DIVISION OF COMPUTER SCIENCE

A Brief History of Time (Before and After Objects) - from the perspective of Process Algebras

(Part I of a series of papers on Process Algebras and their application)

P N Taylor

Technical Report No.238

July 1995

A Brief History of Time (Before and After Objects) - from the perspective of Process Algebras

(Part I of a series of papers on Process Algebras and their application)

Paul N. Taylor
Computer Science Division, School of Information Sciences,
University of Hertfordshire, College Lane, Hatfield Herts. AL10 9AB.

Tel: 01707 284763
Email: comrpnt@herts.ac.uk

July 1995

Abstract

It is intended that this text will lay the foundation for a more approachable path to my work and bears no resemblance to the similarly titled work by Professor S.W. Hawking at Cambridge [8] (although it remains my intention that my own work may be equally as approachable). I intend to introduce the ideas behind three process algebras in such a way as to encourage more people to understand my current research interests and ideas. Rather than launch into a deep philosophical discussion regarding a particular problem I prefer the introductory approach. A first glance at the topic will have a gentler pace. Future material will go into more depth and build upon the ideas presented here.

To set the scene surrounding my own research interests I propose to briefly discuss the motivation behind three process algebras, namely CCS (Milner [13]), CSP (Hoare [9]) and LOTOS (ISO [10]) and highlight the principles upon which they are built. I intend to weave my way through such diverse issues as founding principles, language semantics, application areas, process communication, synchronisation (bi-process and multi-process) and the similarities and differences between the three process algebras. General discussion will centre on modularisation, recursion, action restriction, hidden actions, non-determinism and models of equivalence.

Due to the increased activity in the area of both object-oriented design and programming we can start to foster the idea of modelling object-oriented concepts in the process algebras of CCS/CSP and LOTOS. All three notations pre-date the ideas introduced in object-oriented design so the challenge is to see if these new ideas can be captured. I ask the question, is it possible to provide an abstract specification which is closer to an object-oriented design and implementation than was possible in the past when specifications did not consider either objects or inheritance?

My main focus is on inheritance and the modification and extension of a system. I am particularly concerned with maintaining the integrity of a system whilst allowing changes to take place, both internal to a process and external to the environment as a whole. Problems arising due to nondeterministic processes and synchronising communications will be raised towards the end of the text.

If all goes as planned then the result of this text will be to shed some light on the shady world of processes, synchronising communication and objects. Having lit the way with this first collection of notes on the subject one can proceed further by discussing finer points in future material.

Section:		Page #:
	Introduction	4
Part I		4
1	What is a Process Algebra?	4
2	A Brief History of Process Algebras	5
3	Influences Between the Process Algebras	5
4	What Uses Do They Have?	6
5	What is a Process?	6
	5.1 What are Their Goals?	8
	5.2 Who are Their Rivals?	9
6	Language Similarities	9
	6.0.1 Language Differences	10
	6.1 Formal Definitions	10
	6.1.1 Actions	10
	6.1.2 Distinguished Actions	11
	6.1.3 Processes (CCS Agents)	11
	6.1.4 Environment	11
	6.2 Which Language is Best?	11
7	Basic Concepts	12
	7.1 Event Notation.	12
	7.2 Action Prefixing	13
	7.3 Deterministic Choice	13
	7.4 Nondeterministic Choice	14
	7.5 General Choice	14
8	Building Complex Systems (Process Composition)	15
	8.1 Independent Composition (Interleaving)	15
	8.2 Fully Dependent Composition (Full Synchronisation)	16
	8.3 General Composition	16
9	Process Interaction	17
	9.1 Number of Participants	18
10	Visibility of Interactions	19
	10.1 Explicit Action Concealment	19
11	Internal Actions (I)	21
12	Action Renaming	22
13	Recursion	22
14	Inaction	23
—end of section on basic CSP/CCS.		
15	Successful Process Termination	24
16	Sequential Composition	24
17	Value Passing Between Processes	25
	17.1 Value Passing with Sequential Composition	25
	17.2 Value Input/Output	26
	17.3 Multiple Value/Message Passing	27
18	Communication (Value Passing)	28
	18.1 Number of Participants (Value Passing)	29
	18.2 Direction of Communication (Value Passing)	29
	18.3 Communication Conditions	30
19	Guarded Commands	30
—end of section on value passing calculus of CCS.		
20	Modularisation.	31
—end of section covering complete LOTOS.		

Section:		Page #:
21	Operational Semantics	32
	21.1 Behaviour of Operators Defined by the *traces* Function	32
	21.2 CCS Derivation Trees	33
22	Choice versus Concurrency	33
23	Internal Actions (II)	34
24	Nondeterminism (Basic)	35
	24.1 Refusals (CSP)	35
	24.2 Failures (CSP)	36
	24.3 Divergences (CCS)	36
25	Nondeterminism (CCS)	37
	25.1 Strong Equivalence (~)	37
	25.2 Observational Equivalence (≈)	38
	25.3 Observational Congruence (equality)	39
26	Equivalence (Basic)	39
	26.1 Equivalence (CSP)	39
	26.2 Equivalence (CCS)	40
27	Algebraic Laws (Basic)	40
	27.1 Algebraic Laws (CSP)	40
	27.2 Algebraic Laws (CCS)	41
28	Verification Using Algebraic Laws (Basic)	41
	28.1 Verification Using Algebraic Laws (CSP)	41
	28.2 Verification Using Algebraic Laws (CCS)	42
— end of section on Process Algebras without application.		
Part II		42
29	Object-Oriented Concepts	42
	29.1 What is an Object?	42
	29.2 What is a Class (I)?	43
	29.3 Why are the Last Two Sections in the Wrong Order?	44
	29.4 What is a Class (II)?	44
30	Inheritance and Reuse	45
	30.1 Three Types of Inheritance	47
31	Sub-Type Relationships (Basic)	50
	31.1 Sub-Type Relationships (CSP)	50
32	Method Invocation	52
33	Recursion with Inheritance	53
34	Object Communication	55
	34.1 Why is the System Diagram Partially Connected?	56
— end of section on Process Algebras with Object application.		
35	Conclusions	58
References		59

Introduction

The intentions behind this study are to give you, the reader, some idea as to where process algebras fit into our view of the world when it comes to modelling the 'real world' formally. Often we visualise many components of a *system* but fail to see how those components rely on each other to perform their tasks correctly. We may concentrate on a particular problem area and focus our thoughts on solving a single problem.

With this text I hope to build up a picture of how different process algebras interrelate and how they can be applied to the growth area of object-oriented specification and design. Without some form of formal representation of an object or its communication with other components of a system it is difficult to see how we might progress nearer to some form of implementation of the design.

I will begin from first principles, asking the simplest of questions so that I can be sure that the view of the world that we all share is uniform and consistent. Indeed, if I can explain something in the most basic of terms then I really should understand whatever it is I am talking about. One popular quote from lecturers is that "you never really understand something until you have taught it!". I intend this text to be just that, a platform from which to teach and at the same time learn more about the way that process algebras fit into the scheme of things (in the context of objects and their environments).

I said that we'd start at the beginning (which seems logical) so here goes with the most basic of questions.

Part I

1 What is a Process Algebra?

The reader should be under no illusion at this point (right at the start of the journey), a process algebra is a mathematical language. It allows the specification of ordered sequences of actions (assigned to an entity called a *process* or *agent*) which can then be used to communicate with similarly specified processes in a system (known as the environment). Each process algebra has a formal set of semantics explaining how each of the terms of the language is supposed to act. Process algebras are employed in the specification of concurrent, communicating systems. Although the idea of true concurrency in a single processor environment is perhaps a misnomer as each process interleaves rather than progresses at the same instance in time. Only during communication does a process synchronise with some other entity, otherwise all processes progress together. A process algebra can also be described as a formal reasoning system where variables are used to represent processes. These variables are assigned behaviour which is then executed, one action at a time.

The semantics of a process algebra explain what events occur for each function in the formal language. The communication that processes engage in, either with each other or with the environment, is known as a synchronisation, hence two processes will synchronise with each other. We can visualise this as inter-process communication.

The most common forms of process algebra are:

i) Communicating Sequential Processes (CSP — Hoare 1985 [9])
ii) Calculus of Communicating Systems (CCS — Milner 1989 [13])
iii) Language of Temporal Ordering Specification (LOTOS — ISO 1989 [10])
iv) Algebra of Communicating Processes (ACP — Bergstra/Klop 1984 [2])

For the purposes of this text I shall be introducing the first three languages CSP, CCS and LOTOS. I do not intend to give a complete tutorial as other works are more than capable of

doing so [3,4,9,13]. Various application areas exist for each language. These areas include the specification and design of distributed systems (where each node in the system can be considered an object). Other areas include the specification of network layer protocols, the original purpose of LOTOS [10]. One further area of application is rapid prototyping for numerous communicating systems, from hardware simulation to interface design and implementation.

2 A Brief History of Process Algebras

To put the development of each of the listed process algebras into some chronological perspective consider the following diagram which shows a related pictorial history [7].

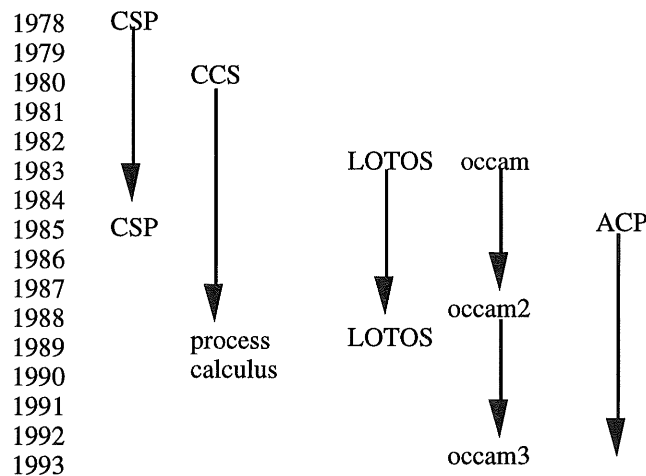


figure 2.1

The further development of each language can be clearly seen from figure 2.1. If we consider the years between either language (specification and programming) we can determine that only a few years separates each new development. At the time of the introduction of CSP a need for some form of formal notation to allow the modelling of communication between interacting process was recognised. Note that LOTOS arrived on the scene after both CSP and CCS. The next section shows the relationship between CSP, CCS and LOTOS.

3 Influences Between the Process Algebras

The origins of LOTOS can be found via a closer examination of both CSP and CCS. Despite the comparatively youthful age of LOTOS it has received ISO (International Standardization Organisation) accreditation and carries the mark of an internationally recognised standard. LOTOS is based on the communication semantics of CSP and elements of CCS to form its own semantics. Figure 3.1 [7] shows how each language borrows elements from the two *grand old men* of process algebras, CSP and CCS.

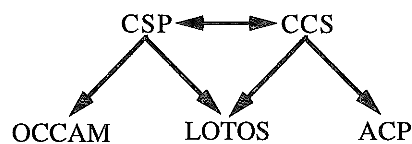


figure 3.1

The concepts of the two founding languages consequently reach far and wide as we explore the different process algebras at our disposal. One important question that we might ask is which language should be chosen for a particular job? A possible answer to this question is to suggest the language which best fits the problem. What if don't know how to use either language? Starting from first principles could be an advantage as no preconceived ideas as to the operation and application of the chosen language exist. Consider the next section.

4 What Uses Do They Have?

By "they" I am, of course, referring to process algebras. An important application of process algebras is their ability to model *real world* objects. What do I mean when I say *real world*? In my opinion I think of the *real world* as a physical entity, containing numerous physical entities; that which surrounds us during our every waking moment (what happens when we're asleep is not decidable!) Objects that we can see, touch and hold could be considered *real world* objects.

Think of an object. What behaviour does it exhibit? If you thought of an animate object, like a bird then you can imagine it flying, swooping and doing the everyday actions that birds engage in. If you thought of an orange then it probably just sat there in you mind and didn't do much until you thought about doing something with it, like eating it. Despite your choice of object it still had attributes. Whether it was animate or inanimate makes little difference, a process algebra can still capture its behaviour and model the communications (interactions) that your object has with its environment.

Now that the discussion has moved on to object interaction let us consider the kinds of interaction that are possible. For an animate object, like the bird, it can instigate an interaction with another object simply because the bird steps through a series of actions; some of which involve other objects, like Mr Worm! If we can model interaction (an inanimate object waiting for some animate object to talk to it) then we can surely reason about the communication going on in our chosen system. So, we can reason about a communicating system, yet another use for process algebras. We can plot an execution path through a series of actions (choosing some action where a choice exists) and observe the outcome. During any progression through a system we regard ourselves as The Almighty, observing all that goes on concerning the external view that each object portrays.

I have briefly discussed the notion of a process and given some hints as to the identification of potential processes around us. Let us now ask the question explicitly.

