DIVISION OF COMPUTER SCIENCE

# Increasing the usability of Formal Specification Techniques through a combination of complementary formal languages and automated verification tools.

P. N. Taylor
C. E. Britton

Technical Report No.210

August 1994

# Increasing the usability of Formal Specification Techniques through a combination of complementary formal languages and automated verification tools.

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

August 1994

## Abstract

This paper addresses three main issues. Firstly, the combination of formal specification languages to model proposed systems. For this paper we introduce the dual specification of a case study system using the formal languages LOTOS [1,11,13] and the Z notation [17] to capture the behaviour of the complete system, including the modelling of data abstraction, information hiding and modularisation. Secondly, the production of an industrial-strength specification using a mechanical, automated CASE tool to verify the syntax of the formal specification. It is hoped that specifications which are verified mechanically will be more widely acceptable to industry because of the consistency enforced by the CASE tools used to check them. Finally, the transition from formal specification to implementation using the dual formal specification approach introduced in this paper.

We show how a formal specification can be developed and then verified using a mechanical syntax/type checking tool running on a desktop PC, Logica's CASE tool *Formaliser* [7,12]. We use a small case study as the foundation for a dual formal language approach to solving a systems development problem. A LOTOS specification is used to capture the concurrent behaviour of the system's components whereas we use the Z notation to capture the structure of the data for each process. Certain issues arise regarding the relationship between system invariants and process behaviour which are not fully covered by LOTOS but are added by the Z model.

Our choice of formal languages to specify the case study problem enables us to mechanise the verification of the syntax of either the LOTOS specification (using a LOTOS interpreter [8,14]) or the Z specification using *Formaliser*. In this paper we concentrate on the use of *Formaliser* to verify the Z specification of the case study system.

Besides the use of a software tool to check the Z we also consider a broader central theme concerning data abstraction and information hiding and how this might best be achieved using both LOTOS and Z. It is widely accepted that the solution to a problem can be simplified by breaking that problem up into parts and solving them with small discrete steps. In computer systems design and specification we can reduce complexity by modularisation and abstraction. The ideas discussed in this section of the paper are used as the foundation for an implementation of the case study system using the object-oriented programming language C++ [18]. We show how information hiding via data abstraction can be achieved in Z, using schema inclusion and schema calculus (and captured by *Formaliser's* multiple-document cross-referencing). We also show how LOTOS uses a hierarchy of process encapsulations together with the hiding of communicating gates to achieve information hiding and encapsulation.

The combination of both formal languages presents a more complete picture of the proposed system to potential implementors, which we believe removes much of the ambiguity surrounding a specification written in just one formal language with just one perspective.

# 1 Introduction

In this paper we show how a dual language approach to the formal specification of a case study system effectively captures the requirements of that system. We also aim to keep the complexity of the specification (and therefore the system) down to a minimum. Experience has shown that the complexity attached to many of the problems that we as computer scientists solve can be reduced. Our approach to solving many problems is the same, regardless of the problem; we break the problem down into manageable parts and then work towards a solution. One obvious improvement with this modular technique is the reduction of complexity and the increased maintainability of the systems that we produce. At each stage we would seek to justify our findings and verify our work. Using mechanical checking tools we can speed up the process of producing system specifications that are internally consistent and therefore acceptable to both academics and industry alike.

Our study introduces both a LOTOS and Z specification of the same system. We show how the specification languages together capture the behaviour and structure of the individual processes that make up the system. LOTOS [1,2,5,11,13] was chosen to capture the system's concurrent behaviour because of its industrial-based background; having been spawned by ISO for use with network and distributed systems protocol specification [11]. We require a formal specification language that can give us the power to model the relationships between the temporal interaction of processes in our system such that the behaviour of those processes can be observed external to the system; that is time ordered communications between processes.

Because of LOTOS' use of ACT-ONE [4] to model data (using strict algebraic notation) we require the use of another language to capture the data model of our system more completely than LOTOS. We chose Z as our second language because it can capture the data structure of the system's processes and the manipulation of that data. The low level structure of each process is modelled more completely by Z than LOTOS and this level of detail justifies its inclusion in our dual specification strategy.

Both of our chosen formal languages have already been proved in industry. LOTOS being extensively used by ISO/OSI for network/distributed systems protocol specification [12] and Z by IBM for its CICS [3] product development. The use of both languages by industry gave us confidence that they are already known to industry, together with the fact that both languages have verification tools provided for them [8,9,13,15]. We felt that the interpretation of the specification and the transition from specification to implementation from academic to industrial environments would be simplified because of the familiarity of both LOTOS and Z in both of these working environments.

To illustrate how formal specifications can be checked mechanically we use a software tool produced by Logica Cambridge Limited, called *Formaliser* [7,12], which can represent the specification on a desktop PC. The formal text is verified and diagnostic errors are produced to aid the specifier in tracking down problems. Explanations regarding the semantics of the formal language specification become simpler and more readily understood. Mechanical checking leads to an increased confidence in the internal correctness of the specification, both as a model of the system and as a basis for implementation.

The structure of this paper is as follows:

- Section 2 describes the case study system used throughout this paper.
- Section 3 describes how the different formal models capture different aspects of the system according to characteristics of the formal languages.
- Section 4 describes the composition of the separate processes in the system, together with alternative views of process composition imposed by the different formal languages and any changes in the system's behaviour as a result of that composition.
- Section 5 describes the interaction between processes, including the concurrent aspects of the system and how each formal language copes with concurrency.
- Section 6 discusses the ideas behind data abstraction and information hiding. We show

how both LOTOS and Z model these two areas of abstraction.

• Section 7 concentrates on the segmentation of the system by grouping processes together in order to capture the system's required behaviour as well as provide the data encapsulation and information hiding discussed in section 6.

• Section 8 describes using *Formaliser* to verify the Z specification of the case study system. The advantages surrounding the use of a CASE tool to aid the specifier in the production of a specification are then discussed. We also present evaluation criteria for mechanical checking tools and show how these are met by *Formaliser*.

• Section 9 shows how a smooth transition from specification to implementation can be achieved as a consequence of our dual specification approach. We show how each formal language can be interpreted to give a faithful implementation of the required system.

• Section 10: Conclusions are drawn from this dual specification approach. We discuss the benefits of using two formal specification languages to model systems, together with the merits arising from the use of mechanical checking tools to verify those specifications.

## 2 The case study system

Our case study originates from a simple problem which is often given to undergraduate students as part of a formal 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 six processes as follows:

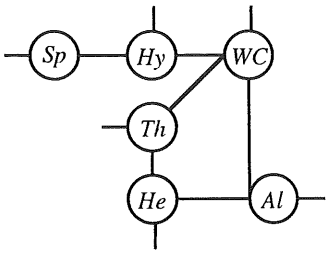


**Figure 1**

The simplified diagram of the complete system in Figure 1 omits the exact details of the communications as it only shows how the processes connect to each other and their environment. It does not show the nature of those connections.

## 3 Modelling the system with complementary formal languages

Using two formal specification techniques we can build two different views of the greenhouse control system (GHCS). For the LOTOS view of the GHCS we can identify those parts of the system that perform actions typically modelled by a concurrent specification language. The communications that take place between the GHCS components and the behaviour of each individual component are the areas of the model that are captured by LOTOS. However, LOTOS cannot give us a complete picture of the system on its own; particularly in view of the data structures associated with each system component and system invariants ranging over the whole system. Therefore, we use the Z notation [18] to model the data and invariants in the GHCS and capture any requirements which LOTOS is unable to model.

We adopt a similar approach for specifying the GHCS in the complementary formal languages used in this paper. We can identify individual GHCS components easily due to the nature of the system; each component being quite independent. The modular approach that we adopt for system design and specification is evident in the specification of the GHCS in both of the chosen formal specification languages. For example, the GHCS components can be specified simply in LOTOS as separate processes thus:

```
process Sprayer[SpGates](s:State) : noexit :=
...

process Hygrometer[HyGates](min:Humid,max:Humid) : noexit :=
...

process Window[WCGates](cw:Level) : noexit :=
...

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

process Heater[HeGates](ch:Level) : noexit :=
...

process Alarm[AlGates](s:State) : noexit :=
...
```

**Figure 2**

where each process can reference operations belonging to other processes in the system if those operations appear as part of the gate list (e.g: as an element of the set *SpGates*). The thread of control within the system woven by these referenced operations gives us the diagram seen earlier in Figure 1. The parameters *[XGates]* specifying available points of entry into the process and

*(p:Q)* specifying some state variable *p* of type *Q* used to capture the current state of the process (LOTOS stores the process' state dynamically, unlike the static data modelled in Z).

