DIVISION OF COMPUTER SCIENCE

Smoothing the transition from Formal Specification to Object-Oriented Implementation

(Introducing *ImpSpec* - A framework for modifying standard specifications to incorporate object-oriented design techniques)

P. N. Taylor
D. E. Smith

Technical Report No 213

January 1995

Smoothing the transition from Formal Specification to Object-Oriented Implementation

(Introducing *ImpSpec* - A framework for modifying standard specifications to incorporate object-oriented design techniques)

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

January 1995

Abstract

The advances in object-oriented design and programming languages have left many formal specification languages behind in terms of the facilities offered to the specifier. The very languages that were developed to formalise the design of systems, prior to implementation, no longer support the design techniques or programming languages that are now being used for that implementation.

In this paper we introduce *ImpSpec*, a series of refinements for the modification of existing specifications to produce comparable object-oriented specifications. Our discussions concentrate on the formal description technique LOTOS [5] and use object-oriented extensions to the language put forward by [3] and [9]. From a standard specification written in LOTOS we perform a series of design steps to transform the original specification into an object-oriented equivalent specification. We seek to justify the behaviour of the object-oriented specification against the original specification. We also consider the issues surrounding inheritance and the encapsulation of both data and communications. Using the *ImpSpec* framework we are able to produce an abstract model that is closer to the structure of an object-oriented programming language implementation, such as C++ [10] or Modula-3 [4].

We argue that it is possible to enhance non-object-oriented specifications to incorporate object-oriented design techniques, thus making the specifications more flexible and open to change.

1 Introduction

In the early 1980's the International Organisation for Standardization (ISO) decided to develop a formal description technique to help with the mathematical modelling of computer networks and open distributed systems [1,2,5,7]. The ideas behind object-oriented design and the programming languages to support such techniques post-date the initial development of LOTOS and many other concurrent formal description languages.

The way in which we design and specify systems has progressed, therefore the languages designed to support such design activities must also progress. In this paper we introduce a framework for modifying existing non-object-oriented LOTOS specifications to incorporate the ideas associated with object-oriented design. Our goal is to produce a formal specification that, using ideas put forward by [3] and [9], will be in a form that closely resembles implementations in current object-oriented programming languages, such as C++ [10] and Modula-3 [4].

We feel that, at present, specifications written without object-oriented features require many assumptions to be made so that they can form the basis of an object-oriented design and implementation. These assumptions can lead to problems concerning the verification of the subsequent implementation against its specification. Consequently, if we write a specification that models techniques available in implementation languages then our resulting implementation will more accurately model the specifications.

In this paper we modify a LOTOS specification of a simple case study system using a series of steps, known as the *ImpSpec* framework. The resulting specification will contain many of the features that are common to object-oriented design. We do not consider, in detail, the actual implementation of the final object-oriented specification as this is beyond the scope of this paper.

Primarily, we concentrate on the abstract model of the system, leaving aside implementation specific details. Although we do make certain assumptions concerning the use of data and class-method encapsulation we do not explicitly state that some implementation language specific constructs should be used. Our objectives are to introduce the reader to a series of explicitly defined steps whereby the final product is proved to be a valid implementation of the object-oriented LOTOS specification.

2 The *ImpSpec* framework (Implementation of the Specification)

We introduce the *ImpSpec* framework as a way of ensuring that an implementation conforms to its specification. The standardised formal language used throughout this paper is that of LOTOS (DIS8807) which does not have an object-oriented extension, although modifications to the language have been proposed [3,9]. We do not want to concern ourselves with whether we could change the standard syntax but whether we should change it! A discussion of the necessary modification to LOTOS to support object-oriented design can be found in [9].

We have already identified that current formal languages are slow to react to the features offered by object-oriented techniques and are not providing the facilities to model formally the systems that we wish to implement using these new techniques. We hope that the *ImpSpec* framework will help to redress the imbalance between both object-oriented formal specification, design and implementation.

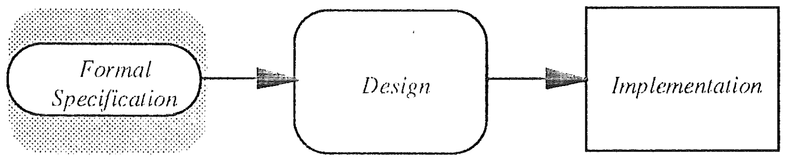


Figure 2.1

In figure 2.1 the imbalance between formal specification, design and implementation is shown as the shaded area surrounding the specification, which represents the requirements of the design and implementation that are not supplied by the specification.

We seek to reduce the number of assumptions made between specification, design and implementation by providing a refinement framework for modifying the original specification which contains no object-oriented features, as shown in the following diagram:

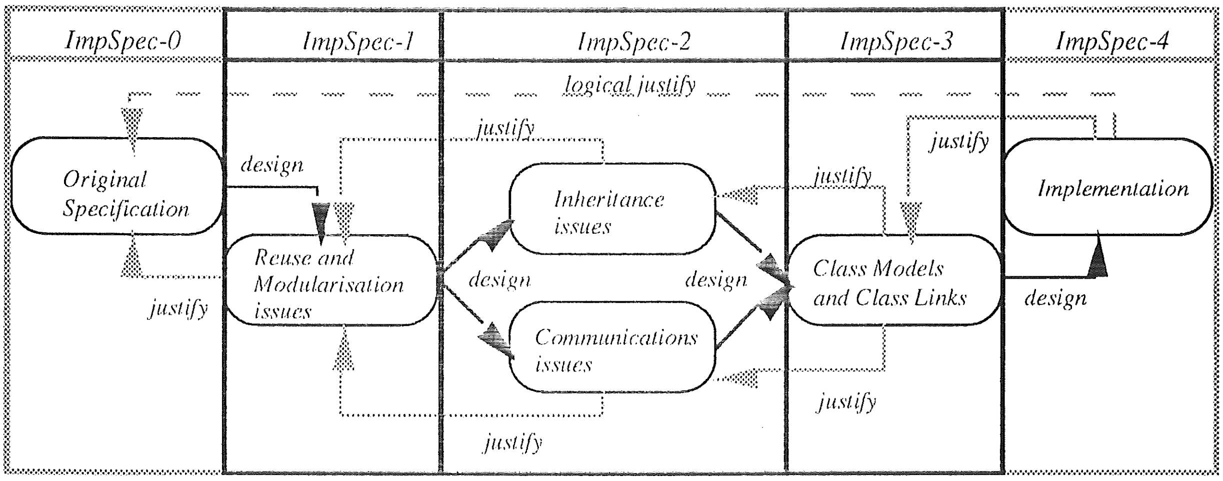


Figure 2.2

The paper concentrates on stages 1, 2 and 3, encapsulated by the bold line in figure 2.2. Each stage of ImpSpec is detailed below:

ImpSpec-0: The original formal specification of a system, prior to modifications.

ImpSpec-1: A first step to identify areas of possible reuse and modularisation within the original specification

ImpSpec-2: The introduction of object-oriented constructs into the post-ImpSpec-1 specification. Issues concerning the inheritance of data and methods, together with information hiding and data/method access restriction are dealt with during this stage.

ImpSpec-3: The combination of the data and communications design defined in ImpSpec-2, into sub-systems to form the complete system. Access restrictions for each sub-system are defined at this stage.

ImpSpec-4: The actual implementation of the object-oriented specification from ImpSpec-3. Considered to be outside the scope of this paper. The ImpSpec framework is primarily concerned with stages 1-3.

We refer to the original specification as falling within the bounds of *ImpSpec-0* (prior to entering our refinement framework in steps 1 to 3).

