DIVISION OF COMPUTER SCIENCE

The Analysis of Formal Models of Communication for the Specification of Reusable Systems (Progress Report)

P N Taylor

Technical Report No.229

July 1995

The Analysis of Formal Models of Communication for the Specification of Reusable Systems (Progress Report)

P. N. Taylor.

Division of Computer Science,
University of Hertfordshire,
College Lane, Hatfield, Herts.
AL10 9AB. U.K.

Tel: +44 (0) 1707 284763
email: comrpnt@herts.ac.uk

July 1995

Abstract

This research focuses upon the reuse of processes in numerous environments. It is primarily concerned with investigating the relationship between sub-types, inheritance [13,28] and reuse within the context of process algebras. A fundamental issue underlying all of the research is the idea of communication and synchronisation between processes and the effect that modification has upon that communication.

Using current process algebras (CCS [15], CSP[10] and LOTOS[3,11]) an object can be modelled as a communicating process; its interface being defined as a set of synchronising ports.

A number of key issues have arisen from the research. Traditional process specification, using the languages of [15,10] and [11], assume that the signature of a process is static. This limits its reuse in different environments; such processes cannot change to meet future needs. It has been shown [24] that using action restriction and renaming with complex processes does not provide the flexibility required to maximise the reuse of a process.

This paper illustrates that extending a system with new processes can cause the system to deadlock. Models of synchronisation and communication play a key rôle in process algebras. In the context of extendible systems the semantics of synchronisation, modelled in [15,10], result in an unacceptable influence on the original behaviour of a system with the introduction of new processes. In order to guarantee behavioural equivalence within extendible systems there is a need to demonstrate conformance [19] within the sub-type relationship. If sub-typing is guaranteed then (at least) the behaviour of the superclass is available within the sub-type.

The key issues discussed in this paper can be summarised as:

- the nature of process synchronisation and its effect on extendible systems.
- the maintenance of system integrity despite the introduction of new processes.
- the investigation and notion of conformance between sub-types.

The main aim of the proposed research is to provide a mechanism for extending current process behaviour using non-strict inheritance [6] (referred to as casual inheritance). It is in this area of extendibility that the proposed investigation lies.

Introduction

In an object-oriented environment the elementary unit is an object, which is a single encapsulated entity containing both state and behaviour. Each object has an explicitly defined interface with its environment and the state of the object can only be changed via this interface. An object may communicate via several interface points (ports) with other objects in the system; either internal system objects or external objects (i.e: users).

One of the important concepts associated with object-oriented design has been the idea of sub-typing which is implemented via inheritance. Sub-typing allows both the extension and reuse of specifications, where one process definition can be substituted for another and still guarantee the behaviour of the system. Objects can be classified hierarchically using sub-typing to derive a new object from an existing one. The sub-type relationship between a superclass and its subclasses is a partial order [12], that is, it satisfies reflexivity ($Q \leq P$), transitivity ($R \leq Q \leq P$ therefore $R \leq P$) and anti-symmetry ($Q \leq P$ and $P \leq Q$ then $P = Q$), where $x \leq y$ denotes that x is a sub-type of y.

The sub-type relationship must guarantee sub-typing to ensure that no ill effects will result in the substitution of superclass and subclass objects. Ideally, a library of objects can be envisaged (i.e: process definitions) which can be used as the basis for numerous systems. Each specialised object would inherit its core behaviour from some central library and extend that behaviour to meet the requirements of the system. Meyer [14] argues that precise specifications of program code should accompany each library module. Specifications for that code should therefore be carried out using formal description techniques. By formally specifying the sub-type relationship and using formal languages to validate the behaviour of processes a system can be given a precise definition. Confidence will then remain high about the integrity of the design and behaviour of its components.

1 Current Investigation

This research focuses upon reusable processes in numerous environments. It is primarily concerned with investigating the relationship between inheritance [13,28] and reuse within the context of process algebras. A fundamental issue underlying all of the research is the idea of communication and synchronisation between processes and the effect that modification has upon that communication. The work so far has covered current practices in the following areas:

- process specification and portability
- specification of generic processes incorporating reuse
- behavioural inheritance; strict, non-strict (a.k.a casual) and transitive inheritance
- dynamic port allocation
- notions of equivalence between processes, such as:
  - testing equivalence
  - failure equivalence
  - conformance and extension testing
  - ≤may/≤must equivalence

• temporal behavioural modification (i.e: the mechanics and effects of non-strict inheritance)

2 Process Specification

Using process algebras such as CCS[15], CSP[10] and LOTOS[3,11] the interface and behaviour of a process can only be defined once during its lifetime, with little scope for subsequent modification. Previous work in [23,24,25] and [26] illustrates the standard form of process specification as defined in LOTOS [11]. It can be argued that a process signature does not provide enough flexibility to make the process portable. Previous work on the Rigorous Object-Oriented Analysis (ROOA) method [17] models systems using a client/server architecture. A process consists of one server port and *n* client ports. Consequently, when a process is modified to communicate with extra clients its interface needs to be redefined. This redefinition makes the process specific to the environment. I regard current process specification methods as too rigid because they limit the application of reusable processes.

