DIVISION OF COMPUTER SCIENCE

A Case Study for Generic Processes and Reusability in LOTOS

P. N. Taylor
D. E. Smith

Technical Report No.209

June 1994

A Case Study for Generic Processes and Reusability in LOTOS

P.N. Taylor and D.E. Smith
Division of Computer Science,
University of Hertfordshire,
College Lane, Hatfield, Herts. AL10 9AB. U.K
email: comrpnt@hertfordshire.ac.uk

June 1994

Abstract

The benefits of reusability have been widely documented and commented upon. When processes exhibit similar behaviour the possibilities for generic process specification exist. It is possible to construct process specifications by instantiating generic process descriptions. In this paper we investigate the identification and specification of reusable components using the formal specification technique (FDT) LOTOS. We discuss the semantics of reusability as defined in LOTOS and apply them to a case study problem. We specify a simple control system for use within a greenhouse environment and concentrate upon the identification and specification of generic, reusable processes found within the case study system.

1 Introduction

In the early 1980's the International Organisation for Standardization (ISO) decided to develop a formal description technique to help with the mathematical modelling of computer networks and open distributed systems [1, 2, 5, 6]. Those systems that were related to the Open Systems Interconnection (ISO) seven layered architecture were to be the main beneficiaries of the work on LOTOS. The idea was to be able to give LOTOS the power to model the relationship between the temporal interaction of processes such that the behaviour of those systems could be observed external to the system; that is time ordered communications between processes. LOTOS builds on the work of CCS [7] and CSP [4] where the syntax and semantics of LOTOS are based on both of these previous works. For more information about the foundations of LOTOS see [3] which compares and contrasts all three languages; CCS, CSP and LOTOS.

2 Process Specification

Our case study originates from a simple problem which is often given to students as part of a design and specification course. It is a greenhouse control system (GHCS) containing six components all working in parallel. The informal specification of each component is as follows:

Sprayer (*Sp*): It can be turned 'on' or 'off' by either accepting communications from the environment or the Hygrometer. If the Sprayer is left 'on' for too long then it will timeout and turn itself 'off'.

Hygrometer (*Hy*): The Hygrometer process accepts humidity readings from the environment. It uses these readings to determine whether to tell the Sprayer to turn 'on' or 'off' and the Window Controller whether to 'open' or 'close'. It can also accept user specified minimum and maximum settings to determine the humidity range.

Window Controller (*WC*): The Window Controller accepts communications from either the Hygrometer or the Heater and the environment which tell it whether to 'open' or 'close'. It has a static minimum and maximum range which it cannot move beyond. If an attempt is made to adjust the window beyond this preset range then a signal is sent to activate the Alarm, thus warning the environment of a problem.

Thermometer (*Th*): The Thermometer accepts temperature readings from the environment. Similar in operation to the Hygrometer. The temperature readings are used to determine whether to tell the Window Controller to 'open' or 'close' and the Heater to 'turn up' or 'turn down'. This process also has minimum and maximum temperature settings to use when validating the current temperature.

Heater (*He*): The Heater accepts communications from the Thermometer which tell it to 'turn up' or 'turn down'. It has a preset range which it cannot be set beyond. Attempts to adjust the Heater beyond its preset limits will result in a signal being sent to the Alarm, warning the environment of a problem.

Alarm (*Al*): The Alarm accepts communications from the Window Controller and the Heater processes. Upon receipt of a signal the alarm will sound. It can be reset by the environment or will timeout and turn itself 'off'. However, the Alarm cannot be activated from the environment, only via some internal communication.

We can represent the lines of communication between these processes as follows:

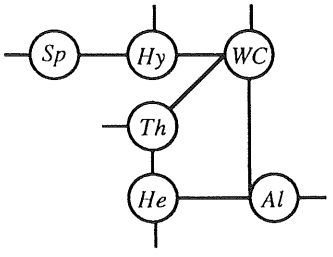

Figure 2.1

The simplified diagram of the complete system, in figure 2.1, omits the exact details of the communications as it only shows how the components connect to each other and the environment. It does not show the nature of those connections.

