DEPARTMENT OF COMPUTER SCIENCE

# Research Issues Associated with Process Algebras and the Object-Oriented Paradigm

P.N. Taylor

Technical Report No. 281

April 1997

# Research Issues Associated with Process Algebras and the Object-Oriented Paradigm

**P. N. Taylor.**

Department of Computer Science, Faculty of Information Sciences,
University of Hertfordshire, College Lane, Hatfield, Herts. AL10 9AB. U.K.
Tel: 01707 284763, Fax: 01707 284303, Email: p.n.taylor@herts.ac.uk

April 9, 1997

**Abstract**

In this paper I present a review of research issues raised by the application of process algebras to the design of object-oriented systems. In particular, I concentrate upon the difficulties associated with behavioral extension and the dynamic reconfiguration of systems with a view to reuse; a concept for which object-oriented design provides many useful methods, including inheritance and operator overriding.

From this review it is clear that several issues are poorly addressed by current languages; these issues form the basis for my research. Basic languages such as CCS [9], CSP [4] and LOTOS [5] and higher-order languages (i.e: CHOCS [14] and $HO\pi$ [13]) are reviewed. My research thus far suggests that elements required to make a formal model that matches closely an object-oriented design are present, in part, in many of the reviewed formal languages but are not all present in one language. This paper attempts to answer the question: 'Which features of the available formal languages for concurrent systems capture the essential features of object-oriented design with a view to aiding the reuse of concurrent systems specification?'.

An overall picture of process algebras and how they map to object-oriented design issues is presented, together with links between elements of the process algebra view of systems design. Issues such as asynchronous communication and its influence on a system's behavioral modification, or the reuse of processes using inherited branch behaviour are discussed. I show that synchronisation between processes hinders any attempt to modify the behaviour of a system containing those processes.

I also show that new processes have the capability to influence the stability of the system due to the dependence between processes enforced by synchronisation. If the systems in question are modeled using an object-oriented design approach and inheritance between processes is required (to aid reuse of components and simplify the model) then only strict inheritance can be applied in a limited manner, based upon restrictions identified by Rudkin and his work on LOTOS.

My work relates many issues which at first appear disjoint. I show that elements of a process algebra need to be combined with other capabilities provided with certain languages in order to enable a more complete formal model of an object-oriented system to be specified.

## Introduction

The main aim of this paper is to review and address key issues surrounding the application of process algebras available today to the formal capture of object-oriented systems. I present a network diagram which helps with the identification of the key areas that are the focus of my research. The aim of the research is to show how available process algebras can be used to capture some, if not all, of the characteristics that we have come to expect from object-oriented design and implementation.

A central theme of this paper is the modification of the internal structure of a process (i.e: behavioral modification) which is process-based, and the reconfiguration of a system's environment (i.e: system modification) which is system-based. Much of my work looks at inheritance and its relationship with communication protocols. To help with the identification of the key issues of my research the diagram in figure 1 was constructed. This diagram lists each area of interest and then links those areas together to form a research network (known as the PARNET: Process Algebra Research NETwork). The structure of this paper is based upon the network diagram in figure 1. Each branch on the network is discussed in the following sections. Languages that provide a facility to meet the criteria of the branch in the appropriate table and subsequently feature in the appropriate sections below. Links between branches are shown as double headed arrows.

This paper is split into 3 parts. Part 1 introduces the problem domain associated with system communication issues. Part 2 introduces the behavioral issues associated with object-oriented design and its application to process algebra system specification. Where appropriate the links between each section, shown as arrows in figure 1, are discussed. Part 3 underlines my research