We need to keep encapsulating the state of each process because LOTOS has its data model founded on an algebraic specification language, namely ACT-ONE [4]. This implies that we cannot specify the storage of some static state inside a process. We cannot model formally the static data (or state) of a process in LOTOS, like the data contained within a record structure of a programming language; that is the function of the Z specification. Consequently, the representation of a process' state is continually referenced when calling the process and is not actually stored anywhere — the state of the process is totally dynamic. Alternative views concerning the formal modelling of data in communicating systems do exist [2], but this topic is beyond the scope of this paper. Examples of encapsulating the current state of a process can be seen throughout the LOTOS specification of the GHCS in the appendices.

For the Z formal specification we specify each GHCS process as a collection of Z schemas. Each schema contains the data structure (or state) of the process. The invariants on the process reinforce the algebraic invariants previously specified in LOTOS. We would specify the *Hygrometer* in LOTOS as:

```
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 *)
```

**Figure 3**

and the Z equivalent as a collection of schemas, starting with the definition of some constants, Boolean type redefinitions and the base state schema.

$$minReading == 0$$
$$maxReading == 100$$
$$on == True$$
$$off == False$$
$$Reading == \{n:N \mid minReading \leq n \leq maxReading\}$$

| *Hygrometer* |
| --- |
| $minHumid : Reading$<br>$maxHumid : Reading$ |
| $minHumid \leq maxHumid$<br>$maxHumid \geq minHumid$ |

**Figure 4**

The basic structure of the *Hygrometer* (Figure 4) will be used during the implementation stages to form a class structure for the object-oriented implementation. Each entry in the basic

declarations part of the schema (above the central dividing line) denoting a field in the class structure. The remaining schemas required to complete the *Hygrometer* specification can be seen below (Figure 5). These schemas provide us with the ability to modify, reference and initialise the state of the *Hygrometer* process respectively.

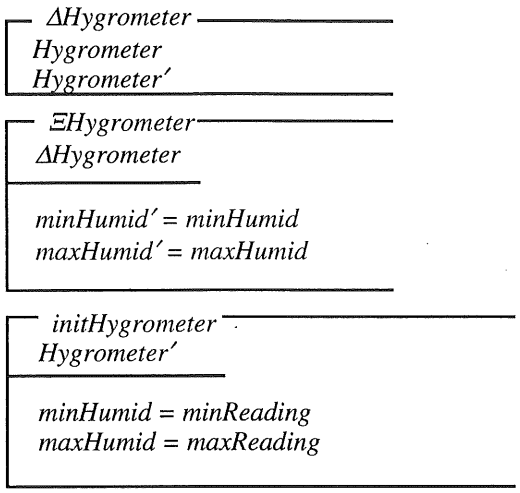


**Figure 5**

The initialisation of the *Hygrometer* process (schema *initHygrometer*) only occurs **once** and dictates the state of the *Hygrometer* when the system becomes 'live'. We can view this process initialisation as a one-off statement in the main body of the executable code (in terms of the system's implementation).

One area that we must be wary of when using two quite distinct formal languages, such as LOTOS and Z, is to remember which model we are currently using. Although both models cover the same system they show different views of that system. To avoid confusion we have found that it is good practice to view each model separately when considering the whole system and jointly only when viewing the separate processes. Ideally the Z model is used when constructing the basic structure of each process as it shows more detail about the data structures at this foundation level. As the specification (or implementation) grows the LOTOS behavioural model is brought into view. It should never be the case that any part of the specification contradicts the complementary specification's model of the system. If a contradiction in either data model, data manipulation or behavioural model presents itself during development then an error in the basic design of the system has been found. We could argue that herein lies another reason for a dual approach to system's design and specification; yet another safety net to catch errors in the specification of the system.

To summarise, as long as we are aware of which view of the system we are currently looking at, in terms of which formal model is being reviewed, then the problems associated with digesting too much formalism and therefore too much complexity are reduced. We feel that the benefits of using two formal languages to capture a system outweigh the potential pitfalls associated with a single formal specification approach.

## 4 Process Composition

The case study system has six separately identifiable processes which can be brought together to form the whole GHCS. In some systems the boundaries between processes can be more difficult to define. It could be possible for one large process to perform the tasks of several smaller ones. However, should this monolithic organisation prove to be the case then we have lost much of the flexibility that modularity provides. The divide-and-conquer strategy that we

adopt to solve many problems steers us away from such large structures. With our LOTOS and Z specifications we seek to provide flexibility and maintainability together with a formal model of the system. The format of both LOTOS and Z specifications provide us with a means of modularising the specification, via separate process and schema definitions.

In LOTOS, the different uses of three composition operators (||| — interleaved, |[x]| — selective parallel and || — full synchronisation) will effect the behaviour of the overall system depending on how those operators are combined together. Both the interleaving and full synchronisation operators can be defined in terms of the selective parallel operator using the following equivalences, noting that the alphabet $\alpha$ of a process *pN* (shown as $\alpha pN$) 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.

Full synchronisation is equivalent to selective parallel composition if the gate-list *x* is the set of all actions defined for both composed processes. Process interleaving is equivalent to selective parallel composition if the gate-list *x* is the empty set; i.e: no synchronisation occurs, leaving the composed processes to proceed independently.

Multiple communications across process can be achieved in LOTOS by using common gate names between more than one process. Synchronisation between two processes can then be extended to multi-process communication (synchronisation) by composing several processes together using selective parallel composition, as shown below:

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

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.

The Z notation does not provide us with a means of scheduling processes that are cojoined, but this composition is not important as part of the Z model. For the complementary Z specification we do not have to worry about alternative behaviour based on the order of composition operators. In Z we can simply use schema reference inside the actual schema that we wish to partner with other processes in the system. Figure 6 is an example of schema inclusion, bringing parts of the GHCS system together:

*Hygrometer*

$\Delta$*Hygrometer*
*SprayerProcess*
*WindowControllerProcess*
...
*sprayOn! : B*
*sprayOff! : B*
...

---

...
$((ReadHumid? \land current? < minHumid) \Rightarrow sprayOn! = True \land \ldots$
...

**Figure 6**

where the *Hygrometer* process uses operations supplied by both the *Sprayer* process and *WindowController* process. Show diagrammatically as:

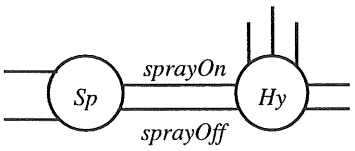


**Figure 7**

which shows the hidden communications *sprayOn!* → *sprayOn?* and *sprayOff!* → *sprayOff?* within the *SpHy* segment. Any components wanting to use the *Sprayer/Hygrometer* pairing would simply include a reference to *SpHy* in their schema's basic declarations section.

## 5 Capturing Concurrent Behaviour

In our GHCS model an expression like *(p1 |[x]| p2) |[x]| p3* would deadlock the *Hygrometer* and the *Thermometer*, together with the *Heater* and *Alarm* which all share the *Window Controller* resource. Deadlock would occur if all of these processes fail to synchronising on the same action. Processes *p1/p2* could progress but they in turn would have to wait on process *p3* to synchronise before the whole system could progress.

Our aim throughout has been to keep processes in certain parts of the system from having influence over processes in other non-related parts of the system. A higher-level view of the system reveals distinct segments in the structure of the GHCS.