The first step towards a full object-oriented specification is to identify areas of the original specification that offer behavioural reuse. The identification of reuse enhances the flexibility of the specification.

We incorporate the work carried out in [3] and [9] to map the specification onto a more recognised object-oriented structure; this mapping is done as part of *ImpSpec-2*. Two sections exist as part of *ImpSpec-2*; inheritance issues concerning commonly accessed data and functions and communications issues concerned with supplying controlled access to those data and functions.

The final refinement stage, *ImpSpec-3*, brings together the inherited child-processes and the hidden communications. At the end of this stage we consider the specification to be in its final object-oriented form. Each child-process can be linked together to form a larger sub-system which can eventually become the complete system. Our aim is to provide a specification that is layered so that sub-systems are completely contained within the system, therefore hiding internal communications from the rest of the system and the environment.
Using notions of equivalence we construct formal proofs to show that the object-oriented version of the specification is a valid implementation of its immediate predecessor. Consequently, the final specification can be shown to be transitively linked to the original specification in *ImpSpec-0*. Refer to the "*logical justification link*" in figure 2.2.

For the purposes of this paper we assume that the standard LOTOS specification for the GHCS case study system already exists (appendix A). The result of applying *ImpSpec-1* to appendix A can be seen in appendix B and the final object-oriented specification of the GHCS (by applying *ImpSpec-2*) can be seen in appendix C.

3 Informal Specification of the Case Study System

Our case study system contains six processes (some of which exhibit common behaviour) all executing in parallel. Each process can be informally specified as follows:

***Sprayer* (*Sp*)**: It can accept *on* or *off* communications from either the environment or the *Hy* process. The process *Sp* will timeout and reset to *off* after some nondeterministic period of time if it remains *on* for too long.

***Hygrometer* (*Hy*)**: The *Hy* process accepts communications from the environment. It then evaluates the value of those communications to determine what actions it should take in terms of telling the *Sp* and *WC* processes to perform some action. Process *Hy* controls the behaviour of *Sp* and *WC*, internal to the system. A minimum and maximum range are associated with *Hy* and these values are used to determine the behaviour of *Hy*.

***Window Controller* (*WC*)**: The *WC* process accepts communications from either *Hy* or *He* and also the environment. These communications tell *WC* whether or not to perform an *open* action or a *close* action. A minimum and maximum range is also associated with *WC*. If an attempt is made to set *WC* beyond these bounds then a communication to *Al* (the Alarm) is made, warning the environment of a problem.

***Thermometer* (*Th*)**: Process *Th* accepts communications from the environment (the temperature reading). Similar in operation to *Hy*, the reading is used to determine what actions the *WC* and *He* processes should take. A minimum and maximum value is used to determine the behaviour of the *Th* process.

***Heater* (*He*)**: Communications destined for *He* originate from the *Th* process. Like process *WC* it also has a preset range which it cannot be set beyond. Attempts to adjust *He* beyond its preset limits will result in a signal being sent to *Al*, warning the environment of a problem.

***Alarm* (*Al*)**: The *Al* process accepts communications from the *WC* and *He* processes. Upon receipt of a communication the alarm will be set to *on*. The environment can reset Al to *off* or, like process *Sp*, it will eventually timeout and turn *off*. However, unlike *Sp*, the *Al* process cannot be activated from the environment, only via some internal communication.

The communication between the six processes can be seen in the following diagram:

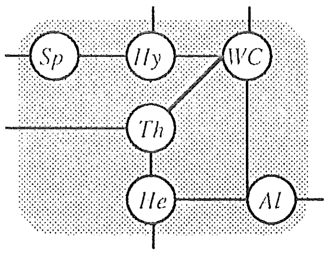


Figure 3.1

Figure 3.1 omits the exact details of the communications in the GHCS as it only shows how the components connect to each other and the environment. It does not show the nature of those connections. Internal communications are shown inside the shaded area and cannot be influenced by the external environment.

4 *ImpSpec-1*: Reuse and Modularity within the Specification

In the GHCS we can identify processes which contain duplicate behaviour. Consequently, generic processes can be derived to substitute those duplicate processes by performing the following steps:

i) identify the common behaviour between two processes. ($\alpha.A \cap \alpha.B$), where $\alpha.X$ is the set of actions of X).

ii) extract the common behaviour, assigning it to a generic process. ($\alpha.C = \alpha.A \cap \alpha.B$).

and

iii) use process and action renaming to construct the specific processes required to build the system. ($D \stackrel{def}{=} C[x/y]$, where x/y is action renaming on the actions of C).

Now, using generic process substitution, we can supply all of the behaviour necessary for the GHCS using just three processes. The following equivalences are described using the language of CCS [8] using a similar convention introduced in [1].

Sp $\stackrel{def}{=}$ *SPAL[on/setSprayOn,off/setSprayOff,setOn/SetSprayOn,setOff/SetSprayOff]*
Al $\stackrel{def}{=}$ *SPAL[on/setAlOn,setOff/SetAlOff] \ {off,setOn}*
Hy $\stackrel{def}{=}$ *HYTH[SetMin/SetMinHumid,SetMax/SetMaxHumid,ReadInput/ReadHumidity,*
on/setSprayOn,off/setSprayOff,open,close]
Th $\stackrel{def}{=}$ *HYTH[SetMin/SetMinTemp,SetMax/SetMaxTemp,ReadInput/ReadTemp,on/inc,*
off/dec,open,close]
He $\stackrel{def}{=}$ *HEWI[SetLevel/SetHeatLevel,open/inc,close/dec,on]*
WC $\stackrel{def}{=}$ *HEWI[SetLevel/SetWindow,open,close,on]*

Although action renaming is not part of standard LOTOS we can use it as a meta-language for talking about LOTOS. The syntax of renaming, as described in [1], can be formally defined as follows:

$$\boxed{\begin{array}{l} B \xrightarrow{g'} B',\ \phi=[g_1/g_1',\ldots,g_n/g_n'],\ \textit{and}\ g/g' \in \phi \\ \textit{implies} \\ B\phi \xrightarrow{g} B\phi' \\ \\ B \xrightarrow{\mu^+} B'\ \textit{and}\ \mu^+ \notin \{g_1',\ldots,g_n'\} \\ \textit{implies} \\ B\phi \xrightarrow{\mu^+} B\phi' \end{array}}$$

Figure 4.1

Process B performs the action g' and evolves to B'. Action gn can therefore be renamed to gn'. With the aid of action renaming we can amend the *Sp* process definition to read:

Sp $\underline{\underline{\textit{def}}}$ *SPAL[on/setSprayOn,off/setSprayOff,setOn/SetSprayOn,setOff/SetSprayOff]*
becomes
Sp[setSprayOn,setSprayOff,SetSprayOn,SetSprayOff]
after gate renaming.

Each of the original six GHCS processes uses gate renaming as a way of specialising those actions offered by the three generic processes. Process *Al* requires further action because it does not use all of the features offered by the generic *SPAL* process. We remove the actions in *Al* that are no longer required using a CCS-style action restriction [8] (shown below as underlined):

Al $\underline{\underline{\textit{def}}}$ *SPAL[on/setAlOn,setOff/SetAlOff]*$\underline{\setminus \{\textit{off},\textit{setOn}\}}$

So far we have identified three separate types of process behaviour, represented by *SPAL, HYTH* and *HEWI*. These are our candidates for generic processes (albeit with some amount of redundancy in the case of *Al*). The use of gate renaming allows us to specialise each process. For example, *Sp* and *Al* are specialisations of the generic process *SPAL*.

The use of generic processes enables us to simplify the specification. However, we still need to show that the behaviour of the generic three process specification conforms to the specific six process specification. We conduct a series of formal proofs upon each generic process to show that it offers the same behaviour as the two processes it replaces. The mechanics of the proof itself is beyond the scope of this paper and is the subject of our current work.