3 Reusable Processes and Generality

The informal description of the GHCS processes suggests that the *Sprayer* and *Alarm*, *Hygrometer* and *Thermometer* and the *Window Controller* and the *Heater* are behaviourally equivalent. We can specify the following generic processes to capture the common behaviour of these processes pairings.

- Generic process *SPAL captures the behaviour of the Sprayer and Alarm processes*
- Generic process *HYTH captures the behaviour of the Hygrometer and Thermometer processes*
- Generic process *HEWI captures the behaviour of the Heater and Window Controller processes*

By instantiating generic processes we can reduce the complexity of our specification. LOTOS supplies a relabelling operator which enables us to rename the gates that connect the process to the rest of the system and the environment. Renaming allows us to define new processes in terms of existing processes. For example, the generic process *HYTH* is used to define both a *Hygrometer* and a *Thermometer* process.

Consider the semantics of the relabelling operator:

$B \text{—} g' \rightarrow B'$, $\Phi = [g1/g1', \ldots, gn/gn']$,
and $g/g' \in \Phi$
implies
$B\Phi \text{—} g \rightarrow B'\Phi$

$B \text{—} \mu+ \rightarrow B'$ *and* $\mu+ \notin \{g1', \ldots, gn'\}$
implies
$B\Phi \text{—} \mu+ \rightarrow B'\Phi$

Figure 3.1

The semantics of relabelling, in figure 3.1, state that process B can progress to a new state B', via the actions of the relabelled gates, g', where the list of relabelled gates exist in the gate list for B. This in turn *implies* that the process B, with gates listed in the set Φ, can perform an action contained in the gate list Φ. The second part of the rule states that an action $\mu+$ may occur and process B will progress to some new state, B'. The actions in $\mu+$ do not include successful termination and internal actions (i), which are not affected by relabelling.

The renaming of process gates takes place at the meta-language level. For our purposes this level of specification will suffice as we intend to use LOTOS to convey ideas about the system and not specific details which are more closely related to implementation.

4 Capturing Generic, Reusable Behaviour

By concentrating on only the *SPAL, HYTH* and *HEWI* processes we can reduce the amount of work required to prove the behaviour of the component parts of the system. We still have to prove the system correct but if we have confidence in the reusable processes then we only have to prove the behaviour of the composed processes and not worry about the validity of the existing reusable processes (we put our trust in the abilities of the original authors). Our first task is to specify formally each generic process for use with our greenhouse control system. The *HYTH* and *HEWI* processes capture the behaviour of the *Hygrometer/Thermometer* and

Heater/Window Controller processes respectively. However, slight modifications to the *SPAL* process are required for it to be used in place of the *Sprayer* and the *Alarm* processes. Therefore we shall deal with the generic process *SPAL* last.

Prior to the formal specification of the generic processes we need to introduce the abstract data types and operations that are used throughout the specification. These data types and operations would, in practice, be based upon recognised base-types, such as natural numbers. Although we recognise the need for such base-types we do not include them in this discussion because of the generic/reusable emphasise of this particular paper. Also, only a minimum number of equations have been specified as these also rely on base-types. The algebraic operations are defined thus:

```
type StateType is Boolean
        sorts State
        opns
                on,off : State
                isOn : State → Boolean
        eqns
                isOn(on) = true;
                isOn(off) = false;
endtype

type LevelType
        sorts Level
        opns
                theMinLevel :: Level
                theMaxLevel :: Level
                isAtMinLevel : Level → Boolean
                isAtMaxLevel : Level → Boolean
                incLevel : Level → Level
                decLevel : Level → Level
        eqns
                ...
endtype

type ReadingType
        sorts Reading
        opns
                > : Reading Reading → Boolean
                < : Reading Reading → Boolean
                ≥ : Reading Reading → Boolean
                ≤ : Reading Reading → Boolean
                = : Reading Reading → Boolean
        eqns
                ...
endtype

type RecStateType
        sorts RecState
        opns
                setMax : Reading RecState → RecState
                setMin : Reading RecState → RecState
                theMax : RecState → Reading
                theMin : RecState → Reading
        eqns
                ...
endtype
```

Figure 4.1

Using the specified types and operations in figure 4.1 we can specify fully the generic operations that capture the behaviour of the GHCS processes.