5 What is a Process?

The originator of CSP, Professor of Computation at Oxford University C.A.R. Hoare, introduces processes in such a way that I cannot find a better way of expressing the same sentiments myself [9]. Here's what Tony Hoare had to say about processes:

> "Forget for a while about computers and computer programming and think instead about objects in the world around us, which act and interact with us and with each other in accordance with some characteristic pattern of behaviour. Think of clocks and counters and telephones and board games and vending machines.
>
> To describe their patterns of behaviour first decide what kinds of event or action will be of interest and choose a different name for each kind."

To illustrate what Professor Hoare says in his opening remarks in the book describing the language of CSP [9] let us think of a object. Let that object be a marker pen that lecturers use

to relay knowledge from their minds to those of their students. In the case of a marker pen there are three kinds of event that we might attribute with such an object. These events are:

i) pickup — the raising of the pen from the podium in the hand of the lecturer
ii) write — the writing of symbols on the board
iii) putdown — the replacing of the pen back on the podium

Using CSP the behaviour of the marker pen can be described formally thus:

(CSP) *E5.1*

$$Pen = (pickUp \rightarrow (write \rightarrow (putDown \rightarrow STOP_{\alpha Pen})))$$

To increase the complexity of the behaviour of the pen we may include additional events that extend the expression in *E5.1*. This extension is shown below in each of the languages that I shall be using throughout this text.

(CSP) *E5.2*

$$Pen = (pickUp \rightarrow (capOff \rightarrow (write \rightarrow (capOn \rightarrow (putDown \rightarrow STOP_{\alpha Pen})))))$$

where the alphabet of *Pen* (the set of valid events that *Pen* can engage in) is defined as:

(CSP) $\alpha Pen = \{pickUp, capOff, write, capOn, putDown\}$ *E5.2.1*

The language of CCS would describe the pen as follows:

(CCS) $Pen \stackrel{def}{=} pickUp\ .\ capOff\ .\ write\ .\ capOn\ .\ putDown\ .\ \emptyset$ *E5.3*

where the sort of *Pen* (CCS equivalent to the CSP alphabet) is defined as:

(CCS) $Pen : \{pickUp, capOff, write, capOn, putDown\}$ *E5.3.1*

LOTOS would use the following behaviour definition to describe the pen:

(LOTOS) *E5.4*

```
process Pen[g] : exit :=
        pickUp;capOff;write;capOn;putDown;STOP
end
```

where the gate list of *Pen* (LOTOS equivalent to the CCS sort) is defined as:

(LOTOS) $g = \{pickUp, capOff, write, capOn, putDown\}$ *E5.4.1*

Note that each of the actions defining the events that *Pen* can carry out are visible to us, the observer. Later we shall discuss hiding events from the observer so that they cannot be influenced by external forces. The reasons behind external influence will become clear as we work through the sections. For now, let us accept that each action of *Pen* is observable from the environment (which is where we are placed as an observer!).

Although we can observe the events of *Pen* we cannot be sure how these events are represented behind the scenes. From the audience's point of view we do not witness *in what way the pen is lifted* off the podium, only that *it is lifted* off the podium. A process is like a black box which we cannot see inside. The environment cannot penetrate the process and consequently has no knowledge about its internal structure. We do not know what the lecturer is thinking or what the pen will write (courtesy of the lecturer) when it gets to the board.

The interface between a process and its environment is defined through a series of ports (in CCS terminology), alternatively known as gates in LOTOS and channels in CSP. We can

represent the *Pen* process pictorially in the following way:

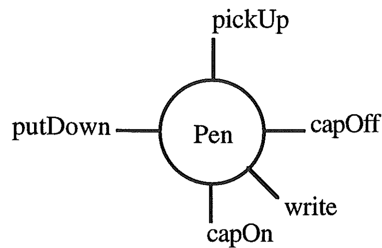


figure 5.1

Inside a process there may be other processes comprising the structure that we are capable of observing. The internal sub-processes will only reveal themselves in the form of ports that are available at the highest level of the structure (the level which is visible to the environment). An entire system is made up of communicating processes (which are discussed in detail in later sections). Consider the following example in **our** *real world.* It illustrates the hiding of processes from the environment (namely hidden from us).

> In a restaurant you interact (communicate) with the waiter when you place your order. After a short time (or a long time in the case of an expensive restaurant — the waiting time being proportional to the cost of the meal coupled with the intensity of the lighting) the meal arrives. Now consider the activities that are taking place behind the scenes whilst the meal is being prepared. As the observer you are unaware of the full range of activities which you have initiated. The interface that you interact with is the waiter, who in turn communicates with the chef and so on.

Now you can see the architecture of sub-processes and hidden communications that exist within the confines of the *black box* that is the restaurant kitchen. Moving on, let's think about what the process algebras aim to achieve.

5.1 What are Their Goals?

A simple break down of each process algebra will show the intentions of the languages under review. For CSP we are provided with a simple model which provides us with an elegant mechanism for defining new operators.

The language of CCS offers a minimal set of operators and a simple model of communication (using complementary named ports). The central concept of CCS is its model of communication which has certain restrictions when compared to CSP and LOTOS (which share a common view of communication). For example, CCS cannot cope with multi-process synchronisation. Only one co-agent can communicate at any one time in CCS. Regardless of this restriction CCS offers a flexible platform from which to build communicating systems.

The main aim of LOTOS, as defined by the ISO ODP (ISO Open Distributed Processing) document is the formal specification of communication protocols to aid in the development of distributed systems and computer networks. These systems would use the standard seven layered organisation model, common throughout the networking community. LOTOS is described as an FDT (Formal Description Technique) as it imposes strict rules in the layout and structure of specifications written using its notation. A LOTOS specification closely resembles program source code in its layout and structure.

5.2 Who are Their Rivals?

Within the realms of formally specifying concurrent systems CSP, CCS and LOTOS have the specification of concurrent systems pretty much to themselves. However, neither language should rest on its laurels. Together with the rivalry between each process algebra other notations do provide some modelling capabilities which could substitute or complement the capabilities of our three review languages [7]. These alternate notations are:

> i) Z and VDM — Both of these notations are abstract and non-constructive, which implies that any events specified within the confines of either notation has no temporal ordering. Any event can be called in any order, consequently we could not use either Z or VDM to enforce a model of a strict sequence of events.
>
> ii) Petri-Nets — This particular notation has one main advantage over either CSP, CCS or LOTOS. Petri-Nets can model true concurrency and causality (the ordering of events) whereas neither of our reviewed process algebras offer true concurrency. Despite the obvious advantage when trying to model *real world* concurrency Petri-Nets have no strict algebraic theory supporting them. Therefore we consider them to be primitive and open to abuse when used for anything but the simplest of system.
>
> iii) Regular Expressions and Finite State Automata — Both of these notations are simple and should be familiar to the reader. We will not illustrate the syntax except to say that FSA are derived from corresponding Regular Expressions. A FSA can provide another medium for representing the actions taken by a process, given some behaviour for that process. Again, there are advantages towards incorporating them into a larger model of the systems we might attempt to specify but due to their simplistic language we find that concepts such as non-determinism (unpredictability given a choice of events) cannot be modelled and is best left to other more capable languages.

Despite some challenges to our three choice process algebras we find that no one single alternative can threaten the completeness offered by the formalism of either CSP, CCS or LOTOS. With this justification for their use in mind we shall continue.

6 Language Similarities

The founding principle of each of the languages CSP, CCS and LOTOS is the concept of an atomic action (known in CSP as an event). At any one time the execution of a process by way of an action cannot be interrupted by another process or by the environment. The action will be allowed to complete before any interruption occurs. Each language obeys this principle and communications rely upon it to ensure that they can synchronise before another process takes control of the system. It may help the reader at this point to think of the environment where the processes execute as some single processor system where each process can take it in turns to execute.

CSP, CCS and LOTOS also view a process as a collection of actions. That is, processes are constructed from sequences of actions. Indeed, the behaviour of a process is determined by the actions contained within it.

The various operators that our languages provide enable us to construct more complex processes. A simple process which contains four discrete actions is enhanced by adding some form of choice operator into its temporal behaviour (action sequence). By applying the choice operator we can define new behaviour from the original process.

6.0.1 Language Differences

Together with language similarities we find that care must be taken to identify the differences between the languages. Fundamental to the idea of CSP and LOTOS is the idea of visible communications that can be determined by the environment. Without this principle no multi-way synchronisations would be possible as interactions between processes could not be observed by the environment.

CCS is different as it internalises communications so that the environment cannot tell what interactions took place. The tau action (τ) represents some form of internal communication but does not give any hints as to what explicit actions actually synchronised. If we restrict a CCS process's action then the environment cannot interfere with that action. However, if no internal communication occurs based upon that restricted action then the process itself will be in a state of deadlock because no communications will be able to occur with any action within the process, causing deadlock. CSP (and LOTOS) assume that some internal communication will force the action to progress and therefore they do not deadlock when their actions are restricted in the same way. Consequently, CSP and LOTOS can distinguish between the interaction and restriction of process actions whereas CCS cannot. These issues underlie the model of communications that each language adheres to. Without the observability of synchronising actions CSP and LOTOS would not be able to offer further communications to multiple processes and therefore engage in multiple synchronisation.

Between the syntax of CSP, CCS and LOTOS certain common operators appear. Despite the syntax appearing to be the same the semantics of the operators can be radically different. Consider the internal action. In CSP it does not appear at all in the sequence of observable actions (the *traces)* of a process, but in CCS and LOTOS the internalised actions are evident as place holders in the sequence.

Similar concepts may also be expressed differently so again we must be aware of subtleties between the semantics of the languages, regardless of their syntax.

6.1 Formal Definitions

For each language we find that certain issues are fundamental to their structure. With either CSP, CCS or LOTOS the idea of an atomic action is central to the notion of process behaviour. The term *atomic* is used to give the idea that the action cannot be further subdivided into sub-actions. Each action must complete before some other process action can execute.

6.1.1 Actions

Together with the concept of atomic actions each algebra insists that actions are uniquely named so as not to cause confusion within the confines of the same process and system. Imagine the consequences of dual naming within a process. For example, how is a process to know which version of action a is to be executed and synchronised upon?

Although atomic actions and unique naming is common to all three notations CCS uses a different approach to ensure process interaction (via synchronisation). CCS uses complementary naming to enforce synchronisation where the actions a and $\overline{a}$ will synchronise when agents $A1$ and $A2$ are composed together; expressed as $(A1 \mid A2)$.

6.1.2 Distinguished Actions

Special actions are resident in each process algebra. These actions have meanings other than simple process behaviour and progression of that behaviour.

(CSP)	*✔ [5, p.23]* *Successful Process Termination* *if P1 ; P2 and ✔ $\notin$ traces(P1) then* *P2 cannot start execution.*	*E612.1*
(CCS)	*τ (tau action) [13, p.39]* *representing internal communication,* *synchronisation and nondeterministic choice when* *τ is an initial action in a choice expression*	*E612.2*
(LOTOS)	***i*** *(hidden action) [3, p.31]* *used to show some hidden action within a process trace* *δ (exit symbol)* *defined as exit — $\delta \rightarrow$ STOP which signifies successful process termination.*	*E612.3*

6.1.3 Processes (CCS Agents)

From the discussion so far we can determine that processes are constructed from actions and references to other processes. Processes interact with their environment and where necessary can pass and receive values as dynamic data storage. With the inclusion of data a process can be envisaged as a single encapsulated entity containing behaviour (its actions) and state (the dynamic storage continually being passed through the system or back to itself) which makes the process a close relative of an object in the object-oriented sense. The interface ports to the process are through its available (visible) actions.

6.1.4 Environment

The environment of a process, or indeed a system, is one of the places where we, the system's users reside. Also a process's environment may include other processes or *real world* entities. The environment can only see processes via their observable actions (unrestricted behaviour). Any process composed with its environment forms another process; forming an infinite number of potential processes as we encapsulate more and more processes to make larger, more complex processes in turn. To view the system we would have to remove ourselves to a higher level, above the current encapsulation so as to see the boundary of the system. It is only from the boundary that we, as observers, would be able to interact with all visible process actions within the system.

6.2 Which language is Best?

For each language we can list many attributes, some of which support the particular language and others which do not. Consider this simple list [7, p.55] of *for and against* points that cover the broad areas of each notation.

(CSP - For)	• + Simple language semantics and verification semantics • + Can be implemented in OCCAM without too much translation
(CSP - Against)	• - Are the laws of CSP complete and/or consistent?
(CCS - For)	• + Elegant • + Minimal Language
(CCS - Against)	• - Minimal Language • - Equivalence versus Congruence (similarity versus substitution)
(LOTOS - For)	• + Tool Support (syntax can be executed) • + *Allows constraint style specification
(LOTOS - Against)	• - Some verbose syntax (akin to programming language syntax)

* Constraint Style Specification is where behavioural constraints are represented as processes. Composition is used to impose constraints on interaction points between processes. It is *the* most abstract approach and is implementation dependent.

7 Basic Concepts

I now move on to discuss the notation for each language and the semantics behind the notation. My journey through each of the next few sections will try to introduce you, the reader, to each language in parallel so that a grounding in CSP, CCS and LOTOS can be gained. Like the previous sections, we'll start from the absolute basics and build up the knowledge-base from there.