5 *ImpSpec-2*:

ImpSpec-2.1 Inheritance and Data Encapsulation

An object-oriented approach to design and implementation enable us to enforce the encapsulation of class data and functions. Access to the class can be restricted by such constructs as C++ private inheritance and explicitly defined friendship between classes. Ideally, we consider private data to be the preferred form of data in a system. Access to this data needs to be strictly controlled by means of specific methods provided for the task. We are also concerned with hiding communications between processes so that we can restrict the influence that processes and the environment have over parts of the system. *ImpSpec-2* covers both of these issues and defines refinement steps for dealing with them.

Process relabelling was used as an initial solution to provide process reuse but it has its limitations. Primarily, because it does allow us to extend the behaviour of a process. Our aim is to be able to model inheritance (in the object-oriented sense), where an inherited class (a child) *can* extend the behaviour of the its base-class (parent-class). In [9] an inheritance extension to LOTOS, as defined by ODP [6], is formally defined. We use this definition as the basis for inheritance in LOTOS.

We consider the previous section's three generic processes as base-classes (parents). Each parent can then be used as a template for a child-class (giving the child its basic form of behaviour before possible extension). Consider the following example:

From ImpSpec-1 (appendix B):

$Sp \underline{\underline{def}}$ *SPAL[on/setSprayOn,off/setSprayOff,setOn/SetSprayOn,setOff/SetSprayOff]*

becomes:

From ImpSpec-3 (appendix C):

```
process Sp[SetSprayOn,SetSprayOff,setSprayOn,setSprayOff](s:State) : self(State) noexit :=
        SPAL[setSprayOn/setOn,SetSprayOff/SetOff](s)
        []
        [not isOn(s)] → SetSprayOn;self(on)
        []
        [isOn(s)] → setSprayOff;self(off)
endproc (* Sp *)
```

Notice that the behaviour offered by *SPAL* is included inside *Sp* as a choice of possible actions. The extension to the basic behaviour of *SPAL* appears as the two remaining choices inside *Sp*; namely *SetSprayOn* and *setSprayOff*.

We make use of the *self* primitive [9] which deals with the redirection of system control. For example, process *Sp* may execute action *setSprayOn*. This action is specified in *Sp's* parent-process, *SPAL*. Therefore, control is passed to *SPAL* to execute the *setSprayOn* action. The *self* primitive is used so that the thread of control that was passed to *SPAL* is returned back to *Sp* and does not remain with *SPAL* when its actions have been executed. If *SPAL*, *Sp* and *Al* were not defined using *self* then control of the system would remain with the last process called. The system would not be able to return control back up the inheritance chain to the original calling point. The following diagram helps to illustrate the need for *self*.

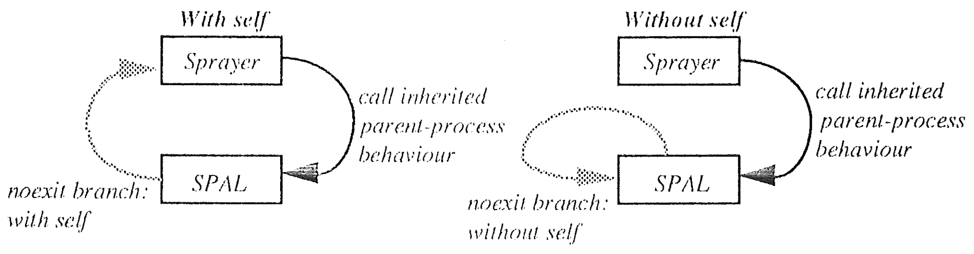

With self
Sprayer
call inherited parent-process behaviour
SPAL
noexit branch: with self
Without self
Sprayer
call inherited parent-process behaviour
SPAL
noexit branch: without self

Figure 5.1

The arrows in figure 5.1 represent the thread of control thread caused by choosing actions supplied by *SPAL*. It is clear from the diagram that control is returned to the caller if a process is defined in terms of *self* and not returned otherwise. *ImpSpec-2* states that each parent and child-process should be defined using *self* so that the thread of control is not trapped in the parent-process.

All of the processes in our system are self-referencing, therefore they all resume execution after they have performed their defined actions. By using *self* we can stop them from recursively calling themselves directly if they were, in fact, called from another process. For processes that do *exit* and therefore cease to be part of the system the *self* primitive rule still holds, allowing the called process to *exit* from the system and cease execution. A process defined in terms of *noexit* surrenders control back to the process that called it.

Aside from the inherited behaviour we must consider inherited data. If processes share common behaviour then we assume that they also share common data. We can determine the nature of this data by identifying the shared actions of the child-processes. Our intention is not to enforce any specific structure on the data within the system. We do not want to say *how* to implement the system, only *what* to implement. For these reasons we only imply the nature of the data by specifying what is to be done with that data.

The actions of process *Hy* tell us about its data requirements, such as the need for static minimum and maximum humidity values (of a type of our choosing). Also a value associated with a humidity reading that changes each time the process receives a reading. Notice that no information about the data types are given here, only the *need* for the data. Having decided upon the need for the data we then decide where that data should reside; in the parent or the child process.

ImpSpec-2.2 Communications Control

Only class methods and friends of a class can access data in the class. The idea of controlling communications between processes (and therefore classes) is integral to the design of a well structured object-oriented system.

In LOTOS we can restrict access to the communications between two processes by using the ***hide*** operator. For example, if processes *Sp* and *Hy* synchronise on the actions *setSprayOn* and *setSprayOff* we can choose to hide these communications from their environment, thus:

From ImpSpec-3 (appendix C):

```
process SpHy[SpGates∪HyGates](s:State, h:RecState) : noexit :=
        hide setSprayOn,setSprayOff in
                *Sp[SpGates](s)
                        |[setSprayOn,setSprayOff]|
                *Hy[HyGates](h)
endproc
```

The asterisk (*) denotes the redirection operator, introduced in [9], which states that once the system chooses one **p* over **q* then the behaviour of the system will remain with **p* and not revert back to the calling process; i.e: *SpHy*. The redirection operator (*) has the effect of stopping *self* from returning control back to a level prior to those processes defined in terms of *.

In C++ we can implement hidden communications by defining functions as *private* or *protected* in a class definition. Friends of that class will then have access to those functions. Interpreting LOTOS in an object-oriented programming language is discussed in section 8.

The system is only accessible via external communications. These will resemble the public methods (functions) defined in each of the classes that make up the object-oriented system.

The hiding of internal process communications plays a crucial part in the definition of the classes implemented during *ImpSpec-4*. The formal proof at the end of this stage should confirm that the specification at the end of *ImpSpec-2* conforms to the specification produced as a result of *ImpSpec-1*, hence the transitivity of the proofs back to the original specification of the system in *ImpSpec-0*.

6 *ImpSpec-3*: Class Models and Links Between Classes

To build each object-oriented process (our class structures) we need to combine the inheritance model of *ImpSpec-2.1* and the communications model of *ImpSpec-2.2* to form a template for each class. The implementation of the design will refer to the results of this stage. We shall use the formal inheritance model to build each class in the system and the communications model to define which classes have access to restricted data and methods.

The GHCS generic process definitions will form the backbone of the complete object-oriented specification. We combine each specific child-class together based upon the internal communications defined as part of the original specification. The hidden communications that take place between the sub-systems are our main interest. The communications hierarchy of the GHCS is shown in figure 6.1, where each of the four sub-systems is shaded independently.