Consider the formal specifications for *HYTH* and *HEWI*.

```
hythGates def {SetMin,SetMax,ReadInput,on,off,open,close}

process HYTH[hythGates](hyth:RecState) : noexit :=
        SetMin ? r:Reading;
                ([r ≤ theMax(hyth)] → HYTH[hythGates](setMin(r,hyth))
                        []
                [r > theMax(hyth)] → HYTH[hythGates](hyth))
        []
        SetMax ? r:Reading;
                ([r ≥ theMin(hyth)] → HYTH[hythGates](setMax(r,hyth))
                        []
                [r < theMin(hyth)] → HYTH[hythGates](hyth))
        []
        ReadInput ? r:Reading;
                ([r < theMin(hyth)] → on;close;HYTH[hythGates](hyth)
                        []
                [r > theMax(hyth)] → off;open;HYTH[hythGates](hyth)
                        []
                [(r ≥ theMin(hyth)) and (r ≤ theMax(hyth))] → HYTH[hythGates](hyth))
endproc (* HYTH *)


hewiGates def {SetLevel,open,close,on}

process HEWI[hewiGates](l:Level) : noexit :=
        SetLevel ? l:Level;HEWI[hewiGates](l)
        []
        open;
                ([not isAtMaxLevel(l)] → HEWI[hewiGates](incLevel(l))
                        []
                [isAtMaxLevel(l)] → on;HEWI[hewiGates](l))
        []
        close;
                ([not isAtMinLevel(l)] → HEWI[hewiGates](decLevel(l))
                        []
                [isAtMinLevel(l)] → on;HEWI[hewiGates](l))
endproc (* HEWI *)
```

Figure 4.2

The behaviour for each process matches the informal process descriptions in section 2. To change a generic process into a specific process we simply relabel the gates in the generic process so that the relabelled gates match those of the new target process. For the *Hygrometer/Thermometer* and *Heater/Window Controller* processes we achieve this transformation with the following definitions:

```
Hygrometer def HYTH[SetMinHumid/SetMin,SetMaxHumid/SetMax,ReadHumid/Read,sprayOn/on,
        sprayOff/off,open,close](hyState:RecState)

Thermometer def HYTH[SetMinTemp/SetMin,SetMaxTemp/SetMax,ReadTemp/Read,inc/on,
        dec/off,open,close](thState:RecState)

Heater def HEWI[SetHeat/SetLevel,inc/open,dec/close,on](heState:Level)

Window Controller def HEWI[SetWindow/SetLevel,open,close,on](wcState:Level)
```

Figure 4.3

The general format for the specification of relabelling a generic process to a specific process is *specificName def generalName[newName/oldName]*. From this format description we

can see how both *HYTH* and *HEWI* can be used as the foundations for two specific processes each.

Due the differences between *Sprayer* and *Alarm* we cannot simply perform the same steps as those employed for the creation of the *HYTH* and *HEWI* processes. We can summarise these differences thus:

> The *Alarm* is activated internally, via communications with other processes in the system. *Sprayer* can be activated in a similar manner but can also be activated by the environment (*Alarm* cannot be activated in this way). Also, the *Sprayer* can be deactivated via an internal communication , whereas the *Alarm* can only be deactivated via an external communication from the environment.

Figure 4.4

We can formalise the informal description in figure 4.4 by specifying *Sprayer* and *Alarm.*

```
sprayGates def {SetSprayOn,SetSprayOff,sprayOn,sprayOff}

process Sprayer[sprayGates](spState:State) : noexit :=
        [not isOn(spState)] →    (SetSprayOn;Sprayer[sprayGates](on) []
                                 sprayOn;Sprayer[sprayGates](on))
        []
        [isOn(spState)] →        (i;Sprayer[sprayGates](off) []
                                 SetSprayOff;Sprayer[sprayGates](off) []
                                 sprayOff;Sprayer[sprayGates](off))
endproc (* Sprayer *)

alarmGates def {SetAlarmOff,on}

process Alarm[alarmGates](alState:State) : noexit :=
        [not isOn(alState)] →    on;Alarm[alarmGates](on)
        []
        [isOn(alState)] →        (i;Alarm[alarmGates](off) []
                                 SetAlarmOff;Alarm[alarmGates](off))
endproc (* Alarm *)
```