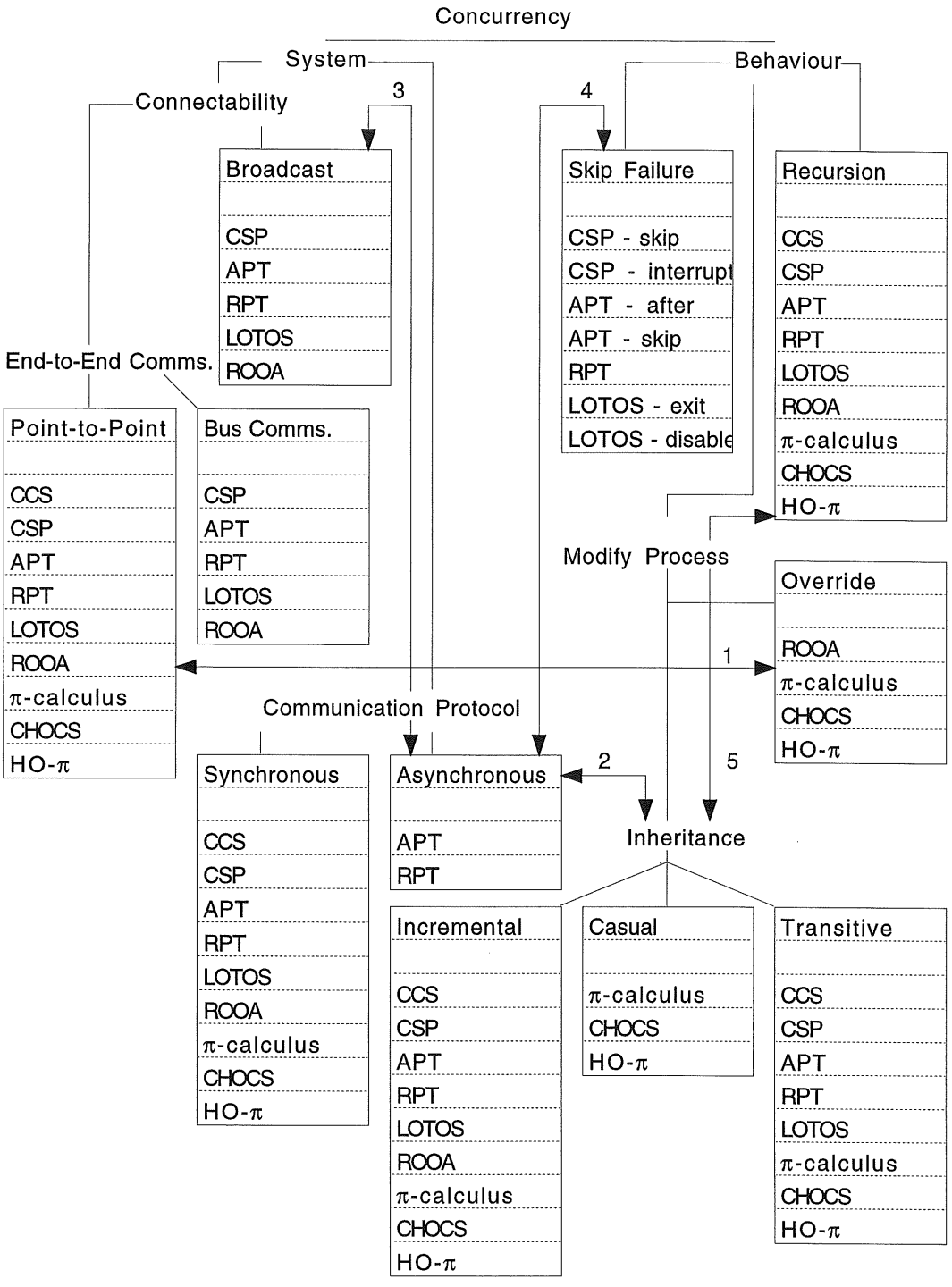


Figure 1: Object-Oriented Concepts and Process Algebras: A relational network

interests and draws conclusions from the current work. Part 3 also suggests future work to be carried out, prior to the final submission of my thesis.

# Part I

# System

## 1 System Considerations

A *system* is a collection of processes that communicate with each other and the environment. Only the interface to each process is considered in this section, rather than the intricacies of a process's internal behaviour. However, the communication protocols discussed are of concern to the behaviour of the process.

## 2 Connectability

Connectability is concerned with whether processes connect along prescribed paths or make use of a more general, shared, communications network. A shared network determines the route and order of any message, leaving the process to simply send and receive those messages and not concerning itself about the location of the receiver. Note that message routing is beyond the scope of this paper and is not discussed further.

### 2.1 Point-to-Point Communication

Point-to-point communication implies that the sender and receiver are linked via a common port or channel, or that the address of the receiver is known to the sender and is sent along with the message so that the routing system that deals with the transportation of the message knows the identity of the recipient. Examples of each type of point-to-point communication in the physical world are: i) making a telephone call where a connection is made for the duration of the call, or ii) posting a letter, where the postal service handles delivery based on the address on the envelope.

Two views of communication can be derived from the two main camps supporting process calculi: i) Hoare's model (Oxford) and ii) Milner's model (Edinburgh). Figure 2 helps to illustrate the different view of process communication as modelled in CCS, CSP and their derivatives.

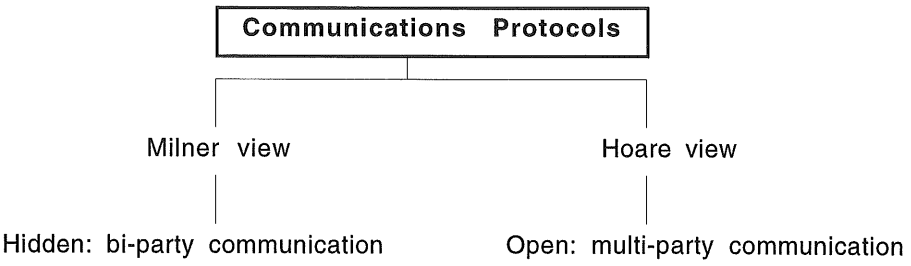


Figure 2: Opposing Views of Process Algebra Communication

Specifically, point-to-point communication occurs between shared (complementary) ports in CCS, channels in CSP or gates in LOTOS. Note that these terms all refer to the same point on the interface and are used interchangeably. All of the review languages support point-to-point communication.

The remaining paragraphs in this section introduce how point-to-point communication is provided by the review languages. Examples are given to illustrate the syntax of the reviewed languages when exercising point-to-point communication.

CCS defines point-to-point communication via complementary ports that are housed in each communicating process. The interface to a process is defined by the ports that are visible to the environment (i.e: exported entry points to a process). Other processes in the environment may communicate only via complementary ports (i.e: of the form $a \leftrightarrow \overline{a}$) [9, p.38]. With each process algebra presented here it should be noted that communication is atomic (i.e: there is no specific send and subsequent receive) as both parties synchronise together at the same instance in time (although time is not strictly modelled in either language under review).

CSP processes use a similar form of communication, although duplicate named ports are preferred rather than complementary ports. The same port names appear in both processes participating in a synchronised communication. CSP communication takes the form $(a \leftrightarrow a)$ [4, p.133].

Asynchronous Process Theory (APT) [7] and Receptive Process Theory (RPT) [6] both use CSP as their foundation language. Consequently, these two languages deal with process communication in a similar way to that of CSP, using similarly named ports in order to communicate. It should be noted that the work on RPT is also built upon the work of Dill [3].

LOTOS uses gates to communicate which provide access to LOTOS processes in much the same way as ports and channels provide access to CCS and CSP processes respectively [8, p.328].

The $\pi$-calculus (a first-order process calculus) was developed from CCS but has the ability to

pass ports as parameters to processes instead of just values. Therefore, the $\pi$-calculus can achieve a large degree of mobility in terms of the systems that it specifies. Despite this mobility these parameterised ports become static during communication. The communication itself is carried out much the same as CCS or CSP. An illustration of the ability to pass a port as a parameter to a process and then have that process continue to use the parameterised port is illustrated in the mobile phone relay example in the $\pi$-calculus tutorial [9, p.12].

CHOCS (Calculus of High Order Communicating Systems) takes the semantics of the $\pi$-calculus one stage further by allowing whole processes to be passed as parameters to a process, rather than just ports or values. With CHOCS, as with CSP, common ports are used to achieve communication between parties that are composed together within the same environment [14, p.15].

High-Order$\pi$ calculus (written as HO$\pi$) is another derivative of Milner's $\pi$-calculus and like CHOCS it too allows processes to be passed as parameters to a process [13]. The mobility of HO$\pi$ is rivaled only by CHOCS as both languages are highly mobile. HO$\pi$ also requires commonly named ports to be present in each communicating party for the communication to be successful. The mobility of both CHOCS and HO$\pi$ and its application to modelling the concepts of the object-oriented paradigm warrants further investigation.