Consider the following example case study system that is referred to throughout this paper. It is a case study of a system designed to control a chemical plant. Processes in the Chemical Plant Control System *(CPCS)* can be defined in the LOTOS notation. The process defined in Figure 2.1 is shown as a generic reusable process and is named after the behaviour that it exhibits; namely Flow Detection Heat Sensor *(FDHS)*.

```
process FDHS[eSetMin,eSetMax,eReadInput,iSetOff,iOpen,iClose](r:RecState) : self(RecState) :=
    eSetMin ? h:Reading;
            ([isLEMax(h,r)] → self(setMin(h,r))
                    []
            [isGTMax(h,r)] → self(r))
    []
    eSetMax ? h:Reading;
            ([isGEMin(h,r)] → self(setMax(h,r))
                    []
            [isLTMin(h,r)] → self(r))
    []
    eReadInput ? h:Reading;
            ([isLTMin(h,r)] → iSetOn;iOpen;self(r)
                    []
            [isGTMax(h,r)] → iSetOn;iClose;self(r)
                    []
            [isGEMin(h,r) and isLEMax(h,r)] → self(r))
endproc (* FDHS*)
```

Figure 2.1

The specific process Heat Sensor *(HS)* in Figure 2.2 is responsible for taking a reading from the temperature sensor in the chemicals. The Heat Sensor's behaviour is inherited from the behaviour of its superclass process, the Flow Detection Heat Sensor *(FDHS)*.

```
process HS[eSetMinTemp,eSetMaxTemp,eReadTemp,iSetALOn,iOpenCF,iCloseCF](r:RecState) :
self(RecState) noexit :=
    FDHS[eSetMinTemp/eSetMin,eSetMaxTemp/eSetMax,eReadTemp/eReadInput,
        iSetALOn/iSetOn,iOpenCF/iOpen,iCloseCF/iClose](r)
endproc (* HS *)
```

Figure 2.2

The process Coolant Flow Heater *(CFHE)* in Figure 2.3 maintains a flow of coolant around the chemical plant to regulate temperature. A heating unit is also included to maintain a constant temperature around the plant.

```
process CFHE[eSetLevel,iOpen,iClose,iSetOn](l:Level) : self(Level) :=
    eSetLevel ? l:Level;self(l)
    []
    iOpen;
        ([not isMaxLevel(l)] → self(incLevel(l))
            []
        [isMaxLevel(l)] → iSetOn;self(l))
    []
    iClose;
        ([not isMinLevel(l)] → self(decLevel(l))
            []
        [isMinLevel(l)] → iSetOn;self(l))
endproc (* CFHE *)
```

Figure 2.3

One specific process that is based upon *CFHE's* generic behaviour is the Coolant Flow Regulator *(CF)* which is specified in Figure 2.4.

```
process CF[SetCF,iOpenCF,iCloseCF,iSetALOn](l:Level) : self(Level) noexit :=
    CFHE[SetCF/SetLevel,iOpenCF/iOpen,iCloseCF/iClose,iSetALOn/iSetOn](l)
endproc (* CF *)
```

Figure 2.4

One final generic process resident in our system is the Sprayer Alarm process *(SPAL)*; as shown in Figure 2.5. The *SPAL* process defines the behaviour of spraying units around the site to help with cooling and decontamination. *SPAL* is also responsible for providing alarm signals, signifying problems with the running of the chemical plant. Note that each generic class is a superclass and is used as a template for defining subclasses. These subclasses provide the specialised templates that are used to create instances of the processes within the Chemical Plant Control System.

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

Figure 2.5

An example of a subclass process of the generic abstract template *SPAL* is the Alarm process *(AL)* in Figure 2.6. Process *AL* can be activated either internally from another process synchronising communication with it or by external panic buttons situated around the chemical plant.

```
process AL[eSetALOn,iSetALOn,eSetALOff,iSetALOff](s:State) : self(State) noexit :=
    SPAL[iSetALOn/iSetOn,eSetALOff/eSetOff](s)
    []
    [not isOn(s)] → eSetALOn;self(on)
    []
    [isOn(s)] → iSetALOff;self(off)
endproc (* AL *)
```

Figure 2.6

Process *AL* illustrates strict inheritance as it first inherits its basic behaviour from the generic process *SPAL* and then extends that behaviour with two extra action choices; namely the guarded expressions *eSetALOn* and *iSetALOff*. Note that with all process action names *"e"* is used to denote an intended external action and *"i"* is used to denote an intended internal action (e.g: *eSetMin* and *iClose*):

The work of Rudkin [19] illustrates that strict inheritance succeeds in offering behavioural extension via an extra choice of actions. However, adhering to the limitations imposed in [19] strict inheritance is not applicable to any but the simplest of systems. Rudkin states that it is not possible to show that strict inheritance can be applied to recursive processes. This particular restriction makes the ideas introduced in [19] unusable in practice and requires further investigation to see if a more suitable approach can be found.

As an extension to strict inheritance, non-strict inheritance is considered (which I refer to as casual inheritance). To highlight the key issues arising from casual inheritance consider the following problem. Suppose that we modify the Chemical Plant Control System *(CPCS)* to include a Warning Light process *(WL)*. The new Warning Light process behaves similar to the Alarm process except that it has no external panic button to activate it. Figures 2.7 and 2.8 illustrate the extension to the control system *CPCS* with the new Warning Light process *(WL)*.

