DIVISION OF COMPUTER SCIENCE

The Influence of the Formal Description Technique LOTOS on Concurrent System Design

P. N. Taylor
D. E. Smith

Technical Report No.203

June 1994

The Influence of the Formal Description Technique LOTOS on Concurrent System Design

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

In this paper we investigate the applicability of the formal description language LOTOS for specifying a concurrent system. We are particularly interested in how a specifier might capture a system's requirements, given LOTOS as a choice of target formal description language. We show how a LOTOS specification can be modularised using encapsulated processes. A simple case study is introduced, specified using LOTOS. Questions about different design issues concerning process composition with regard to capturing the required behaviour of the case study system are then raised, together with possible solutions.

Introduction

In this paper we specify, in LOTOS, a simple control system for use with a greenhouse environment. A full version of the system's specification can be seen in appendix B. From the initial specification of the component parts of the system we shall compose these processes to form the complete control system itself.

In the early 1980s 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, 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 Milner's CCS at Edinburgh [8] and Hoare's CSP at Oxford [4]. 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.

1 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*): 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*): 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 (*WC*): 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 these preset ranges then a signal is sent to activate the Alarm, thus warning the environment of a problem.

Thermometer (*Th*): 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 temperature range.

Heater (*He*): Accepts communications from the Thermometer which tell it to 'turn up' or 'turn down'. It has preset, static, heat settings 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*): 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'.

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

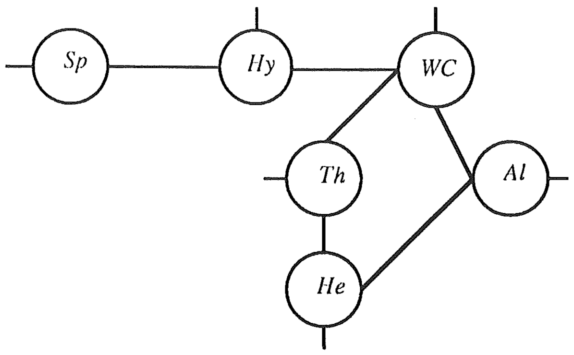

Figure 1.1

This simplified diagram of the complete system omits the exact details of the communications because it only shows how the components connect to each other and not the nature of those connections. Later in this paper we look at the actual communication channels that are represented by these connections. Our model will show the communication gates that exist for each process.

Typically, the LOTOS process descriptions will have the following format:

```
process processName[gateList](parameterList:parameterType)
.
.
.
process description
.
.
.
endproc (* comments *)
```

Section 1 of appendix A presents those elements of the LOTOS syntax that we require to model the GHCS, together with simple examples of how each expression is used. For the greenhouse control system the first process that we formally specify is that of the *Sprayer*.

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

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

The signature of the process *Sprayer* has a gate-list (*[SpGates]*) through which the process communicates with the environment. The parameter *s:State* encapsulates the state of the process. In this case the fact that the *Sprayer* is 'on' will be recorded in the parameter *s:State* when a call to *Sprayer* is made. We discuss this encapsulation of state in more detail later. The Sprayer is defined recursively and as such will not terminate. It is said to have the functionality *noexit*.

Inside the main body of the process definition there are two guarded sequences of expressions, denoted by *[x] → processA*. We can view the functionality of the guards as:

[false] → processA ≡ stop (deadlock)

and

[true] → processA ≡ processA

where the truth value of the guard will determine the behaviour of the process. In the first case a value of *false* will cause the process to *stop*, which denotes deadlock in the system. A value of *true* simply causes the process to behave as specified in the expression following the guard.

The *Sprayer* can only be in one of two states, 'on' or 'off', so we combine the two guarded expressions together using the nondeterministic choice operator (*[]*). If the *Sprayer* is 'on' then it can only be turned 'off'(either by the environment, internal communication or hidden *i*-action) and vice versa.

A distinction should be made between the use of the square brackets to define both the guards and the nondeterministic operator (e.g: *[x] → action1; processA and processA [] processB*). They are not related in any sense. For nondeterminism we can categorise degrees of visibility in view of the system boundary and observable actions. Although this may sound strange consider the following diagram.

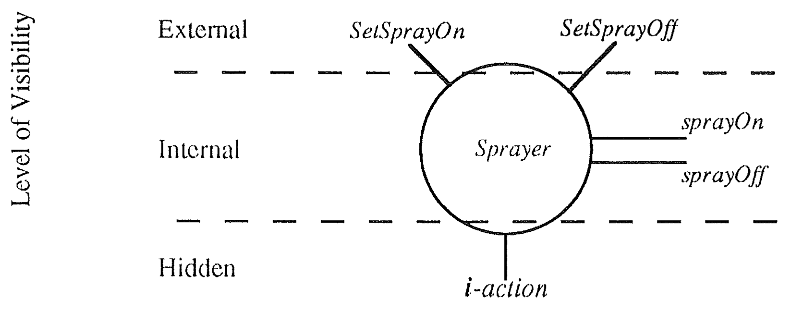

Figure 1.2

From the *external* environment's point of view the choice of actions may appear to be more deterministic because we can observe every action that occurs at this level and influence it. We can think of this observation as cause-and-effect where causing some action effects the behaviour of the process. We can therefore 'determine' what action occurred. Internal actions can be restricted from the environment so that they cannot be observed. Consequently, there is less cause-and-effect at this level of determinism because actions elsewhere may have caused the effect that it observed by the environment. Hidden actions occur deep within the processes of a system and are not influenced at all by external or internal actions. The environment does not have any control over internal actions, either explicitly or implicitly.