### 2.1.1 Bus/Multi-party Communication

This section concentrates upon communication involving two or more parties, using the same port. In the previous section specific ports where used to synchronise and communicate values. Theoretically, bus or multi-party communication allow numerous processes to engage in communications across shared ports. CCS is excluded from this section because its communications are limited to only bi-party and are also hidden from the environment which excludes further parties from joining the communication. The $\tau$ action in CCS signals that a synchronising communication has occurred but gives no information as to the nature of that communication [9, p.39]. In CCS the only way that communication can be *observed* is by noting the change in state of the participating processes; an implicit rather than explicit view of communication.

CSP differs from CCS as it does not hide its communications. These communications appear in the action trace of a process, shown between angle brackets (e.g: $\langle a, b, b, c, d, \ldots \rangle$) [4, p.142]. For example, $(c!v \rightarrow P) \parallel (c?x \rightarrow Q(x)) = c!v \rightarrow (P \parallel Q(v))$. Note that $c!v$ (on the left side of the equation)is an observable action, akin to tapping the wires of the system to log internal

communications as they occur.

With both APT and RPT their formal basis is that of CSP. Therefore, communication in both of these language follows CSP primitives. Both APT and RPT can therefore share ports and engage in multi-party communication.

LOTOS is a key language for sharing communication gates. Many examples of gate sharing for multi-party communication in LOTOS exist [11, p.16]. LOTOS borrows much of its syntax and semantics from CSP, hence the similarity between communication expressions in CSP and LOTOS. The symbols ? and ! are used to represent input and output respectively. In the ROOA (Rigorous Object-Oriented Analysis) method a banking system is used to show port sharing across several processes. The ROOA method attempts to map object-oriented concepts onto LOTOS, therefore processes are treated as objects [11, p.16]. The semantics of communication in LOTOS can be seen in [1, p.33]. Basically, a gate taking part in a communication is not hidden and is available for subsequent communication and synchronisation.

It should be noted that all of the communications methods discussed so far require some form of synchronisation. That is, the simultaneous joining of similar actions on common ports to affect a communication. Processes that are not ready to engage in communication force the ready processes to wait until such time as they can all synchronise together. With bus and multi-party communication the problems with waiting are compounded because a series of processes may all have to wait for a slow process to catch up. A chain reaction of waiting processes will eventually deadlock the entire system. I refer to this chain reaction as *domino waiting*. The resolution of *domino waiting* is one of the key areas that concerns my research.

## 2.2 Broadcast Capability

The ability to broadcast a message is linked implicitly with the type of communication adopted by the particular formal language; namely synchronous or asynchronous communication. To broadcast a message one must not care about receiving an acknowledgment from any message recipient. To do so could entail an infinite wait while all recipients confirm that they have received the original broadcast; analogous to a lecturer waiting for each student to signal understanding before moving on to the next piece of information—a timely exercise!

Note that only some of the reviewed languages can cope with broadcasting and it is still not clear as to the success of their applicability to this particular type of communication.

Asynchronous Process Theory (APT) [7] and Receptive Process Theory (RPT) [6] are likely

candidates for a process algebra that can supply a broadcast communication primitive. However, because they are based on CSP it is not clear whether the asynchronous process algebras supply all of the elements required for broadcast communication. In theory, in APT and RPT the waiting time endured by CSP during synchronisation should not occur as acknowledgments are not required in either RPT or APT as they can send and receive at will.

The visibility of communication in both APT and RPT remains the same as that of CSP, allowing further processes to engage in an asynchronous (speed independent) communication along common shared channels.

# 3 Communication Protocols

By far the most common form of communication protocol is synchronous communication. CCS, CSP and languages derived from them (i.e: LOTOS, APT, RPT, CHOCS and HO$\pi$) are all based on synchronous communication. Although APT and RPT offer asynchronous communication via unbounded buffers the buffers themselves still synchronise with the target of the communication.

Synchronous communication have the potential to stall the execution of a process, insisting upon a common state between communicating parties before allowing communication to commence. Alternatively, asynchronous communication do not require the participation of other processes. Processes may send communications without the requirement to first wait until all parties are ready to receive the message. In theory, an asynchronous process may send as many messages as fast as it wishes. In practice both APT and RPT offer synchronisation with a personal unbounded buffer which is always ready to synchronise with its parent process.

## 3.1 Synchronous Communication

In CCS, processes $P$ and $Q$, when composed together with the composition operator $|$, will communicate on the common ports that they both share. Formally we express this notion via the intersection of the sorts of $P$ and $Q$ which are sets holding the actions contained in each process. Described formally as: $\mathcal{L}(P) \cap \mathcal{L}(Q)$, where communication occurs across the ports contained in the resulting set. Consider the definitions of processes $P$ and $Q$:

$$P \stackrel{\text{def}}{=} a.b.\overline{c}.P$$

$$Q \stackrel{\text{def}}{=} \overline{a}.b.c.Q$$

In this example, $(P \mid Q)$ forces $P$ and $Q$ to synchronise via $a$ and $c$. The port $b$ will not be used between $P$ and $Q$ as it is not complementary (i.e: not of the form $b \leftrightarrow \overline{b}$).

CSP uses a similar synchronisation technique to CCS, except the language of CSP does not complement its ports [4, p.66]. Ports (known as channels in CSP) with the same name are used to determine the synchronisation points between processes. These points become active when the processes are composed together in the same environment. If the previous CCS example were transformed into CSP then $P$ and $Q$ would synchronise on $a, b$ and $c$. The complementary port symbol $\overline{a}$ would be removed from the actions of $P$ and $Q$ as the language of CSP does not support the concept of a complementary channel name. As with CCS, the intersection of each participating process's alphabet yields those ports eligible to engage in communication. With CCS the expression $(\alpha P \cap \alpha Q)$ defines the common ports between the communicating processes. An alphabet $(\alpha)$ of a process in CSP is the same as the sort $(\mathcal{L})$ of a CCS agent. CSP allows more than two processes to engage in a synchronising communication as it does not hide the communication from the surrounding environment.

LOTOS processes also synchronise on common gates [8, p.328]. A common gate in numerous LOTOS processes will be used to synchronise all of the processes together provided that each process is in a fit state to receive the communication. If any participating process is not ready then the communication's initiator will have to wait until all processes are ready to synchronise, only then can communication commence.

APT is primarily an asynchronous process algebra but it can be used to model synchronous communication. To perform this transformation an acknowledgment to each message is built into the communicating system. By removing the unbounded buffers from the output ports on an APT process we effectively reduce its capability to that of a CSP process [7, p.5].