7.1 Event Notation

Regardless of whether we talk about events, actions or gates we are still talking about the same concept. The progression of a process is through the execution of actions that define a process's behaviour. We model events (the actions of a process), processes (collections of actions to form complete behaviour), alphabets (valid actions pertaining to a process) and process interfaces (the entry point into a process) in each language accordingly [7, p.6]:

Event Notation:

(CSP)	*events (e1,...,en)*	*E71.1*
(CCS)	*actions (a1,...,an)*	*E71.2*
(LOTOS)	*gates (g1,...,gn)*	*E71.3*

Process Notation:

(CSP)	*process (P1,...,Pn)*	*E71.4*
(CCS)	*agent (A1,...,An)*	*E71.5*
(LOTOS)	*behaviour expression (B1,...,Bn)*	*E71.6*

Alphabet Notation:

(CSP)	*alphabet* (αP)	*E71.7*
(CCS)	*sort (L(A))*	*E71.8*

(LOTOS)	*events or actions?*	*E71.9*

Interface Notation:

(CSP)	*channels (c1,...,cn)*	*E71.10*
(CCS)	*ports (p1,...,pn)*	*E71.11*
(LOTOS)	*gates (g1,...,gn)*	*E71.12*

With these basic components we are in a position to model simple actions for a process by prefixing that action onto a process expression.

7.2 Action Prefixing

Each process algebra incorporates action prefixing as a way of expressing one action occurring before some other behaviour or choice of actions. In fact one action prefixing another (and so on) is the recursive definition of a sequence of actions. An everyday example of action prefixing is the order of events that you perform before you leave the house for work each day. One task is completed, then another and then another. Only one task may be completed at any one time and upon reflection these tasks, when listed together, form the sequence of actions that make up your morning ritual.

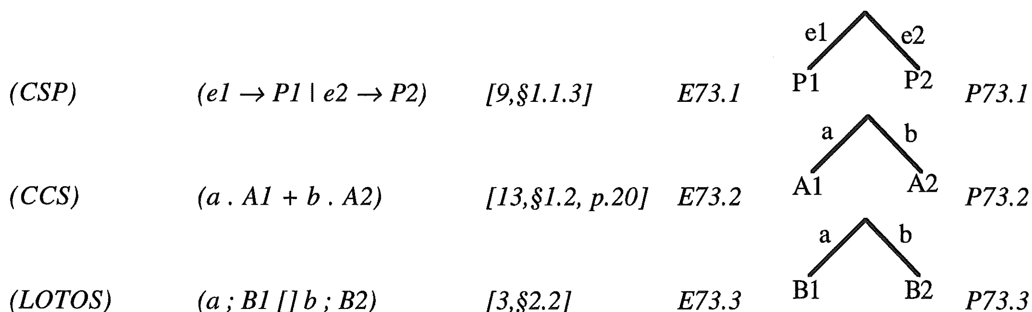
Consider the action prefixing syntax for each language:

(CSP)	$e \rightarrow P$	*[9, §1.1.1]*	*E72.1*
(CCS)	$a\ .\ A$	*[13,§1.3, p.27]*	*E72.2*
(LOTOS)	$g\ ;\ B$	*[3,§2.2]*	*E72.3*

The action *e, a* or *g* occurs, followed by some behaviour. After an action has taken place other processes in the system may proceed, holding up other processes whilst they do so.

7.3 Deterministic Choice

The term *deterministic choice* refers to the influence that the environment has over a possible choice of distinct actions that a process offers. If a process is deterministic then we can *determine* its behaviour. Processes that offer a deterministic choice of actions are considered to be stable as their outcome is known (i.e: predictable). Consider the syntax for deterministic choice that each language uses and also the graphical representation of that choice.



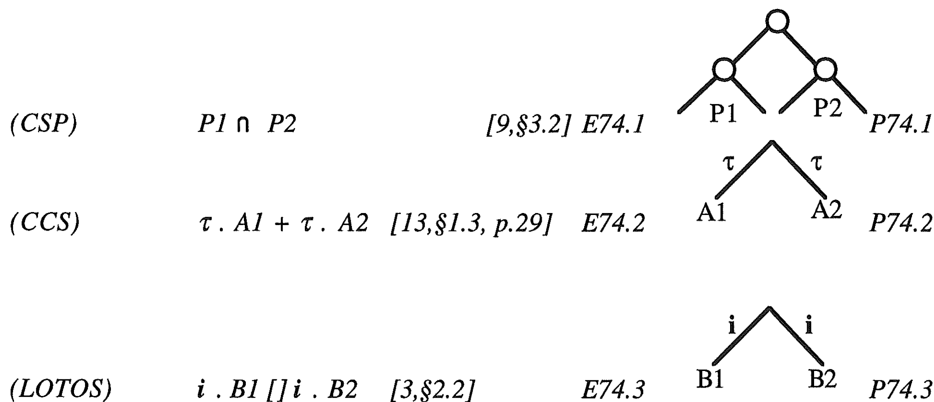
(CSP)	$(e1 \rightarrow P1 \mid e2 \rightarrow P2)$	*[9,§1.1.3]*	*E73.1*	*P73.1*
(CCS)	$(a\ .\ A1\ +\ b\ .\ A2)$	*[13,§1.2, p.20]*	*E73.2*	*P73.2*
(LOTOS)	$(a\ ;\ B1\ []\ b\ ;\ B2)$	*[3,§2.2]*	*E73.3*	*P73.3*

Notice how the expressions are now starting to increase in complexity! Here we see how each language copes with offering one of two choices to the environment. The diagrams *P73.1* → *P73.3* show graphically the choice between the two actions. To use the diagrams (referred to in CCS as derivation trees) we simply start at the root (top) and choose a branch to descend. Each path down the tree denotes some action being taken by the process. With all three

languages the choice of one action over another causes the non-chosen branch to disappear, hence that particular choice is no longer available until the process starts from the root again. In the examples above the first action choice determines the subsequent behaviour of the process. This difference between available choices once a particular choice has been made helps to further underline the subtleties between the semantics of the three languages, something which is not apparent from the syntax alone.

7.4 Nondeterministic Choice

If we are offered choice by a process and the outcome cannot be predicted then we can assert that the behaviour of the process is nondeterministic (i.e: unpredictable). The environment cannot influence the outcome of some decision regarding choice within a nondeterministic process. The mechanism which decides the outcome of the choice within the process is not observable it just happens and the process proceeds down the path that it, itself, has chosen. Note that if one choice is continually chosen over another then there is nothing that we, as observers, can do about it. There is no concept of fairness in either CSP, CCS or LOTOS.



Notice how the CSP notation differs from both the CCS and LOTOS notation. In CSP the operator ⊓ signifies the nondeterminism of a choice of actions for a process. However, CCS and LOTOS continue to use the same choice operator as before (see section 7.3) but prefix their sequence of actions on either side of the choice operator with a special action that dictates internal choice (namely τ or i). Again, neither choice can be influenced by the environment nor can some sense of time be imposed by the environment. All that we know as observers is that one of the actions (branches) will get chosen at sometime in the future. We'll only know the outcome of the nondeterministic choice when we observe the state of the process at sometime after the choice has been made.

7.5 General Choice

General choice can be regarded as an option between one choice of behaviour over another for which the environment can influence that choice. The environmental influence implied here must be taken on the very first action of either process in the choice expression. Subsequent behaviour of the system is determined once the first action has been taken. Here are the different syntax expressions for each language.

(LOTOS) *B1 [] B2* *[3,§2.2]* *E75.3*

The CSP notation again provides us with a clue as to the difference between the expression in *E75.1* and that of *E74.1*. The choice between either process behaviour (*P1* or *P2*) is ours, that is, the environment's and will determine what the process does next. Again, note the reuse of the CCS choice operator + and that of LOTOS *[]*. These last two languages make full use of their limited choice operators and consequently care must be taken to ensure that the context within which they are used is fully understood; is the process offering deterministic or nondeterministic choice?

So far we have only illustrated the foundations of the three languages using very simplistic processes which would not model anything but the simplest object in the *real world.* It would be more interesting if we were to build more complex processes. For this task we need to introduce more language constructs.

8 Building Complex Systems (Process Composition)

The behaviour of the *Pen* process that was specified earlier (*E5.1* onwards) has no connection with other processes. Only processes that are composed together can communicate. Each of our target languages uses synchronisation in order to communicate. Similarly named actions come together to communicate (i.e: synchronise). A simple example involves two processes (we'll call them Bonnie and Clyde).

> Bonnie and Clyde turn to one another and shout "Oh no! It's the cops. Looks like they've rumbled us this time. I bet it was those darn kids who couldn't keep a secret!"

Hence, Bonnie and Clyde engage in a synchronised event. Processes, like Bonnie and Clyde synchronise together. From the previous discourse we get the impression that our villainous couple shout together in unison. Indeed, this is what they did. The semantics of CSP, CCS and LOTOS state that both processes simultaneously engage in the communication. Using our gangster example both villains would have to turn to each other at the same time; using some nefarious form of ESP perhaps. Here's a simple diagram showing the act of synchronisation that Bonnie and Clyde engaged in:

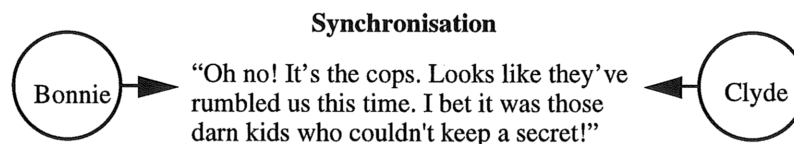


figure 8.1

Different forms of synchronisation exist. Let us examine them next.

8.1 Independent Composition (Interleaving)

It is possible to compose processes together to form a complete system whilst still enforcing strict divisions between the processes. Interleaving is the independent (concurrent) progression of processes related by |||. It might prove necessary to share some resource (cunningly disguised as a process) between several processes so that no interaction occurs between the different camps sharing the resource. In CSP and LOTOS the same symbol and semantics are used, namely:

(CSP) *P1 ||| P2* *[9,§3.6]* *E81.1*

(LOTOS)	*B1* ||| *B2*	*[3,§2.4, p.34]*	*E81.2*

The operational semantics of ||| enforce interleaving but ||| actually states that *P1* and *P2* will progress concurrently. CCS requires an invariant to guarantee interleaving, namely that no two processes should share complementary actions, if they do then they have a duty to synchronise if those particular actions are restricted. Non-restricted actions can occur at any time, singularly, or they can come together to form a synchronised communication between two processes.

(CCS)	*(A1 | A2)*	*iff* $L(A1) \cap \overline{L}(A2) = \{\ \}$	*[7,p10]*	*E81.3*

Note that the rules surrounding CCS communication are stricter than those for both CSP and LOTOS. CCS requires the synchronisation of complementary ports between processes. In *E81.3* the way to specify interleaving is to ensure that no complementary port exists within the sort of the other composed agents (remember, in the language of CCS an agent is a process). If we call upon the services of a Venn diagram we can see the meaning of the rule introduced in *E81.3*. For $L(A1)$ use the sort $\{a,b,c\}$ and $\overline{L}(A2)$ use the sort $\{d,e,f\}$.

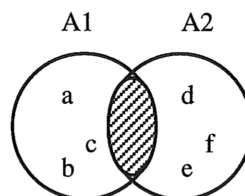


figure 81.1

Provided that there are no actions common to both agents (no actions in the shaded area) then the interleaving of both *A1* and *A2* is guaranteed as they will both continue separately.

8.2 Fully Dependent Composition (Full Synchronisation)

At the other end of the synchronisation scale we find full synchronisation. Each action in the composed processes must be performed by each process simultaneously. The rules for this type of composition are very strict and consequently CCS suffers from its own requirements concerning complementary port synchronisation. Consider the CSP and LOTOS syntax and semantics for full synchronisation.

(CSP)	*P1* || *P2*	*[9,§2.2]*	*E82.1*
(LOTOS)	*B1* || *B2*	*[3,§2.4, p.34]*	*E82.2*

In CCS we find fully dependent composition difficult to model due to the way that CCS conceals interactions between processes. In short, CCS interactions are not observable by the environment, all that occurs is the τ action which does not yield any information as to the cause of the interaction only that some form of synchronisation has taken place.

8.3 General Composition

A more flexible form of synchronisation is provided via general composition which allows processes to interact via a subset of their actions. Other, non-similar actions are performed

independently. With the following examples each process synchronises on the intersection of common events in each process's respective alphabet.

(CSP)	*P1 ‖ P2*	*[9,§2.3]*	*E83.1*
	where $\alpha P1 \cap \alpha P2$ contains the events that are synchronised.		
(CCS)	*A1 | A2*	*[13,§2.5,p.46]*	*E83.2*
	where A1 and A2 may, or may not interact on complementary actions.		
(LOTOS)	*B1 |[g1,...,gn]| B2*	*[3,§2.4]*	*E83.3*
	where B1 and B2 must synchronise on the gates g1,...,gn.		