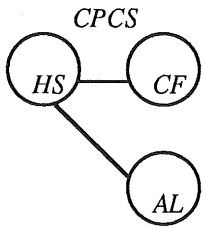


Figure 2.7

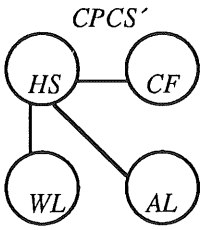


Figure 2.8

The Heat Sensor *(HS)* needs to be synchronised with both *WL* and *AL* during the same sequence of actions. Using the LOTOS definitions given in Figures 2.1 to 2.8 the problem can be solved in one of two ways.

The first course of action would be to configure *WL* to synchronise on the same port as *AL*; namely *iSetALOn*. LOTOS provides a facility for multiple synchronisation which would allow *HS* to communicate with both *AL* and *WL* on port *iSetALOn* at the same time. CSP also provides multiple synchronisation courtesy of its semantics (which LOTOS is based heavily upon) [10, §2.3.1] but CCS only allows bi-party communication due to the restrictive nature of the internal tau (τ) action. With CCS it is not possible for an observer to determine the nature of an inter-process communication [15, §2.2, p.39].

As a second proposed solution to the system extension introduced in Figures 2.7 and 2.8 it is possible to introduce a new communication port into HS, changing the interface of HS so that it can synchronise with WL whilst maintaining separate communications from AL. This second approach underlines the concept of casual inheritance because HS would have to be modified with a new synchronising action to WL.

The first approach uses multiple synchronisation between HS, WL and AL to solve the problem of reuse but has undesirable side-effects. The problems associated with multiple synchronisation are discussed later in section 5, regarding dynamic port allocation.

If casual inheritance is chosen then the temporal behaviour of HS would have to be modified. This modification poses a different set of problems which are related to the signature and temporal behaviour of HS. These problems are also discussed in detail later, in section 5.

3 Specification of generic processes incorporating reuse

The issues discussed in [23,24] and [26] recognise the need for reusable processes so that each specific process can use an existing generic core behaviour as its foundation. Process reuse in [24] adopts a strategy of capturing common behaviour to be copied and restricted by other subclasses. In [24] specific processes are defined in terms of a generic process (see Figure 3.1). The specific processes then restrict any unwanted behaviour from the inherited generic process. The following diagram shows how Q and R inherit behaviour from P (via straightforward duplication) and then restrict that behaviour in order to specialise.

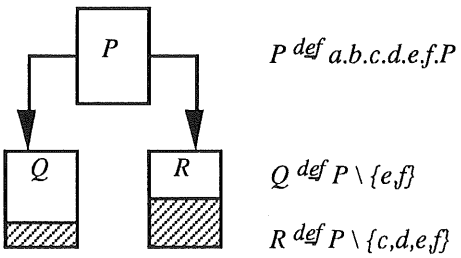


Figure 3.1

The ideas presented in [24] illustrate a method which is opposed to the inheritance that I am primarily concerned with. Action restriction is limited to small systems that do not offer many different forms of behaviour. Work in [24] has shown that restriction soon fails to support the ideas of inheritance that lead to greater reuse and flexibility.

Ideally, I seek to capture the common behaviour of a superclass and then extend that behaviour via strict and casual inheritance. This behavioural extension makes reuse more efficient and flexible because simple superclasses can be reused more widely than complex (specific) superclasses.

4 Behavioural Inheritance; strict, non-strict (casual) and transitive

There has been some work covering inheritance in LOTOS [12, 17] and [19]. However, many of the issues surrounding inheritance have not been adequately addressed and cannot be applied to large recursive systems. Consequently, this work concentrates upon the concepts of inheritance [13,28] and its application. There are three types of inheritance that my research is primarily concerned with:

i) *strict inheritance* which extends superclass behaviour.

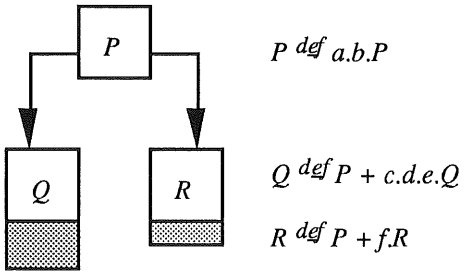


Figure 4.1

Processes Q and R in Figure 4.1 inherit behaviour from P and then extend that behaviour by offering extra action choices. The shading in Q and R illustrates the choice extension over the original behaviour supplied by process P.

ii) *casual inheritance* which modifies and extends existing process behaviour, such as supplied by the operator $\oplus$ in Figure 4.2.

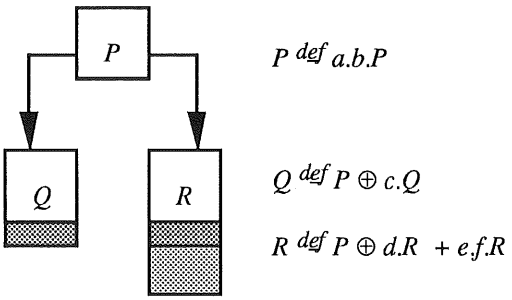


Figure 4.2

The definition of process Q shows the inherited behaviour of P being modified to include action c, prior to the recursion of Q. Process Q would therefore be written as $Q \stackrel{def}{=} a.b.c.Q$. Process R shows temporal behavioural extension together with action choice extension (as used in strict inheritance). Process R can be written in full as $R \stackrel{def}{=} a.b.d.R + e.f.R$.

iii) *transitive inheritance* which states that R inherits from Q, which in turn inherits from P. Consequently, R transitively inherits from P.