When trying to predict the levels of determinism that a process exhibits we tend to think in terms of the whole process itself, not the individual actions that make the process up. For example, consider the following process definitions:

$$VM_1 \overset{def}{=} i;tea.VM_1 + i.coffee.VM_1$$

$$VM_2 \overset{def}{=} tea.VM_2 + coffee.VM_2$$

The first vending machine (VM_1) may, or may not offer tea or coffee whereas the second machine will always supply the requested beverage. The i-action makes VM_1 less deterministic than VM_2 which is usually shown as $VM_1 \leq VM_2$. One way to test for determinism is to compose a test process (which might ask for tea) with the VM processes and look at how many times the VM process passes the test.

At this point it is worth discussing the internal *i*-action in more detail. In LOTOS this action is used to denote some form of action that cannot be observed by the environment. In our example the internal *i*-action specifies that when the *Sprayer* is in the on' state some nondeterministic amount of time elapses before the *Sprayer* evolves into an'off' state. The actual time that elapses, denoted by the *i*-action, cannot be predicted as it is absolutely nondeterministic. In practice this could be at an instance (now) or at any time in the future. We use it to show some method of elapsed time but at the implementation level we would have to define exactly what period of time would be required. The *Alarm* process uses the *i*-action in a similar way for the same reasons at stated here.

The *Hygrometer* process introduces new notation which we discuss after its formal specification.

```
HyGates def {SetMinHumid,SetMaxHumid,ReadHumid,sprayOn,sprayOff,open,close}

process Hygrometer[HyGates](min:Humid,max:Humid) : noexit :=
        SetMinHumid ? h:Humid;
                ([h le max] → Hygrometer[HyGates](h,max)
                        []
                [h gt max] → Hygrometer[HyGates](min,max))
        []
        SetMaxHumid ? h:Humid;
                ([h ge min] → Hygrometer[HyGates](min,h)
                        []
                [h lt min] → Hygrometer[HyGates](min,max))
        []
        ReadHumid ? h:Humid;
                ([h lt min] → sprayOn;close;Hygrometer[HyGates](min,max)
                        []
                [h gt max] → sprayOff;open;Hygrometer[HyGates](min,max)
                        []
                [h ge min and h le max] → Hygrometer[HyGates](min,max))
endproc (* Hygrometer *)
```

The expression *SetMinHumid ? h:Humid;* accepts a value of type *Humid*, stored in the variable *h*, at the gate named *SetMinHumid*. This reads as some input value into the process via a named gate and stored in a variable. The value of *h* is then used within the guards to determine the choice of action. According to the outcome of the guards new values are stored using recursive calls that change the state of the Hygrometer (in terms of its *min* and *max* values). To see this effect look closely at the parameters of the following extracts of *Hygrometer*:

- *[h le max] → Hygrometer[HyGates](h,max)*
- *[h ge min] → Hygrometer[HyGates](min,h)*

The current humidity reading is accepted at the *ReadHumid* gate and the Hygrometer's actions are dictated according to that reading. If the humidity is within an acceptable range (*h ge min **and** h le max*) then *Hygrometer* is simply called recursively with no changes to its state or communications to other processes.

It is worth mentioning why we need to keep encapsulating the states of the processes in the system when we refer to them. The reason for this encapsulation is because LOTOS has its data model founded upon an algebraic specification language, namely ACT-ONE [7]. This implies that we cannot specify the storage of some static state inside a process. We cannot model static data (or state) like the data contained within a record structure in some programming language. Consequently, the representation of a process's state is continually referred to when calling the process and is not actually stored anywhere—the state of the process is totally dynamic. Alternate views concerning the formal modelling of data in communicating systems do exist [2], but we will not discuss them here. Examples of encapsulating the current state of a process can be seen throughout the GHCS and in the next process definition, the *Window Controller.*

```
WCGates def {SetWindow,open,close,on}

process Window[WCGates](cw:Level) : noexit :=
        SetWindow ? cw:Level;Window[WCGates](cw)
        []
        open;
                ([not isMaxLevel(cw)] → Window[WCGates](incLevel(cw))
                        []
                [isMaxLevel(cw)] → on;Window[WCGates](cw))
        []
        close;
                ([not isMinLevel(cw)] → Window[WCGates](decLevel(cw))
                        []
                [isMinLevel(cw)] → on;Window[WCGates](cw))
endproc (* Window *)
```

The current state of the window is stored in the variable *cw* and is used to determine whether the window can be opened or closed, according to some predefined range. The *Window* process itself can receive communications from either the environment, the *Hygrometer* or the *Heater*. The *Window* process may be set to some level, determined by the type *Level,* or it can be incremented or decremented via some internal communication from another process. It is not possible to set the window to some value that is out of range in the *SetWindow* action because of the type restrictions on *Level* (see appendix B). However, if one of the hidden communications tries to set the window past the defined range of *Level* then the *Alarm* process is signalled.