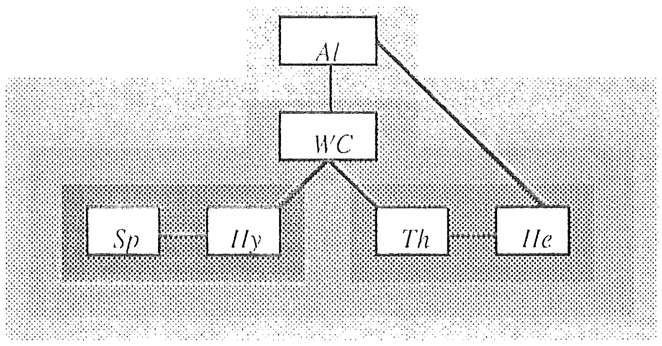


Figure 6.1

The modular structure of figure 6.1 is used as the basis of the implementation (described in section 8). The six processes in the system will not be directly accessible from the environment, only via special ports that are not hidden at each sub-system level. All unrestricted ports on each process will propagate up to the externally visible part of the system.

7 Process Equivalence during *ImpSpec* refinement.

The transformation of the original LOTOS specification into an object-oriented equivalent require us to formal proofs including *observational equivalence*, *testing equivalence* and *failure equivalence*. At the initial refinement stage (*ImpSpec-1*), where we initially identify reusable behaviour across processes, we can use observational equivalence (as defined in [8]) to prove that a new generic process conforms to the observable behaviour of the original specific processes that it is derived from. The following diagram illustrates the comparison between processes *Sprayer* and *SPAL*.

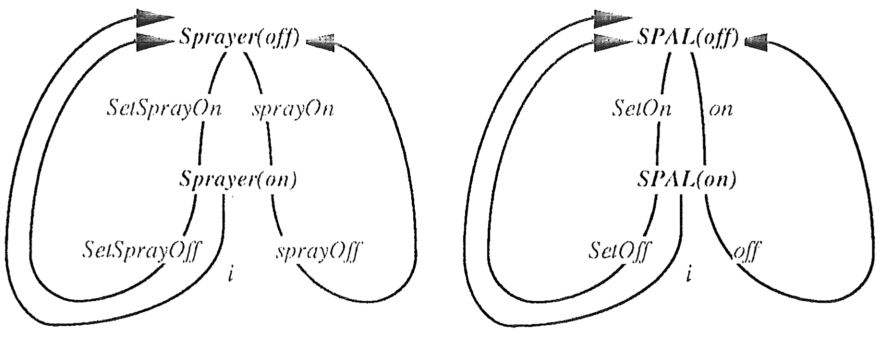


Figure 7.1

The identical transition diagrams reveal that both processes are indeed observational equivalent, proving that *SPAL* is a valid implementation of *Sp*. At each branch of process *Sprayer* we test to see if *SPAL* can also perform similar actions. Process equivalence allows us to substitute one process for another without altering the behaviour of the overall system. A bisimulation equivalence between *Sprayer* and *SPAL* would take the following form:

Possible Actions:

Sprayer	SPAL
$Sprayer(off) \to Sprayer(on)$	$SPAL(off) \to SPAL(on)$
$Sprayer(on) \to Sprayer(off)$	$SPAL(on) \to SPAL(off)$

A bisimulation B is a set of pairs of P which is closed under the following rules:

i) if $p \xrightarrow{\tau} p'$ *then* $q \xrightarrow{\tau^*} q'$ *and* $(p',q') \in B$, *and also*

if $p \xrightarrow{\alpha} p'$ *then* $q \xrightarrow{\tau^* \alpha \tau^*} q'$ *and* $(p',q') \in B$

and conversely:

ii) if $q \xrightarrow{\tau} q'$ *then* $p \xrightarrow{\tau^*} p'$ *and* $(p',q') \in B$, *and also*

if $q \xrightarrow{\alpha} q'$ *then* $p \xrightarrow{\tau^* \alpha \tau^*} p'$ *and* $(p',q') \in B$

The two processes p and q are said to be observational equivalent if there exists a bisimulation B such that $(p,q) \in B$. So, for the test of bisimulation equivalence we begin with the basic set B, presently defined as incomplete and written as:

$B = \{(Sprayer(off), SPAL(off)),....$

$$(Step\ 1.1)\ Sprayer(off) \xrightarrow{SetSprayOn} Sprayer(on)$$

where an equivalent action by *SPAL* is sought to allow *SPAL* to progress to some other state (*SPAL´*).

$$(Step\ 1.2)\ SPAL(off) \xrightarrow{SetOn} S,$$

where *(Sprayer(on), S)* $\in$ *B* is proposed as true and state $S \equiv SPAL(on)$ $\therefore$
(Sprayer(on),SPAL(on)) $\notin$ *B*, $\therefore$

$$B' = \{(Sprayer(off),SPAL(off)),(Sprayer(on),SPAL(on)), ...$$

$$(Step\ 2)\ Sprayer(off) \xrightarrow{sprayOff} Sprayer(on)\ and\ SPAL(off) \xrightarrow{off} S,$$

where *(Sprayer(on),S)* $\in$ *B* is proposed to be true and state $S \equiv SPAL(on)$ $\therefore$
(Sprayer(on),SPAL(on)) $\in$ *B*, $\therefore$
B does not change because the element *(Sprayer(on),SPAL(on))* already exists in the set B.

$$B' = \{(Sprayer(off),SPAL(off)),(Sprayer(on),SPAL(on)), ...$$

$$(Step\ 3)\ Sprayer(on) \xrightarrow{SetSprayOn} Sprayer(off)\ and\ SPAL(on) \xrightarrow{SetOn} S,$$

where *(Sprayer(off),S)* $\in$ *B* is proposed to be true and state $S \equiv SPAL(off)$ $\therefore$
(Sprayer(off),SPAL(off)) $\in$ *B*, $\therefore$
B does not change because the element *(Sprayer(off),SPAL(off))* already exists in the set *B*.

$$B' = \{(Sprayer(off),SPAL(off)),(Sprayer(on),SPAL(on)), ...$$

$$(Step\ 4)\ Sprayer(on) \xrightarrow{sprayOn} Sprayer(off)\ and\ SPAL(on) \xrightarrow{on} S,$$

where *(Sprayer(off),S)* $\in$ *B* is proposed to be true and state $S \equiv SPAL(off)$ $\therefore$
(Sprayer(off),SPAL(off)) $\in$ *B*, $\therefore$
B does not change because the element *(Sprayer(off),SPAL(off))* already exists in the set *B*.

All actions have now been performed without contradiction or failure so we close the set *B* and conclude that our feelings were right and that *SPAL* is indeed a valid implementation of *Sprayer*. The two processes are consequently *bisimulation equivalent*.

$$B' = \{(Sprayer(off),SPAL(off)),(Sprayer(on),SPAL(on))\}$$

There is no need to choose to perform the *SPAL* actions first in these tests as they are symmetrical. Only actions not performed by *Sprayer* but performed by *SPAL* would yield an expression where the *SPAL* transition was written first in the previous steps.

We carry out bisimulation tests on the remaining processes in *ImpSpec-0* and their generic equivalents in *ImpSpec-1*.

In systems which contain generic processes with radically different behaviour, over and above that of the original processes, more equivalence testing is required. Conformance testing [9], which deals with proving inheritance between processes will be used to prove equivalence between *ImpSpec-2* and *ImpSpec-3* specification process definitions.