In the $\pi$-calculus, once the mobility of the system has been defined communication between processes occur in much the same way as those between CCS agents [10, p.6]. A sending agent may send a signal across a named port. However, it may be unclear as to the identity of the receiver. In the $\pi$-calculus mobile phone example [10, p.12] the use of one base station over another does not affect the success of the message getting through to the receiving vehicle, only the route of the message.

In CHOCS processes may also be passed as parameters to be used at some later stage. Note that ports are also valid parameter types. Synchronisation occurs across shared ports that are labeled with a direction of communication, similar to ? and ! in CSP. See [14, p.143] and [10,

p.12]

## 3.2 Asynchronous Communication

One may think of asynchronous communication as supplying the foundations from which synchronous communication is built. Indeed, the synchronous communication that was discussed in the previous section can be thought of as two asynchronous communications combined together, one part for send and another for acknowledge. However, it would be unfair to think of asynchronous communication in this way because all of the synchronous communications in the reviewed process algebras that we have already seen occur at the same instance in time. Both parties come *together* to form the communication rather than one initiator followed by a reply.

Asynchronous communication can be modelled in a synchronous communication environment by adding buffers of infinite size on each input and output channel of a process. The resulting system's processes can then send countless messages, synchronising with the buffer which is always ready to receive the process's messages. Recipients synchronise with the buffer when they are in a fit state to receive.

With an asynchronously communicating process any inherited behaviour that results in failure (i.e: unstable behaviour leading to deadlock) need not crash the entire system. I look upon the potential of an asynchronous communication protocol as one possible solution to the problem of maintaining a system. This maintenance problem can be categorised as:

1. dynamic (i.e: processes and interface points being added and removed from the original specification)

2. subject to modification via inherited processes to replace original ones (i.e: processes remaining externally similar (accepting port replacements/additions) but their internal behaviour changes)

Asynchronous Process Theory (APT) also offers asynchronous communication and uses buffers of infinite size on each input and output channel [7]. APT processes can be converted into CSP processes by the removal of these buffers. Alternatively, CSP processes can be converted into APT processes by placing an infinite buffer on each input and output channel. By using CSP as the foundation language and enabling APT and CSP processes to be viewed as equivalent (using APT $\rightarrow$ CSP conversion) one may then use CSP language constructs to prove properties about an APT specification. Consequently, many of the theoretical results gained using CSP

remain valid for APT [7, p.1]. The problems associated with using CSP as a model are that any misgivings that CSP has will be carried over into APT; these misgivings are inevitable due to the equivalences between the two languages.

Comments associated with APT in the previous paragraph also hold for Receptive Process Theory (RPT). Synchronisation can be defined using a send/receive protocol [6]. A receptive process is viewed as a triple *(I,O,F)*, where $I$ = *input alphabet,* $O$ = *output alphabet* and $F \subseteq (I \cup O)*$ which denotes the set of failures for the process.

# Part II

# Behaviour

The behaviour of a process relates to its defined sequence of actions. These action sequences provide a service to the process's environment. Each process algebra offers such a sequence of actions, including both behavioural choice and recursion. Choice may be deterministic (where the environment influences then choice) or nondeterministic (where the process influences the choice). The interface to a process is defined by its action sequence, where each action is a point of entry into the process.

## 4 Behavioral Considerations

Many aspects of communication between processes will be covered during this section. It is normal practice for a process to execute actions sequentially (i.e: in the order that they are stated in the process's behavioral definition). I view this ordering as a constraint w.r.t object-oriented behavioral modification (via inheritance). Basically, once a process has started down a sequential path of actions then that path must be completed before the system can execute some other branch of that process's behaviour. All of the process algebras under review insist upon completion of their current execution path.

### 4.1 Skip Failure Redundancy

Due to the possibility of *domino waiting*, where a collection of processes are all waiting on deadlocked actions, a method of bypassing a halted process action would be of benefit. I hope

to avoid a situation where, eventually, the entire system may halt with all processes waiting on an initial event that has failed to occur. My work will attempt to address the issue of *domino waiting*.

The languages CSP, RPT and APT offer a way for a process to continue processing and therefore continue executing rather than wait (possibly) indefinitely for a deadlocked action. This section contains descriptions of those languages and how they attempt to provide a service, regardless of the availability of some of their actions for communication.

CSP offers the an interrupt operator, represented by the symbol $\hat{}$, which does not wait for the successful termination of a process but instead *forces* execution of the next process in the expression [4, p.180]. For example, the expression $(P \hat{} Q)$ denotes that $P$ is interrupted by the first event of $Q$; $P$ is never resumed. The trace of events for $(P \hat{} Q)$ is simply the trace of $P$ followed (at some arbitrary point when the interrupt occurs) by the trace of $Q$. Successful termination (denoted by the symbol $\checkmark$) must not exist in the set of $\alpha P$ to ensure that $Q$ interrupts $P$. It may prove useful for a process to interrupt a hung process using the $\hat{}$ operator so that *domino waiting* does not entail; particularly if a modified object has caused the system to crash as it can then be avoided in future.

CSP also offers the *catastrophe* operator (represented as a lightning bolt symbol) and the *restart* operator ($\hat{P}$) [4, p.181]. The use of the *catastrophe* operator means that a process behaves like $P$ until a catastrophe occurs, after which it behaves like $Q$.

The *restart* operator in CSP, as its name suggests, restarts a process's behaviour in the event of catastrophe. After each catastrophe $P$ behaves like $P$ did originally (i.e: from the beginning of its definition) [4, p.181].

LOTOS offers the *exit* operator which allows a process to terminate successfully, passing a value to the environment when doing so. The ROOA method uses *exit* functionality in an example to implement the superclass/subclass relationship [11, p.28]. Note the following definition of a superclass which passes its methods to the environment:

```
process Superclass[g](state:StateSort) :  exit(StateSort) :=
  g!selector1!GetID(state) ...;
  ...
  exit(state);
[]
```

```
    g!modifier2!GetID(state) ...;
    ...
    exit(F1(state))
  []
    ...
endproc
```

LOTOS also provides a disable operator (denoted by the symbol [>), which denotes that in the expression ($B1$[> $B2$) process $B1$ may be disabled by $B2$. The LOTOS designers required such a notation in case the normal course of actions in a system became disrupted. The semantics for [> are defined below:

Rule 1: $B1 \xrightarrow{\mu} B1'$ implies $(B1[> B2) \xrightarrow{\mu} (B1'[> B2)$

Rule 2: $B1 \xrightarrow{\delta} B1'$ implies $(B1[> B2) \xrightarrow{\delta} B1'$

Rule 3: $B2 \xrightarrow{\mu^+} B2'$ implies $(B1[> B2) \xrightarrow{\mu^+} B2'$

$B1$ may (rule 1), or may not (rule 2) be interrupted by the first action of process $B2$. Control is irreversible in rule 1 as $B2$ gains control over $B1$. In the rule 2 $B1$ performs an action; if the action is not that of successful termination (using rule 1) then $B2$ survives. If the action of $B1$ is that of successful termination (using rule 2) then $B2$ disappears as the process that $B2$ was expecting to interrupt has disappeared, thus disabling the disabling process $B2$ [1]. It is clear that disabling in LOTOS is taken from the CSP interrupt operator as the rules governing its use are the same. Only the actual symbol used in the language is different, the semantics remain the same.