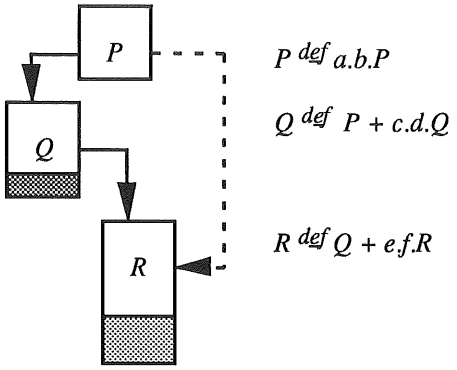


Figure 4.3

The dashed line in Figure 4.3 shows the transitive link between superclass process *P* and subclass process *R*, where *P* is the superclass of both *Q* and *R*. It can be seen from the diagram that the modules surrounding each process expand as each one inherits from its superclass. Such is the case with any form of inheritance where each subclass encapsulates and (possibly) extends inherited behaviour.

Previous work containing examples of strict inheritance can be found in [12,17] and [18]. However, each work has a different view of how inheritance should be modelled. The work of [12] uses process renaming to implement recursion but fails to show clearly if recursion is possible with an inheritance hierarchy of greater than one layer from the superclass. Recursion appears trapped in the superclass class offering the synchronising action such that the full range of actions of an inherited process are only on offer once and then cease to be available. This trapping of the flow of control between classes in the inheritance hierarchy is due to the recursion failing to return to the point of origin for the action call.

In [17] process enabling is used to provide recursion between superclasses and sub-types. Our studies have shown that process enabling does not support transitive inheritance. To illustrate the restriction imposed by the LOTOS process enabling operator (i.e: >>) consider the following example, based upon work in ROOA [17] and applied to the Chemical Plant Control System *(CPCS)* case study:

```
process SPAL[α](s:State) : exit (State) :=
        [not isOn(s)] → α ! iSetOn;exit(on)
        []
        [isOn(s)] → (i;exit(off) [] α ! eSetOff;exit(off))
endproc (* SPAL *)

process AL[β](s:State) : noexit :=
        (SPAL[β](s) >> accept  s:State in exit(s)
        []
        [not isOn(s)] → β ! eSetOn;exit(on)
        ) >> accept  s:State in AL[β](s)
endproc (* AL *)

process SP[γ](s:State) : noexit :=
        AL[γ](s) >> accept  s:State in exit(s)
        []
        [isOn(s)] → γ ! iSetOff;exit(off)
        ) >> accept  s:State in SP[γ](s)
endproc (* SP *)
```

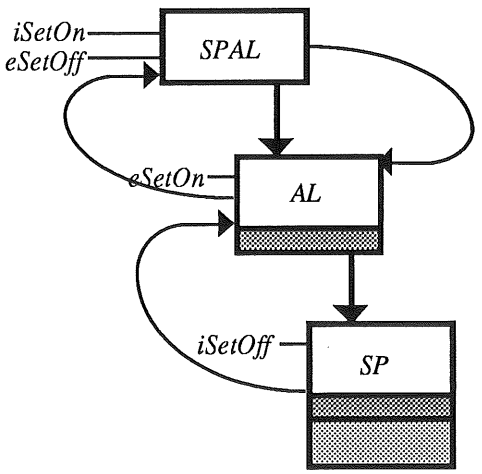


Figure 4.4

The superclass for the Sprayer/Alarm process *SPAL* is defined with *exit* functionality so that its services can be invoked from any of the subclasses which inherits from it. These services are called using delegation which searches up through the inheritance hierarchy until it finds either the appropriate action or fails. Both *AL* and *SP* can access the *SPAL* services *iSetOn* and *eSetOff*. After the *SPAL* services have been handled the services offered by its subclasses must be made available again. Recursion must be invoked on the subclass and not the superclass to ensure that the subclass services are made available after servicing a request to the superclass. By defining the functionality of processes in this way means that all superclass processes in LOTOS are abstract class templates [17] and are subsequently not created as physical entities within a system.

Notice that process *AL* in Figure 4.4 does not conform to the rules of functionality as defined above. Ideally, *AL* should be defined with the functionality *exit* for a superclass or *noexit* for a subclass. In order for *AL* to be a subclass of *SPAL* and a superclass of *SP* it is necessary to be able to specify that *AL* can delegate service requests to its superclass and receive requests from it subsequent subclasses. A mechanism is required for returning service requesters to their point of origin for each service call.

The method illustrated in Figure 4.4 cannot cope with transitive inheritance as the functionality for process *AL*, in the centre of the inheritance hierarchy, is required to have both the functionality of *noexit* and *exit* which is not possible. The bold downward pointing arrows in Figure 4.4 denote inheritance whilst the arc arrows denote recursion. Notice that the flow of control fails to return from *AL* to *SP* due to the *noexit* functionality of *AL*; recursion is trapped in *AL* when called from *SP* and therefore the system will fail to offer the services of *SP* again.

For recursion with transitive inheritance I prefer the notion of *self* [19,28] where each process can delegate the responsibility of servicing a request to another process, higher in the inheritance hierarchy; requests can therefore traverse numerous layers of inheritance (i.e: ≥1 inheritance layer). The *redirection* operator (*) [19] enforces the scope of recursion such that delegation in an inheritance hierarchy of $R \leq {}^*Q \leq P$ will not go beyond the redirection operator (*).

