on the left as ***bh#***. The behaviour whose number is bh0 is the root of the tree. Alternative behaviours are shown on the same column. The fact that a behaviour has reached its last action (stop), it is represented by DEADLOCK. If a behaviour contains in its sequence another predescribed behaviour, the symbolic tree of the former is represented as the initial sequence of its actions followed by an ***again*** to the included behaviour.
A sample sequence of events is shown in the tree using the bold style.

bh0 * 1 i (specified explicitly) [27]
bh1 * | 1 ***pipe !heat:state !yes:is_needed*** [27]
bh2 * | | 1 ***pipe !gas:state !on:switch*** [28]
bh3 * | | | 1 ***pipe !ign:state !on:switch*** [30]
bh4 * | | | | 1 pipe !igniting:state [57]
bh5 * | | | | | 1 pipe !igniting:state [57] ==> again bh5
* | | | | | 2 pipe !flame:state !no:is_needed [35]
bh6 * | | | | | | 1 pipe !gas:state !off:switch [36]
bh7 * | | | | | | | 1 pipe !ign:state !off:switch [38]
bh8 * | | | | | | | | 1 i (specified explicitly) [27] ==> again bh1
* | | | | | | | | 2 pipe !heat:state !no:is_needed [24] DEADLOCK
* | | | | | | 2 pipe !ign:state !off:switch [38]
bh10* | | | | | | | 1 pipe !gas:state !off:switch [36] ==> again bh8
* | | | | | 3 pipe !flame:state !yes:is_needed [44]
bh11* | | | | | | 1 ***pipe !ign:state !off:switch*** [45]
bh12* | | | | | | | 1 pipe !burning:state [62] ==> again bh12
* | | | | | | | 2 ***pipe !flame:state !no:is_needed*** [48]
bh13* | | | | | | | | 1 ***pipe !gas:state !off:switch*** [49] ==> again bh0
* | | | | | | | 3 pipe !heat:state !no:is_needed [51] ==> again bh13
* | | | | 2 pipe !flame:state !no:is_needed [35] ==> again bh6
* | | | | 3 ***pipe !flame:state !yes:is_needed*** [44] ==> again bh11
* | | 2 pipe !ign:state !on:switch [30]
bh14* | | | 1 pipe !gas:state !on:switch [28] ==> again bh4
*2 pipe !heat:state !no:is_needed [24] DEADLOCK

An informal description of this sample sequence would be: *heat* required (***action 1 of bh1***), turn *gas on* (***action 1 of bh2***), turn *ignition on* (***action 1 of bh3***), there is *flame* (***action 3 of bh4, then go to bh11***), turn *ignition off* (***action 1 of bh11***), there is *no flame* (***action 2 of bh12***), turn *gas off* (***action 1 of bh13, then go back to idle state (bh0)***).

Whenever the execution tree of a specification is finite, as in this case, it represents the behaviour of the specification in the form of a finite state automaton. Such an automaton represents explicitly all the execution paths of the specification and can be thought to constitute a sort of "verification" of the specification.

Note also that this tree could be easily written in the form of a "monolithic" LOTOS specification, i.e. using only the **[]** operator. This specification would be observationally equivalent to the first one.

<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
==> go 1
(* go back to the check point set above. Try path 3. *)
<1>- pipe !igniting:state ---> bh1 [61]
<2>- pipe !flame:state !no:is_needed ---> bh2 [35]
<3>- pipe !flame:state !yes:is_needed ---> bh3 [44]
==> 3
(* after igniting, there is flame. Therefore turn ignition off. *)
<1>- pipe !ign:state !off:switch ---> bh1 [45]
==> 1
(* it is now burning. *)
<1>- pipe !burning:state ---> bh1 [66]
<2>- pipe !flame:state !no:is_needed ---> bh2 [48]
<3>- pipe !heat:state !no:is_needed ---> bh3 [51]
==> 1
<1>- pipe !burning:state ---> bh1 [66]
<2>- pipe !flame:state !no:is_needed ---> bh2 [48]
<3>- pipe !heat:state !no:is_needed ---> bh3 [51]
(* set a check point, to be able to try both actions 2 & 3. *)
==> cpset
Ok!!!
==> 2
(* with action #2, there is no flame. Must turn the gas off. *)
<1>- pipe !gas:state !off:switch ---> bh1 [49]
==> 1
(* back to the initial state (idle) of the system. *)
<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
(* back to the second check point in this execution. *)
==> go 2
<1>- pipe !burning:state ---> bh1 [66]
<2>- pipe !flame:state !no:is_needed ---> bh2 [48]
<3>- pipe !heat:state !no:is_needed ---> bh3 [51]
==> 3
(* with action #3, there is no need for heat. Must turn the gas off. *)
<1>- pipe !gas:state !off:switch ---> bh1 [52]
==> 1
(* back to the initial state (idle) of the system. *)
<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
(* exit from ISLA interpreter. *)
==> exit
EXIT ISLA ? (n/y) => y