Figure 4.5

A comparison between the two gate lists for *Sprayer* and *Alarm* reveals a different set of links to the environment. In order to specify a generic process that captures both *Sprayer* and *Alarm* we must use action restriction on the external action *SetSprayOn* so that we can specify *SPAL.* If *SetSprayOn* were an internal actions then the LOTOS ***hide*** operator would suffice and we could define *Alarm* in terms of *Sprayer* by using action hiding only. The remaining similarities between the two processes can be expressed formally with the following expression, taken from the language of CCS.

$$Alarm \approx Sprayer \setminus \{sprayOff, SetSprayOn\}$$

Figure 4.6

where \' is the restriction operator borrowed from CCS. The restriction of the actions from the *Sprayer* process differs from the LOTOS hide operation in that the restricted actions are not available either internally or externally with respect to their environment.

To generate a reusable process to capture both *Sprayer* and *Alarm* we take a maximal view and provide all of the functionality used by both processes. This gives rise to the following generic process specification.

```
spalGates def {SetOn,SetOff,on,off}

process SPAL[spalGates](spalState:State) : noexit :=
        [not isOn(spalState)] →  SetOn;SPAL[spalGates](on) [] on;SPAL[spalGates](on)
        []
```

```
        [isOn(spalState)] →     (i;SPAL[spalGates](off) [] SetOff;SPAL[spalGates](off) []
                                off;SPAL[spalGates](off)
endproc (* SPAL *)

Sprayer def SPAL[SetSprayOn/SetOn,SetSprayOff/SetOff,sprayOn/on,sprayOff/off](spalState:State)

Alarm def SPAL \ {SetOn,off}[SetAlarmOff/SetOff](spalState:State)
```

Figure 4.7

By using action restriction we can define new processes that only require a subset of the actions provided by the reusable process. We could also think of this relationship between general and specific processes as a *parent/child* relationship. The *parent* provides all of the necessary actions for the *child* to use in order to allow the child to capture its own required behaviour.

With all of the generic processes now specified we can consider how they might be composed together to form the greenhouse control system (GHCS). We would like to encapsulate processes together so that details particular to process groups are hidden from the rest of the system and the environment. The next diagram (taken from figure 2.1) shows the partitions that we wish to enforce within the complete GHCS.