The *Thermometer* process is similar in behaviour to the *Hygrometer* process. Readers might like to think about future specifications where the renaming of gates in a generic process would yield both a *Hygrometer* and *Thermometer* process. Due to the similarities between *Hygrometer* and *Thermometer* we can keep our commentary to a minimum.

```
ThGates def {SetMinTemp,SetMaxTemp,ReadTemp,inc,dec,open,close}

process Thermometer[ThGates](min:Temp,max:Temp) : noexit :=

        SetMinTemp ? t:Temp;
                ([t le max] → Thermometer[ThGates](t,max)
                        []
                [t gt max] → Thermometer[ThGates](min,max))
        []
        SetMaxTemp ? t:Temp;
                ([t ge min] → Thermometer[ThGates](min,t)
                        []
                [t lt min] → Thermometer[ThGates](min,max))
        []
        ReadTemp ? t:Temp;
                ([t lt min] → inc;close;Thermometer[ThGates](min,max)
                        []
                [t gt max] → dec;open;Thermometer[ThGates](min,max)
                        []
                [t ge min and t le max] → Thermometer[ThGates](min,max))

endproc (* Thermometer *)
```

The *Thermometer* communicates with both the *Window Controller* and the *Heater*. If the temperature drops below a certain level then the *Heater* is told to turn 'up' (*inc*) and the *Window* is told to close (*close*). The opposite actions are requested if the temperature gets too high. If the current temperature is within the predefined range then the *Thermometer* simply reverts back to its original state (i.e: no changes to its state).

Moving on to *Heater* we again notice similarities between this process and *Window.*

```
HeGates def {SetHeat,inc,dec,on}

process Heater[HeGates](ch:Level) : noexit :=
        SetHeat ? ch:Level;Heater(ch)
        []
        inc;
                ([not isMaxLevel(ch)] → Heater[HeGates](incLevel(ch))
                        []
                [isMaxLevel(ch)] → on;Heater[HeGates](ch))
        []
```

```
            dec;
                        ([not isMinLevel(ch)] → Heater[HeGates](decLevel(ch))
                                    []
                        [isMinLevel(ch)] → on;Heater[HeGates](ch))
    endproc (* Heater *)
```

The *Heater* behaves in a similar way to the *Window* process. It is linked to the *Thermometer* and the *Alarm*. If the heater is asked to turn up above its maximum setting, or turn down below its minimum setting (*inc; [isMaxLevel(ch)] or dec; [isMinLevel(ch)]*) then the *Alarm* is signalled to alert the environment of a problem. This behaviour certainly underlines its similarities with the *Window* process.