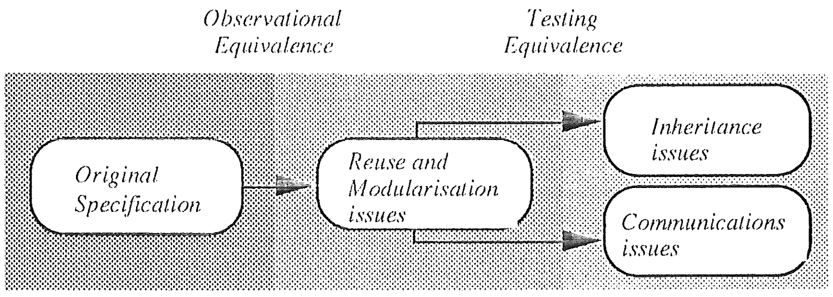


Figure 7.2

The boundaries that lie between the different LOTOS specifications is shown in figure 7.2 and highlights where the different notions of equivalence can be applied.

8 *ImpSpec-4*: Implementation

ImpSpec-4.1 Base-Class Definitions and Private Inheritance

We can construct a working model of the system by literally taking elements from the specification and interpreting them in our chosen object-oriented programming language (appendix D.2). We chose C++ as our implementation language because of its widespread use and general acceptance as a full-bodied object-oriented programming language [10]. Certainly, C++ provides all of the features that we require to faithfully model the GHCS using the object-oriented techniques incorporated thus far into the final version of the specification.

Object-oriented languages can be categorised using the work of Wegner [11] regarding the concepts of *object*, *class* and *inheritance*. Wegner defines *Object-based* languages as those supporting *objects*, *Class-based* languages as those supporting *objects* and *classes* and finally *Object-oriented* languages as those that support *objects*, *classes* and *inheritance*.

Using C++ notation process *SPAL* will be defined as a base-class, whereas processes *Sp* and *Al* will be defined as sub-classes, inheriting from *SPAL*. The following code segment illustrates part of their implementation:

```
class SPAL
{
        private:
                char        processName[20];
                State       processState;

        // use protected so that functions can be derived
        protected:
                SPAL(char *name, State initState);
                void setOn(void);
                void SetOff(void);
                Boolean isOn(void);
                void SetProcessState(State state);
                void PrintSPAL(void);
}; // end class

class Sp : private SPAL
{
        friend void Hy::setSprayOn(void);
        friend void Hy::setSprayOff(void);

        private:
                // private class functions
                void setSprayOn(void);
                void setSprayOff(void);

        public:
                Sp(char *name, State initState);
                void SetSprayOn(void);
                void SetSprayOff(void);
                void PrintSp(void);
}; //end class
```

The data area of both *Sp* and *Al* is actually defined in the private area of their parent-class (*SPAL*). Private inheritance protects this data, restricting access to it so that only direct descendants of *SPAL* can access the private area. Note that friends of *Sp* cannot access the private area of *Sp* (thanks to private inheritance).

ImpSpec-4.2 Communications Control using C++ Function Friendship

In C++, normal class friendship will give the friend of a class access to the private and protected areas of that class. This gives the friend class access to any methods declared in the class offering that friendship.

Recall that we are concerned with implementing the idea of hidden (restricted) communications between processes. In C++, this restriction takes the form of controlling the access to the methods of a class by specifying either specific classes as friends of a class (class friendship), or particular functions (function friendship).

For example class *Sp* declares class *Hy* to be its friend. Notice that friendship is *given*, not requested. Any method in class *Hy* can reference any method or data in class *Sp* directly, as if it were itself part of class *Sp*. This type of friendship is weak and open to abuse. It allows access to an area of a class which we consider strictly private; the class's own private area. To strengthen the weakness between class friends we prefer function friendship, thus reducing the availability of class functions to a subset of those functions present in the friend class.

The C++ language defines class function friendship as a more controlled way of restricting access to class methods. Having such control over methods simplifies the tracing of errors found during execution because we can track the access path to the methods, from local class methods to friend methods. In LOTOS we can specify this tight bonding between classes (LOTOS processes) using the following process composition:

Sp[SpGates](s) |[setSprayOn,setSprayOff]| Hy[HyGates](h)

The synchronisation between processes *Sp* and *Hy* takes place between specified communication ports *[setSprayOn,setSprayOff]*. The C++ implementation of this function friendship takes the form:

```
class Sp : private SPAL
{
        friend void Hy::setSprayOn(void);
        friend void Hy::setSprayOff(void);
...

class Hy : private HYTH
{
...
        private:
                ...
                void setSprayOn(void);
                void setSprayOff(void);

   ...
```

We can show graphically the links between the different classes and the areas accessible to them according to their defined friendship:

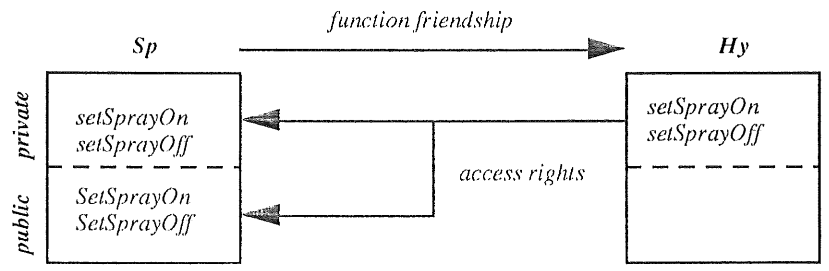


Figure 8.1

The private area of processes *Sp* and *Hy* appears above the dashed line and represents the area of each class that is accessible to its selected friends. An analogy of this friendship is akin to letting a friend use your ATM bank card and telling them the PIN for the card as well. The public area of a class is accessible to every other class by default.

To keep *Sp* and *Hy's* private area hidden from outsiders we need to privately inherit from *SPAL*. Private inheritance stops external classes accessing the private area of the parent-class even if the external classes are defined as friends of the inherited class. Only the methods of the inherited class can access the parent-class's private area.

ImpSpec-4.3 Further Communications Control using Static Class-Pointers

Each of the subsystems in the implementation of the GHCS are required to offer access to specific functions located in similar child-classes. Pointers to other classes need to be stored in child-classes to implement the hidden communications between classes. These pointers are stored in the private data areas of each child-class. Consider the following code extract where only the parent-class methods can change the private data values, hence control to the class data is absolute.

```
class Hy : private HYTH
{
        private:
                Sp      *spPtr;
                WC      *wcPtr;
                ...
}; //end class
...
```

```
void Hy::SetLinks(Sp *sp, WC *wc)
{
        spPtr = sp;
        wcPtr = wc;
}

void Hy::setSprayOn(void)
{
        spPtr->setSprayOn();
}
...
```

Pointer variables are stored within each child-class to enable them to connect to other instances of other classes. For example, an instance of *Hy* will call an instance of *Sp*, referred via the pointer to *Sp* which is stored in *Hy*. The expression *spPtr->setSprayOn()*, in the class method *Hy::setSprayOn()* illustrates this hidden communication. For each communication we require a pointer for each class-type. In *Hy* we define two links, one to an instance of *Sp* and one to an instance of *WC*.

Figure 8.2 shows an interpretation of the C++ code which includes the inheritance chain, function friendship and pointers to specific classes. The full diagram can be seen in appendix D.1.

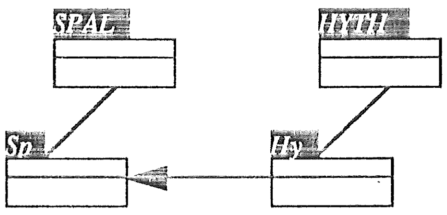


Figure 8.2

Parent-classes appear at the top of the diagram. Child-classes are shown below their parents. The arrow (pointing right to left) defines both friendship between two child-classes and the pointer variable linking an instance of *Hy* to an instance of *Sp*.