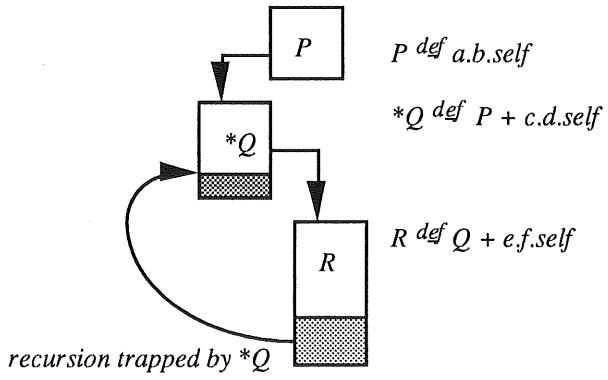


Figure 4.5

Certain restrictions imposed in [19] with regard to the sub-typing problem [7,19,20] make the ideas of *self* and *redirection* less applicable. These restrictions concern recursive processes and maintaining the integrity of the existing system prior to inheritance and possible sub-type substitution. These issues will need to be addressed if *self* and *redirection* are to be used widely in support of the

ideas concerning casual inheritance.

5 Dynamic port allocation

Static process signatures and behaviour prevent simple modification without rewriting the process itself. The processes defined in earlier work, such as [24, 25], cannot communicate with new processes as they are added to the system. Similar to my own models, the process definitions of ROOA [17] require new ports to be defined as a process becomes the client of a server process.

In a dynamic environment processes may leave and join the environment over time and will communicate with other processes whilst they are attached; similar in architecture to many computer networks. Such an environment can be modelled using a shared communication medium (shown as α). The structure of an inter-process communication is modelled as an event containing *<receiverIdentity, msg>*, as illustrated in the following LOTOS example:

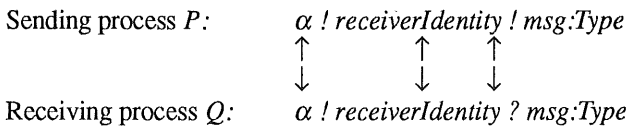


Figure 5.1

The communication illustrated in Figure 5.1 synchronises at three points in the event with a (unique) complementary partner. The synchronisation points in the communication are denoted by the vertical arrows in Figure 5.1. Using this form of communication, *n* processes can be added to the environment and remain confident that each process can communicate with the other existing processes (using α).

A key issue associated with the use of a shared communications medium is the problem encountered with multiple synchronisation. Processes with the same identity synchronise on the same action but if either process fails then the remaining processes that also synchronise on that action will also fail, causing the system to deadlock. Consequently, failure to synchronise across multiple processes communicating on the same port is a critical factor in the failure of a system.

Consider the process signatures for the processes Heat Sensor *(HS[s(HS),c(1),c(2)])*, Coolant Flow Regulator *(CF[s(CF)])* and Alarm *(AL[s(AL)])*. The contents of *[...]* signify the set of communicating ports for each process. From these process signatures the following system diagram can be derived. As shown in Figure 5.2.

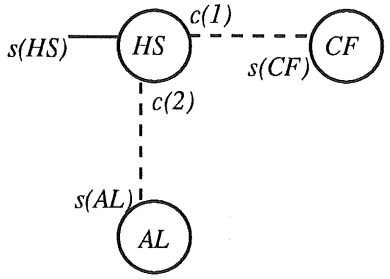


Figure 5.2

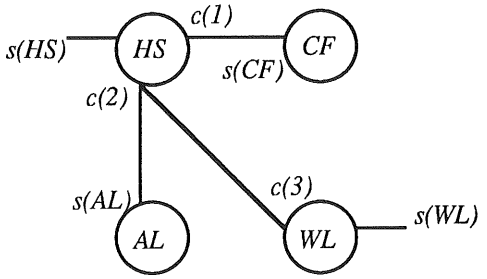


Figure 5.3

The gates present in *HS* are renamed to match those present in both *CF* and *AL* (i.e: *HS[s(HS),s(CF)/c(1),s(AL)/c(2)]*). Each process has a minimum of one server port, offering its services to the environment. If a process is also a client of other processes in the system then it also has *n* client ports for each process that it requires services from. As the model expands, with *CF* and *AL* themselves becoming clients *c(n)* of some other process, note that renaming alone cannot cope with the added complexity.

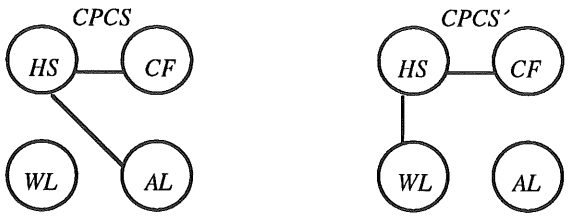
If an extra process is added to the system (as shown in Figure 5.3) then all of the consequences of this extension will have to be considered. Firstly, assume that *WL* communicates with *AL* on the same port as *HS*; namely port *s(AL)*. All three processes *(HS, AL* and *WL)* will have to synchronise on *s(AL)*. If, for example, *WL* fails then so will both *HS* and *AL* because they cannot synchronise with *WL* as it no longer offers the synchronising action *s(AL)*. Multiple synchronisation is very powerful. Any process using which makes use of multiple synchronisation has the power to stop all of the other processes from progressing, hence deadlocking the entire system.