The final process is, the *Alarm*. Although it bears some similarities with *Sprayer* it is not completely identical in behaviour.

```
    AlGates def {on,SetAlarmOff}

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

The *Alarm* process only has two named gates which link it to the environment; communications via the gate named *on* originate from both the *Heater* and *Window* processes. As with similar behaviour in the *Sprayer* process the *Alarm* process is either in an 'off' state or in an'on' state. There is no external environment communication to set the *Alarm* on, only the internal communications themselves can turn the *Alarm* 'on'.

The complete structure for the GHCS can now be represented in the following diagram, showing all of the defined processes and the communication gates between them

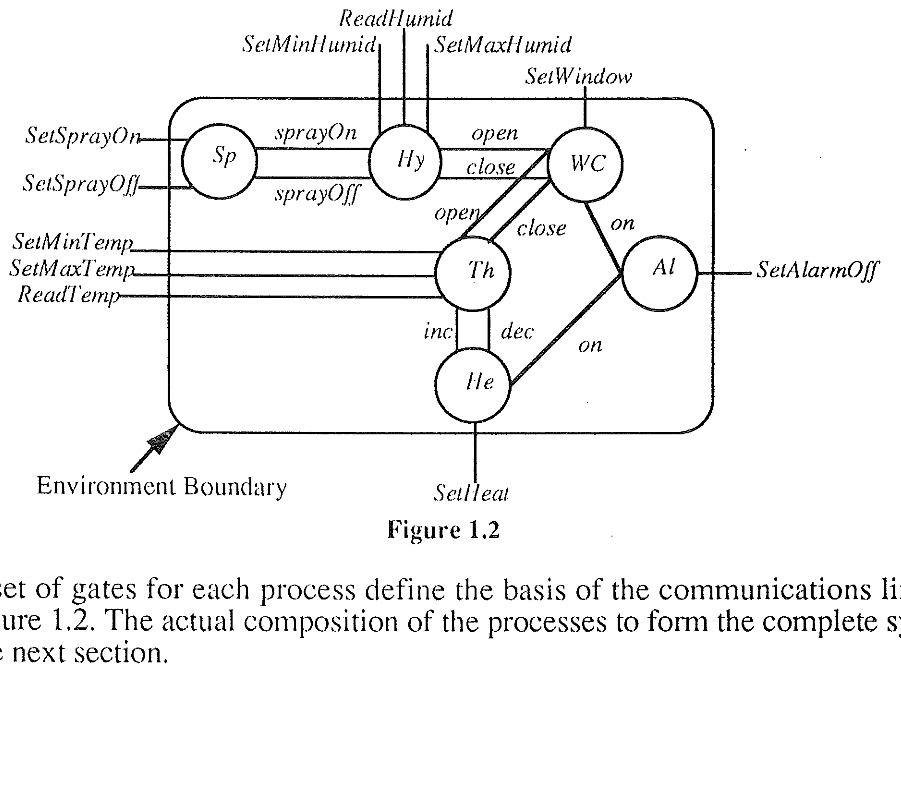

Figure 1.2

The set of gates for each process define the basis of the communications links that are present in figure 1.2. The actual composition of the processes to form the complete system is the subject of the next section.

2 Process Composition

Large individual processes are not ideal when it comes to building large systems. It would not be advisable to build one large monolithic process that performs the functions of many smaller ones. Experience has taught us that the flexibility enjoyed by large systems built from smaller component parts is a design technique that should be encouraged. To build these systems we need to be able to bring processes together so that they can communicate with each other to form concurrent systems. Section 2 of appendix A introduces the various LOTOS composition operators that give us control of process composition.

In LOTOS, the use of three composition operators (*|||*, *|[x]|* and *||*) will effect the behaviour of the overall system depending upon how those operators are used. Both the interleaving and full synchronisation operators can be defined in terms of the selective parallel operator. Consider the following equivalences, noting that the alphabet α of a process *pN* (shown as *αpN*) is the set of actions that process *pN* can engage in:

$\alpha p1 = \{a11, a12\}$ *and* $\alpha p2 = \{a21, a22\}$

$\therefore$

- $p1 \; || \; p2 \equiv p1 \; |[\alpha p1 \cup \alpha p2]| \; p2$
- $p1 \; ||| \; p2 \equiv p1 \; |[\emptyset]| \; p2$ where $\emptyset$ denotes the empty set.

From the equivalences it is shown that full synchronisation is equivalent to selective parallel composition using all of the actions specified for the composed processes and process interleaving is equivalent to selective parallel composition where no synchronisation occurs, leaving the composed processes to proceed independently.

LOTOS achieves multiple communications by using gate names that are common across more than one process. This multi-way communication is a powerful feature of LOTOS and to avoid problems with processes waiting on each other selective parallel composition should be used carefully. Synchronisation between two processes can then be extended to multi-process communication (synchronisation) by composing several processes together using selective parallel composition. Like so:

$((p1 \; |[x]| \; p2) \; |[x]| \; p3) \; |[x]| \; p4$

where the processes *p1* and *p2* synchronise together, then with *p3* and finally with *p4*. The system will not progress until they all synchronise together but we can view the expression as occurring in the order dictated by the brackets. For LOTOS specifications, with their lack of unique gate-naming, the gates *x*, in the selective parallel operator (*|[x]|*), take part in a multi-process communication where each named gate must take part in a synchronised communication. The fact that LOTOS provides this flexibility with its three composition operators underlines its power as a specification language. LOTOS forces the specifier to consider the issues surrounding multi-process communication, synchronisation between processes and the overall system's behaviour.

3 Capturing Concurrent Behaviour

In our GHCS model an expression like *(p1 |[x]| p2) |[x]| p3* would deadlock the *Hygrometer, Window* and *Thermometer*, together with the *Window*, *Heater* and *Alarm* until all of the processes could agree to perform the same synchronising action using the commonly named gate. Although the *p1/p2* composition could progress the whole system would have to wait on *p3*.

Our aim is to keep processes in certain parts of the system from having influence over other non-related processes in other parts of the system. If we maintain a high-level view of the system we notice that there are two distinct parts to the GHCS with boundaries that can be drawn around parts of the whole system. Both the *Sprayer/Hygrometer* and *Thermometer/Heater* parts communicate with the *Window* part of the system. These first two parts do not (and should not) communicate with each other, as they would if we simply used selective composition alone throughout the construction of the system. The composition is quite straight forward but we must consider restricting the composed actions so as to hide unnecessary communications from other parts of the system and the environment.

4 Hiding Communications

In LOTOS a method of hiding internal actions is provided (as presented in section 3 of appendix A). Each system can have its actions restricted from observation by the environment using the ***hide*** operator. For each action that is hidden from the environment an internal ***i***-action occurs. The example below shows the syntax of the ***hide*** operator.