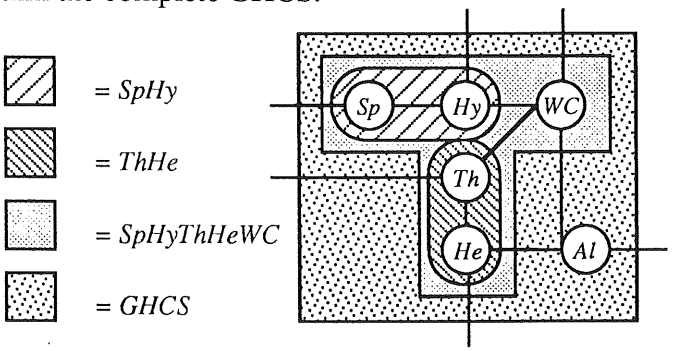

Figure 4.8

From the diagram in figure 4.8 we can see that *SpHy* comprises of the processes built from the generic processes *SPAL* and *HYTH* and *ThHe* uses *HYTH* and *HEWI*. *SpHyThHeWC* then uses *SpHy* and *ThHe*, together with *HEWI*. The complete system (*GHCS*) uses *SpHyThHeWC* and *SPAL* (acting as an *Alarm* process). Each part hides information from those around them using the LOTOS ***hide*** operator. We can specify the components formally like so:

```
SpGates def {SetSprayOn/SetOn,SetSprayOff/SetOff,sprayOn/on,sprayOff/off}
HyGates def {SetMinHumid/SetMin,SetMaxHumid/SetMax,ReadHumid/ReadInput,sprayOn/on,
            sprayOff/off,open,close}
ThGates def {SetMinTemp/SetMin,SetMaxTemp/SetMax,ReadTemp/ReadInput,inc/on,dec/off,
            open,close}
HeGates def {SetHeat/SetLevel,inc/open,dec/close}
WCGates def {SetWindow/SetLevel,open,close}
AlGates {SetAlarmOff/SetOff,on}

SpHyThHeWCGates def {SpGates∪HyGates∪ThGates∪HeGates∪WCGates} \ {sprayOn,sprayOff,
                    inc,dec,open,close}

process GHCS[SpHyThHeWCGates∪AlGates](spalSP,hythHY,hythTH,hewiHE,hewiWC,spalAL)
: noexit :=
    hide on in
        SpHyThHeWC[SpGates∪HyGates∪ThGates∪HeGates∪AlGates \
```

```
        {sprayOn,sprayOff,inc,dec}](spalSP,hythHY,hythTH,hewiHE,hewiWC,spalAL)
        |[on]|
        SPAL \ {SetOn,off}[SetAlarmOff|SetOff](spalAL)
where
process SpHy[SpGates∪HyGates](spalSP:State,hythHY:RecState) : noexit :=
        hide sprayOn,sprayOff in
                SPAL[SpGates](spalSP) |[sprayOn,sprayOff]| HYTH[HyGates](hythHY)
endproc (* SpHy *)

process ThHe[ThGates∪HeGates](hythTH:RecState,hewiHE:Level) : noexit :=
        hide inc,dec in
                HYTH[ThGates](hythTH) |[inc,dec]| HEWI[HeGates](hewiHE)
endproc (* ThHe *)

process SpHyThHeWC[SpGates∪HyGates∪ThGates∪HeGates∪WCGates \
{sprayOn,sprayOff,inc,dec}](spalSP:State,hythHY,hythTH:RecState,hewiHE,hewiWC:Level) :
noexit :=
        hide open,close in
                (SpHy[SpGates∪HyGates \ {sprayOn,sprayOff}](spalSP,hythHY)
                |||
                ThHe[ThGates∪HeGates \ {inc,dec}](hythTH,hewiHE))
                        |[open,close]| HEWI[WCGates](hewiWC)
endproc (*SpHyThHeWC *)

endproc (* GHCS *)
```

Figure 4.9

To reduce the complexity of the process definitions we have made use of the set restriction notation which should not be confused with the CCS action restriction notation introduced in figure 4.6. The gates in each generic process are renamed to produce the individual GHCS processes that make up our complete system, as shown in figure 4.8.

5 Generalising Generality

To make our specification as general as possible we have abstracted away much of the implementation specific details that would normally be associated with a LOTOS specification (i.e: specific variable typing). We have found that LOTOS is ideal for communicating our ideas about concurrent systems to potential implementors because the structure of LOTOS lends itself to many programming languages. For our purposes programming language specifics concerning data structures and data types are not required at the level of abstraction at which we have chosen to specify. We keep these data oriented details away from our abstract specifications and in particular the generic issues that this paper concentrates upon. Our approach is at the meta-level of specification and therefore is above the level of a typical 'typed' LOTOS specification. The abstract nature of our specification allows us to introduce easily the notions of reusability within large systems where common behaviour amongst processes is more likely to be found. As work continues in this area we can envisage a library of reusable components being created and maintained so that specifiers can simply use supplied (proved) sections of LOTOS to build their models; similar to the abstract data type/object reuse approach used by software houses.

6 Conclusions

We have shown that even a simple concurrent system can contain processes that exhibit common behaviour. Our first goal has been to elicit this behaviour in order to capture it in generic reusable components. Our study has shown that it is possible to reuse certain aspects of a LOTOS specification, albeit at the meta-level of specification. With combinations of gate renaming and action restriction we have produced new processes which (to use the term loosely) inherit the behaviour of the *parent* processes upon which they are based. For more information on LOTOS and inheritance (in the object-oriented sense) we suggest [8]. In our study a maximal approach was used to provide enough functionality to *child* processes so that their behaviour could be captured. Restricting the maximal generic processes provided only as much observational behaviour as was necessary to create the *child* process itself.