It could be argued that no process should have the right to influence the operation of any other process. If *WL* is a low priority process then it would be beneficial if the system were to ignore *WL* if it fails to synchronise with any other process. Certainly, critical processes should halt the system if they fail because the system cannot function without them. However, prioritising processes is a complex issue and beyond the scope of this particular work.

Returning to the problem posed by the extension to the system in Figure 5.2 (as shown in Figure 5.3). If *WL* communicates with *HS* on a different port then its failure will not affect any other process. However, complications lie in the failure of *WL* to progress with its communications to *HS*. Further communications with *AL* will be impossible as *HS* has not progressed. Therefore, a *domino effect* occurs with one process causing the failure of another which eventually propagates throughout the entire system, causing its failure.

To summarise the issues associated with multiple synchronisation. Firstly, introducing a new process into a system which communicates on the same channel as other processes forces those third-party processes into a synchronisation with which they were not previously associated. Subsequently, the new process may cause them to deadlock as it exerts an influence over them. Secondly, adding a new process which communicates on a new port can still cause the original processes to deadlock if the new process then fails. Further to deadlocking an original process the introduction of a new process which uses a new communication port will also require the host process to be redefined with that new port to enable it to communicate with the new process; one of the original problem areas in this research.

As part of my work I have briefly investigated the π-calculus [16] which itself offers a method of switching between processes whilst a system is executing. Related work using LOTOS is discussed in [18]. The following diagram in Figure 5.4 illustrates a π-calculus solution to the dynamic switching between communication ports in the Chemical Plant Control System *(CPCS)* example. It shows a Heat Sensor *(HS)* connecting to a Warning Light *(WL)* rather than an Alarm *(AL)* at some time during the execution of the system. Similar to system *CPCS* in Figure 5.4 changing state to become system *CPCS´* in Figure 5.5.



$HS(\alpha,\beta,setXOn) \stackrel{def}{=}$ *if temp < min* α *closeCF* β *setXOn HS*
else
if temp > max α *openCF* β *setXOn HS*
else
if (temp ≥ min) and (temp ≤ max) then HS

$CF(\alpha,action) \stackrel{def}{=} \alpha(action)$ *action CF*

$AL(\beta,action) \stackrel{def}{=} \beta(action)$ *action AL*

$WL(\beta,action) \stackrel{def}{=} \beta(action)$ *action WL*

Figure 5.4

The number of ports and actions in the process's signature (e.g: $HS(\alpha,\beta,setXOn))$ are constant and the names of the synchronising ports and actions are also constant. To modify HS to synchronise on port γ the process would have to be redefined accordingly; similar to the LOTOS solution discussed earlier regarding the addition of an extra client port. I want to avoid the limitation of changing the signature of a process explicitly by using casual inheritance to extend the signature of a process.

Limitations with the π-calculus suggest that a review of high order (second and ω-order) process algebras be carried out, such as the High Order π-calculus [21] and the Calculus of High Order Communicating Systems [27]. Although these high order calculi offer parameterisation of processes to aid modification it is not clear whether they address the problems of behavioural extension and the effects of multiple synchronisation.

6 Notions of equivalence between processes

The proposed research intends to investigate the notions of failure and testing equivalence ($\leq may$, $\leq must$) [8] to determine whether one process can be substituted for another in an environment where the original process was expected. The new process must still provide (at least) the same behaviour [4,5] (congruence) as the original process. I am primarily interested in applying notions of equivalence to the inheritance issues that the work has raised so far.

Conformance testing (Q *conf* P)[5,20] and its derivatives [5] will be used as part of the study to equate the behaviour of a subclass process Q with its superclass P. I am concerned about the behaviour of sub-types when viewed from the behaviour of their superclasses. Much of the work on equivalences will draw upon the use of conformance testing, as illustrated in [19], where the main theme of the testing falls within the area of Hoare's failures model [10]. If a superclass fails to perform an action then the subclass should also fail to perform the same action. I am also concerned with how process equivalence is affected by inheritance and modularisation, building upon the work carried out in [1] and extending the ideas of [19] regarding the rules of conformance.

Using notions of equivalence I would like to capture the behaviour of a modified process (for example Q) and show it to be the same as its superclass (P) if the modifications are ignored; thus $P \approx$

$Q \backslash \{c\}$ where $P \stackrel{def}{=} a.b.P$ and $Q \stackrel{def}{=} a.b.c.Q$. The semantics of CCS [15], CSP [10] and LOTOS [11] do not allow us to say that P and Q are equivalent, regardless of the restriction on c because of the differences between the traces of P and Q. Conformance states that the traces are subsets of one another; $T(P) \subseteq T(Q)$. The subset rule is violated by the previous definitions for P and Q above. Process Q cannot be substituted for P because Q may deadlock if it fails to synchronise on c, whereas P does not; thus the stability of the system has been reduced.

7 Temporal behaviour modification (casual inheritance)

The effects of casual inheritance are recognised as the modification of existing process behaviour together with possible behavioural extension. The term *temporal behaviour* relates to the sequence of actions present in a process. Casual inheritance presents a complex challenge and is illustrated by the following example, which again uses the Chemical Plant Control System *(CPCS)* as its basis:

A basic system contains three processes which all communicate with each other via the shared port α. A heat sensor process *(HS)* synchronises with both a Coolant Flow Regulator *(CF)* and an Alarm *(AL)*, via the shared communications medium α.