```
process System[a,b,c] : exit :=
        hide a in
                a;b;c;SystemA a,b,c]
        ...
endproc
```

The sequence of actions that *System* performs will resemble the action trace $<i \rightarrow b \rightarrow c>$, where ***i*** is the hidden action. By restricting certain actions we can enforce the behaviour of our system to keep the environment from gaining access and influencing the processes within. A combination of action hiding, selective parallel composition and interleaving will keep the processes separate. Therefore maintaining the encapsulation and modularity that we require to capture the required behaviour of the system. This encapsulation allows us more freedom when specifying complex systems.

5 Building the System

For our system we need to construct capsules (or parts) which will house the composed processes that we need to synchronise together. These parts have to be specified so that they cannot be tampered with by their environment (whether internally or externally). We would like to model absolute modularity, hiding any internal communications that take place within a capsule. The following definitions will therefore provide the encapsulated structure that we are looking for.

Firstly, let us remind ourselves of the contents of the gate lists for each process,

Sprayer	*spGates* $\stackrel{def}{=}$ *{SetSprayOn,SetSprayOff,sprayOn,sprayOff}*
Hygrometer	*hyGates* $\stackrel{def}{=}$ *{SetMinHumid,SetMaxHumid,ReadHumid,sprayOn,sprayOff,open,close}*
Window	*wcGates* $\stackrel{def}{=}$ *{SetWindow,open,close,on}*
Thermometer	*thGates* $\stackrel{def}{=}$ *{SetMinTemp,SetMaxTemp,ReadTemp,inc,dec,open,close}*
Heater	*heGates* $\stackrel{def}{=}$ *{SetHeat,inc,dec,on}*
Alarm	*alGates* $\stackrel{def}{=}$ *{on,SetAlarmOff}*

and then introduce the shorthand notation for use with the specification of the parameter lists for each of the parts,

spState $\underset{=}{def}$ *{sprayer}*
hyState $\underset{=}{def}$ *{minHumid,maxHumid}*
wcState $\underset{=}{def}$ *{window}*
thState $\underset{=}{def}$ *{minTemp,maxTemp}*
heState $\underset{=}{def}$ *{heater}*
alState $\underset{=}{def}$ *{alarm}*

The sections which make up the GHCS can be referred to as *parts* of the GHCS. The *Sprayer* and *Hygrometer* both need to synchronise on common actions. We compose them using selective parallel composition and then encapsulate them with the ***hide*** operator. This encloses the part *SpHy* and stops any influence on the communications that take place within the part via the gates *sprayOn* and *sprayOff*.

- ***process*** *SpHy[spGates∪hyGates](spState:State,hyState:Humid)* : ***noexit*** *:=*
  hide *sprayOn,sprayOff* ***in***
  Sprayer[spGates](spState) |[sprayOn,sprayOff]| Hygrometer[hyGates](hyState)
  endproc

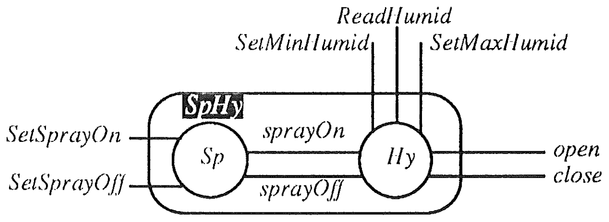

Figure 5.1

Part *ThHe* uses the same techniques to keep itself immune from external influence. The common actions are again used for composure and subsequently used to hide the internal communications from the environment, thus completing part *ThHe*.

- ***process*** *ThHe[thGates∪heGates](thState:Temp,heState:Level)* : ***noexit*** *:=*
  hide *inc,dec* ***in***
  Thermometer[thGates](thSate) |[inc,dec]| Heater[heGates](heState)
  endproc

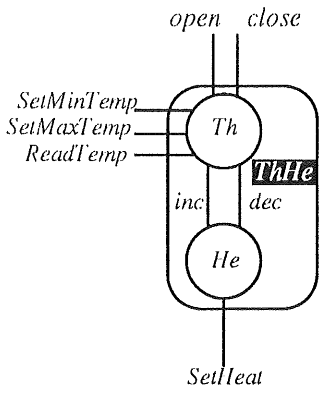

Figure 5.2

Part *STWin* brings the first two parts together so that they do **not** interfere with each other. Parallel composition is used to achieve this requirement. We not want to broadcast any information about the internal workings of *SpHy* or *ThHe*. As an example of information hiding consider the gates *sprayOn, sprayOff, inc* and *dec* which are defined in *SpHy* and *ThHe*. They are hidden from *STWin* and cannot be accessed by it.

We use interleaving at this stage of the system's composition to enforce a strict separation between *SpHy* and *ThHe* (we need process independence). We do not want these two modules talking to each other because they contain common gates and would consequently have to wait on each other, forcing delays and possibly deadlock. However, we can selectively compose them with *Window* because they communicate via common gates and are interleaved in doing so. Either *SpHy* or *ThHe* can talk to *Window*, but not both at the same time and not to each other.

```
• process STWin[spGates,hyGates,thGates,heGates,wcGates]
          (spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level) : noexit :=
          hide open,close in
                    (SpHy[spGates∪hyGates](spState:State,hyState:Humid) |||
                              ThHe[thGates∪heGates](thState:Temp,heState:Level)) |[open,close]|
                                        Window[wcGates](wcState)
  endproc
```

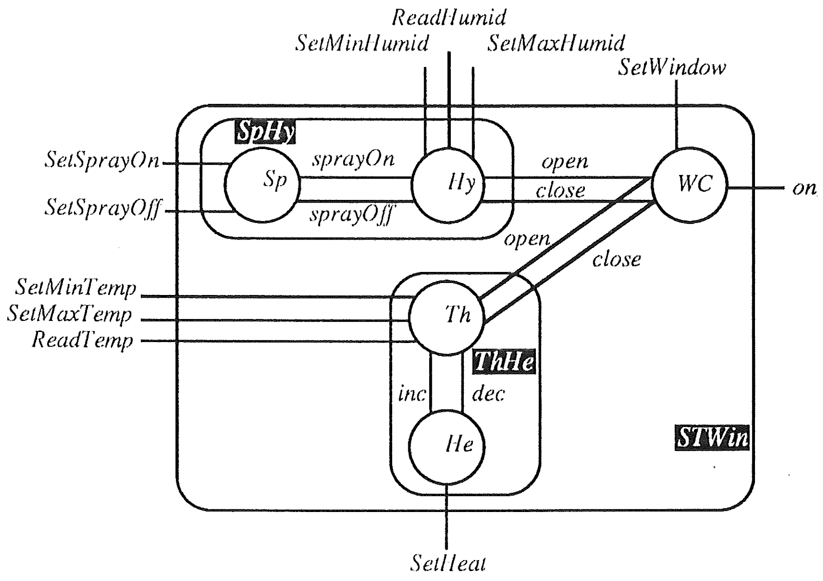

Figure 5.3

We can define the GHCS to be complete by including the *Alarm* process and hiding the communications with it so that the environment cannot influence the activation of the *Alarm*. The complete modularised specification follows, together with a diagrammatic representation of the same system.