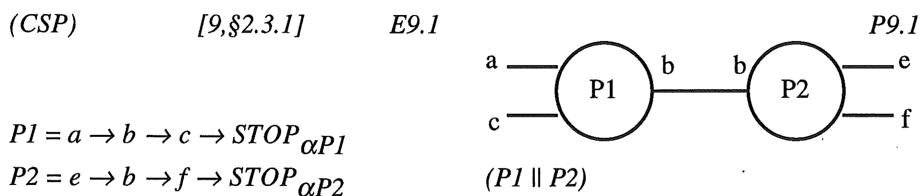
In the LOTOS example *(E83.3)* note that we can use general composition *|[g1,...gn]|* to specify both of the other types of composition (interleaving ||| and full synchronisation ||).

Consider:	*B1 ||| B2 is equivalent to B1 |[Ø]| B2*	*[17,§2]*	*E83.4*
	B1 ‖ B2 is equivalent to B1 |[$\alpha B1 \cap \alpha B2$]|	*[17,§2]*	*E83.5*
	Therefore, both ||| and ‖ can be defined in terms of |[...]|		

9 Process Interaction

Both CSP and LOTOS synchronise on identically named actions. However, as I have briefly discussed, CCS uses the notion of complementary actions as its basis for communication (e.g: interaction between α and $\overline{\alpha}$).

(CSP) *[9,§2.3.1]* *E9.1* *P9.1*



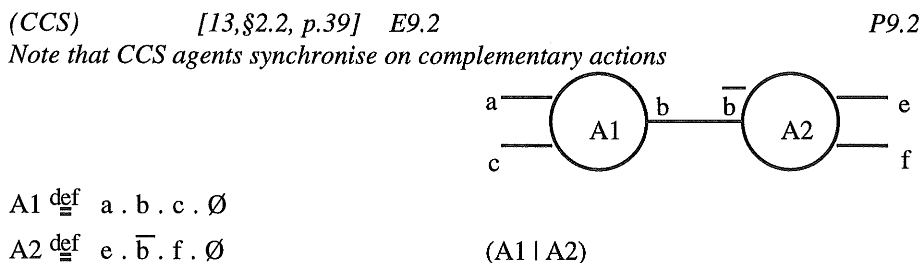
$P1 = a \rightarrow b \rightarrow c \rightarrow STOP_{\alpha P1}$

$P2 = e \rightarrow b \rightarrow f \rightarrow STOP_{\alpha P2}$

(P1 ‖ P2)

(CCS) *[13,§2.2, p.39]* *E9.2* *P9.2*

Note that CCS agents synchronise on complementary actions

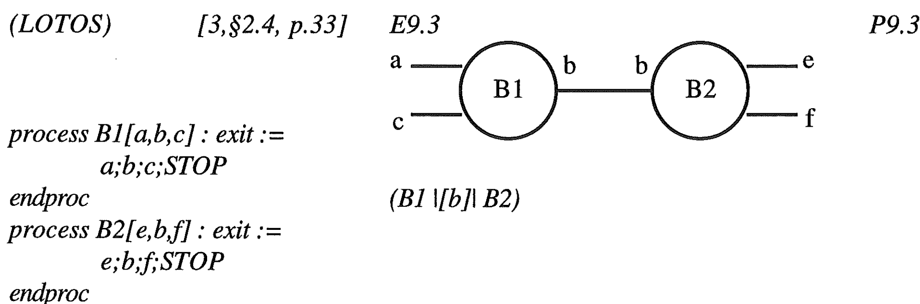


A1 $\stackrel{def}{=}$ a . b . c . Ø

A2 $\stackrel{def}{=}$ e . $\overline{b}$. f . Ø

(A1 | A2)

(LOTOS) *[3,§2.4, p.33]* *E9.3* *P9.3*




```
process B1[a,b,c] : exit :=
        a;b;c;STOP
endproc
process B2[e,b,f] : exit :=
        e;b;f;STOP
endproc
```

(B1 |[b]| B2)

One further point regarding process interaction. Actions are used for the communication between the processes, not the process names, these are simply used as a means of encapsulating sequences of actions and choice.

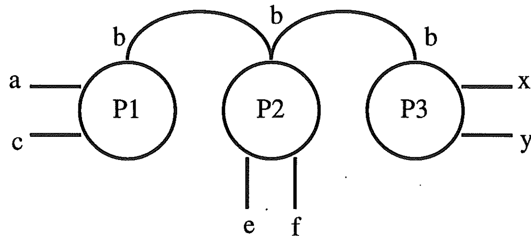
9.1 Number of Participants

In section 6.0.1 (Language Differences) I underlined the fundamental difference between CSP, LOTOS and CCS. Remember that the model of communication used by CSP and LOTOS is radically different from that of CCS. Recall that CCS cannot cope with multi-way communication (the interaction of more than two processes) because interactions in CCS are hidden from the environment (although work has been carried out to redress this issue [1]). Restricted actions are stopped from occurring (from the environment's point of view) and only internal communication will allow a process to continue. In both CSP and LOTOS it is valid to compose two or more processes in the following way.

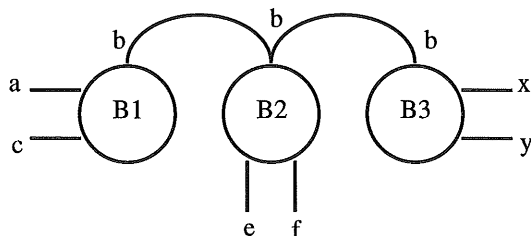
(CSP)	*(P1 ‖ P2) ‖ P3*	*[9,§2.3.1]*	*E91.1*
(LOTOS)	*(B1 |[g]| B2) |[g]| B3*	*[3,§2.4,p.33]*	*E91.2*

Note that ‖ and |*[...]*| are associative. Using the process definitions in *E9.1* and *E9.3* let us extend the previous composition by adding a third process *P3* and *B3* respectively to each example shown, as follows.

(CSP)　　$P3 = x \rightarrow y \rightarrow b \rightarrow STOP\ \alpha P3$　　*E91.3*
(P1 ‖ P2) ‖ P3　　*P91.1*



(LOTOS)　　*process B3[x,y,b] : exit :=*　　*E91.4*
x;y;b;STOP
endproc
(B1 |[b]| B2) |[b]| B3　　*P91.2*



Consider the following potential problem. What would happen to the system if either one of the three processes failed to supply the action *b*? Would the system be able to cope with such a failure of one of its components or would it deadlock? I shall return to this issue later in the text.

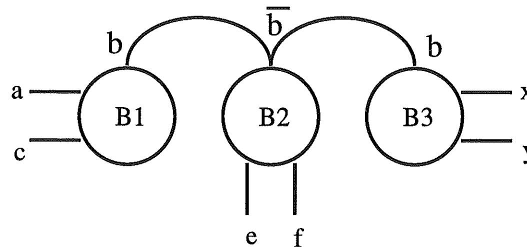
Now consider an attempt to build a similar system in CCS. As we have already seen, CCS cannot cope with anything other than bi-party interaction (a maximum of two processes)

[13,§2.2, p.39]. If we attempt to compose *A3* with the existing *(A1|A2)* system what will happen?

(CCS) $A3 \overset{def}{=} x \,.\, y \,.\, b \,.\, \emptyset$ E91.5

$((A1 \mid A2) \mid A3)$

P91.3



Due to the way that CCS models communication we are left in somewhat of a dilemma. With which complementary action does action $\bar{b}$ synchronise? Is it *B1(b)* or *B3(b)*? The answer is either! The problem lies with the nondeterministic nature of the expression in E91.5. The environment could also play a part in the synchronisation if action *b* is not restricted. The system illustrated in *P91.3* will not function deterministically!

10 Visibility of Interactions

I reiterate the comments of the previous section by reminding the reader about the visibility of interactions between CSP and LOTOS processes. As observers and members of the environment of a process we can see which actions are part of any communication between the processes [9,§2.3.1],[3,§2.4, p.33]. We know what's going on! We can see the specific actions involved in the communications!

By now I have explained the main difference between CCS and the two other process algebras presented in this text. The internalisation of CCS communications prevents outside influence from the environment in such a way as to require internal synchronisation to permit any process to continue. Consequently, it is simpler to deadlock a CCS process than a CSP or LOTOS process. Again, CSP and LOTOS share a common view of communication and set this view aside from the notion of restricting actions so that the environment cannot influence them. With CCS the observer will see a τ action in place of a synchronising communication and that is all. As observers we only know that a process has performed an interaction by referencing the new state of the process and the existence of the τ action in the traces of that process.

10.1 Explicit Action Concealment

Concealing process actions has the power to stop the environment from influencing, participating with or observing those actions. If we chose to build complex processes made up from sub-processes then we hide the communications between the sub-processes using action restriction (referred to as action hiding in LOTOS). The interface to an externally visible process is made up from the unrestricted actions of that external process and those actions that are unrestricted belonging to the sub-processes that lie beneath it. Let's look at an example from each language.

(CSP - Concealment) $P1 = a \rightarrow b \rightarrow c \rightarrow STOP\ \alpha P1$ *[9,§3.5]* *E101.1*

$P2 = b \rightarrow f \rightarrow x \rightarrow STOP\ \alpha P2$

$P3 = x \rightarrow y \rightarrow b \rightarrow STOP\ \alpha P3$

$((P1 \parallel P2) \backslash \{b\}) \parallel P3$

Within the composition *(P1 || P2)* the concealed action *b* allows *P3* to proceed independently if capable of doing so. However, *P3* requires the synchronisation with action *x* at the end of *P2* so it will not proceed until *P2* reaches that point. One aspect of CSP that has also been captured in LOTOS is the ability to progress should synchronisation fail. In the example in E101.1 process *P1* could drop the action *b* and continue if *b* fails to be offered by any other process in the system. If *b* is present elsewhere then P1 will wait but if *b* is not a valid action of any other process in the system then P1 will simply pass over its own *b* action and continue with action *c*.

If we fail to restrict action *b* in the composition of *P1* and *P2* then the entire system will deadlock because all three processes will be looking to synchronise on *b*. The reason lies behind the fact that *P1* and *P2* will hang on *b*, waiting for *P3(b)*. Incidentally, *P3(b)* cannot occur until *P3(x)* occurs earlier in its temporal behaviour. *P3(x)* is subsequently waiting for *P2(x)*, which itself cannot occur until *P2(b)* occurs. Aha! A catch-22 situation. A classic circular argument involving two parties saying "I'm not moving until you move and you won't move until I do!"; rather reminiscent of Irish peace negotiations and those of the former Yugoslavia. What about the LOTOS equivalent, action hiding?

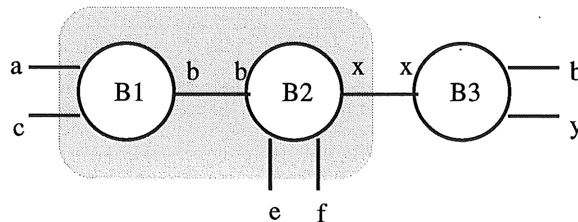
(LOTOS Action Hiding) *[3,§2.5]* *E101.2*

```
process B1[a,b,c] : exit :=
        a;b;c;STOP
endproc

process B2[b,f,x] : exit :=
        b;f;x;STOP
endproc
```

E101.3

```
process B3[x,y,b] : exit :=
        x;y;b;STOP
endproc

process B4[a,c,f,x,y] : exit :=
        hide b in
        (B1 |[b]| B2) |[x]| B3
endproc
```

P101.1



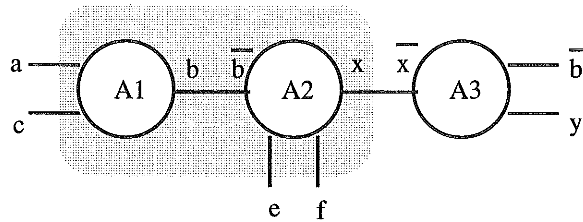
The shaded area of *P101.1* denotes the restricted synchronisation that we, as observers have no access to. Notice from the diagram in *P101.1* that *B3* is not actually related to *B1* and *B2* by action *b*. The use of *|[g1,...gn]|* precludes *b* in *B3* from interacting with *B1* and *B2*. The ***hide*** operator keeps the environment from influencing and observing the interaction of *b* between *B1* and *B2*.

Now let us consider the mechanics of CCS action restriction. In CCS restricted actions are prevented from occurring by influence from the environment. We can therefore prevent interaction by restricting certain actions. This restriction forces processes to wait until internal communications occur which allow them to proceed.