Asynchronous Process Theory (APT) provides us with an *after* operator [7, p.17]. The process $(P/a.v)$ behaves like $P$ after $v$ has been communicated by the environment on channel $a$. The infinite buffer attached to the input channel of $P$ maintains the value $v$ until $P$ is ready to use it. Effectively, by using *after* we can specify the behaviour of a system before and after a specific action occurs.

The operator *Skip* is also available to Receptive processes (RPT) processes due to the foundations of RPT being based on CSP [6, p.18]. The semantics for *Skip*, when used in RPT, are the same as for CSP [4, p.171].

Receptive Process Theory (RPT) provides *skip guarded choice* which means that the process $(Skip \rightarrow P \mid ?x \rightarrow Q_x)$ eventually chooses to behave like $P$ unless the process's environment supplies it with some input $x$, earlier, whereupon it behaves like $Q_x$ [6, p.23].

## 4.2 Process Behavioral Modification

In most cases the behaviour of a process is regarded as static. However, under certain circumstances this behaviour can be modified. The simplest form of modification is behavioral extension where new branches of behaviour are added to the existing behaviour of the process. This type of process modification is referred to as *incremental modification.* In object-oriented texts this form of modification via reuse is known as strict or incremental inheritance.

## 4.3 Inheritance

Inheritance allows the creation of new encapsulations of behaviour out of old, previously defined, behaviors. In other words, inheritance permits the reuse of components in a specification in order to define a new specification. For example, a process $P$ is defined thus: $P \stackrel{\text{def}}{=} a.b.c.P + d.e.f.0$. Using strict inheritance we can define a new process $Q$ which uses the behaviour of $P$ and then extends it: $Q \stackrel{\text{def}}{=} P + x.y.z.Q$. Certain issues arise which we must address in order to confirm that $Q$ inherits from $P$. Does $Q$ behave like $P$ if branch $a$ or $d$ is chosen? Also, how is the recursion defined if branch $a$ or $x$ is chosen, given that $P$ is defined as the target of the recursion in the original branch $a.b.c.P$ and $Q$ for $x.y.z$? Recursion is an important issue w.r.t inheritance. Consequently an entire section of this paper has been devoted to the study of recursion.

Other forms of inheritance are shown in the following list, each item of which is discussed in the sections that follow:

- **Incremental Inheritance.** Also known as *strict inheritance.* Behaviour is reused and (optionally) extended and behavioural compatibility is guaranteed.
- **Casual Inheritance.** Branch behaviour in a process is modified so that the branch itself changes. Behavioural compatibility is **not** guaranteed. A central theme of my research is to investigate methods for achieving casual inheritance in existing process algebras.
- **Transitive Inheritance.** A hierarchy of inheritance can be defined. $P$ is the parent class of $Q$, which is the parent class of $R$. Therefore, $P$ is a parent class of $R$ and so on. Ancestral

relationships between classes can be defined provided that transitive inheritance is present. Restricting inheritance to just one generation denies this type of inheritance (a restriction imposed in ROOA, [11, p.28]).

- **Multiple Inheritance** states that a child class can have more than one parent class. The behaviour of each parent is present in the child. The child may also, if required, extend that behaviour with some of its own specific behaviour, therefore becoming more specialised.

### 4.3.1 Incremental Modification

The simplest form of inheritance or modification that current process algebras can provide is incremental modification. As illustrated in the introduction to this section, a process's behaviour can be extended by using the process name in another process's definition. The new definition would normally contain more choices of behaviour in order to offer an extended behaviour to the environment than that of the original. Note that choice extension is not necessary. One reason for leaving the behaviour the same and simply copying the original process behaviour is to enable the specification to use that behaviour under a different name, to enhance the readability of an inherited class in a certain part of the specification.

In order to qualify as a valid subclass (i.e: the inherited child of some parent) the behaviour of the child must conform exactly to that of the parent. If the parent fails to provide some action $a$ at a certain point then the child must also fail to provide the same action at that same point during execution. Also, if an action *is* provided then it must also be provided by the child. Basically, the child must be capable of substituting the parent under all circumstances; success or failure. This notion of replacement is known as *conformance* and has been formalised in [2, p.11]:

**Definition 1 Conformance**

*Let $Q$ and $P$ be processes.*

*($Q$ conf $P$)* iff

| | |
|---|---|
| $\forall s \in Traces(P). \forall A \subseteq L(P)$ | If $Q$ fails to offer an action $a$ (after sequence |
| if $\exists Q' \bullet \forall a \in A \bullet Q \xrightarrow{s} Q' \not\xrightarrow{a}$ | $s$) then $P$ must also fail to offer the same |
| then $\exists P' \bullet \forall a \in A \bullet P \xrightarrow{s} P' \not\xrightarrow{a}$ | action $a$ (after the same sequence $s$). |

A common failing of each of the review languages is that neither of them provides for recursive

definitions in inherited processes without some form of syntactic and semantic modification. The *hard-coded* process names that appears at the end of a behaviour branch (such as $P \stackrel{\text{def}}{=} a.b.c.P$) must be replaced with a generic pointer that can be substituted for the calling process's address at run-time. We shall return to this important language restriction in the section on recursion; namely the creation of a *self* operator.

Using LOTOS, Rudkin presents some good examples of strict inheritance. Consider the following simple examples taken from [12, p.416–417].

```
process Buffer4[in,out](q:queue) :self(queue) :=
  in?x:element; self(x appends q)
[]
  [q ne empty] -> out!hd(q); self(tl(q))
end proc

process Buffer[in,out,flush,delete](q:queue) :  self(queue) exit(nat) :=
  Buffer4[in,out](q)
[]
  flush;self(empty)
  []
  delete;exit(length(q))
endproc
```

The primitive *self* is used in the buffer example to enable `Buffer4` and `Buffer` to return to the calling procedures rather than get stuck in an infinite recursive loop. *self* is instantiated with the new name of the caller so that recursion is correctly defined.