```
• process GHCS[spGates,hyGates,thGates,heGates,wcGates,alGates]
                    (spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level,
                              alState:State) : noexit :=
          hide on in
                    STWin[spGates,hyGates,thGates,heGates,wcGates]
                              (spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level)
                              |[on]|
                                        Alarm[alGates](alState)
  endproc
```

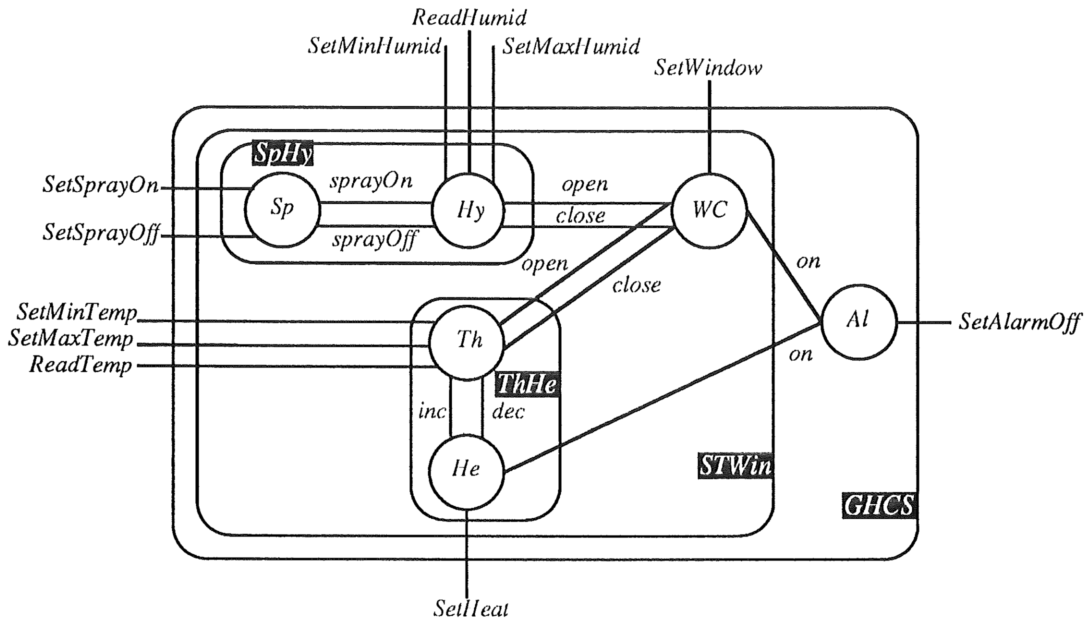

Figure 5.4

We have built the GHCS specification to be flexible in terms of modularisation. By using the language constructs to separate each process we have kept the system partitioned and free from unnecessary complexity. Also, a deeper understanding of the semantics of the composition operators has allowed us to manipulate the expressions to capture the correct behaviour of our system.

7 Conclusions

One of the initial goals of this work has been to determine whether LOTOS could be used as an equal platform (compared with other formal specification languages) from which we could specify the behaviour of systems. From a potential implementor's point of view the LOTOS specification (shown in appendix B) would be easier to comprehend due to its layout and use of familiar programming-style format.

One key part of our investigation has centred around the design approach that we might adopt if we already knew that LOTOS was the specification language. The decision to modularise the system was an obvious choice and LOTOS certainly provides the necessary syntax to allow us to model processes individually. The way that the specific communication gates have been included in the LOTOS process definitions together with the ability to model multiple communications has helped us to keep track of which processes can talk to which. Consistent gate naming is therefore enforced throughout the specification.

One of the main advantages of LOTOS came to light when we tried to link the separate GHCS processes together. LOTOS has a strict view of process composition and consequently the language forced us to adopt a stricter approach to the composition of the system components. The use of the same gate names in LOTOS, across different processes, was the main issue. One advantage with this view of a system is that multiple communications between numerous processes can be achieved because all of the processes talk via the same gates. We viewed this multi-process communication as a direct result of the networking requirements that LOTOS was originally intended to satisfy. From one process multiple communications with many other processes can be achieved. This type of communication satisfies a standard network requirement where a broadcast message is sent to all current users. With this type of multiple communication available to us we had to think carefully about the composition of the GHCS processes.

available to us we had to think carefully about the composition of the GHCS processes.

The LOTOS specification of the GHCS is very close to one possible implementation of the same system. Although we found the formality of LOTOS to be more useful in terms of thinking about the problem we concluded that a high level first draft specification was a better way to begin the task of specification; due to its more relaxed structure. A good method to adopt for the writing of a LOTOS specification can be derived from a typical step-by-step approach.

(1) Write a draft specification using an abstract notation such as CCS.
(2) Convert the abstract notation into LOTOS syntax.
(3) Implement the specification in a programming language such as Ada.

These three stages of development might prove to be more productive than jumping straight into LOTOS itself. We certainly found them useful when trying to think about specifying the behaviour of the GHCS in terms of a new language

We could not claim to have covered all of the issues that even our simple GHCS system has raised; later papers are aimed at filling some of these gaps. However, the main approach to specifying a system in LOTOS has now been covered, together with the design approaches that the LOTOS language implies through its syntax and semantics.

In conclusion, we have shown how LOTOS supplies us with enough tools to enable us to attempt many problems that would normally be solved using the more traditional formal specification languages. We could argue that for certain problems a specification written in LOTOS is a better document for communicating a formal specification about a concurrent system than one written in a less programming language style. Such problems may fit a LOTOS-style approach better than other languages which claim to cover the same ground.

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] de Meer, J., Roth, R. and Vuong,S. (1992). Introduction to algebraic specifications based on the language ACT ONE. *Computer Networks ISDN Systems.* **23**. pp363—392.

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

Appendix A — LOTOS Syntax

1 Language Syntax

The LOTOS language is made up of the following expressions. The examples give some idea as to the way in which these expressions can be used.

Action Prefixing
Perform one action and then progress into a state where the next action can be performed.

Symbol	Example
;	*s;t*

Non-deterministic choice.
A choice between two possible actions, where either choice may be chosen with no preference.

Symbol	Example
[]	*s [] t*

Guarded Actions
Perform some guarded action only if the guard itself holds *true.*

Symbol	Example
[Boolean expr] → y	*[x=0] → pass [] [x ≠ 0] →fail*

Recursion
The power to call the process *x* from within the actual process *x*; causing self-reference.

Symbol	Example
noexit	*process z[p,q,r](s,t) :* ***noexit*** *:=* *c;z[p,q,r](s,t)* . . . *endproc*

Boolean Operators
Used within the guard's expressions to determine truth value.

Symbol	Example
eq,ne,lt,	*[x eq y] →, [x ne y] →, [x lt y] →,*
gt,le,ge	*[x gt y] →, [x le y] →, [x ge y] →*

Extra Boolean operator, exclusive to LOTOS guarded expressions

not	*[not x] →*

Input and Output (I/O)
Data passed into and out of the processes within the specified environments.

Input Symbol	Output Symbol I/O	Examples
Accept x, of type X on gate s	*Output y on gate t*	
s?x:X	*t ! y*	*Read ? x:X* *Write ! y*

Internal Actions
Some unobservable action that is performed by a process and is hidden from the environment. Often used to show some internal action or user interaction in a non-deterministic time frame (timing/timers). Processes with hidden actions can progress given some external communication but no evidence as to the nature of the communication will be visible to the environment.

Symbol	Example
i	*[x ≠ 0] → s;i;t.P(x)*

2 Composition Operators

Process composition is achieved via the following operators. Combinations of these operators will change the behaviour of the system so care must be taken to ensure that the semantics of each operator is understood. Future work on this subject will introduce the semantics more thoroughly.

Full Synchronisation
Processes joined together in full synchronisation must synchronise on every action and cannot proceed independently ('If I can do it then you must do it, else neither of us can proceed'). For *p1* to proceed *p2* must also be in the same state, ready to proceed. This is the strictest form of synchronisation because it ranges over all processes composed using it, forcing the processes that are ready to communicate to wait for those taking part in the full synchronisation.

Symbol	*Example*
||	*p1 || p2 || p3*

Process Interleaving
In some ways we can think of process interleaving as the opposite of full synchronisation. Interleaved processes can engage in separate actions, independent of other processes in the system. For example, two people at a desk in an office can communicate with each other or the rest of the office, or carry on with their own tasks, independently. If some communication does occur with these two people (which we model as processes) then some synchronisation with the receivers must also occur (like any other process communication).

Symbol	*Example*
|||	*p1 ||| p2*

Selective Synchronisation
A selective parallel operator can be used to synchronise processes on common actions listed within the operator. For example *p |[x]| p2* will selectively synchronise on action *x* and proceed in parallel for other actions. The selective parallel operator is binary and associative and if used in a multi-way communication will require *x* to synchronise across all listed processes containing *x*. The rules for the use of *|[x]|* also state that in a case such as *(p1 |[x]| p2) |[y]| p3* the process *p3* will synchronise in a non-deterministic way with either process *p1* or *p2* on events exclusive to *p3*; that is to say that *p3* will synchronise with *p1* and *p2* if they share some common gate. Remembering that if all of the processes share the same gate name then they will all have to synchronise on that gate with the appropriate action.

Symbol	*Example*
|[x]|	*p1 |[x]| p2*

Appendix B