(CCS Action Restriction) *[13, p.40]* *E101.5*

$$((A1 \mid A2) \backslash \{b\}) \mid A3$$

P101.2



where the agents are defined as follows:

$$A1 \stackrel{def}{=} a\,.\,b\,.\,c\,.\,\emptyset \qquad A2 \stackrel{def}{=} e\,.\,\overline{b}\,.\,f\,.\,x\,.\,\emptyset \qquad A3 \stackrel{def}{=} \overline{x}\,.\,y\,.\,b\,.\,\emptyset$$

From the CCS example above note that $\backslash\{b\}$ also restricts the complementary action of the defined set element ($\overline{b}$). Therefore we assume that some restriction expression $\backslash\{\alpha\}$ also restricts its complement $\backslash\{\overline{\alpha}\}$; there is no need for it to appear within the restriction set.

11 Internal Actions (I)

Within the process algebras internal actions can result from:

- interaction within the language of CCS
- concealment in CSP and LOTOS (using the ***hide*** operator).

In CCS and LOTOS internal actions may also be explicitly defined as action prefixes using the special actions τ and $\boldsymbol{i}$.

(CCS Silent action τ) *[13, p.39]* *E11.1*

$\tau\,.\,A$ *e.g:* $a\,.\,\tau\,.\,A$

P11.1

(LOTOS unobserved action prefix $\boldsymbol{i}$*)* *[3,§2.2]* *E11.2*

$\boldsymbol{i}\;;B$ *e.g:* $a\,.\,\boldsymbol{i}.\,B$

P11.2

Whereas the internal actions appear within the derivation trees for both of the previous languages a CSP trace would not contain any evidence of an internalised action, it simply disappears from the trace as if it never happened.

12 Action Renaming

If we are required to define one action in terms of another then it is possible in all three languages to rename actions with new identities. Renaming alters the action names of a process but leaves the behaviour of the process the same. In [17] I use action renaming to reuse certain generic components of a case study system and then incorporate action restriction to specialise those renamed processes.

(CSP Symbol Change)	*f(P)*	*[9,§2.6]*	*E12.1*
(CSP Process Labelling)	*l : P*	*[9,§2.6.2]*	*E12.2*
(CCS Relabelling)	*A[f]* where *[f] is of the form [newLabel/oldLabel]*	*[13, p.32]*	*E12.3*

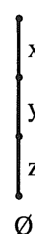
Example: $A1 \stackrel{def}{=} a . \overline{b} . c . \emptyset$

$A2 \stackrel{def}{=} A1[x/a, y/b, z/c]$

P12.1
A1

Ø

P12.2
A2



Ø

(LOTOS meta-language relabelling) *[3,§2.6, p.35]* *E12.4*

process B[x/a,y/b,z/c]

or using parameter shifting during recursion.

```
process B[a,b,c] : noexit :=
        a ; b ; c ; B[c,a,b]
endproc
```
E12.5

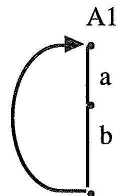
Note that the LOTOS example of shifting parameters effectively renames the meaning of the actions when the next recursive call to the process is made.

13 Recursion

So far, I have only dealt with processes that terminate (cease to exist) after one pass through their sequence of actions. In reality, the *real world* objects that we trying to model continues through numerous (possibly infinite) iterations. A machine repeats a certain sequence of actions many times during the course of its life. To model useful processes each algebra allows the name of the process to be used at the end of an action sequence, allowing the process to repeat. Formally recursion is defined as a fix point expression.

(CSP)	$\mu X . X$	*[9,§1.1.2]*	*E13.1*
(Example)	$P1 = a \rightarrow b \rightarrow c \rightarrow P1$		

(CCS)	$fix(X = a.X)$ $A \stackrel{def}{=} fix(X = a.X)$	[13,§2.9]	E13.2
(CCS Example)	$A1 \stackrel{def}{=} a\ .\ b\ .\ c\ .\ A1$ $\equiv$ $fix(A1 = a\ .\ b\ .\ c\ .\ A1)$		Ex13.1
			P13.1



(LOTOS)	*process B[g] : noexit :=* *g ; B[g]* *endproc*	[3,§2.6, p.36]	E13.3

Note that for all three languages, CCS, CSP and LOTOS, for unique solutions under recursion the recursion must be guarded by a preceding action prior to the recursive call, otherwise there may be many possible solutions.

14 Inaction

The word itself says quite a lot about what we can expect from a process which contains inaction; not a lot! Inaction is defined formally as being a process that cannot do anything. It cannot proceed in any further actions or interactions.

(CSP)	STOP	[9, p.25]	E14.1
(CCS)	$\emptyset$	[13,§2.4]	E14.2

note that $\emptyset$ *is not basic.* $\emptyset \stackrel{def}{=} \Sigma_{i \in \{\}} A_i$ *[7, p.16] which states that there are no more actions to be undertaken by the process A.*

(LOTOS)	STOP	[3,§2.2]	E14.3

And so ends the first instalment of this text. With the inclusion of *inaction* we are at the end of the sections introducing basic CSP and CCS, although we haven't quite finished with LOTOS yet!

—end of section on basic CSP/CCS.

15 Successful Process Termination

To show that a process has completed its task successfully we need some way of specifying task completion.

(CSP)	*SKIP*	*[9,§5.1]*	*E15.1*
(LOTOS)	*EXIT*	*[3, p.36]*	*E15.2*

The CCS equivalent is a trifle more complicated but it can be achieved. The label $\overline{done}$ indicates the termination of an agent. The CCS definition of successful termination can be formalised thus:

(CCS) *[13,§8.2, p173]* *D15.1*

P terminates successfully if, for every derivative P′ of P

(a child process of P) $P' \xrightarrow{\overline{done}}$ *is impossible.*

No more actions of P if terminated.

Note that the definition in *D15.1* does not state that *P* **should** terminate, only that when it performs the label $\overline{done}$ that it **does** terminate. This label is required so that other processes that may be waiting on the successful completion of *P* can recognise that the completion of *P* has occurred and start executing themselves.

16 Sequential Composition

The idea behind a sequential process is that events occur one after another. Sequential composition entails the starting of some second process after the successful termination of a leading process. We can model sequential composition in CSP and LOTOS, whereas CCS (a simpler language; as we are finding out) involves a little more work.

(CSP) *P1 ; P2* *[9,§5.1]* *E16.1*

Example:

$P1 = a \rightarrow b \rightarrow c \rightarrow STOP_{\alpha P1}$

$P2 = d \rightarrow e \rightarrow f \rightarrow STOP_{\alpha P2}$

where a valid set of traces P1 ; P2 would be $\langle a,b,c,d,e,f \rangle$

(LOTOS) *B1 >> B2* *[3,p.36]* *E16.2*

For a comparative model in CCS we require a new operator titled *Before*, which is defined thus:

(CCS) *[13, p.173]* *E16.3*

$A1 \ Before \ A2 \stackrel{def}{=} (A1[b/done] \mid b \,.\, A2) \setminus \{b\}$

where $A1 \xrightarrow{\overline{done}} A2$ *signifies successful termination*

Note that the CCS definition in *D15.1* is used in *E16.3,* together with action renaming and restriction to enforce internal interaction between only *A1* and *A2*. With this synchronisation

we can follow a successful path from the execution of one process to another.

17 Value Passing Between Processes

The processes that we have seen so far only communicate with each other and their environment on simple terms. I will now discuss an extension to these terms to include parameterisation. The aim is to pass values along with the interactions that processes engage in. Each notation allows a certain degree of parameterisation.

(CSP Equations)	$N(v1,...,vn) = P2$	*[9,p.32]*	*E17.1*
Example:	$P(x,y) = a(x) \rightarrow b(y) \rightarrow STOP_{\alpha P1}$		*Ex17.1*
(CCS Parameterised Constant)	$N = (v1,...,vn) \overset{def}{=} A$	*[13,§2.8]*	*E17.2*
Example:	$A1(x,y) \overset{def}{=} a(x) . b . A2(y)$ $A2(y) \overset{def}{=} c(y) . \emptyset$		*Ex17.2*
(LOTOS Parametric Processes)	*process N[...](v1:tn,...,vn:tn) : f :=*	*[3,§5.4]*	*E17.3*
Example:	*process B[g](v : t) : noexit :=* *g ? x : t; B[g](x)* *endproc*		*Ex17.3*

In LOTOS types *t* are defined using the algebraic specification language ACT ONE. The functionality *f* of a process *B* (the operator that appears after the gate list *[g]* and value parameters *(v : t)*) is described as either:

- *noexit* functionality, which indicates that the process never terminates successfully
- *exit* functionality, which indicates that a process successfully terminates
- *exit(t1,...,tn)* functionality, which indicates that the process will terminate and also offer data values for receipt by other processes.

For LOTOS I shall discuss the notion of passing values upon successful termination of a leading process in a sequential composition.

17.1 Value Passing with Sequential Composition

In LOTOS values may be passed between processes. That is to say that values may be passed from *B1* to *B2* in the context of *B1 >> B2*. Processes which pass values are to end with functionality *exit(t1,...,tn)*. For example:

process B1[g](v:t) : exit(e1,...,en) := ... *endproc*	*[3,§5.5.1]*	*E171.1*

Processes that are to receive values are to be specified in a complementary state to the sender and take the following form:

process B2[g](v:t) : f := [3,§5.5.2] E171.2

...

>> accept X1:t,...,Xn:t in B1

endproc

For the transfer of information to be successful the parameters and their types from source to target must match exactly. If the transfer succeeds then *B2* will continue, *B1* will cease to exist in the system but its legacy (the passed information) will continue to be resident within the system for as long as *B2* continues to function.

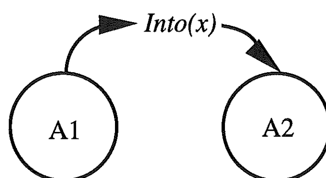
CCS value passing on sequential composition is defined as a process *Ax*, terminating by yielding up its exit values via the label $\overline{res}$. The CCS operator *Into* provides the value passing medium. Consider the definition *D171.1*.

(CCS) [13,§8.2, p.174] D171.1

$$A1\ Into(x)\ A2 \overset{def}{=} (A1 \mid res(x)\ .\ Q) \setminus res$$

where A2 refers to the result x, passed from A1.

P171.1



17.2 Value Input/Output

To allow a process to model the acceptance of values from the environment, as well as from each other in the same system we need to be able to specify that a value *v* enters the process through some recognised port. Here's how the three languages cope with value input and output.

Value Input:

(CSP Channel Input)	$c\ ?\ v \rightarrow P(v)$	[9,§4.2]	E172.1
(CCS Input Port)	$p(v)\ .\ A(v)$	[13,p.17]	E172.2
(LOTOS Input Gate)	$g\ ?\ v : t\ ;\ B(v)$	[3,§5.1.3]	E172.3

The post-fix value *v* within each example (shown as *P(v)*, *A(v)* or *B(v)*) illustrates that the input value has the ability to change the local state of the process after the recursive call at the end of each expression. Now consider the syntax for output:

Value Output:

(CSP Channel Output)	$c\ !\ v \rightarrow P(v)$	[9,§4.2]	E172.4
(CCS Output Port)	$\overline{p}(v)\ .\ A(v)$	[13,p.17]	E172.5
(LOTOS Output Gate)	$g\ !\ v : t\ ;\ B(v)$	[3,§5.1.3]	E172.6

Notice that although it is not necessary it is still possible for the value *v* to remain with the process despite being output to some other interacting process or the environment.

One more type of value passing remains to be discussed and it is only applicable within the realms of LOTOS.

17.3 Multiple Value Message Passing

In LOTOS a gate *g* may be followed by a list *c1,...,cn* of offered (*!e* output) and accepted (*? v:t* input) events. The general form of these lists is shown as:

(LOTOS) *g c1,...,cn ; B* *[14,§3.3.10, p.27]* *E173.1*

Three forms of communication exist which follow the general form. These are:

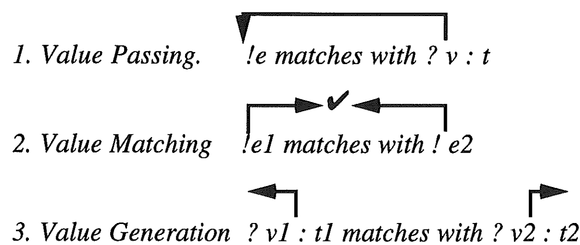


figure 173.1

Each method of value passing can be illustrated using the following simple examples. We tend to think of the first example, value passing, as the value passing synchronisation that takes place between two processes. Note that passing values restricts the number of participants in a communication (see section 18.1). Value passing in CSP and LOTOS requires an output and input communication to contain similar sorts along the same channel.

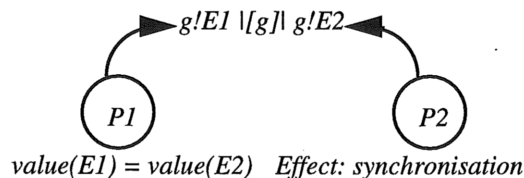


figure 173.2

Value matching brings two processes together on the same channel (port) with equivalent values being passed onto that channel (output signals from bother processes).