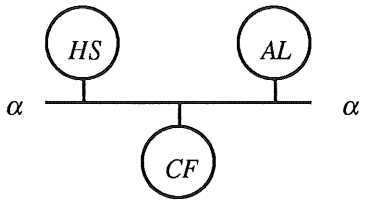


Figure 7.1

To illustrate how the specification of the processes change when a shared communications medium is used consider the formal definitions for the processes Heat Sensor *(HS)*, Coolant Flow Regulator *(CF)* and Alarm *(AL)*.

```
process HS[α]{CF(iOpenCF,iCloseCF),AL(iSetALOn)}(min:State,max:State) : self(State) :=
    eReadValue ? v : Value; ([v lt min] → α ! CF ! iOpenCF; α ! AL ! iSetALOn;self(min,max)
                            []
                [v gt max] → α ! CF ! iCloseCF; α ! AL ! iSetALOn;self(min,max)
                            []
                [(v ge min) and (v le max)] → self(min,max))
    []
    eSetMinValue ? v : Value;
                ([v le max] → self(v,max)
                            []
                [v gt max] → self(min,max))
    []
    eSetMaxValue ? v : Value;
                ([v ge min] → self(min,v)
                            []
                [v lt min] → self(min,max))
endproc (* HS*)
```

```
process CF[α]{AL(iSetALOn)}(s:Level) : self(Level) :=
    α ! me ? msg : Action;([msg == openCF] →
                                    ([s == max] → α ! AL ! iSetALOn; self(s)
                                    []
                                    [s < max] → self(s+1))
                    []
                    [msg == closeCF] →
                                    ([s == min] → α ! AL ! iSetALOn; self(s)
                                    []
                                    [s > min] → self(s-1)) )
    []
    eSetLevel? l : Level; self(l)
endproc (* CF *)

process AL[α]{}(s:State) : self(State) :=
    α ! me ? msg : Action;self(action)) )
    []
    i;self(off)
endproc (* AL *)
```

Problem: The Warning Light process *(WL)* is added to the existing environment containing *HS*, *CF* and *AL*. Communications to *WL* need to be incorporated into process *HS* together with communications to the existing processes *CF* and *AL*. Ideally, communications to *WL* after the original communications to *CF* and *AL* would need to be inserted, such that the modified behaviour of *HS* would allow communication with *CF*, *AL* and *WL* together during the same atomic transmission. The following LOTOS process definitions formalise the structure of *WL*, together with the extension to *HS*:

```
process HS'[α]{...}(...) : self(State) :=
        eReadValue ? v : Value;
            ([v lt min] → α ! CF ! iOpenCF; α ! AL ! iSetALOn; α ! WL ! iSetWLOn; self(min,max)
                        []
            [v gt max] → α ! CF ! iCloseCF; α ! AL ! iSetALOn;α ! WL ! iSetWLOn; self(min,max)
                        []
            [(v ge min) and (v le max)] → self(min,max))
        []
                ...


process WL[α]{}(s:State) : self(State) :=
        α ! me ? msg : Action;self(action)) )
        []
        i;self(off)
endproc (* WL *)
```

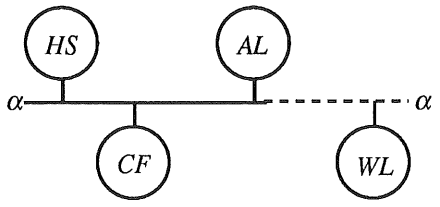


Figure 7.2

The aim for a proposed solution to the problem is to modify the structure for the behaviour of

HS; as shown in bold text as part of the modified heat sensor *HS´*. Complex issues associated with multiple synchronisation and deadlock are brought to the forefront when considering adding new processes to a system. Consider the sphere of influence that a new process has over its neighbours when they are all required to synchronise on the same actions. There is no concept of priority; if one fails then they all fail!

Section 5 illustrates fully the issues raised as part of the discussion into extending an existing system. Further to these issues, the work of [9] addresses part of the problem of process dynamics in a different way to the current work presented here; using a constant number of ports for communication, similar to the π-calculus [16] and switching between available communications channels and processes during run-time. Current methods, including [12,17,19], do not address casual inheritance. Further work is needed to support the ideas of reuse using process algebras.

8 Proposed Investigation

The investigation will centre on the reusability of processes using inheritance. Primarily I shall address the issues arising from the temporal modification of process behaviour and the effect that this modification has on the behaviour of the rest of the system. Casual inheritance can be defined as a method of extending the sequence of actions that comprise the temporal behaviour of a process. By incorporating casual inheritance I can make the target process more flexible and portable.

I intend to investigate the application of *self* and *redirection* (*) [19] and address the restrictions imposed upon processes that use these two constructs to enable them to meet conformance criteria; increasing their application area whilst maintaining the sub-type relationship.

Other issues imposed by the introduction of casual inheritance will need to be addressed in order to approach a more complete solution. Issues such as i) multiple synchronisation between processes which can lead to deadlock if one action fails to synchronise and ii) introducing new communication ports into existing processes if a new process synchronises on a new channel.