- *Segment 1 (SpHy) = Sprayer/Hygrometer*
- *Segment 2 (ThHe) = Thermometer/Heater*
- *Segment 3 (STWin) = SpHy/ThHe/Window Controller*
- *Segment 4 (GHCS) = STWin/Alarm*

shown diagrammatically as:

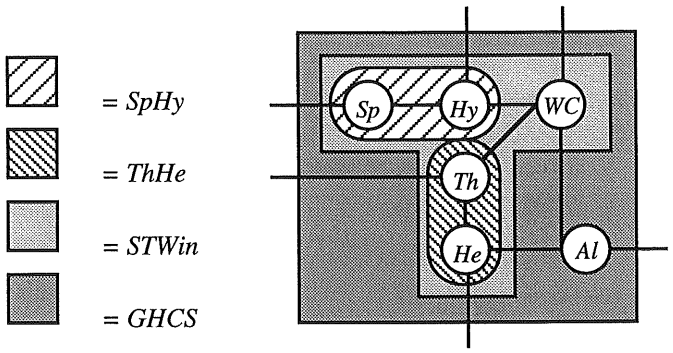


**Figure 8**

Both the *SpHy* and *ThHe* segments communicate with the *Window Controller* to form the *STWin* segment. The *SpHy* and *ThHe* segments are not permitted to communicate with each other because they are required to share the *Window Controller* resource and consequently use the same lines of communication; namely *open* and *close*. We enforce this mutual exclusion using the interleaving operator (|||) to compose *SpHy* and *ThHe* segments together.

In Z this exclusion is not necessary because there are no timing constraints to worry about. The Z concentrates on the fact that a communication occurs to some remote operation and not when that communication occurs in the temporal model. Here we see the different views of the GHCS captured by our two distinct formal models.

In LOTOS the composition is quite straight forward provided that we fully understand the behaviour of the system according to our combination of the composition operators. However, in Z the order of events cannot be dictated by the order of the included schemas in a

segment schema, such as *SpHy*. The Z model is concerned with what processes are connected together, not the nature of those connections. Our Z specification does not need to know about the scheduling of the processes in terms of concurrency.

## 6 Information Hiding and Abstraction

In LOTOS and Z there are ways of hiding internal actions from separate processes in the system and from the environment (the outside world). LOTOS provides the ***hide*** operator to restrict the observation of process gates. 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
```

**Figure 9**

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. In LOTOS 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.

With our Z version of the GHCS specification the hiding of internal communications between segments inside the system cannot be performed as they are in LOTOS because there is no distinction between different operations used to modify the same part of the state. For example, the use of either *sprayOn!* and *SetSprayOn!* would not be distinguished by the system as either will imply some modification to the *Sprayer* state. LOTOS has the problem of different gates being linked to the same parts of the state so it must differentiate between them by hiding the internal gates and allowing the external gates to be observed (e.g: *SetSprayOn* is observable whereas *sprayOn* is hidden). If we restrict a schema by some process' field then neither the system or its environment will have access to that part of the process' structure. Ideally we would simply restrict the hidden operations to stop them being accessed outside a segment, as shown below:

*SpHy*

$SprayerProcess \setminus \{sprayOn?, sprayOff?\}$

$HygrometerProcess \setminus \{sprayOn!, sprayOff!\}$

**Figure 10**

## 7 Building the System

The separate segments that make up the modular GHCS can be brought together via process composition (LOTOS) and schema inclusion and schema calculus (Z). For LOTOS we require the following definitions for the gate lists per process to help in the simplification of the specification:

$spGates \stackrel{def}{=} \{SetSprayOn, SetSprayOff, sprayOn, sprayOff\}$

$hyGates \stackrel{def}{=} \{SetMinHumid, SetMaxHumid, ReadHumid, sprayOn, sprayOff, open, close\}$

$wcGates \stackrel{def}{=} \{SetWindow, open, close, on\}$

*thGates* $\underline{\underline{def}}$ *{SetMinTemp,SetMaxTemp,ReadTemp,inc,dec,open,close}*
*heGates* $\underline{\underline{def}}$ *{SetHeat,inc,dec,on}*
*alGates* $\underline{\underline{def}}$ *{on,SetAlarmOff}*

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

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

The *Sprayer* and *Hygrometer* processes 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 segment *SpHy* and stops any influence on the communications that take place within the segment via the gates *sprayOn* and *sprayOff*. Below, in Figure 11 are LOTOS and Z versions of the *SpHy* segment, together with a diagram showing a representation of the segment:

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

*SpHy*
*SprayerProcess* \ *{sprayOn?, sprayOff?}*
*HygrometerProcess* \ *{sprayOn!, sprayOff!}*

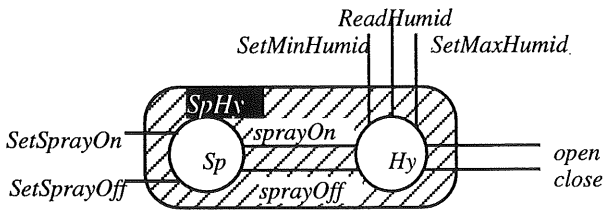


**Figure 11**

The segment *ThHe* encapsulates the *Thermometer* and *Hygrometer* processes in much the same way as that of *SpHy*:

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

*ThHe*
*ThermometerProcess* \ *{inc!,dec!}*
*HeaterProcess* \ *{inc?,dec?}*

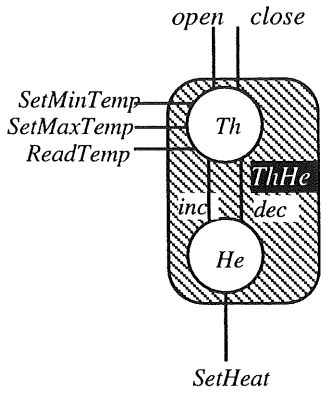


**Figure 12**

At a level above the *SpHy* and *ThHe* segments, is the *STWin* segment which brings these two low-level components together whilst maintaining their individuality. Parallel composition is used to achieve this requirement. We not want to broadcast any information about the internal workings of *SpHy* or *ThHe*. For example, 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 to enforce a strict separation between *SpHy* and *ThHe* because they contain common gates and would consequently have to wait on each other, forcing delays and possibly deadlock. We can selectively compose *SpHy* and *ThHe* with *Window Controller* so that they can talk to *Window Controller*, 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***

*STWin*
*SpHy* \ {*open!, close!*}
*ThHe* \ {*open!, close!*}
*WindowControllerProcess* \ {*open?, close?*}

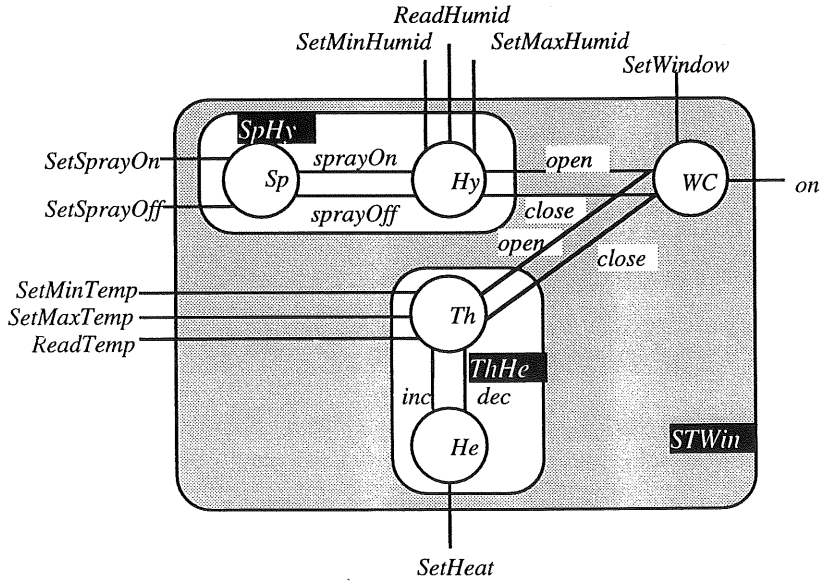


**Figure 13**

We can complete the GHCS 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 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

┌ GHCS ──────────────────
| STWin \ {on!}
| AlarmProcess \ {on?}
└────────────────────────
```