The composition of the generic processes is the second stage in capturing the behaviour of the target system. Each reusable process must join together to form the whole system using process composition. Some renaming of common gates will be required depending upon the original choice for generic gate names.

With this paper we have shown how the introduction of a different method of reusing processes in LOTOS gives us a new insight into how formal specification languages can be used to mimic the practices commonly reserved for programming languages (i.e: inheritance and reusability). We have seen many developments in the approach to implementing systems over the years but few advances in developing formal specification languages to meet those new approaches. We feel that further work in harmonising the work of specifiers and specification languages together with programmers and programming languages will create a smoother transition between system specification and implementation.

References

[1] Bolognesi, T and Brinksma, E. (1987). Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems.* **14**(1):25—59.

[2] Bustard, D.W, Norris, M.T., Orr, R.A. and Winstanley, A.C. (December 1992). An Exercise in Formalizing the Description of a Concurrent System. *Software Practice and Experience.* **22**(12): 1069—1098.

[3] Fidge, C. (1993). *A Comparative Introduction to CSP, CCS and LOTOS.* Key Centre for Software Technology, University of Queensland, Australia. Technical Note.

[4] Hoare, C.A.R., (1985). *Communicating Sequential Processes.* Prentice-Hall.

[5] International Standardization Organisation, (1987). Information Processing System—Open Systems Interconnection, *LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, DIS 8807, 1987.

[6] Logrippo, L., Faci, M and Haj-Hussein, M. (1992). An Introduction to LOTOS: learning by examples. *Computer Networks and ISDN Systems.* **23**(1):325—342.

[7] Milner, R., (1989), *Communication and Concurrency.* Prentice-Hall.