I seek to provide a mechanism for implementing casual inheritance. The introduction of additional operators into standard LOTOS [11] may also be required to allow the temporal behaviour of a process to be extended. These operators, if applicable, will be investigated. This new operator would bind communications together (similar to the operator discussed in [1] and introduced in [2]) so that they would occur during the same atomic communication.

To complete the proposed research an investigation into various formal proofs of equivalence will need to be carried out. The equivalence proofs, taken from [4,5,8] and [19], are intended to show that the modifications I propose do not compromise the semantics of the formal description techniques upon which the process definitions are based.

To conclude, my goal is to provide a mechanism for modifying current process behaviour whilst maintaining the integrity of the modified system. Formal proofs will need to be constructed to show that a new modified process is (at least) behaviourally equivalent to the original process.

Finally, future work concerning the implementation of my proposed research ideas could be considered using various programming languages such as C++ [22] and various actor languages [29].

References

[1] Baillie, E.J. (April 1992). *Towards a Satisfaction Relation between CCS Specifications and their Refinements*. Ph.D. Thesis. University of Hertfordshire, U.K.

[2] Baillie, E.J. and Smith, D.E (May 1994). A Conservative Extension to CCS for True Concurrency Semantics. **TR200**, Division of Computer Science, University of Hertfordshire. U.K.

[3] Bolognesi, T and Brinksma, E. (1987). Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*. **14**(1): pp25—59.

[4] Bolognesi, T. and Caneve, M. (1989). Equivalence Verification: Theory, Algorithms and a Tool. *The Formal Description Technique LOTOS*. P.H.J van Eijk, C.A. Vissers and Diaz, M. (Editors). Elsevier Science Publishers B.V: North Holland. pp303—326.

[5] Brinksma, E. (1989). A theory for the derivation of tests. *The Formal Description Technique LOTOS*. P.H.J van Eijk, C.A. Vissers and Diaz, M. (Editors). Elsevier Science Publishers B.V: North Holland. pp235–247.

[6] Cusack,E. (1988). Fundamental aspects of object oriented specification. *BT Technology Journal*, **6**(3): pp76—81.

[7] 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.

[8] de Nicola, R. and Hennessy, M.C.B. (1984). *Testing Equivalences for Processes*. Theoretical Computer Science. Elsevier Science Publishers B.V: North Holland. **34**: pp83—133.

[9] Fredland, L. and Orava, F. (1992). *Modelling Dynamic Communication Structures in LOTOS*. Proceedings of the IFIP.

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

[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] Mayr, T. (1989). Specification of Object-Oriented Systems in LOTOS. K.J. Turner (Editor). *Formal Description Techniques*. Elsevier Science Publishers B.V:North Holland. pp107—119.

[13] Meyer, B. (1988). *Object oriented software construction*. Prentice-Hall International.

[14] Meyer, B. (December 1990). The New Culture of Software Development. *Journal of Object-Oriented Programming*. November/December 1990.

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

[16] Milner, R. (October 1991). *The Polyadic π-Calculus: A Tutorial*. LFCS Report **ECS-LFCS-91-180**. Department of Computer Science, University of Edinburgh.

[17] Moreira, A.M.D. (August 1994). *Rigorous Object-Oriented Analysis Method*. **TR CSM-132**. Ph.D. Thesis. Department of Computing Science and Mathematics, University of Stirling, Scotland.

[18] Najm, E. and Stefani, J. (1992). *Dynamic Configuration in LOTOS*. Proceedings of the IFIP.

[19] Rudkin, S. (1992). Inheritance in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques*, IV, pp409—423. Elsevier Science Publishers B.V: North Holland.

[20] Rudkin, S. (July 1993). Template, Types and Classes in Open Distributed Processing. *BT Technology Journal*, **11**(3): pp32—40.

[21] Sangiorgi D. (May 1993). *Expressing Mobility in Process Algebra: First-Order and Higher-Order Paradigms*. Technical Report (Ph.D. Thesis). **ECS-LFCS-93-266** (CST-99-93). Department of Computer Science, University of Edinburgh, U.K.

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

[23] Taylor, P.N. and Britton, C.E. (August 1994). *Increasing the usability of Formal Specification Techniques through a combination of complementary formal languages and automated verification tools*. **TR210**, Division of Computer Science, University of Hertfordshire. U.K.

[24] Taylor, P.N and Smith, D.E (June 1994). *The Influence of the Formal Description Technique LOTOS on Concurrent System Design*. **TR203**, Division of Computer Science, University of Hertfordshire. U.K.

[25] Taylor, P.N and Smith, D.E (June 1994). *A case study for Generic Processes and Reusability in*

LOTOS. **TR-209**, Division of Computer Science, University of Hertfordshire. U.K.

[26] Taylor, P.N and Smith, D.E (January 1995). *Smoothing the transition from Formal Specification to Object-Oriented Implementation.* **TR-213**, Division of Computer Science, University of Hertfordshire. U.K.

[27] Thomsen, B., (1990). *Calculi for higher-order communicating systems.* Ph.D. Thesis. Imperial College, London University.

[28] Wegner, P. (1987). *The Object-Oriented Classification Paradigm.* Research Directions in Object-Oriented Programming. B. Shriver and P. Wegner (Editors). MIT Press.

[29] Yonezawa, A and Tokoro, M.(Editors) (1987). *Object-Oriented Concurrent Systems..* MIT Press.