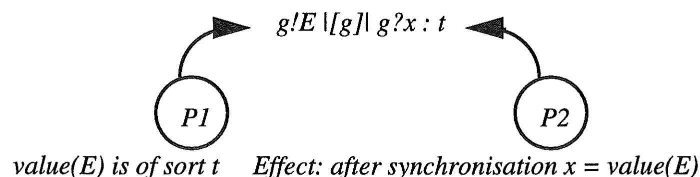
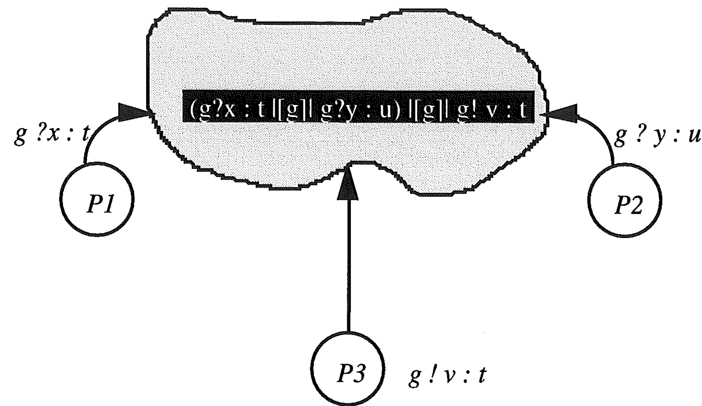


Figure 173.3

Value generation is the most complex form of value passing as it requires two processes to wait on some channel for values of a certain sort (type) from some third process. The third process

supplies the signal, allowing the first two processes to synchronise. The variables from the three processes will all be equivalent and of the same sort.



sort t ≡ sort u Effect: after synchronisation x = y = v, where v is some value of sort t

Figure 173.4

18 Communication (Value Passing)

Communication within the process algebras is strictly synchronous. Two processes come together to synchronise on some common action. The actual event occurs simultaneously at both ends of the interaction. When we think of this type of synchronous communication we tend towards a *signal* → *acknowledgement* model. This view is incompatible with the semantics of the process algebras that are presented in this text. If we think of both sides of the interaction engaging at once then we've got the idea!

If required we can model asynchronous communications. A simple solution requires some form of buffer where the final target process is removed from the source's immediate environment. Basically, using E.W. Dijkstra's metaphor, "we have solved yet another problem in Computer Science by adding one more level of abstraction". For example, consider the following buffers to allow us to model asynchronous communication.

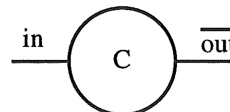
(CSP Buffer) $BUFFER = P\langle\rangle$, *where* [9,§4.2, p.138] E18.1

$$P\langle\rangle = left\ ?\ x \rightarrow P\langle x\rangle$$

and

$$P\langle x\rangle\hat{\ }s = (left\ ?\ y \rightarrow P\langle x\rangle\hat{\ }s\hat{\ }\langle y\rangle \mid right\ !\ x \rightarrow Ps$$

P18.1



(CCS Buffer) [13,§1.2] E18.2

$$C \stackrel{def}{=} in(x)\ .\ C'(x)$$

$$C'(x) \stackrel{def}{=} \overline{out}(x)\ .\ C$$

With the buffer process inserted between two communicating processes the source

process never explicitly requires an acknowledgement with the target process, only the buffer process. Consequently, the source can continually send messages without requiring any acknowledgements from the target, only the buffer. This is actually a bit of a hack! Synchronisation is still occurring but if we restrict our view to just the source and target then it appears that messages are being sent out without the need to actually synchronise with the final target process.

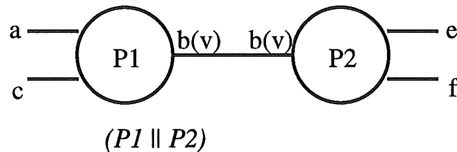
18.1 Number of Participants (Value Passing)

In previous sections I have mentioned the number of processes that are synchronised during one value passing interaction. In CSP and CCS only bi-party communication [9,§4.2, p.134] [13, p.15] **with value passing** is possible, whereas in LOTOS full multi-party communication is possible with value passing. CSP adopts a different strategy with normal synchronising communications without value passing. CSP allows multi-party communication in this instance. CCS disallows any form of multi-party communication or value passing. Consider further the differences between the three languages, as illustrated in the following examples.

(CSP) *E181.1* *P181.1*

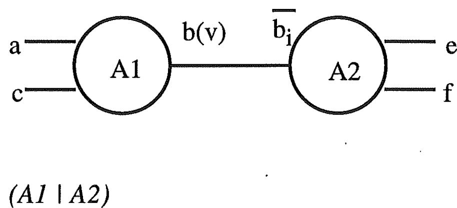
$P1 = a\ ?\ v \rightarrow b(v) \rightarrow c \rightarrow STOP_{\alpha P1}$

$P2 = e \rightarrow\ b\ !\ v \rightarrow f \rightarrow STOP_{\alpha P2}$



(P1 || P2)

(CCS) *E181.2* *P181.2*



$A1 \stackrel{def}{=} a\ .\ b(V)\ .\ c\ .\ \emptyset$

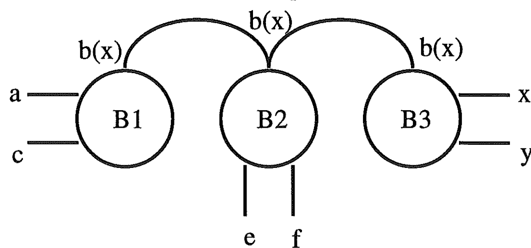
$A2 \stackrel{def}{=} e\ .\ \overline{b_i}\ .\ f\ .\ \emptyset$

where data item $i \in$ data set V

(A1 | A2)

(LOTOS)

B1 |[b(x)]| B2 |[b(x)]| B3 *E181.3* *[3,§2.4, p.33]* *P181.3*

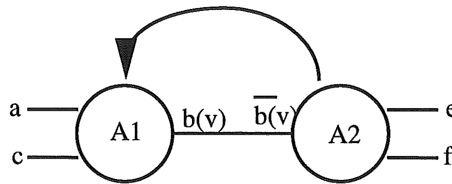


18.2 Direction of Communication (Value Passing)

Both CSP and CCS use unidirectional communication; one direction only from source to target [9,§4.2, p.134] [13, §1.1]. However, LOTOS, having a slightly more complex communications model, allows for multi-directional communication [3, §5.1.3]. For a better

understanding of this type of communication consider the direction of the arrows in the value matching and value passing forms of message passing in section 17.3 (figure 173.1).

(CSP and CCS) *E182.1* *P182.1*

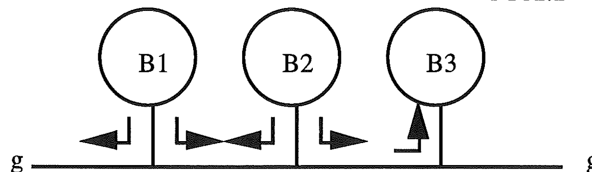


A1 $\stackrel{def}{=}$ *a . b(v1) . c .* Ø

A2 $\stackrel{def}{=}$ *e .* $\overline{b}$*(v2) . f .* Ø

(A1 | A2)

(LOTOS) *E182.2* *P182.2*



process B1[g](v1 : t) : noexit :=
g ! v1 : t ; B1[g](v1)
endproc

process B2[g](v2 : t) : noexit :=
g ! v2 : t; B2[g](v2)
endproc

process B3[g](v3 : t) : noexit :=
g ? v3 : t ; B3[g](v3)
endproc

((B1 |[g]| B2) |[g]| B3)

18.3 Communication Conditions

When communications occur in CSP, CCS and LOTOS processes they may be guarded [9,§5.5,p186], [13,§3.2, p65], [3, §5.2.1]. This means that certain Boolean conditions have to be met before the communication will occur. The next section introduces guarded commands.

19 Guarded Commands

During the course of process execution we may need to enforce a logical path through its sequence of actions, rather like the path that a computer program will choose based upon the evaluation of some Boolean expressions. Each target language supplies the means to guard action sequences using similar Boolean expressions. If the expression evaluates to *true* then the subsequent action sequence is executed, just like our program source code.

(CSP Condition) $P1 \lhd b \rhd P2$ *[9,§5.5, p.186]* *E19.1*
Translation: *"P1 if b else P2"*
Note: Many users of CSP use an if...then...else construct, the result of the previous expression would be:

if b then P1 else P2

(CCS Condition) *if b then A1* *[13,§2.8, p.55]* *E19.2*

(LOTOS Guarded Expression) $[b] \rightarrow B$ *[3, §5.2.2]* *E19.3*
Translation: *"if b then B"*

— *end of section on value passing calculus of CCS.*

20 Modularisation

Up until this point in the text I have discussed very little about objects, except in the case of processes purporting to be objects in view of their state and behaviour. The concept of modularisation is a key issue supporting the modelling of objects and their environments. Modularisation supports large scale specification. We need to be able to model large systems as these are common in the *real world* that we are attempting to model.

Each process algebra provides notation for modularising specifications to help break them up into component parts and make them more readable.

(CSP Process) *E20.1*

$$P1 = a \rightarrow b \rightarrow P1$$
$$P2 = b \rightarrow c \rightarrow P2$$
$$P3 = c \rightarrow d \rightarrow P3$$
$$P4 = ((P1 \parallel P2) \setminus \{b\}) \parallel P3$$

(CCS Agent) *E20.2*

$$A1 \stackrel{def}{=} a \,.\, b \,.\, A1$$
$$A2 \stackrel{def}{=} \overline{b} \,.\, c \,.\, A2$$
$$A3 \stackrel{def}{=} \overline{c} \,.\, d \,.\, A3$$
$$A4 \stackrel{def}{=} ((A1 \mid A2) \setminus \{b\}) \mid A3$$

(LOTOS Process Definition) *[3,§5, pp.47-48]* *E20.3*

```
process B[a,b,c,d] : noexit :=

        process B1[a,b] : noexit :=
                a ; b ; B1[a,b]
        endproc (* B1 *)

        process B2[b,c] : noexit :=
                b ; c ; B2[b,c]
        endproc (* B2 *)

        process B3[c,d] : noexit :=
                c ; d ; B3[c,d]
        endproc (* B3 *)

endproc (* B *)
```

(LOTOS Specification) *[3,§5, pp47-48]* *E20.4*

```
specification System[a,c,d] : noexit :=
        behaviour
                hide b in
                (B1 |[b]| B2) |[c]| B3
endspec
```

Each process can be encapsulated within a sub-system which itself can be further encapsulated in other subsystems. Layering a specification keeps the information within it hidden from undesirable influence. The specification can therefore be made more resilient and robust. We adopt the same strategy for modularising a formal specification as we would for modularising program code.

— end of section covering complete LOTOS.

21 Operational Semantics

The operational semantics for each process algebra dictates **how** each operator will behave. For example, in CSP the observational behaviour of a process is defined by the set of all traces it may perform.

A trace is a finite sequence of event symbols taken from the definition of a process. The trace sequence only records events visible to the environment (i.e: unrestricted events). At some arbitrary point in time the trace sequence will contain only those events that have occurred. Therefore, a set of traces will hold all possible trace sequences of a process. Due to recursion the set of traces for a process may be infinite.

21.1 Behaviour of Operators Defined by *traces* Function

In CSP the formal definition of operations on the *traces* function [5, §1.5] are defined as follows:

- *traces(STOP) = {<>}.* *[5,§1.8.1]*
  The empty sequence is the only valid sequence for a process that does nothing. No actions occur, therefore no set of action sequences is possible except the empty sequence which belongs to every set of sequences.

- *traces(e → P) = {<>} ∪ {<e>^t | t ∈ traces(P)}* *[5,§1.8.1]*
  The set of sequences for *P*, which initially only includes the empty set, is made up from the union of all possible sequences of *P* concatenated onto the end of the initial action sequence *e*.

- *traces(P1 [] P2) = traces(P1) ∪ traces(P2)* *[5,§1.8.1]*
  Deterministic choice over either process *P1* or *P2* yields the union of both possible sets of sequences for each branch of the choice.

- *traces(a → STOP [] b → c → STOP) = {<>,<a>,<b>,<b,c>}* *[5,§3.3.3]*
  Choice between one action sequence or another yields a set of all possible sequences that either choice can engage in. Note that no repetition occurs in the set as duplicate elements are ignored (re: two occurrences of <> from each part of the choice expression).

The previous definitions of traces apply soley to that function in the CSP language. CCS has its own method of tracing the actions and choices available within in its own process definitions.

21.2 CCS Derivation Trees

The behaviour of a CCS agent is expressed as a property of its derivation tree. Transitions within the tree represent the ability to perform an action; denoted by:

$A \xrightarrow{a} A'$, where a may be the internal action τ which is unobservable to the environment [7, p.33].

Derivation trees collect all possible agent derivations (surprisingly enough!) and are similar in concept to the CSP *traces* function. As with the contents of a set of sequences derivation trees may be infinite in length. An incomplete tree is termed a *partial* tree. Study the following derivation tree in figure *212.1*. Notice how the choice between the behaviour in the expression places us on one branch of the tree or another. Both choice and recursion can be modelled using a tree of this sort. Recursion, although not covered here but later in the text, links the base of a branch to the root. The root always appears at the top of the diagram (think of the tree as inverted). For this particular derivation tree (in figure 212.1) we shall refer to the marker pen example process that was introduced back in section 5.