[8] Rudkin, S. (1992). Inheritance in LOTOS. *IFIP Proceedings 1992.*

Appendix

```
Specification GreenHouse_Control_System(...)

type BasicType is Boolean, Level
        sorts Basic

type StateType is Boolean
        sorts State  {on,off}
        opns
                isOn : State → Boolean
        eqns
                isOn(on) = true;
                isOn(off) = false;
endtype

type LevelType
        sorts Level
        opns
                succ(_) : Level → Level
                _lt_ : Level Level → Boolean
                _le_ : Level Level → Boolean
                _gt_ : Level Level → Boolean
                _ge_ : Level Level → Boolean
                setLevel: Level → Level
                incLevel: Level → Level
                decLevel: Level → Level
                isMinLevel: Level → Boolean
                isMaxLevel: Level → Boolean
        eqns
                forall n:Level (* in the range min...max *)
                ofsort Level
                        incLevel(setLevel(minLevel)) = setLevel(succ(minLevel));
                        incLevel(setLevel(n)) = setLevel(succ(n)) if n lt maxLevel;
                        incLevel(setLevel(n)) = setLevel(maxLevel) if n ge maxLevel;
                        decLevel(setLevel(minLevel)) = setLevel(minLevel);
                        decLevel(setLevel(succ(n))) = setLevel(n) if n le maxLevel-1;
                ofsort Bool
                        isMinLevel(setLevel(minLevel)) = true;
                        isMinLevel(setLevel(succ(n))) = false;
                        isMaxLevel(setLevel(maxLevel)) = true;
                        isMaxLevel(setLevel(n)) = false if n lt maxLevel;
endtype

type RecStateType
        sorts RecState, Reading
        opns
                setMax : Reading RecState → RecState
                setMin : Reading RecState → RecState
                isLTmin : Reading RecState → Boolean
                isGTmax : Reading RecState → Boolean
                isGEmin : Reading RecState → Boolean
                isLEmax : Reading RecState → Boolean
endtype


process GHCS(spalState:State, hythState:RecState, hewiState:Level) : noexit :=
        hide spalState in
                (SpHyHewi2[spalGates∪hythGates∪hewiGates](spalState,hythState,hewiState)) |[on]|
                        SPAL[spalGates](spalState:State)
        where

                process SpHy[spalGates∪hythGates](spalState:State, hythState:RecState) : noexit :=
                        hide on,off in
                                SPAL[spalGates](spalState:State) |[on,off]|
```

```
                    HYTH[hythGates](hythState:RecState)
endproc

*relabel common gates across processes, otherwise a gate name mismatch will occur where
        Inc/Dec in HYTH is not recognised as the same gate as open/close in HEWI.
*HYTH = HYTH[Inc/on, Dec/off](hythState:RecState)
*HEWI = HEWI[Inc/open, Dec/close](p:Level)

process HyHewi[hythGates∪hewiGates](hythState:RecState,hewiState:Level) : noexit :=
        hide Inc,Dec in
                HYTH[hythGates](hythState) |[Inc,Dec]| HEWI[hewiGates](hewiState)
endproc

process SpHyHewi2[spalGates∪hythGates∪hewiGates]
        (spalState:State,hythState:RecState,hewiState:Level) : noexit :=
        hide open,close in
                (SpHy[spalGates∪hythGates] ||| HyHewi[hythGates∪hewiGates])
                        |[open,close]|
                                HEWI[hewiGates](hewiState)
endproc

spalGates def {SetOn,SetOff,on,off}

process SPAL[spalGates](s:State) : noexit :=
        [not isOn(s)] → SetOn;SPAL[spalGates](on) [] on;SPAL[spalGates](on)
        []
        [isOn(s)] → (i;SPAL[spalGates](off) [] SetOff;SPAL[spalGates](off) []
                off;SPAL[spalGates](off)
endproc (* SPAL *)


hythGates def {SetMin,SetMax,ReadInput,on,off,open,close}

process HYTH[hythGates](r:RecState) : noexit :=
        SetMin ? h:Reading;
                ([isLEMax(h,r)] → HYTH[hythGates](setMin(h,r))
                        []
                [isGTMax(h,r)] → HYTH[hythGates](r))
        []
        SetMax ? h:Reading;
                ([isGEMin(h,r)] → HYTH[hythGates](setMax(h,r))
                        []
                [isLTMin(h,r)] → HYTH[hythGates](r))
        []
        ReadInput ? h:Reading;
                ([isLTMin(h,r)] → on;close;HYTH[hythGates](r)
                        []
                [isGTMax(h,r)] → off;open;HYTH[hythGates](r)
                        []
                [isGEMin(h,r) and isLEMax(h,r)] → HYTH[hythGates](r))
endproc (* HYTH *)


hewiGates def {SetLevel,open,close,on}

process HEWI[hewiGates](cw:Level) : noexit :=
        SetLevel ? cw:Level;HEWI[hewiGates](cw)
        []
        open;
                ([not isMaxLevel(cw)] → HEWI[hewiGates](incLevel(cw))
                        []
                [isMaxLevel(cw)] → on;HEWI[hewiGates](cw))
        []
        close;
                ([not isMinLevel(cw)] → HEWI[hewiGates](decLevel(cw))
```

```
                                        []
                                    [isMinLevel(cw)] → on;HEWI[hewiGates](cw))
            endproc (* HEWI *)

endproc (* GHCS *)
```

Hygrometer def HYTH[SetMinHumid/SetMin, SetMaxHumid/SetMax, ReadHumid/Read, SprayOn/on, SprayOff/off, open/open, close/close](r:RecState)

Thermometer def HYTH[SetMinTemp/SetMin, SetMaxTemp/SetMax, ReadTemp/Read, Inc/on, Dec/off, open/open, close/close](r:RecState)

Heater def HEWI[SetHeat/SetLevel, Inc/open, Dec/close, on/on](p:Level)

Window Controller def HEWI[SetWindow/SetLevel, open/open, close/close, on/on](p:Level)

Sprayer def SPAL[SetSprayOn/SetOn,SetSprayOff/SetOff,sprayOn/on,sprayOff/off]

Alarm def SPAL[SetAlarmOff/SetOff,on/on] \ {SetOn,off}