***b. Generation of the Symbolic Execution Tree***

**The available processes are:**
**[1] Gas_Burner[pipe]()**
**[2] Burner[p]()**
**[3] Igniting[p]()**
**[4] Burning[p]()**

PROC: lp[rocesses] | ? | h[elp] | <process number> | <command>
==> 1
Enter actual process gates for execution
==> Gas_Burner
Applying inferences rules !
***(* initial state (idle). To begin execute action # 2. *)***
<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
==> 1
***(* while heat is not needed, the system is idle. *)***
<1>- pipe !heat:state !no:is_needed ---> bh1 [24]
<2>- i (specified explicitly) ---> bh2 [27]
==> 1
<1>- pipe !heat:state !yes:is_needed ---> bh1 [27]
==> 1
***(* heat is needed, turn gas and ignition on. *)***
<1>- pipe !gas:state !on:switch ---> bh1 [28]
<2>- pipe !ign:state !on:switch ---> bh2 [30]
==> 1
<1>- pipe !ign:state !on:switch ---> bh1 [30]
==> 1
***(* it is igniting. Wait for flame. *)***
<1>- pipe !igniting:state ---> bh1 [61]
<2>- pipe !flame:state !no:is_needed ---> bh2 [35]
<3>- pipe !flame:state !yes:is_needed ---> bh3 [44]
==> 1
<1>- pipe !igniting:state ---> bh1 [61]
<2>- pipe !flame:state !no:is_needed ---> bh2 [35]
<3>- pipe !flame:state !yes:is_needed ---> bh3 [44]
==> cpset
Ok!!!
***(* a check point is set here to execute the two paths (2 & 3). First path 2 is chosen. *)***
==> 2
***(* after igniting, no flame. Therefore turn gas and ignition off. *)***
<1>- pipe !gas:state !off:switch ---> bh1 [36]
<2>- pipe !ign:state !off:switch ---> bh2 [38]
==> 1
<1>- pipe !ign:state !off:switch ---> bh1 [38]
==> 1
***(* the system is now idle. *)***

```
      ( p !flame !no ;
   (* if no flame, turn gas and ignition off. *)
        ( ( p !gas !off ; exit  |||  p !ign !off ; exit )
          >>
        (* go back to idle state. *)
            Burner [p]
        )
        []
        p !flame !yes ;   p !ign !off ;
    (* if there is flame, turn ignition off. The device is burning. *)
        ( Burning [p]
          [>
        (* burning is disabled by lack of flame or heat is not needed. *)
        (* in either case, turn the gas off. *)
            ( p !flame !no ;   p !gas !off ; Burner [p]
             []
             p !heat !no ;   p !gas !off ; Burner [p]
)  ) ) )  )

   where
  (* loops on Igniting. *)
    process Igniting [p] :noexit :=
      p !igniting ;  Igniting [p]
    endproc (* Igniting  *)

  (* loops on Burning. *)
    process Burning [p] :noexit:=
      p !burning ;  Burning [p]
    endproc (* Burning  *)
 endproc (* Burner *)
endspec (* Gas_Burner *)
```

## *IV . Executing LOTOS Specifications Using an Interpreter.*

Because of the fact that the specification is written in executable LOTOS style, it is possible to test it  with an appropriate tool, such as the University of Ottawa LOTOS interpreter. Two execution modes were used: step-by-step and symbolic tree generation.

### *a. Step-By-Step Execution*

0

The step-by-step execution of the specification is given below. At each level, the user may choose any action among the offered ones. To choose an action, type in the number shown on its right.

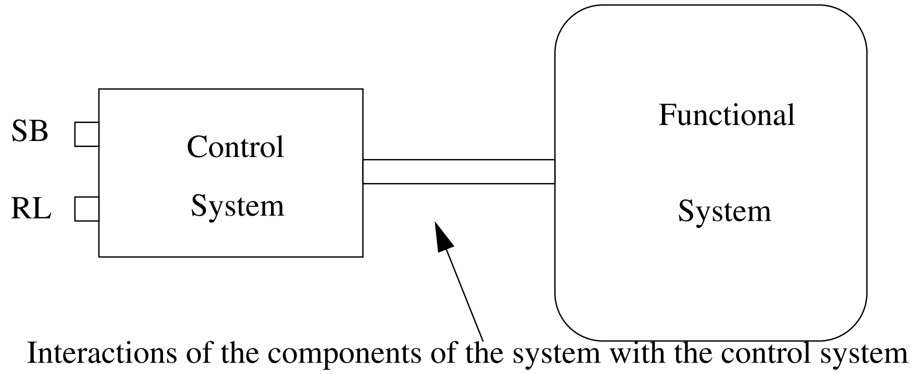
```
              ISLA
   Interactive System for LOTOS Applications
```

The **Abstract Data Types** used in both specifications use three type definitions; type of material, type for switching a device (on/off) , and a type to check whether some material is needed or not.