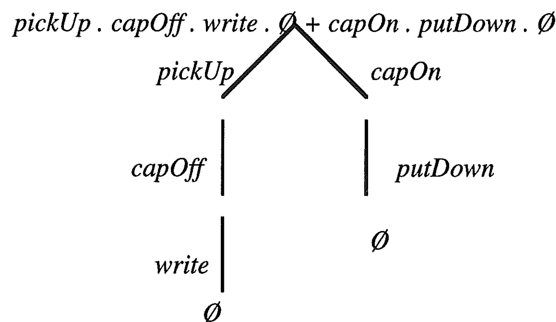


figure 212.1

Each branch of the tree represents a single action. It is customary to rewrite the remaining actions that are left at the junctions along a particular branch, such as:

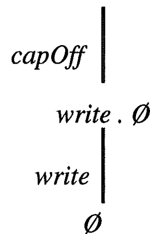


figure 212.2

Note that the remaining actions at each junction in figure *212.1* were left out to aid clarity. I did not want to swamp the reader with too much information first time around!

22 Choice versus Concurrency

Within the syntax of CSP and CCS it is not possible to determine some of the properties that distinguish one type of process from another. Consider the following examples.

(CSP) E22.1
The results of the traces function in CSP cannot identify choice from composition.

[9,p.120][7,p.35]
$traces(a \rightarrow b \rightarrow STOP \; ||| \; c \rightarrow STOP) = \{\langle\rangle, \langle a\rangle, \langle a,b\rangle, \langle a,c\rangle, \langle a,b,c\rangle, \langle a,c,b\rangle\}$

which is equivalent to the result obtained by:

[9,p.120]
$traces\,(a \rightarrow b \rightarrow c \rightarrow STOP \;[]\; c \rightarrow b \rightarrow STOP) = \{\langle\rangle, \langle a\rangle, \langle a,b\rangle, \langle a,c\rangle, \langle a,b,c\rangle, \langle a,c,b\rangle\}$

Likewise, a CCS derivation tree cannot distinguish between choice and composition. [13,p.69]

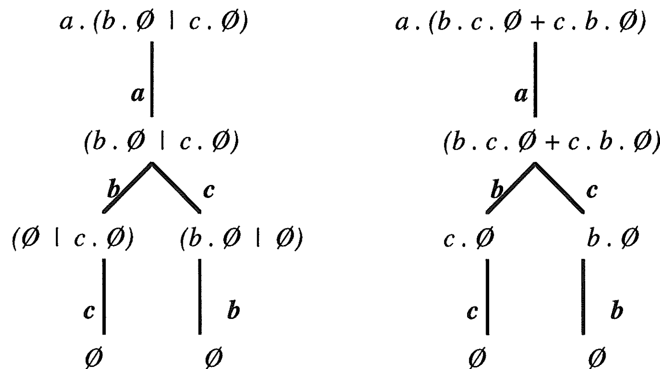


figure 22.1

The similarity between the trees gives us an indication that the behaviour of the composed processes is identical. Agents in CCS can be tested for equivalence based upon the state of their respective derivation trees (more about process equivalence later).

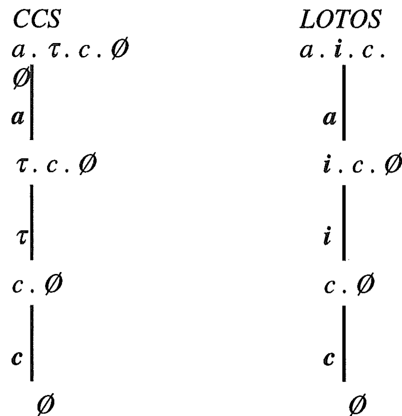
23 Internal Actions (II)

Actions that are restricted are considered to be internal actions. CCS and LOTOS have specific actions that are automatically considered to be internal. In CSP hidden actions are those that are internalised. Although internal actions are valid actions for a CSP process they never appear in the traces of that process, not even as place holders for some unknown action (as they would in a CCS derivation tree). From the point of view of the *traces* function internal actions are not registered.

$(CSP) \quad traces((a \rightarrow b \rightarrow c \rightarrow STOP) \setminus \{b\}) = \{ \langle\rangle, \langle a\rangle, \langle a,c\rangle \}$ *[9,§3.5.3]* *[E23.1*

Note that the restricted action *b* is conspicuous by its absence in the set of traces for *E23.1*. Both CCS and LOTOS include their internalised actions in their derivation trees or traces that are derived from the behaviour expressions of their own processes [13, p.40][3, p.35]. Consider the following diagram which shows how internal actions feature in both CCS and LOTOS derivation trees. Officially, CCS does not have a *traces* function, it only uses derivation trees as a method of expressing the performed actions of a process.

P23.1



Note that under normal circumstances, despite the fact that the internal actions are included in any traces that CCS and LOTOS processes might have, they are ignored and consequently have no overall effect upon the behaviour of the process. However, one important case when internal actions do have an effect is when they are initial (first) actions. To explain this important case in more detail we need to look again at nondeterminism.

24 Nondeterminism (Basic)

If we say that a process is nondeterministic we imply that the behaviour of the process cannot be predicted from its specified sequence of actions. Nondeterministic processes can contain choices between specific actions or start with a hidden action. Examples of nondeterministic processes usually contain choice but initial hidden (internal actions) will still yield the same amount of nondeterminism as we, the observers, cannot predict if (and when) the process will progress. A hidden initial action has the power to stop a process from doing any action, alternatively it may proceed, we have no way of predicting what it will do!

CSP process traces (*traces(P)*) do not capture nondeterminism, Here is an example:

(CSP) $traces((a \rightarrow STOP) \sqcap (b \rightarrow STOP)) = \{ <>,<a>,<b>\}$ *[9,§3.4]* *E24.1*

(CSP) $traces((a \rightarrow STOP) \mathbin{[]} (b \rightarrow STOP)) = \{ <>,<a>,<b>\}$ *[9,§3.4]* *E24.2*

As we can see from the set of traces in *E24.1* and *E24.2*, they are the same.

24.1 Refusals

For each operator the *refusals* set records sets of events that may lead to deadlock when offered to the environment. These events may be restricted initial actions or actions that do not occur in some behaviour that we are trying to justify as equivalent to another process. Note the definitions for the function *refusals*.

(CSP) $refusals(P1 \sqcap P2) = refusals(P1) \cup refusals(P2)$ *[9,§3.4..1]* *D241.1*
where the set of refusals for a nondeterministic choice of actions of P1 or P2 is the same as the union of the set of both the refusals of each process.

(CSP)	*refusals(P1 [] P2) = refusals(P1)* $\cap$ *refusals(P2) [9,§3.4.1]* *where the set of refusals for the deterministic choice of actions for P1 or P2 is the same as the common set of refusals of both processes.*	*D241.2*

Example:[7, p.37]i) refusals((a$\rightarrow$ *STOP)* $\sqcap$ *(b* $\rightarrow$ *STOP)) = { {}, {a}, {b} }* *Ex241.1*
ii) refusals((a$\rightarrow$ *STOP) [] (b* $\rightarrow$ *STOP)) = { {} }* *Ex241.2*

The refusals set of Ex241.1 contains both action a and action b because due to its nondeterministic state it may refuse to offer either action. As observers in its environment we have no control over which action it will carry out first.

Because Ex241.2 is deterministic we can predict which action will occur first.

Note that a deterministic process is one that **never** refuses any event it is capable of performing at the next step.

24.2 Failures

Failures are defined as being the relation between the traces of a process and the sets of events that cause a refusal. Put simply, the *failures* set of a process lists those actions which would cause the process to fail were they to be requested as the next possible action, in contrast to the explicitly defined behaviour of the process. Study the following example and then relate that to the explanatory text.

(CSP) [9,§3.9] failures(a $\rightarrow$ *b* $\rightarrow$ *STOP) = { (<>,{}), (<>,{a}), (<a>,{}),* *Ex242.1*
(<a>,{a}), (<a,b>,{}), (<a,b>,{a}),
(<a,b>,{b}), (<a,b>,{a,b}) }

where the tuple (<x>,{y}) denotes "after action trace <x>, fail on providing actions in {y}".

Quote: Reference CSP by C.A.R Hoare. p129, [9]. "(s,X) is a failure of P if P can engage in the sequence of events s and then refuse to do any more, despite the environment prepared to engage in any events of X."

24.3 Divergences

Divergence in a process is extremely undesirable if we expect our systems to terminate successfully. Divergence defines chaos to be the result of unguarded recursion. That is to say that a divergent process no longer engages in any further events, it simply ceases to respond to any interactions and acts as if it is locked into some infinite loop, making the process endlessly unavailable. In CCS the symbol Ω is used to show a process which becomes divergent. In CSP we can define chaos as a worst case process, thus:

(CSP)	$CHAOS = \mu X.X$	*[9,§3.8]*	*Ex243.1*

and only a slightly more stable process (slightly less divergent) as:

(CSP)	$LessCHAOS = \mu X.\ (c \rightarrow (X \setminus \{c\})) = c \rightarrow CHAOS$	*Ex243.2*

The second process in *Ex243.2* is slightly less divergent because we can predict the initial action *c*. Only then does this process fall into the depths of divergent behaviour.

Any process satisfies divergence, for instance the process *CHAOS* can do anything and it can also refuse to do anything. The function *diverges* can be defined as the set of traces leading to chaos [9,§3.5.3].

25 Nondeterminism in CCS

As we have seen with the CSP examples in section 24, nondeterminism affects observable behaviour. However, internal actions cannot always be ignored, especially if they are leading actions in a process's behaviour [13,§2.3]. For example:

(CCS) $\tau . a . b . c . \emptyset + d . e . f . \emptyset$ *E25.1*

would not be as predictable as:

(CCS) $a . b . c . \emptyset + d . e . f . \emptyset$ *E25.2*

because of the leading tau (τ) in the left hand side of the expression in *E25.1*. If the left hand side is chosen then we cannot tell when the actions *a*, *b* and *c* will occur.

In CCS there are different types of equivalence, where one process is deemed to be comparable to another or a valid implementation of a specification.

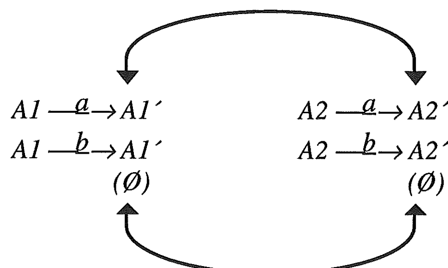
25.1 Strong Equivalence (~)

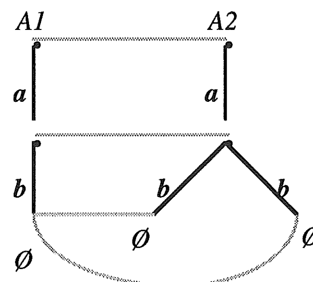
To test one process against another for equivalence we can perform a strong bisimulation over the two processes. For this test we do not regard τ as a special action. Strong bisimulations are defined as follows:

(CCS) $(A1,A2) \in S \Rightarrow$ *whenever* $A1 \xrightarrow{a} A1'$ *then,* *D251.1*
for some $A2'$, $A2 \xrightarrow{a} A2'$ *and*
$(A1',A2') \in S$ *and vice versa. [13,§4.2, p.88]*

Which translates into plain English as "whenever A1 performs some action, including τ, and progresses into some new state A1′ then A2 also performs that action and progresses into a new state itself". The tuple (A1,A2) and new state tuple (A1′,A2′) denote the contents of the bisimulation set B which must be a closed set if there is to be a strong equivalence between the two processes. An open set would mean that one of the processes could not perform some action that the other process could, which would denote that strong equivalence does not exist between the two processes.

For example: $A1 \stackrel{def}{=} a . b . \emptyset \quad A2 \stackrel{def}{=} a . (b . \emptyset + b . \emptyset)$ *[7, p.40]Ex251.1*





$B = \{ (a . b . \emptyset, a . (b . \emptyset + b . \emptyset)), (b . \emptyset, b . \emptyset + b . \emptyset), (\emptyset,\emptyset) \} \equiv \{(A1,A2), (A1',A2'),(\emptyset,\emptyset)\}$

therefore A1 ~ A2 if there exists a set S such that (A1,A2) ∈ S [13,§4.2, p.90].

A lot of work has been done in the area of process equivalence. Some of the work concentrating on certain process algebras [4, 6].

25.2 Observational Equivalence (≈)

As potential observers we are interested in the behaviour of processes from the viewpoint of a third party entity which can observe the process perform its unrestricted actions to satisfy some form of equivalence. Other names for observational equivalence are *bisimularity* or *weak equivalence*.