Upon execution of the GHCS instances of all of the child-classes are created. Links between the various sub-systems are then established. The friendship is defined in the code itself but the links can only be established after each object has been created and has a specific memory address. Control of the hidden communications between the objects is maintained using the two-tier system of defining function friendship and class pointers within child-classes. The function friendship mechanism defines *what* type of class can have access to class methods. The pointer variables define exactly *who* can access the class methods in a child-class.

We do not allow parent-classes to define their friends or point to other classes. This restriction of parents increases their so that they are not tied to specific forms of friendship or access by other classes.

9 Conclusions

In this paper we have introduced a framework for converting non-object-oriented specifications into object-oriented specifications; the framework of *ImpSpec*. We have shown how to refine the specification so that each evolution of the specification can be shown to be a valid implementation of its predecessor. Notions of equivalence are used as the basis for this justification. We assume that the implementation language will be capable of enforcing tight control over the access to class methods and data; such as C++ [10] and Modula-3 [4].

The identification of reusable behaviour within the specification is the first step towards an object-oriented design. If reuse can be established within the original system then we have a basis for constructing base-classes for use later in the framework.

Decisions need to be made to house the data and communications in certain processes. We interpret processes in LOTOS as classes in C++. The location of the data can be determined by the access that we want to provide to that data. In LOTOS we can define the access to the data using hidden communications and process composition. We localise the communications that will access the data. These communications (in the form of process actions) will form the basis of the class methods that are only accessible explicitly.

We use the language of C++ to capture the behaviour of the processes and the hidden communications between them. Pointers to specific instances of classes are used to implement hidden communications. The syntax of C++ provides us with a final implementation that is closely related to the object-oriented LOTOS specification by means of clearly identifiable classes and hidden communications.

We recognised the need for more structured specification techniques to aid the process of modifying standard systems to incorporate object-oriented design. Much of the flexibility supplied by highly modular object-oriented systems is missing from initial formal specifications. The specification refinement steps presented here can be checked formally against the existing system to prove that it offers (at the very least) the behaviour of the original specification. The benefits of the new object-oriented specification include a more flexible and maintainable specification that can be modified easily to incorporate new features without disturbing the formalism already in place.

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] Cusack, E., Rudkin, S. and Smith, C. (1989). *An Object-Oriented Interpretation of LOTOS*. The 2nd International Conference on Formal Description Techniques (FORTE89).December 1989. pp211—226.

[4] Harbison, S. (1992). *Modula-3*. Englewood Cliffs: 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] ISO/IEC. *Information Retrieval, Transfer and Management for OSI* - Working Document - Architecture Semantics, Formalisms and Specification Techniques. ISO/IEC JTC1/SC21 N6082, June 1991.

[7] 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.

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

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

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

[11] Wegner, P. (September 1987). *Dimensions of object-based language design*. Object Oriented Programming, Systems, Languages and Applications 87. Orlando, Florida.

Appendix A: *ImpSpec-0*: Original LOTOS Specification

```
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 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:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level,alState:State);noexit
        where

            process GHCS[spGates∪hyGates∪thGates∪heGates∪wcGates∪alGates\{sprayOn,sprayOff,open,close,inc,dec,on}](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\{sprayOn,sprayOff,open,close}](spState:State,hyState:Humid) : noexit :=
                        hide sprayOn,sprayOff in
                            Sprayer[spGates](spState) |[sprayOn,sprayOff]| Hygrometer[hyGates](hyState)
                    endproc


                    process ThHe[thGates∪heGates\{inc,dec,open,close}](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\{sprayOn,sprayOff,open,close,inc,dec}](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 *)
```

Appendix B: *ImpSpec-1*: Reuse and Relabelling LOTOS GHCS Specification

```
specification GreenHouse_Control_System(...)
        type
                .......... see original ImpSpec-0 GHCS specification for full description of types and operations.
        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}

        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\{sprayOn,sprayOff,open,close,Inc,Dec,on}](spState:State,hyState:Humid,thState:Temp,heState:Level,wcState:Level,alState:State) : noexit :=
                        hide spalState in
                                (SpHyHewi2[spalGates∪hythGates∪hewiGates](spalState,hythState,hewiState)) |[on]| SPAL[spalGates](spalState:State)
                where

                process SpHy[spalGates∪hythGates\{sprayOn,sprayOff,open,close,on}](spalState:State , hythState:RecState) : noexit :=
                        hide sprayOn,sprayOff in
                                SPAL[spalGates](spalState:State) |[on,off]| HYTH[hythGates](hythState:RecState)
                endproc
                *relabel common gates across processes, otherwise a gate name mismatch will occur where Inc/Dec in HYTH is not recognised as the same gate as open/close in HEWI.
                *HYTH = HYTH[Inc/on, Dec/off](hythState:RecState)
                *HEWI = HEWI[Inc/open, Dec/close](p:Level)

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

                process SpHyHewi2[spalGates∪hythGates∪hewiGates\{sprayOn,sprayOff,open,close,Inc,Dec,on}](spalState:State,hythState:RecState,hewiState:Level) : noexit :=
                        hide open,close in
                                (SpHy[spalGates∪hythGates] ||| HyHewi[hythGates∪hewiGates]) |[open,close]| HEWI[hewiGates](hewiState)
                endproc

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

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

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

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

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

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

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

hewiGates def {SetLevel,open,close,on}

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

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

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

endproc (* GHCS *)
```