## ***III. An Unstructured LOTOS Specification***

In this LOTOS specification, the sequence of events involved in the process of burning the gas is given in the same behaviour expression. No additional system is needed to control this behaviour.

```
Specification Gas_Burner [pipe] :noexit

type  material is
  sorts state
  opns
    heat,ign,gas,flame,igniting,burning : -> state
endtype (* material *)

type switchs is
  sorts switch
  opns
    on, off : -> switch
endtype (* switchs *)

type is_needed is
  sorts is_needed
  opns
    no, yes : -> is_needed
endtype (* is_needed *)

behaviour
  Burner [pipe]

  where
(* the whole system described as one process. *)
  process Burner [p] :noexit :=
(* if no heat, the system stays idle. *)
    p !heat !no ;   Burner [p]
    []
(* after some internal events ('i'), heat is needed. *)
    i; p !heat !yes ;
(* turn gas and ignition on. *)
    ( ( p !gas !on ; exit   |||   p !ign !on ; exit )
      >>
      ( Igniting [p]
        [>
```




SB : Start Button
RL : Red Lamp

Figure 1.  Top Level of the Structure of a Gas Burner

The Functional System (Figure 1.) consists of :

1. a Gas Valve (**GAS**) that may be turned on or off,
2. an Ignition Device (**IGN**) that may also be turned on or off,
3. a Flame Sensor (**FS**) that indicates if the gas is burning, and
4. a Thermostat (**TH**) that indicates whether heating is required or not.

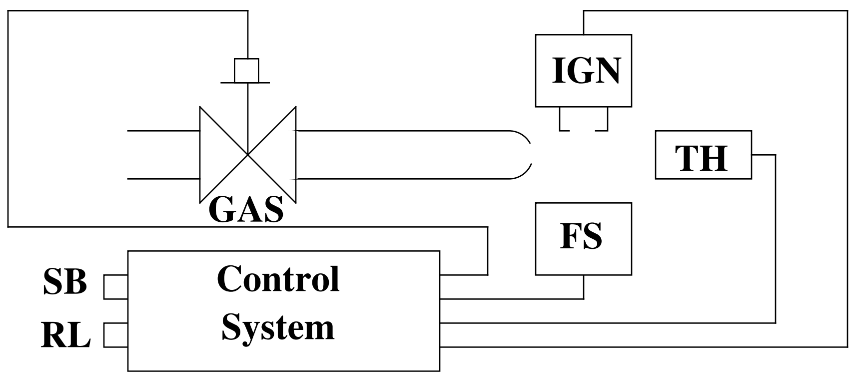



Figure 2. Functional Structure of a Gas Burner

The behaviour of this Gas Burner may be specified in LOTOS in two ways:

1. the behaviour of the system is given as a sequence of a single process,
2. the behaviour of the system reflects its structure. Using a constraint-oriented style of LOTOS specifications, the system may be represented as a parallel composition of the components of the system with the Control System.

# *Using LOTOS To Specify a Gas Burner :*
## *An Example for Beginners*

*by*
*Rezki Boumezbeur*
*Computer Science Department*
*University of Ottawa*

### *Abstract*

In this paper, we use LOTOS (Language Of Temporal Ordering Specification) to formally describe the behaviour of a Gas Burner. The LOTOS specification is presented in two different styles. We present also some sample executions using tools provided by an interpreter. The graphical representation is also presented.

## ***I . Introduction***

The use of LOTOS in the formal description of systems is receiving an increasing interest from researchers. In this paper, we demonstrate its use in the formal description of a system taken from real life, where any would be familiar with and can easily understand a LOTOS specification. For beginners to LOTOS, this is a good example to start with, and threfore can easily undrstand the behaviour defined in LOTOS.

We give two types of descriptions of the system. First, the system is described using any operator of LOTOS, then we restrict our use to some specific operators to write a constraint-oriented specification. To execute the specification, we have used tools provided by the University of Ottawa LOTOS interpreter.

## ***II . Structure of the Gas Burner:***

The Gas Burner consists of two parts; the **Functional System** and the **Control System**. The general structure is given by Figure