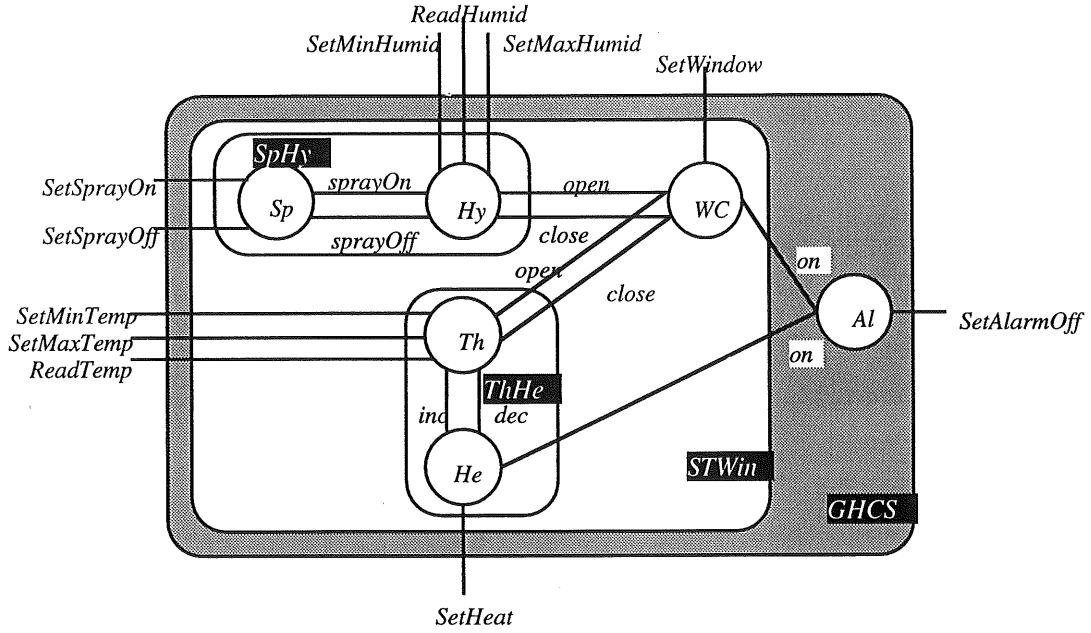


**Figure 14**

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. The restrictions on the Z equivalent schemas for each segment perform the same task as the ***hide*** operator in LOTOS; the abstraction of the segments operation so that unauthorised references cannot occur.

## 8 Verifiable 'Industry-Strength' Z using Formaliser

After we have produced our specifications we would like to assure ourselves that what we have written is consistent and syntactically correct. In this section we concentrate on the verification of the Z specification of the GHCS using *Formaliser* [7,12] and discuss the benefits that such a tool can bring to the specification process.

One of the problems with formal specifications is that an internal inconsistency in the specification will allow any property to be proved as a theorem. Consider the following example:

We can complete the GHCS 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 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
```

*GHCS*
*STWin* \ {*on!*}
*AlarmProcess* \ {*on?*}

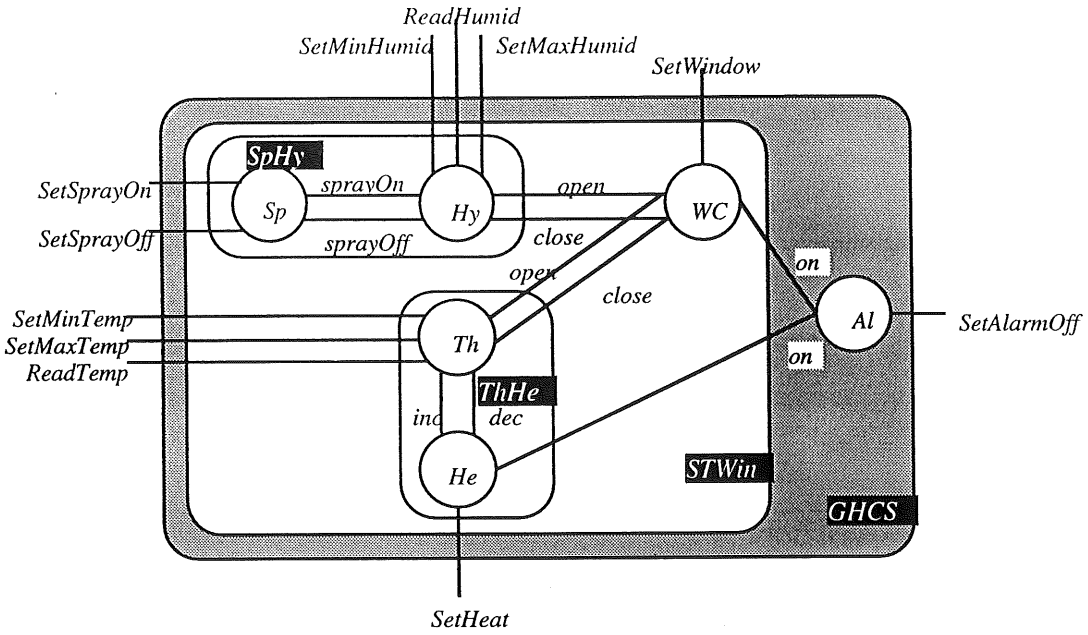


**Figure 14**

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. The restrictions on the Z equivalent schemas for each segment perform the same task as the ***hide*** operator in LOTOS; the abstraction of the segments operation so that unauthorised references cannot occur.

## 8 Verifiable 'Industry-Strength' Z using Formaliser

After we have produced our specifications we would like to assure ourselves that what we have written is consistent and syntactically correct. In this section we concentrate on the verification of the Z specification of the GHCS using *Formaliser* [7,12] and discuss the benefits that such a tool can bring to the specification process.

One of the problems with formal specifications is that an internal inconsistency in the specification will allow any property to be proved as a theorem. Consider the following example:

*value = open ∧ ¬open ⊢ system will terminate ∧ nuclear reactor is safe*

Consistency in a formal specification can be difficult to prove, but inconsistency renders the specification useless as we cannot trust any conclusions that we derive. Formal specifications are notoriously difficult to produce and maintain, so if they are to be used in earnest, tools are required that will help to overcome these problems, leaving the specifier free to concentrate on the important aspects of the specification. This section establishes metrics for the evaluation of formal language checking tools. We then evaluate *Formaliser* in the light of these metrics.

*Formaliser* is an interactive software tool which can check the internal consistency and syntax of formal specifications written in the Z notation [17]. It is produced by Logica Cambridge Limited (U.K) as a CASE tool for software engineers [7,12]. The latest version of *Formaliser* runs under Windows 3.1 on IBM compatible PC's and is therefore easily accessible, both to academics and those working in industry. It makes use of the standard Windows environment and only permits valid commands to be entered via pull-down menus, thereby enforcing a strict control on the user.

Each Z expression, using a specific grammar, forms part of a parse tree structure which is displayed to the user in the standard Z schema format. A typical *Formaliser* screen layout is shown in the following diagram:

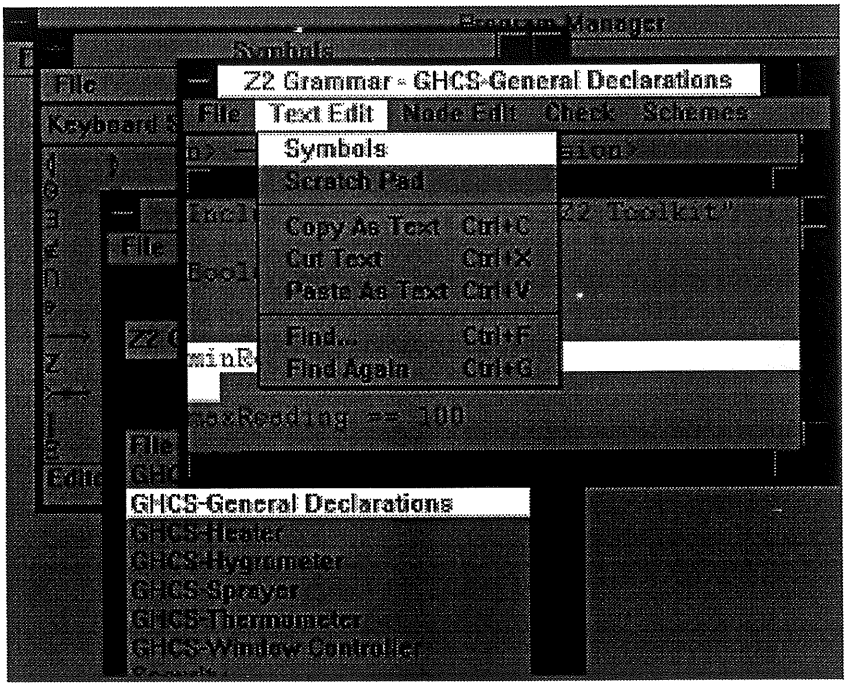


**Figure 15**

Unlike some other formal language checking tools [9] *Formaliser* can check the syntax of Z specifications using the **complete** Z notation, rather than a subset of the notation. External documents that make up the complete specification can be referenced by linking them together. For the GHCS we use this referencing to enforce the encapsulation of GHCS processes to form groups of process (segments), such as *SpHy* and *ThHe* (see Figures 11 to 14). Any changes to the specification's documents will be included in the checking process — internal consistency across the whole specification is therefore guaranteed.

*Formaliser* has two ways of catching errors that might appear in formal specifications.

1. Text which is typed directly into the tool is parsed immediately — on-the-fly. Any syntax errors are displayed in an error window, requiring the user to remove the errors before being allowed to proceed.

2. Type errors are identified when parts of the specification are selected for on-board checking. The Z grammar is used to determine the validity of the selected expressions, where the syntax and types of variables are checked for consistency — a diagnostic error window being displayed if there are any problems, as seen below:

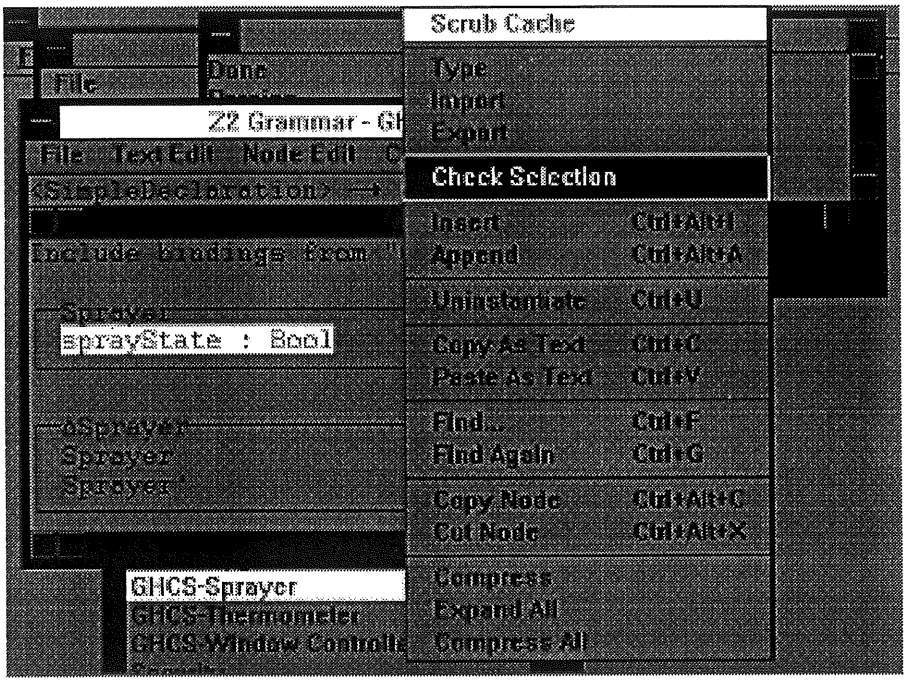


**Figure 16**

The selected text in Figure 16 would pass the syntactic check on text entry because the expression conforms to the Z notation's syntax. However, once the schema is complete and a full check is performed *Formaliser* would catch the type mismatch error and display the following information in a diagnostic error window:

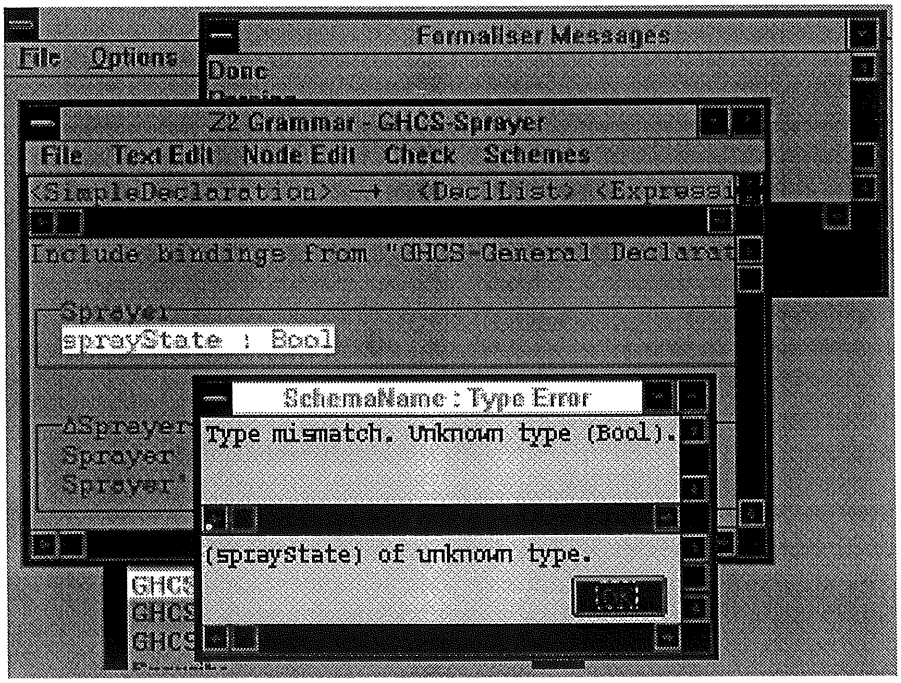


**Figure 17**

Any errors found in the specification will be relayed back to the user so that the appropriate action can be taken. The solution to the previous example would be to declare *k?* as a power set of type *Key* and resubmit the schema for checking.

In order to evaluate *Formaliser* as a formal language checking tool, to aid the software engineer, we need to ensure that it meets a pre-defined set of criteria, such as those found in Fisher (1991) [6]. These criteria are listed below, together with brief evaluations of the relevant aspects of *Formaliser*:

1. Produce quantitative and verifiable designs.

The production of quantitative and verifiable designs is central to *Formaliser's* own design. The on-the-fly and on-board checking facilities stop any inconsistencies and errors from creeping into the specification's text. By referencing the complete Z notation, together with local and external definitions, confidence can be established that the specification is internally consistent and free from error.

2. Simplify and decompose requirements and designs into manageable components.

The multi-document editing that *Formaliser* provides can help with the decomposition of the specification into manageable components. Details from one specification document can be copied into any other document and logically linked together using the "`Inclusion`" statement at the start of any *Formaliser* document.

3. Support change by being adaptable.

Change and modifications to the specification are provided for by *Formaliser's* built-in editor. The cross-referencing of any additions to the specification is provided by the linking facilities within *Formaliser.*

4. Save time and money. Aid the production of cheaper and more efficient formal specifications.

The overall speed of production of Z specifications can be increased thanks to *Formaliser's* checking facilities. The savings in time and money are further enhanced by the automatic transformation of the specification's text into a $\LaTeX$ source file, for the production of high quality hard copies.

From our own experiences with this and previous systems we have found that it is considerably easier to write consistent Z specifications using *Formaliser* to check our work for errors. As with many institutions, there are few experts available to check our work at short notice. We have used *Formaliser* to overcome the problem of finding syntactic and type related errors in our initial work. Moreover, some further issues surrounding the difficulties with producing formal specifications are also addressed by *Formaliser* — such as availability and accessibility. The user-friendly interface and availability on IBM compatible PC's makes it easily accessible and therefore provides more people with the chance to use a formal language checking tool in the production of their specifications.

## 9 Implementation in C++

As a consequence of producing two formal specifications and mechanically checking them we can proceed with the implementation of the specified system with the knowledge that many areas of ambiguity and assumption have been removed due to the increased formalism. We have introduced the idea of the dual formal specification of a system, rather than the single view modelled by just one language.

At the implementation stage we can begin to see how each formal language can help in the development of a working system. The identification of potential objects or abstract data

types has already been performed. For example, the separate processes within the GHCS are idea candidates for classes in the object-oriented programming language C++ [18]. We can define the structure of these classes using the state schemas provided by our Z specification of the system. Any relationships between the classes can then be derived from the composition of the processes in LOTOS and schema inclusion or schema calculus in Z. The initial stage in the development (having defined the classes to be modelled by the software) is to use the Z specification to structure each class and provide functions to operate upon that class. Consider the following class definitions written in C++:

```
// define Hygrometer first to resolve forward referencing error.
class Hygrometer;

class Sprayer
{
        private:
                Boolean sprayerState;

        public:
                ...
                void sprayOn(void);
                ...
};

class Hygrometer
{
        friend void Sprayer::sprayOn(Hygrometer *hyPtr);
        ...
        private:
                short minHumid;
                short maxHumid;

        public:
                ...
                void SetMinHumid(short min);
                ...
};
```

The relationship between the *Sprayer* and *Hygrometer* is defined as `friend` to enable the *Hygrometer* to access only those functions of another class that are listed. We can use either specification to tell us about the relationships between processes. However, the Z provides us with a clearer picture of the internal structure of each class and the LOTOS with a view of how the classes interact. For the operations required by each process we use the LOTOS to identify the names and functionality of the operations. Because of the close resemblance between LOTOS and a structured programming language the transition between the specification and the source code is minimal. Consider the following LOTOS and C++ extracts for the behaviour of the *SetMinHumid* function for the *Hygrometer* process :

```
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))
        ...
endproc (* Hygrometer *)
```

```
Hygrometer::SetMinHumid(void)
{
        short   min=minHumid;

        cout << "Please enter new Minimum Humidity: ";
        scanf("%d",&min);
        cout << "\n";
```