```
specification GreenHouse(spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level,alState:State) : noexit

        library
                NaturalNumber, Boolean
        endlib

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

        type HumidType is NaturalNumber
                sorts Humid < Nat
        endtype (* HumidType *)

        type TempType is NaturalNumber
                sorts Temp < Nat
        endtype (* TempType *)

        type LevelType is NaturalNumber,Boolean
                sorts Level < Nat
                opns
                        setLevel: Nat → Level
                        incLevel: Level → Level
                        decLevel: Level → Level
                        isMinLevel: Level → Boolean
                        isMaxLevel: Level → Boolean
                eqns
                        forall n:Nat (* in the range 0..9 *)
                        ofsort Level
                                incLevel(setLevel(0)) = setLevel(succ(0));
                                incLevel(setLevel(n)) = setLevel(succ(n)) if n lt 9;
                                incLevel(setLevel(n)) = setLevel(9) if n ge 9;
                                decLevel(setLevel(0)) = setLevel(0);
                                decLevel(setLevel(succ(n))) = setLevel(n) if n le 8;
                        ofsort Bool
                                isMinLevel(setLevel(0)) = true;
                                isMinLevel(setLevel(succ(n))) = false;
                                isMaxLevel(setLevel(9)) = true;
                                isMaxLevel(setLevel(n)) = false if n lt 9;
        endtype (* LevelType *)

(* Set definitions for use throughout the specification to help keep the text minimal, commented out because not part of the LOTOS language.