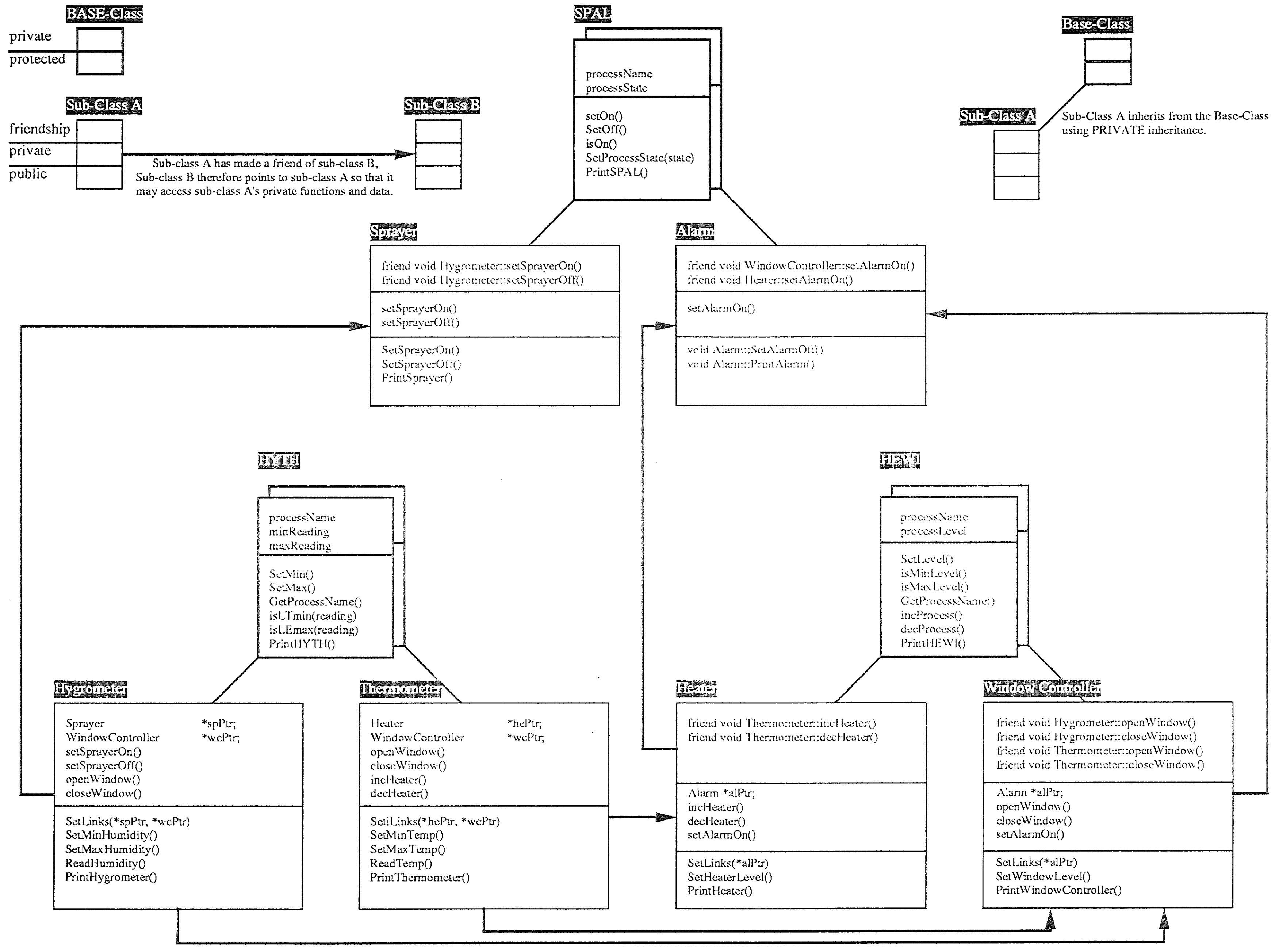
Appendix C: *ImpSpec-3*: Object-Oriented Specification in LOTOS

```
specification GreenHouse_Control_System(...)

        type
                ..........refer to original ImpSpec-0 GHCS specification for type and operations specification

        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}

        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[sprayerGates∪hygrometerGates∪thermometerGates∪heaterGates∪windowGates∪alarmGates\{setSprayOn,setSprayOff,openWindow,closeWindow,incHeat,decHeat,setAlarmOn}]
                        (s:State, h:RecState, t:RecState, he:Level, wc:Level, a:State) : noexit :=
                        hide setAlarmOn in
                                SpHy_ThHe_WC[sprayerGates∪hygrometerGates∪thermometerGates∪heaterGates∪windowGates](s,h,t,he,wc) |[setAlarmOn]| *Alarm[alarmGates](a)
                where

                process SpHy[sprayerGates∪hygrometerGates\{setSprayOn,setSprayOff}](s:State, h:RecState) : noexit :=
                        hide setSprayOn,setSprayOff in
                                *Sprayer[sprayerGates](s) |[setSprayOn,setSprayOff]| *Hygrometer[hygrometerGates](h)
                endproc

                process ThHe[thermometerGates∪heaterGates\{incHeat,decHeat}](t:RecState, he:Level) : noexit :=
                        hide incHeat,decHeat in
                                *Thermometer[thermometerGates](t) |[incHeat,decHeat]| *Heater[heaterGates](he)
                endproc

                process SpHy_ThHe_WC[sprayerGates∪hygrometerGates∪thermometerGates∪heaterGates∪windowGates\{setSprayOn,setSprayOff,openWindow,closeWindow,incHeat,decHeat}](s:State, h:RecState, t:RecState, he:Level, wc:Level) :
noexit :=
                        hide openWindow,closeWindow in
                                (SpHy[sprayerGates∪hygrometerGates](s, h) ||| ThHe[thermometerGates∪heaterGates](t, he)) |[openWindow,closeWindow]| *WindowController[windowGates](wc)
                endproc


                process SPAL[setOn,SetOff](s:State) : self(State) :=
                        [not isOn(s)] → setOn;self(on)
                        []
                        [isOn(s)] → (i;self(off) [] SetOff;self(off))
                endproc (* SPAL *)
```

```
sprayerGates def {SetSprayOn,SetSprayOff,setSprayOn,setSprayOff}

process Sprayer[sprayerGates](s:State) : self(State) noexit :=
        SPAL[setSprayOn/setOn,SetSprayOff/SetOff](s)
        []
        [not isOn(s)] → SetSprayOn;self(on)
        []
        [isOn(s)] →setSprayOff;self(off)
endproc (* Sprayer *)


alarmGates def {setAlarmOn,SetAlarmOff}
process Alarm[alarmGates](s:State) : self(State) noexit :=
        SPAL[setAlarmOn/setOn,SetAlarmOff/SetOff](s)
endproc (* Alarm *)

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


hygrometerGates def {SetMinHumid,SetMaxHumid,ReadHumidity,setSprayOn,setSprayOff,openWindow,closeWindow}

process Hygrometer[hygrometerGates](r:RecState) : self(RecState) noexit :=
        HYTH[SetMinHumid/SetMin,SetMaxHumid/SetMax,ReadHumidity/ReadInput,setSprayOn/setOn,setSprayOff/setOff,openWindow/open,closeWindow/close](r)
endproc (* Hygrometer *)


thermometerGates def {SetMinTemp,SetMaxTemp,ReadTemp,incHeat,decHeat,openWindow,closeWindow}

process Thermometer[thermometerGates](r:RecState) : self(RecState) noexit :=
        HYTH[SetMinTemp/SetMin,SetMaxTemp/SetMax,ReadTemp/ReadInput,incHeat/setOn,decHeat/setOff,openWindow/open,closeWindow/close](r)
endproc (* Thermometer *)
```

```
        process HEWI[SetLevel,open,close,setOn](l:Level) : self(Level) :=
                SetLevel ? l:Level;self(l)
                []
                open;
                        ([not isMaxLevel(l)] → self(incLevel(l))
                                []
                        [isMaxLevel(l)] → setOn;self(l))
                []
                close;
                        ([not isMinLevel(l)] → self(decLevel(l))
                                []
                        [isMinLevel(l)] → setOn;self(l))
        endproc (* HEWI *)


        heaterGates def {SetHeat,incHeat,decHeat,setAlarmOn}

        process Heater[heaterGates](l:Level) : self(Level) noexit :=
                HEWI[SetHeat/SetLevel,incHeat/open,decHeat/close,setAlarmOn/setOn](l)
        endproc (* Heater *)


        windowGates def {SetWindow,openWindow,closeWindow,setAlarmOn}

        process WindowController[windowGates](l:Level) : self(Level) noexit :=
                HEWI[SetWindow/SetLevel,openWindow/open,closeWindow/close,setAlarmOn/setOn](l)
        endproc (* WindowController *)

endproc (* GHCS *)
```