```
        if(min <= maxHumid)
                minHumid = min;
} // end-SetMinHumid
```

The structure of the C++ code follows closely the LOTOS structure. Each of the remaining processes and their operations can be encoded by using the LOTOS specification as a guide. The relationships between classes (processes) using the `friend` facility and the subsequent implementation of the process operations directly from the LOTOS all point towards a smooth transition from specification to implementation.

Notice how we used the Z early on in the implementation stages to derive class structures. The use of Z at the low-level design of the code underlined the main contribution that Z gave us as part of our development strategy. The LOTOS specification then proved to be useful during the development of the behavioural model. The hard work throughout the whole course of our system's development was done at the start of the cycle; during the specification stages. The dual specification approach that we have adopted forces us to be more rigorous during the specification stages. However, the scope of the two specifications leave us relatively few issues to resolve in order to implement the design. Subsequent maintenance of the system is also reduced as a direct result of this early work as the modularity of the code enhances our ability to single out problem areas and modify them without disturbing other parts of the system.

## 10 Conclusions

The work carried out as part of the research for this paper has given us a new insight into a different approach to specifying computer systems using formal languages. The use of two languages, chosen for their acceptability to industry and formal expressive power, gives us the ability to address issues surrounding vague areas in a system's specification. In the past, using just one formal language, certain assumptions had to be made to cope with the lack of formalism in key areas of a system's design. In this paper the GHCS would have a partial model of the structure of each process if LOTOS were the only formal specification language available to the software development team. A decision about the structure of the processes may be left until the last possible moment because of the lack of any strict guidelines regarding the internal workings of each system process. We feel that this vagueness and possible ambiguity should be removed from the development process at the very start of a system's design. By maintaining a tight grip on all aspects of design and specification we can ensure that errors in development do not occur as a direct result of a lack of formalism attached to certain parts of a system.

Together with our choice of formal languages we must also ensure that support is available to check the validity of the languages. Our example case study was modelled using two languages which are well supported by software tools supplied for mechanical verification. We have concentrated on the Z notation [17] and *Formaliser* tool [7,12] but the LOTOS language also has software tool support [8,14]. Further work in this area could be based on the evaluation of tools used to support LOTOS, as opposed to those for Z.

The benefits of using software tools to check formal specifications are obvious to those of us who struggle through pages of unfamiliar Z specifications in order to find errors with the syntax/scope and types. With *Formaliser* we can begin to see how the production of correct and consistent specifications can become possible. Syntactically incorrect expressions cannot be submitted to the specification because of the on-the-fly parser. Type inconsistencies can also be spotted by the on-board checking facilities. Therefore, contradictions in the specification can be traced. One important point to remember is that *Formaliser* cannot be held responsible for the completeness of the specification or ensure that it meets the user's requirements — these areas are still the responsibility of the specifier. Productivity of the software engineers who use *Formaliser* can be improved because constant checking of the Z by hand does not have to be carried out. The consistency and syntactic correctness of our specification has already been checked. As long as we have confidence in the tool then there is no need to repeat the checking process. With tools such as *Formaliser* and its LOTOS equivalents software engineers can have more confidence that their specifications do not contain contradictions and that conclusions drawn from the specifications can be trusted, thus yielding better results. For more information

the reader is referred to *Formaliser's* user guide [12], and a recent evaluation report [19].

The transition from specification to implementation can be made easier if the specification adopts a certain style or layout similar to the final source code. LOTOS has such a recognised style and programmers who use LOTOS specifications can easily recognise areas of the specification that will immediately translate into a programming language. Although the Z notation is not directly related to programming languages, it does have the power to express complex data structures used in implementation. The exact implementation of those structures does require some degree of knowledge as to the best way to interpret certain parts of the specification but informed programming choices can be made based on a knowledge of both the specification and implementation languages. The more complete the formal model of a system is then the better the implementation that will come from that model.

The different perspectives of a proposed system that are provided by a dual specification approach help us to provide a system that meets the requirements of the user better than partial-model specifications, simply because more of the system has been captured formally and cannot be improvised. Fewer gaps exist in the formal model and therefore ambiguity and assumption cannot weaken the structure of the specification. The combination of alternative formal approaches and checking tools provide us with a solid foundation from which to develop future systems.

## 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] Collins, Nicholls and Sorensen. (1988). *Introducing formal methods: the CICS experience with Z.* IBM United Kingdom Laboratories Limited, Hursley Park and Programming Research Group (PRG), Oxford University.

[4] H. Ehrig and B. Mahr. (1985). *Fundamentals of Algebraic Specification 1.* Springer-Verlag: Berlin.

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

[6] Fisher, A. S. (1991). *CASE: Using Software Development Tools.* New York: John Wiley & Sons Inc.

[7] Flynn, Hoverd and Brazier. (1990). *Formaliser — An Interactive Support Tool for Z.* Logica Cambridge Limited: Cambridge, U.K.

[8] Gravavel, H. and Sifakis, J. (1990). Compilation and verification of LOTOS specifications in: Logrippo, Probert R.L. and Ural, H. eds. *Protocol Specification, Testing and Verification.* X (Proceedings IFIP WG6.1 10th International Symposium, Ottawa, Ontario, Canada, 12—15 June 1990). North-Holland, Amsterdam. pp379—394.

[9] Henderson, P. (February, 1986). Functional Programming, Formal Specification and Rapid Prototyping. *IEEE Trans.* **SE—12.** (2). pp241—250

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

[11] 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.

[12] Logica Cambridge Limited. (April 1994). *Formaliser User Guide — (Z Specific Version for MS-Windows 3.1) Version 7.*1. Logica Cambridge Limited, U.K.

[13] 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.

[14] Logrippo, L, Obaid, A, Briand, J.P and Fehri, M.C. (1988). An Interpreter for LOTOS, a specification language for distributed systems. *Software Practice and Experience*. **18**. pp265—385.

[15] 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.

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

[17] Sprivey, J. M. (1987). *The Z Notation*. Exeter: Prentice-Hall.

[18] Stroustrup, B. (1991). *The C++ Programming Language*, 2nd. ed. Addison-Wesley, Reading: MA.

[19] Taylor, P.N. (April 1993). *The Evaluation of Formaliser as a CASE tool and the Development of Graphical Z*. University of Hertfordshire, Technical Note: UHCS-93-N2.

# Appendix A

## A.1 Basic Declarations and Redefinitions

The following basic declarations are used throughout the text of the specification. These types can be regarded as constants which only need be defined once at the top of the specification.

$$Boolean ::= True | False$$
$$minReading == 0$$
$$maxReading == 100$$
$$Reading == \{x : \mathbb{N} | minReading \leq x \leq maxReading\}$$
$$minLevel == 0$$
$$maxLevel == 9$$
$$Level == \{y : \mathbb{N} | minLevel \leq y \leq maxLevel\}$$

## A.2 Sprayer process state definitions

The *Sprayer* process state definition only stores the process' state which can be modified via the operations defined in the *SprayerProcess* schema.

$$\begin{array}{|l} \textit{Sprayer} \\ \hline sprayState : \mathbb{B} \\ \hline \end{array}$$

$$\begin{array}{|l} \Delta \textit{Sprayer} \\ \hline \textit{Sprayer} \\ \textit{Sprayer}' \\ \hline \end{array}$$

$$\begin{array}{|l} \Xi \textit{Sprayer} \\ \hline \Delta \textit{Sprayer} \\ \hline sprayState' = sprayState \\ \hline \end{array}$$

An initial state for the *Sprayer* is off (or False). This initialisation is only called once during the lifetime of the system.

*initSprayer*

$$Sprayer'$$

---

$$sprayState' = False$$

The behaviour of the *Sprayer* is defined in the following schema. The predicates for *Sprayer* state that neither of the inputs into the schema may hold the same Boolean value. If this were true then all inputs would occur at once. The first predicate stops multiple communications to a single process. Only one input can be dealt with at any one time. All processes in the GHCS carry the same predicate to prevent input overloading.

The remaining predicates state that, provided that the input value ($X?$) is true then the right-hand-side of the expression may be evaluated. The predicates in the schema *SprayerProcess* conform to the behaviour of the LOTOS equivalent specification in appendix B, but one main difference is that the same action results from a *true* evaluation of the left-hand-side of the *SetSpray...* prdicates. LOTOS had two distinct operations for this state modification. For example, LOTOS evaluates *SetSprayOn?* and *sprayOn?* as separate inputs and deals with them separately.

*SprayerProcess*

$$\Delta Sprayer$$
$$SetSprayOn? : \mathbb{B}$$
$$SetSprayOff? : \mathbb{B}$$
$$sprayOn? : \mathbb{B}$$
$$sprayOff? : \mathbb{B}$$

---

$$\neg(SetSprayOn? \Leftrightarrow SetSprayOff? \Leftrightarrow sprayOn? \Leftrightarrow sprayOff?) \land$$
$$(((SetSprayOn? \lor sprayOn?) \Rightarrow sprayState' = True) \lor$$
$$((SetSprayOff? \lor sprayOff?) \Rightarrow sprayState' = False))$$

## A.3 Hygrometer process state definitions

The *Hygrometer* process state is defined below, together with initialisation and behavioural definitions.

*Hygrometer*

$$minHumid : Reading$$
$$maxHumid : Reading$$

---

$$minHumid \leq maxHumid$$
$$maxHumid \geq minHumid$$

$\Delta Hygrometer$

$$Hygrometer$$
$$Hygrometer'$$

$\Xi Hygrometer$

$\Delta Hygrometer$

---

$minHumid' = minHumid$
$maxHumid' = maxHumid$

$initHygrometer$

$Hygrometer'$

---

$minHumid' = minReading$
$maxHumid' = maxReading$

The predicate part of *HygrometerProcess* follows the conventions introduced in section A.1 (*Sprayer*) where a guard is placed in the process to prevent multiple inputs evaluating to true and influencing the process' state and those connected to it (e.g: *Sprayer* and *Window Controller*). Notice the use of *HYopen!* and *HYclose!* to signify the origin of the messages aimed at the *Window Controller*.