The $\pi$-calculus allows for the definition of extended behaviour by including any new behaviour as a parameterised name to a process. Consider the mobile telephones example, given in [10, p.12], as a basis from which inheritance could be defined.

Both CHOCS [14] and HO$\pi$ provide the same features for defining processes by using existing process behaviour as part of their definitions. Further work will determine the extent to which these high-order calculi can be put regarding object-oriented specification.

### 4.3.2 Casual Inheritance

To my knowledge casual inheritance is not supported by any process algebra to date. I define casual inheritance as follows:

> The ability to inherit behaviour from some template parent class and then modify actions sequences embedded within the inherited behaviour.

In effect, to alter the sequence of events defined as a branch of behaviour in the parent class. To illustrate further what I mean by casual inheritance consider the following examples:

$$P \stackrel{\text{def}}{=} a.b.c.P + d.e.f.0$$

$$Q \stackrel{\text{def}}{=} P[R/a] + x.y.z.Q$$

Process $Q$ inherits the behaviour defined in $P$, extends that behaviour with $x.y.z.Q$ and then modifies the branch $a.b.c.P$ with some new behaviour defined in $R$. Process $R$ is defined as: $R \stackrel{\text{def}}{=} a.b.k.Q$. Process $Q$ can be written out in full as:

$$Q \stackrel{\text{def}}{=} a.b.k.Q + d.e.f.0 + x.y.z.Q$$

From this simple illustrative example many aspects of my research can be seen. Questions arising from my work so far on casual inheritance can be identified as:

1. How is recursion defined when the target of the inheritance is unknown?
2. What rules govern the syntax and semantics of the $P[R/a]$ modification?
3. How is the new process verified as being a valid subtype of the inherited parent class?

It is my intention to attempt to answer each of these questions as part of this and future papers. For now, I will briefly discuss the issues associated with each of the three questions.

To answer the first question, a mechanism known as *self* can be introduced. *self* permits recursion within process definitions where the initiator of the action sequence is instantiated at run-time. Think of *self* as a pointer variable instantiated with the address of the process that called the servicing process. More on *self* can be found in [12, p.415] w.r.t LOTOS. The notion of *self* can be equally applied to the other process algebras that are reviewed within this paper.

The second question requires that a form of action renaming or relabelling be carried out to permit our chosen process algebra to enable us to modify processes. At this time work in this

area is in the initial stages and the mechanics of casual inheritance cannot yet be reported. Later work will provide details as to how a process can be modified dynamically or in small increments.

The third and last question relies upon conformance *(Q conf P)* being met by the newly modified process. Actions that are permitted by the parent must be made available by the child process. Actions that are prohibited by the parent must also remain so for the new child process. The example process $Q$, from the previous section, appears to fail the conformance test, as defined by the rule on page 15. Certainly, some of the actions in the original branch $a.b.c$ have been replaced (i.e: $k$ replaces $c$). Perhaps one further question may help to clarify what is meant by *conformance.*

> Is conformance met if only the actions present in the original behaviour are met when those actions also appear in the new child behaviour, rather than any new actions that appear in the action sequence?

### 4.3.3 Transitive Inheritance

Provided that some form of law for recursion can be found to cope with inheritance (i:e *self*) then transitive inheritance is achievable in the review languages. The requirement is for the method caller to be made known to the method owner (a method being perceived as a branch of behaviour). The notion of *self* has already been introduced and I shall leave it for a later section on recursion to explain fully the implications for using *self.*

A process $R$ can be defined as transitively inheriting from $P$ using the following definitions:

$$P \stackrel{\text{def}}{=} a.b.c.P + d.e.f.0$$

$$Q \stackrel{\text{def}}{=} P + 0 + g.h.i.Q$$

$$R \stackrel{\text{def}}{=} Q + x.y.z.R$$

The definition of $Q$ inherits from $P$ and in turn, $R$ inherits from $Q$. The subtype relationship between $P$, $Q$ and $R$ can be written thus: $P \sqsubseteq_{subtype} Q \sqsubseteq_{subtype} R$. Process $R$ can then be written in full:

$$R \stackrel{\text{def}}{=} a.b.c.P + d.e.f.0 + g.h.i.Q + x.y.z.R$$

CCS offers a facility to transitively inherit from a superclass by using the same procedure as that used for strict inheritance. The extension to the language in terms of *self* is still required but

if we assume that CCS can cope with *self* then the syntax of including an agent definition in another agent definition is valid [9, p.20].

CSP also includes the facility to inherit and therefore reuse behaviour. Transitive inheritance is again defined using inheritance as illustrated above [4, p.106].

APT and RPT are both based on CSP and if either of these asynchronous languages can be defined in terms of CSP then the semantics for CSP also apply [7, p.14].

The semantics of LOTOS [1] offer the ability to define one LOTOS process in terms of another. In the ROOA method the LOTOS processes that are used to define inheritance use *exit functionality* to define the exit condition of a process which is treated as an object. ROOA insists that a superclass must terminate successfully and that a child subclass must be non-terminating [11, p.28]. A dilemma results where transitive inheritance is concerned. A subclass that is itself a parent of a subclass, further down the inheritance hierarchy is required (by the rules of exit functionality used in ROOA) to maintain both `exit` and `noexit` functionality. The reader should recognize that a process in LOTOS cannot be defined with two different exit functionalities, hence ROOA does not support transitive inheritance.

The mobility of specifications offered by the $\pi$-calculus [10], CHOCS [14] and HO$\pi$ [13] all allow strict inheritance. As they are all based on the same set of semantics so we assume that these process calculi also allow transitive inheritance to occur.

### 4.3.4 Multiple Inheritance

The theory behind multiple inheritance is that more that one parent class is used to create a new child class which then houses all of the attributes of those parents. The new child class can extend or overwrite the inherited methods of its parents if required but the behaviour of the process must not be compromised in any way (i.e: the conformance rule must hold: *(Q conf P)*). To achieve multiple inheritance the target process being defined must include references to all of the required superclasses.