                        spGates ≝ {SetSprayOn,SetSprayOff,sprayOn,sprayOff}
                        hyGates ≝ {SetMinHumid,SetMaxHumid,ReadHumid,sprayOn,sprayOff,open,close}
                        wcGates ≝ {SetWindow,open,close,on}
                        thGates ≝ {SetMinTemp,SetMaxTemp,ReadTemp,inc,dec,open,close}
                        heGates ≝ {SetHeat,inc,dec,on}
                        alGates ≝ {on,SetAlarmOff}

                        spState ≝ {sprayer}
                        hyState ≝ {minHumid,maxHumid}
                        wcState ≝ {window}
                        thState ≝ {minTemp,maxTemp}
                        heState ≝ {heater}
                alState def {alarm}
*)

        behaviour
                [(min gt max)] → (exit)
            []
                [(min le max)] → GHCS(spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level,alState:State);noexit
            where

                process GHCS[spGates,hyGates,thGates,heGates,wcGates,alGates](spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level,alState:State) : noexit :=
                        hide on in
                                STWin[spGates,hyGates,thGates,heGates,wcGates](spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level) |[on]| Alarm[alGates](alState)

                        where


                        process SpHy[spGates∪hyGates](spState:State,hyState:Humid) : noexit :=
                                hide sprayOn,sprayOff in
                                        Sprayer[spGates](spState) |[sprayOn,sprayOff]| Hygrometer[hyGates](hyState)
                        endproc


                        process ThHe[thGates∪heGates](thState:Temp,heState:Level) : noexit :=
                                hide inc,dec in
                                        Thermometer[thGates](thSate) |[inc,dec]| Heater[heGates](heState)
                        endproc


                        process STWin[spGates,hyGates,thGates,heGates,wcGates](spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level) : noexit :=
                                hide open,close in
                                        (SpHy[spGates∪hyGates](spState:State,hyState:Humid) ||| ThHe[thGates∪heGates](thState:Temp,heState:Level)) |[open,close]| Window[wcGates](wcState)
                        endproc


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


                        process Hygrometer[Q](min:Humid,max:Humid) : noexit :=
                                SetMinHumid ? h:Humid;
                                        ([h le max] → Hygrometer[Q](h,max)
                                                []
                                        [h gt max] → Hygrometer[Q](min,max))
                                []
                                SetMaxHumid ? h:Humid;
                                        ([h ge min] → Hygrometer[Q](min,h)
                                                []
                                        [h lt min] → Hygrometer[Q](min,max))
                                []
                                ReadHumid ? h:Humid;
                                        ([h lt min] → sprayOn;close;Hygrometer[Q](min,max)
                                                []
                                        [h gt max] → sprayOff;open;Hygrometer[Q](min,max)
                                                []
                                        [h ge min and h le max] → Hygrometer[Q](min,max))
                        endproc (* Hygrometer *)


                        process Window[R](cw:Level) : noexit :=
                                SetWindow ? cw:Level;Window(cw)
                                []
                                open;
                                        ([not isMaxLevel(cw)] → Window[R](incLevel(cw))
                                                []
                                        [isMaxLevel(cw)] → on;Window[R](cw))
                                []
                                close;
                                        ([not isMinLevel(cw)] → Window[R](decLevel(cw))
                                                []
                                        [isMinLevel(cw)] → on;Window[R](cw))
                        endproc (* Window *)
```

```
                                process Thermometer[S](min:Temp,max:Temp) : noexit :=
                                        SetMinTemp ? t:Temp;
                                                ([t le max] → Thermometer[S](t,max)
                                                        []
                                                [t gt max] → Thermometer[S](min,max))
                                        []
                                        SetMaxTemp ? t:Temp;
                                                ([t ge min] → Thermometer[S](min,t)
                                                        []
                                                [t lt min] → Thermometer[S](min,max))
                                        []
                                        ReadTemp ? t:Temp;
                                                ([t lt min] → inc;close;Thermometer[S](min,max)
                                                        []
                                                [t gt max] → dec;open;Thermometer[S](min,max)
                                                        []
                                                [t ge min and t le max] → Thermometer[S](min,max))

                                endproc (* Thermometer *)


                                process Heater[T](ch:Level) : noexit :=
                                        SetHeat ? ch:Level;Heater(ch)
                                        []
                                        inc;
                                                ([not isMaxLevel(ch)] → Heater[T](incLevel(ch))
                                                        []
                                                [isMaxLevel(ch)] → on;Heater[T](ch))
                                        []
                                        dec;
                                                ([not isMinLevel(ch)] → Heater[T](decLevel(ch))
                                                        []
                                                [isMinLevel(ch)] → on;Heater[T](ch))
                                endproc (* Heater *)


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

                endproc (* GHCS *)

endspec (* behaviour-GHCS *)
```