$HygrometerProcess$

$\Delta Hygrometer$
$SprayerProcess$
$WindowControllerProcess$
$SetMinHumid? : \mathbb{B}$
$SetMaxHumid? : \mathbb{B}$
$ReadHumid? : \mathbb{B}$
$min? : Reading$
$max? : Reading$
$current? : Reading$
$sprayOn! : \mathbb{B}$
$sprayOff! : \mathbb{B}$
$HYopen! : \mathbb{B}$
$HYclose! : \mathbb{B}$

---

$\neg(SetMinHumid? \Leftrightarrow SetMaxHumid? \Leftrightarrow ReadHumid?) \land$
$(((SetMinHumid? \land min? \leq maxHumid) \Rightarrow minHumid' = min?) \lor$
$((SetMaxHumid? \land max? \geq minHumid) \Rightarrow maxHumid' = max?) \lor$
$((ReadHumid? \land current? < minHumid) \Rightarrow sprayOn! = True \land HYclose! = True) \lor$
$((ReadHumid? \land current? > maxHumid) \Rightarrow sprayOff! = True \land HYopen! = True)))$

## A.4 Thermometer process state definitions

$Thermometer$

$minTemp : Reading$
$maxTemp : Reading$

---

$minTemp \leq maxTemp$
$maxTemp \geq minTemp$

$\Delta Thermometer$

$Thermometer$
$Thermometer'$

$\Xi Thermometer$

$\Delta Thermometer$

---

$minTemp' = minTemp$
$maxTemp' = maxTemp$

$initThermometer$

$Thermometer'$

---

$minTemp' = minReading$
$maxTemp' = maxReading$

$ThermometerProcess$

$\Delta Thermometer$
$HeaterProcess$
$WindowControllerProcess$
$SetMinTemp? : \mathbb{B}$
$SetMaxTemp? : \mathbb{B}$
$ReadTemp? : \mathbb{B}$
$min? : Reading$
$max? : Reading$
$current? : Reading$
$inc! : \mathbb{B}$
$dec! : \mathbb{B}$
$THopen! : \mathbb{B}$
$THclose! : \mathbb{B}$

---

$\neg(SetMinTemp? \Leftrightarrow SetMaxTemp? \Leftrightarrow ReadTemp?) \land$
$(((SetMinTemp? \land min? \leq maxTemp) \Rightarrow minTemp' = min?) \lor$
$((SetMaxTemp? \land max? \geq minTemp) \Rightarrow maxTemp' = max?) \lor$
$((ReadTemp? \land current? < minTemp) \Rightarrow inc! = True \land THclose! = True) \lor$
$((ReadTemp? \land current? > maxTemp) \Rightarrow dec! = True \land THopen! = True)))$

## A.5 Heater process state definitions

$Heater$

$heatLevel : Level$

---

$minLevel \leq heatLevel \leq maxLevel$

$\Delta Heater$

$Heater$
$Heater'$

$\Xi Heater$

$\Delta Heater$

---

$heatLevel' = heatLevel$

$initHeater$

$Heater'$

---

$heatLevel' = minLevel$

$HeaterProcess$

$\Delta Heater$
$AlarmProcess$
$SetHeat? : \mathbb{B}$
$level? : Level$
$inc? : \mathbb{B}$
$dec? : \mathbb{B}$
$on! : \mathbb{B}$

---

$\neg(SetHeat? \Leftrightarrow inc? \Leftrightarrow dec?) \land$
$((SetHeat? \Rightarrow heatLevel' = level?) \lor$
$((inc? \land heatLevel < maxLevel) \Rightarrow heatLevel' = heatLevel + 1) \lor$
$((inc? \land heatLevel \geq maxLevel) \Rightarrow on! = True) \lor$
$((dec? \land heatLevel > minLevel) \Rightarrow heatLevel' = heatLevel - 1) \lor$
$((dec? \land heatLevel \leq minLevel) \Rightarrow on! = True))$

## A.6 WindowController process state definitions

$WindowController$

$windowLevel : Level$

---

$minLevel \leq windowLevel \leq maxLevel$

$\Delta WindowController$

$WindowController$
$WindowController'$

$\Xi WindowController$

$\Delta WindowController$

---

$windowLevel' = windowLevel$

*initWindowController*

$$WindowController'$$

---

$$windowLevel' = minLevel$$

The *Window Controller* process receives communications from two separate sources (as does the *Alarm*) and it must differentiate between those two sources. Either the *Hygrometer* or the *Thermometer* process can request an *open* or *close* operations from the *Window Controller*. To stop contradiction between two inputs (which can carry different messages) some individual identity is required, hence the *TH/HY* prefix on the *open* inputs. See the double implication predicate in the segment schema *STWin* in section A.6 to see how we can ensure that both inputs have different values. The LOTOS equivalent of this process' behaviour uses one common gate name to address the multiple process communication using *open?* and *close?*because LOTOS will buffer the inputs (as it is capable of modelling ordered events – unlike Z).

*WindowControllerProcess*

$$\Delta WindowController$$
$$AlarmProcess$$
$$SetWindow? : \mathbb{B}$$
$$level? : Level$$
$$HYopen? : \mathbb{B}$$
$$HYclose? : \mathbb{B}$$
$$THopen? : \mathbb{B}$$
$$THclose? : \mathbb{B}$$
$$on! : \mathbb{B}$$

---

$$\neg(SetWindow? \Leftrightarrow open? \Leftrightarrow close?) \land$$
$$((SetWindow? \Rightarrow windowLevel' = level?) \lor$$
$$(((HYopen? \lor THopen?) \land windowLevel < maxLevel) \Rightarrow$$
$$windowLevel' = windowLevel + 1) \lor$$
$$(((HYopen? \lor THopen?) \land windowLevel \geq maxLevel) \Rightarrow$$
$$on! = True) \lor$$
$$(((HYclose? \lor THclose?) \land windowLevel > minLevel) \Rightarrow$$
$$windowLevel' = windowLevel - 1) \lor$$
$$(((HYclose? \lor THclose?) \land windowLevel \leq minLevel) \Rightarrow$$
$$on! = True))$$

## A.7 Alarm process state definitions

*Alarm*

$$alarmState : \mathbb{B}$$

*ΔAlarm*

$$Alarm$$
$$Alarm'$$

$\Xi Alarm$

$\Delta Alarm$

---

$alarmState' = alarmState$

$initAlarm$

$Alarm'$

---

$alarmState' = off$

The *Alarm* process also receives one input from two sources along the same communications channel; namely *on?*. However, a conflict does not exist between *Alarm, Heater* and *Window Controller* because the same message is sent from either source process and not a potentially contradicting message. Regardless of who sends the Alarm a message the same message will get through so there is no need to impose an invariant on the input values from the same source to ensure that they are always different. Only separate inputs (i.e: *SetAlarmOff?* and *on?*) need to be restricted in such a manner – as is the case with all of the process schemas in the *GHCS* specification.

$AlarmProcess$

$\Delta Alarm$

$SetAlarmOff? : \mathbb{B}$

$on? : \mathbb{B}$

---

$\neg(SetAlarmOff? \Leftrightarrow on?) \land$

$((SetAlarmOff? \Rightarrow alarmState' = off) \lor$

$(on? \Rightarrow alarmState' = on))$

## A.8 Information hiding with segments

The GHCS can be organised into segments where each part holds two or more individual processes. In LOTOS we use the *hide* operator to restrict the observability of the hidden actions. In Z we can restrict the actions using set subtraction. Each segment restricts the same actions as those found in the LOTOS equivalent specification in appendix B. We use Z schema calculus to define the segments with their restricted elements.

$SpHy$

$SprayerProcess \setminus \{sprayOn?, sprayOff?\}$

$HygrometerProcess \setminus \{sprayOn!, sprayOff!\}$

$ThHe$

$ThermometerProcess \setminus \{inc!, dec!\}$

$HeaterProcess \setminus \{inc?, dec?\}$

The *STWin* segment is a special case as it has to restrict certain actions and impose conditions on the interaction of the *Hygrometer* and *Thermometer* processes using *open*

and *close* communications. Both of these links to the *Window Controller* must be mutually exclusive otherwise contradicting messages can arrive at *WindowControllerProcess* at the same time (unlike LOTOS which will buffer the messages). To stop this contradiction we use double implication ($\Leftrightarrow$) to enforce similar Boolean values for each input from separate processes and different inputs from complementary ports on the same process (i.e: HYopen! and HYclose!).

*STWin*

$$SpHy \setminus \{SpHyopen!, SpHyclose!\}$$
$$ThHe \setminus \{ThHeopen!, ThHeclose!\}$$
$$WindowControllerProcess \setminus \{open?, close?\}$$

---

$$(HYopen! \Leftrightarrow THopen!) \wedge$$
$$(HYclose! \Leftrightarrow THclose!) \wedge$$
$$\neg(HYopen! \Leftrightarrow HYclose!) \wedge$$
$$\neg(THopen! \Leftrightarrow THclose!)$$

Finally the complete system itself, the *GHCS*. Notice that we do not imposed any invariants on the *GHCS* schema, as we do in *STWin*, because no contradicting messages can be sent to the *Alarm* from its separate sources. Each connecting component sends the same message so we don't concern ourselves with message contamination.

*GHCS*

$$STWin \setminus \{on!\}$$
$$AlarmProcess \setminus \{on?\}$$

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

    library
        NaturalNumber, Boolean
    endlib

    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 (* StateType *)

    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: Nat → Level
            incLevel: Level → Level
            decLevel: Level → Level
            isMinLevel: Level → Boolean
            isMaxLevel: Level → Boolean
        eqns
            forall n:Nat (* 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 (* LevelType *)

    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 (* RecStateType *)
```

```
(* Set definitions for use throughout the specification to help keep the text minimal. *)

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

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

behaviour
    [(min gt max)] -> (exit)
  []
    [(min le max)] -> GHCS(spState,alState:State, hyState,thState:RecState, heState,wcState:Level);noexit
  where
    process GHCS[spGates,hyGates,thGates,heGates,wcGates,alGates](spState,alState:State, hyState,thState:RecState, heState,wcState:Level) : noexit :=
      hide on in
        STWin[spGates,hyGates,thGates,heGates,wcGates](spState:State, hyState,thState:RecState, heState,wcState:Level) |[on]| Alarm[alGates](alState)
      where


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


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


        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 *)
```

```
process Hygrometer[hyGates](HyState:RecState) : noexit :=
        SetMinHumid ? h:Reading;
                ([isLEmax(h,HyState)] → Hygrometer[hyGates](setMin(h,HyState))
                        []
                [isGTmax(h,HyState)] → Hygrometer[hyGates](HyState))
        []
        SetMaxHumid ? h:Reading;
                ([isGEmin(h,HyState)] → Hygrometer[hyGates](setMax(h,HyState))
                        []
                [isLTmin(h,HyState)] → Hygrometer[hyGates](HyState))
        []
        ReadHumid ? h:Reading;
                ([isLTmin(h,HyState)] → sprayOn;close;Hygrometer[hyGates](HyState)
                        []
                [isGTmax(h,HyState)] → sprayOff;open;Hygrometer[hyGates](HyState)
                        []
                [isGEmin(h,HyState) and isLEmax(h,HyState)] → Hygrometer[hyGates](HyState))
endproc (* Hygrometer *)


process Window[wcGates](cw:Level) : noexit :=
        SetWindow ? cw:Level;Window(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 *)


process Thermometer[thGates](ThState:RecState) : noexit :=
        SetMinTemp ? t:Reading;
                ([isLEmax(t,ThState)]→ Thermometer[thGates](setMin(t,ThState))
                        []
                [isGTmax(t,ThState)] → Thermometer[thGates](ThState))
        []
        SetMaxTemp ? t:Reading;
                ([isGEmin(t,ThState)] → Thermometer[thGates](setMax(t,ThState))
                        []
                [isLTmin(t,ThState)] → Thermometer[thGates](ThState))
        []
        ReadTemp ? t:Reading;
                ([isLTmin(t,ThState)] → inc;close;Thermometer[thGates](ThState)
                        []
                [isGTmax(t,ThState)] → dec;open;Thermometer[thGates](ThState)
                        []
                [isGEmin(t,ThState) and isLEmax(t,ThState)] → Thermometer[thGates](ThState))

endproc (* Thermometer *)
```

```
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 *)


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 *)

endproc (* GHCS *)

endspec (* behaviour-GHCS *)
```