In CSP we can define multiple inheritance syntactically thus: $R = P \Box Q$, where $P$ and $Q$ are defined elsewhere. $R$ now behaves like $P$ or $Q$ and the choice of behaviour is made by the environment. CCS and the remaining calculi under review (i.e: APT, RPT, $\pi$-calculus, CHOCS and HO$\pi$), which conform to similar semantics, also allow deterministic choice between process behaviour.

In ROOA, Moreira mentions that multiple inheritance is a future goal. However, we may

assume that the same methods found in [11, p.27] will be used again for multiple inheritance (i.e: exit functionality).

## 4.4 Behavioral Overriding

In object-oriented programming languages function overriding is used to modify an existing function that existed in the parent class; in process algebras we call this *behavioral overriding.* Instances of type *child* with the new overridden method will use the new version of the method when required. Instances that used strict inheritance as a means of getting access to the parent's original methods are bound to the parent when the required method is requested; they have no alternative (local) method with the same name to call upon.

Behavioral overriding entails the complete redefinition of some pre-defined behaviour. Some process $P$ may be defined to behave in a certain way according to the choice branches offered by the process. To override a process's behaviour we are required to redefine the complete behaviour of that process. Ideally, we would wish to redefine only the branch that changes; the task assigned to casual inheritance. The ability to override a part of some behaviour enhances reuse and yields a more accurate system than can be achieved by overriding a complete process description.

Problems associated with behavioral overriding occur when a leading action is duplicated and synchronisation with other processes is lost due to the removal of some synchronising action. For example a port name may change and deadlock may be introduced. Any references to other process's ports in a process that has altered will need to be reviewed. In a dynamic system the integrity of the links between processes is under constant review as each process checks to see if the synchronisation links to other processes are still valid. Consider the following example of behavioral overriding:

$$P \stackrel{\text{def}}{=} a.b.c.P + d.e.f.0 + x.y.z.0$$

The new behaviour, using current methods, would require the following expression:

$$P' \stackrel{\text{def}}{=} a.b.c.P + d.e.f.0 + x.y.z.P$$

Note that the last branch of behaviour in $P$ has been redefined as recursive, back to the process $P$.

The complete redefinition of $P$ is unnecessary and increases the chance of error during any copying of the original definition. Assuming that the original specification for $P$ was validated a

simple copying error whilst redefining $P$ could result, leaving $P$ in an unstable state. Also, the new behaviour defined for $P$ could cause an error, possibly causing deadlock with some other component in the system. Whereas $P$ was originally considered to be stable, it is not necessarily the case that $P'$ is to be considered stable.

In the earlier section on *casual inheritance* the same problems were faced. Indeed, it is the goal of casual inheritance to enable us to be able to perform behavioral overriding on a process with the minimum of effort. ROOA documentation provides a means of redefining a process's behaviour [11, p.29]. In ROOA the elimination of the services to be redefined takes place before new versions of those services can be defined.

An auxiliary superclass is used to hold the services that are still required. A new subclass is then defined which includes the auxiliary class and also the original superclass, synchronising on a common port with the auxiliary class to restrict the services available from the superclass (i.e: only the intersection $(\alpha P \cap \alpha Q)$ of services in the super and auxiliary classes are used). New services can then be added as usual. To illustrate this description the LOTOS specification extract, adapted from [11, p.29], is presented.

```
process ModifyRBUFFER2[in,out,delete](q:queue):exit(queue):=
    RBUFFER[in,out](q)
  []
    delete; exit(length(q))
endproc
process RedefineRBUFFER2[in,out,delete,flush](q:queue):noexit:=
    (RBUFFER[in,out,delete](q) |[in,out,delete]|
      ModifyRBUFFER2[in,out,delete](q)
  []
    flush; exit(empty)
  ) >> accept q:queue in RedefineRBUFFER2[in,out,delete,flush](q)
endproc
```

Note that the process `RBUFFER2` adds a new function `flush` to the set of available initial actions. Due to the parallel composition across specific channels, as defined by the parallel composition operator (`|[...]|`) between the processes `RBUFFER2` and `ModifyRBUFFER2`, the scope of `RBUFFER2` is restricted as it cannot engage in any actions not present in `ModifyRBUFFER2` and vice versa.

Due to the mobility of its process definitions the $\pi$-calculus can achieve a degree of behavioral overriding. The example given in [10, p.12] shows how a mobile car radio system can first receive new channels as parameters and then use those channels in future communications. Also, the ability for a process to modify itself and effectively switch from being a 'live' process to being an 'idle' one is illustrated. Obviously, these changes occur during run-time. For a specification change we are restricted by the same constraints that encompass each of the review languages.

CHOCS, with its ability to pass process around as parameters seems to offer a powerful way for processes to be redefined. Like the $\pi$-calculus, CHOCS performs these redefinition tasks during run-time. However, normal inheritance mechanisms would again restrict CHOCS to redefining whole processes at a time rather than in part.

HO$\pi$ is similar to CHOCS and although it too offers the ability to dynamically alter a process by adopting some behaviour passed to it as a parameter the same problems of modifying a subset of the complete behaviour are present [13, p.32].

It is clear that some new language construct that can benefit from object-oriented design and implementation practices is required to enable current process algebras to be reused with more efficiency than is currently the case. My research is seeking to find such a mechanism that can be adopted by a language (as yet undefined) and be used to prove that modification has occurred and that the resulting behaviour is stable and free from error. At the very least the resulting behaviour should conform to those actions that were present in the original process (unless directly affected by the behavioral modification) and offer actions that were offered previously.

# 5 Recursion

Recursion is the final section describing the PARNET. In many ways recursion is *the* most important topic discussed in this paper. Without recursion we are reduced to specifying simple processes that are not capable of modelling objects in the real world. Therefore, without recursion we cannot attempt to apply process calculi specifications to real problems; this relegates process algebras to strictly an academic pursuit.

Inheritance (in its many forms) and recursion are closely linked. The use of the recursion operator *self* in many object-oriented languages (including LOTOS [12]) enables inheritance to return control to the calling function. Under normal circumstances recursion is defined with the specific name of the process 'tagged' onto the end of the process definition. However, to overcome

the problems of becoming trapped in a recursive loop (which results from the 'hard-coded' process name appearing at the end of the description) we require some operator with much more power to keep track of the process that initiated the current branch that the recursion is on and return to that process afterwards. Each of the review languages carry a recursive operator and use it in much the same way; explicitly stating the process to execute at the end of the behaviour branch. In practice a generic modifiable reference variable is needed to substitute the explicit reference used in a process's definition. My research has already examined Rudkin's *self* for LOTOS [12, p.415] and thus far no other operator has been proposed. Work continues in this area.