Appendix D.1: GHCS C++ Function Friendship Hierarchy and Structure



Appendix D.2: *ImpSpec-4*: GHCS C++ Class Definitions

```
class SPAL
{
        // use private on data area so that only derived protected can access via SPAL
        // protected functions
        private:
                char      processName[kMaxNameSize];
                State     processState;

        // use protected so that functions can be derived
        protected:
                SPAL(char *name, State initState);
                void setOn(void);
                void SetOff(void);
                Boolean isOn(void);
                void SetProcessState(State state);
                void PrintSPAL(void);
};
```

```
#include "GHCS-SPAL.h"
#include "GHCS-HYGROMETER.h"

//------------------------------------------------------------------------
//        DERIVED CLASS 'SPRAYER', INHERITED FROM 'SPAL' AND EXTENDING BASE CLASS
//        WITH INTERNAL setOff() FUNCTION AND EXTERNAL SetOn() FUNCTION
//------------------------------------------------------------------------
class Sprayer : private SPAL
{
        friend void Hygrometer::setSprayerOn(void);
        friend void Hygrometer::setSprayerOff(void);

        private:
                // private class functions
                void setSprayerOn(void);
                void setSprayerOff(void);

        public:
                Sprayer(char *name, State initState);
                void SetSprayerOn(void);
                void SetSprayerOff(void);
                void PrintSprayer(void);
};
```

```
#include "GHCS-SPAL.h"
#include "GHCS-HEATER.h"
#include "GHCS-WC.h"

//------------------------------------------------------------------------
//        DERIVED CLASS 'ALARM' FROM BASE CLASS 'SPAL'. (NO EXTENSIONS TO BASE CLASS
//        EXCEPT A MODIFIED CONSTRUCTOR FUNCTION WITH NAME/STATE PARAMETERS
//------------------------------------------------------------------------
class Alarm : private SPAL
{
        friend void WindowController::setAlarmOn(void);
        friend void Heater::setAlarmOn(void);

        private:
                // private class functions
                void setAlarmOn(void);

        public:
                Alarm(char *name, State initState);
                void SetAlarmOff(void);
                void PrintAlarm(void);
};
```

```
//------------------------------------------------------------------------
//      BASE CLASS HYTH, USED AS THE GENERIC TEMPLATE FOR THE HYGROMETER AND
//      THERMOMETER SUB-CLASSES. NO INTERNAL COMMS PORTS DEFINED IN-TERMS OF
//      PRIVATE FUNCTIONS.
//------------------------------------------------------------------------
class HYTH
{
        private:
                char    processName[kMaxNameSize];
                short   minReading,maxReading;

        protected:
                HYTH(char *name, short min, short max);
                void SetMin(void);
                void SetMax(void);
                char *GetProcessName(void);
                Boolean isLTmin(short reading);
                Boolean isLEmax(short reading);
                void PrintHYTH(void);
};
```

```
#include "GHCS-HYTH.h"

class Sprayer;
class WindowController;

//------------------------------------------------------------------------
//      DERIVED SUB-CLASS, INHERITING FROM BASE HYTH CLASS. INCLUDES 2 POINTERS
//      AS LOCAL PRIVATE DATA TO ENABLE INTERNAL COMMS TO TAKE PLACE WITH THE
//      FRIENDLY CLASSES SPRAYER AND HYGROMETER. COMPLEMENTARY ACTIONS (DEFINED
//      AS INTERNAL (PRIVATE) FUNCTIONS ARE ALSO DEFINED. THESE ARE DEFINED HERE
//      BECAUSE OF THE DIFFERENT SUB-CLASSES CALLED INTERNALLY, ALTHOUGH THEY
//      (open/close/setOn/setOff) APPEAR TO PERFORM THE SAME ACTIONS THEY CALL
//      DIFFERENT CLASSES DEPENDING ON THEIR IDENTITY. (E.G: HY CALLS SP, TH CALLS HE)
//------------------------------------------------------------------------
class Hygrometer : private HYTH
{
        private:
                Sprayer                 *spPtr;
                WindowController        *wcPtr;

                void setSprayerOn(void);
                void setSprayerOff(void);
                void openWindow(void);
                void closeWindow(void);

        public:
                Hygrometer(char *name, short min, short max);
                void SetLinks(Sprayer *sp, WindowController *wcPtr);
                void SetMinHumidity(void);
                void SetMaxHumidity(void);
                void ReadHumidity(void);
                void PrintHygrometer(void);
};
```

```
#include "GHCS-HYTH.h"

class Heater;
class WindowController;

//------------------------------------------------------------------------
//      SUB-CLASS OF HYTH BASE CLASS DEFINED TO ACT AS A THERMOMETER OBJECT.
//      LOCALLY STORES POINTERS TO THE FRIENDLY HEATER AND WINDOWCONTROLLER
//      SUB-CLASSES
//      DEFINED INTERNAL (PRIVATE) COMPLEMENTARY FUNCTIONS WHICH USE THE POINTER
//      VALUES TO THE FRIENDLY SUB-CLASSES.
//------------------------------------------------------------------------
class Thermometer : private HYTH
{
        private:
                Heater                  *hePtr;
                WindowController        *wcPtr;
                void openWindow(void);
                void closeWindow(void);
                void incHeater(void);
                void decHeater(void);

        public:
                Thermometer(char *name, short min, short max);
                void SetLinks(Heater *he, WindowController *wc);
                void SetMinTemp(void);
                void SetMaxTemp(void);
                void ReadTemp(void);
                void PrintThermometer(void);
};
```

```
//--------------------------------------------------------------------------
//      BASE CLASS HEWI DEFINING COMMON DATA STRUCTURE AND INTERNAL (PRIVATE) AND
//      EXTERNAL (PUBLIC) COMMUNICATIONS
//--------------------------------------------------------------------------
class HEWI
{
        private:
                // protected data structure
                char    processName[kMaxNameSize];
                short   processLevel;

        protected:
                HEWI(char *name, short level);
                void SetLevel(void);
                Boolean isMinLevel(void);
                Boolean isMaxLevel(void);
                char *GetProcessName(void);
                void incProcess(void);
                void decProcess(void);
                void PrintHEWI(void);
};
```

```
#include "GHCS-HEWI.h"
#include "GHCS-THERMOMETER.h"

class Alarm;

//--------------------------------------------------------------------------
//      DERIVED CLASS HEATER, BASED ON HEWI BASED CLASS. EXTENDS HEWI CLASS WITH
//      A POINTER DATA STORE TO A CLASS OF TYPE ALARM. POINTER IS USED TO GIVE
//      ACCESS TO ALARM'S INTERNAL COMMS PORT 'setOn()'. POINTER VARIABLE IS
//      ASSIGNED DURING CLASS INSTANCE CREATION USING CLASS CONSTRUCTOR VARIABLE
//      ALLOW THERMOMETER TO ACCESS PRIVATE FUNCTIONS USING FRIEND ASSOCIATION
//--------------------------------------------------------------------------
class Heater : private HEWI
{
        friend void Thermometer::incHeater(void);
        friend void Thermometer::decHeater(void);

        private:
                // private data
                Alarm *alPtr;

                // private functions
                void incHeater(void);
                void decHeater(void);
                void setAlarmOn(void);

        public:
                Heater(char *name, short level);
                void SetLinks(Alarm *al);
                void SetHeaterLevel(void);
                void PrintHeater(void);
};
```

```
#include "GHCS-HEWI.h"
#include "GHCS-HYGROMETER.h"
#include "GHCS-THERMOMETER.h"

class Alarm;

//-------------------------------------------------------------------------------
//      DERIVED CLASS FROM HEWI BASE CLASS. ALLOW HYGROMETER/THERMOMETER CLASSES
//      TO BE FRIENDS AND ACCESS INTERNAL OPEN/CLOSE FUNCTIONS. STORED LOCALLY A
//      POINTER TO ALARM DERIVED CLASS SO THAT INTERNAL COMMS CAN TAKE PLACE BETWEEN
//      THIS CLASS AND THE ALARM. DEFINED INTERNAL open()/close() FUNCTIONS AND A
//      COMPLEMENTARY setOn() FUNCTION USED TO CALL ALARM'S EQUIVALENT FUNCTION.
//-------------------------------------------------------------------------------
class WindowController : private HEWI
{
        friend void Hygrometer::openWindow(void);
        friend void Hygrometer::closeWindow(void);
        friend void Thermometer::openWindow(void);
        friend void Thermometer::closeWindow(void);

        private:
                // private data
                Alarm *alPtr;

                // private functions
                void openWindow(void);
                void closeWindow(void);
                void setAlarmOn(void);

        public:
                WindowController(char *name, short level);
                void SetLinks(Alarm *al);
                void SetWindowLevel(void);
                void PrintWindowController(void);
};
```