CCS and CSP both define recursion in the same way, by explicitly naming the continuing process [9, p.57], [4, p.27]. As APT and RPT are based on CSP the syntax and semantics of their recursive operator are similar [6].

Work on LOTOS tends to illustrate the standard form of recursion using explicitly named processes [1, p.35]. However, ROOA documentation uses the `>> accept state:StateSort in SomeClass[g](StateSort)` value passing format to yield control from a class process to some calling process [11, p.29]. Unfortunately ROOA cannot build transitively inherited processes due to the restrictions of `exit` and `noexit` functionality, where a process can only hold one type of functionality depending upon whether it is a superclass or subclass. The inability to support transitive inheritance reduces LOTOS's capability to model all types of object-oriented systems. Only a change to the language of LOTOS (and hence a change to the LOTOS ISO standard) will permit more flexibility to be incorporated into its specifications; like the use of *self*.

Recursion in the $\pi$-calculus is the same as for CCS, with an explicit process name being defined as the last action in a process description; that process name obviously being the same as the process in which it is defined [10, p.12].

Both HO$\pi$ and CHOCS use a similar format for recursion and this format follows all of the languages seen previously in this paper. The same semantic model is used for the vast majority of process calculi. One may argue that there are only a limited number of ways of formalising concurrency in a mathematical context.

# Part III

# Summary and the Future

The work presented here has laid the foundations for a rich and full investigation of how present process algebras can be put to more productive use in the context of object-oriented specification. The final part of this paper concentrates upon the direction of my research and the results that can be expected from it in order to add to the foundations of knowledge in this area.

## 6 Future Work

At present no formal language has shown its suitability for being directly mapped onto an object-oriented specification. Many languages reviewed in this text claim to support object-oriented concepts but, as I have shown, few actually deliver their promises. Indeed, some of the languages under review do not claim any object-oriented support whatsoever, still, this should not stop us from attempting to use them to specify systems that contain reuse and extendibility.

I hope to provide a basis for continued study into the suitability of present languages (with minimal syntactic and semantic modification) that can be adapted for use with object-oriented specification whilst still remaining robust. Ideally, a language such as CSP and LOTOS would be likely candidates as these languages have a full operational semantics. CSP is quite complete in terms of the facilities that it offers. LOTOS has one advantage over CSP, it is the subject of ISO standardisation [5]. Obviously, changes to either of these languages are not a trivial undertaking and would require lengthy discussions with other formal language practitioners in order to justify. The question is not whether we *can* change the language to incorporate object-oriented concepts but whether we *should* change the language!

## 7 Summary and Conclusions

In this paper I have identified and highlighted a correlation between certain aspects of process algebras and both process communication and behaviour. My main concern throughout this text has been the reuse and extendibility of existing process specification. It seems clear that we should try to make some of the concepts of the object-oriented paradigm available to the formal languages used to specify concurrent systems.

I have shown that two main categories of a communicating system exist, system and behaviour (see figure 1). Within the category 'system' communication between processes is the key issue. With reuse and extendibility the nature of communication is affected when synchronisation points between processes change (possibly due to inheritance). To shown that a new process is based on the behaviour of an existing process and can replace that older process we must ensure that the behaviour of the new process conforms to that of the old, hence the use of *conformance* [12, p.420].

Different types of inheritance require different issues to be raised. Each type of inheritance requires recursion to be addressed. Recursion has been a recurrent theme throughout this paper and is central to the inheritance issue. I intend to expand my work in the area of process definition, reuse and recursion with a view to inheritance.

High-order process calculi seem to offer assistance to aid my research. Certainly, passing new channels and even processes themselves can help to remove some of the problems of static interfaces between processes. Further work on the application of these high-order languages to object-oriented specification, particularly inheritance, will help to determine whether the original *low-order* languages, such as CSP and LOTOS may be supplanted by the newer, more dynamic formalisms of CHOCS or HO$\pi$. Clearly, more work is required before a definite decision is made regarding the correct approach to capturing object-oriented specification.

Finally, a brief word on the links between the PARNET main themes; system and behaviour. My work thus far shows that when one makes changes to the behaviour of a process a knock-on effect occurs. I argue that the areas identified in the PARNET diagram (figure 1) help to make the distinction between which areas are affected. PARNET keeps the reader aware of the consequences of alterations in any part of the system. I found that corroborating the details pertaining to each review language made me aware of the similarities that each language has and also how those languages are related across the boundaries that are, normally, regarded as separate issues.

## References

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

[2] E. Brinksma and G. Scollo. Formal notations of implementation and conformance in LOTOS. Memorandum INF-86-13, Department of Informatics, University of Twente, The Netherlands, 1986.

[3] L.D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits.* ACM Distinguished Dissertation. MIT Press, Cambridge: MA, 1988.

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

[5] Information Processing System Open Systems Interconnection. LOTOS–a formal description technique based on the temporal ordering of observational behaviour. Technical Report DIS 8807, International Standardization Organisation, 1987.

[6] M.B. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

[7] M.B. Josephs, C.A.R. Hoare, and H. Jifeng. A theory of asynchronous processes. Technical Report PRG-TR-6-89, Programming Research Group, University of Oxford, Oxford University Computing Laboratory, 11 Keble Road, Oxford. OX1 3QD, 1989.

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

[9] R. Milner. *Communication and Concurrency.* Prentice-Hall, London, 1989.

[10] R. Milner. The polyadic $\pi$-calculus: A tutorial. LFCS Report, Department of Computer Science, University of Edinburgh, October 1991. ECS-LFCS-91-180.

[11] A.M.D. Moreira and R.G. Clark. ROOA: Rigorous Object-Oriented Analysis Method. Technical Report CSM-109, Department of Computing Science and Mathematics, University of Stirling, 1993.

[12] S. Rudkin. Inheritance in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques*, volume VI, pages 409–423. Elsevier Science Publishers B.V: North Holland, 1992.

[13] D. Sangiorgi. *Expressing Mobility in Process Algebra: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, Scotland, May 1993. ECS-LFCS-93-266 (CST-99-93).

[14] B. Thomsen. A calculus of high-order communicating systems. In *ACM 6th Annual Symposium on the Principles of Programming Languages*. ACM Press, January 1989.