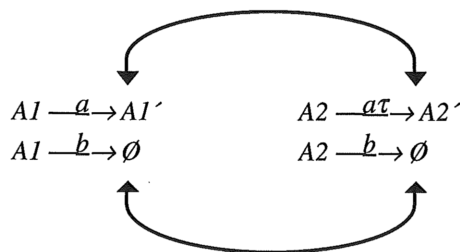
For this type of equivalence we ignore tau (τ) actions and use a new transition system (a system whereby a process moves from one state to another). This new transition system can be defined formally thus:

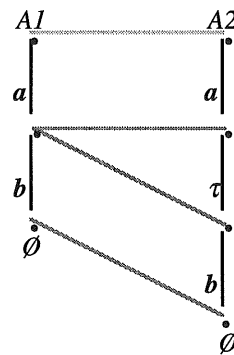
(CCS)[13, p.106] $A \stackrel{\tau'}{\Longrightarrow} A'$ *where* τ' *is a sequence of actions with* τ *removed* D252.1

Weak Bisimulations are defined using a tuple, as before in D251.1.

(CCS)[13,§5.1, p.108] $(A1,A2) \in S \Rightarrow$ *whenever* $A1 \xrightarrow{a} A1'$ *then,* D252.2
for some $A2'$, $a' = a \xrightarrow{\tau^*} a$
$A2 \stackrel{a'}{\Longrightarrow} A2'$ *and*
$(A1',A2') \in S$ *and vice versa.*

For example: $A1 \stackrel{def}{=} a . b . \emptyset$ $A2 \stackrel{def}{=} a . \tau . b . \emptyset$ *[7, p.41]* *Ex252.1*





$B = \{ (a . b . \emptyset, a . \tau . b . \emptyset), (b . \emptyset, \tau . b . \emptyset), (b . \emptyset, b . \emptyset), (\emptyset,\emptyset) \}$

therefore A1 ≈ A2 if there exists a set S such that (A1,A2) ∈ S (as per Ex251.1).

Note how the state of A1, after action a, maps to two states in A2, surrounding the τ action This dual mapping is caused because of the placement of τ. It is sandwiched between two actions and consequently these actions will occur, albeit after the τ has occurred (at whatever time it chooses to occur - remember that we have no control over τ). Refer to D272.7 in section 27.2.

25.3 Observational Congruence (equality)

To enable us to substitute one process for another we need to be able to show observational congruence. A congruence relationship between two processes is stronger than other types of equivalence as it states that processes must perform exactly the same actions so that a new process can offer the same actions as the one it is replacing. Consider the transition system which defines progression from some initial state *(A)* to some new state *(A´)*.

(CCS) $A \overset{a}{\Longrightarrow} A'$ *if* $A(\overset{\tau}{\longrightarrow})^* \overset{a}{\longrightarrow} (\overset{\tau}{\longrightarrow}) A'$ *[13,§5.3]* *D253.1*

where process A transforms into some new state A´ after some action a which can be surrounded by a number of possible τ actions.

Equality states that:

(CCS) $A1 = A2$ *if whenever* $A1 \overset{a}{\longrightarrow} A1'$ *then,* *D253.2*

for some A2´, $A2 \overset{a}{\Longrightarrow} A2'$ *and* $A1' \approx A2'$ *and vice versa.*

In plain language this definition states that A2 gets into some new state A2´after performing the same actions as A1 did when it transformed into A1´.

If both processes therefore perform the same actions, regardless of some embedded τ actions (which are ignored) then we can say that one process can be substituted for another in every circumstance that we would expect to find the original process.

26 Equivalence (Basic)

So far we have dealt with a few different views of equivalence between processes. I hope that the reader has grasped the concept that there are many different ways of determining equivalence between processes. Each language has its own ways of telling whether one process conforms to the behaviour (observable or otherwise) of another process. As we have seen, it may be necessary to test for all possible actions (including hidden actions). Alternatively, we might only be concerned with testing a process against what is observable. Let us explore some more views of equivalence.

26.1 Equivalence (CSP)

In CSP we can use the previously defined functions *failures* and *diverges* to determine the equivalence of one process over another. A process *P* can be uniquely defined by the following tuple:

(CSP) *(αP, failures(P), diverges(P))* *[9, p.130]* *D261.1*

The tuple in *D26.1* portrays a combination of the set of failures for a process, related to its set of divergences, all based upon its original alphabet. Compared to CCS we might think that the CSP view of equivalence is simplistic, however we must consider the work that the functions *failures* and *diverges* are doing behind the scenes. Refer to sections *24.1* and *24.2* for more details about the mechanics of both of these testing functions.

Due of the unique definition of a process, using the tuple, a comparison between the

contents of the results of the tuple when applied to two processes would be able to tell us if two processes were the same in all but name (i.e: equivalent behaviour).

26.2 Equivalence (CCS)

Using the language of CCS we can derive three views of process equivalence, namely:

i) Strong Equivalence (~)
ii) Equality (=)
iii) Observational Equivalence (≈)

We could consider these views to be ordered in some way so as to infer that *Strong Equivalence* is the most binding form of equivalence, followed by *Equality* and finally *Observational Equivalence*. Put simply, we can view the relationship between the different forms of equivalence as:

$$(A1 \sim A2) \geq (A1 = A2) \geq (A1 \approx A2) \qquad [13, p.154] \qquad D262.1$$

Note that when using CCS it is possible to distinguish between nondeterministic processes that are regarded as deterministic processes in CSP. This ability to identify the difference in behaviour has some relevance when considering the equivalence of one process against another if the newcomer contains nondeterminism.

27 Algebraic Laws (Basic)

For each language (CSP, CCS and LOTOS) I have introduced the semantics and syntax of many operators. Certain algebraic laws govern the mathematical manipulation of these laws such that we can only perform certain defined tasks using the laws. We map a law to an expression, apply that law and derive a new form of the expression.

27.1 Algebraic Laws (CSP)

Algebraic laws for the basic CSP operators are shown as follows:

(CSP) [9,§			
	$P \setminus \{\} = P$	[9,§3.5.1]	D271.1
	$P \,[]\, STOP = P$	[9,§3.5.1]	D271.2
	$(P \setminus B) \setminus C = P \setminus (B \cup C)$	[9,§3.5.1]	D271.3
	$(e \rightarrow P) \setminus C = e \rightarrow (P \setminus C)$ if $e \neq C$	[9,§3.5.1]	D271.4
	$= P \setminus C$ *otherwise*		
	$(P1 \sqcap P2) \setminus C = (P1 \setminus C) \sqcap (P2 \setminus C)$ *(distributive law)*	[9,§3.5.1]	D271.5

Each of these algebraic laws can be used forwards or backwards as either the left hand side can be replaced by the right hand side or vice versa. Moving on to CCS we find that the language provides many of the same laws.

27.2 Algebraic Laws (CCS)

We can subdivide the algebraic laws for CCS into three groups. These are specified as those laws for:

i) Static Operators (composition, restriction and relabelling).
ii) Dynamic Operators (action prefix and summation).
iii) Expansion Law (which links the previous two groups).

The algebraic laws for CCS are defined as:

(CCS)	$A + \emptyset = A$		*[13, p.62]*	*D272.1*
	$A \setminus K \setminus L = A \setminus (K \cup L)$		*[13, p.80]*	*D272.2*
	$(a.A) \setminus L = a\ .\ (A \setminus L)$	*if* $a \notin L \cup \overline{L}$	*[13, p.70]*	*D272.3*
	$= \emptyset$	*otherwise*		

Further to these basic laws we also include the monoid laws:

(CCS)	$A1 + A2 = A2 + A1$ *(commutative)*	*[13, p.62]*	*D272.4*
	$A1 + (A2 + A3) = (A1 + A2) + A3$ *(associative)*	*[13, p.62]*	*D272.5*
	$A1 + A1 = A1$	*[13, p.62]*	*D272.6*

Tau (τ) *laws further expand the set of laws available to us.*

(CCS)	$a\ .\ \tau\ .\ P = a\ .\ P$	*[13, p.62]*	*D272.7*
	$A + \tau\ .\ A = \tau\ .\ A$	*[13, p.62]*	*D272.8*
	$a\ .\ (A1 + \tau\ .\ A2) + a\ .\ A2 = a\ .\ (A1 + \tau\ .\ A2)$	*[13, p.62]*	*D272.9*

Armed with these laws (for both CSP and CCS) we can put our newly gained knowledge to good use, proving properties about certain behavioural expressions in either language. Consider the following sections which set out to prove equivalence properties between two different expressions written in either notation.

28 Verification Using Algebraic Laws (Basic)

A mathematical language provides us with the tools to prove properties about expressions in the language using formal laws. It is our intention to show how such laws can be used together to prove simple theorems, such as those that appear in the following two sections.

28.1 Verification Using Algebraic Laws (CSP)

(CSP)		*[7, p.49]*	*E281.1*
Prove	$(e1 \rightarrow STOP \ \square\ e2 \rightarrow STOP) \setminus \{e1\} = ((e1 \rightarrow STOP) \setminus \{e1\}) \ \square\ ((e2 \rightarrow STOP) \setminus \{e1\})$		
LHS	$= STOP \sqcap (STOP \ \square\ (e2 \rightarrow STOP))$	*using law: if* $c \cap B \neq \{\}$ *and is finite then* $(x{:}B \rightarrow P(x)) \setminus C = Q \sqcap (Q \ \square\ (x{:}(B - C) \rightarrow P(x)))$	
	$= STOP \sqcap (e2 \rightarrow STOP)$	*using law:* $P \ \square\ STOP = P$, *which incidentally is not equivalent to* $(e2 \rightarrow STOP)$	

$= ((e1 \rightarrow STOP) \setminus \{e1\}) \, [] \, ((e2 \rightarrow STOP) \setminus \{e1\}) = RHS \therefore$ Q.E.D.

using law:

$(x \rightarrow p) \setminus C = x \rightarrow (P \setminus C)$ *if* $x \notin C$

$= P \setminus C$ *if* $x \in C$

28.2 Verification Using Algebraic Laws (CCS)

(CCS) *[7, p.49]* *E282.1*

Prove	$A1 + \tau . (A1 + A2) = \tau . (A1 + A2)$	
LHS	$= A1 + (A1 + A2) + \tau . (A1 + A2)$	*using law:* $A + \tau . A = \tau . A$
	$= (A1 + A1) + A2 + \tau . (A1 + A2)$	*using law:* $A1 + (A2 + A3) = (A1 + A2) + A3$ *(associative)*
	$= (A1 + A2) + \tau . (A1 + A2)$	*using law:* $A1 + A1 = A1$
	$= \tau . (A1 + A2) = RHS \therefore$ *Q.E.D.*	*using law:* $A + \tau . A = \tau . A$

— *end of section on Process Algebras without application.*

Part II

29 Object-Oriented Concepts

This section brings with it a discussion of objects and their structure. I will leave behind some of the formalisms of the previous sections (thankfully you may say!) to concentrate upon the notion of an object and related issues. I will, of course, return to the process algebras to relate the two sides of this text together (what would be the point of this whole document otherwise?). Let us begin much the same as we did in the first half of our journey, from the beginning with first principles.

29.1 What is an Object?

An object is an elementary unit encapsulating both state and behaviour. In short, an object contains data and functions which act over that data. The functions (referred to as methods) are used to access the data and form the interface to the object.

We view an object as having an explicitly defined interface with its environment. Already we can correlate the form of an object with that of a process. Consider the similarity between the following diagram and that of the processes shown in earlier sections.

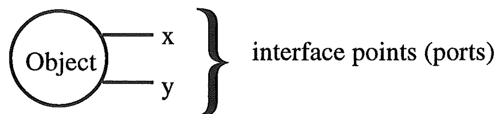


figure 291.1

Similar to an abstract data type (ADT), where the methods (a.k.a functions) are considered related to the hidden data structure an object's methods reside within the bounds of the object itself. Therefore, an ADT share many attributes normally associated with objects. However, ADTs are not strictly objects and the functions which correspond to an ADT are not as closely related to the data as an objects methods are related to its data area. Figure 291.2 illustrates the separation that can be observed when making a distinction between these two types of entity.

Despite being related at the data/function level objects and ADTs depart from the same model when inheritance becomes an issue. ADTs cannot cope with inheritance (the ability to copy and extend/overwrite an existing data definition). Inheritance is one area where objects take over the modelling role.

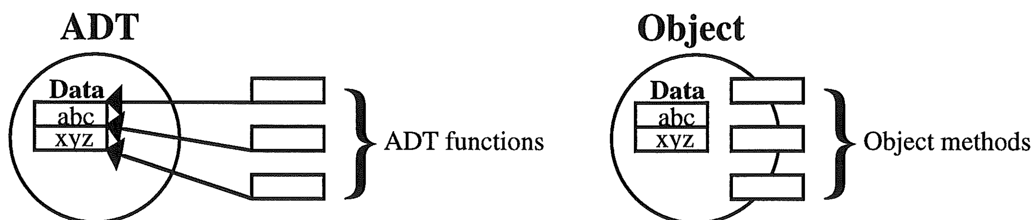


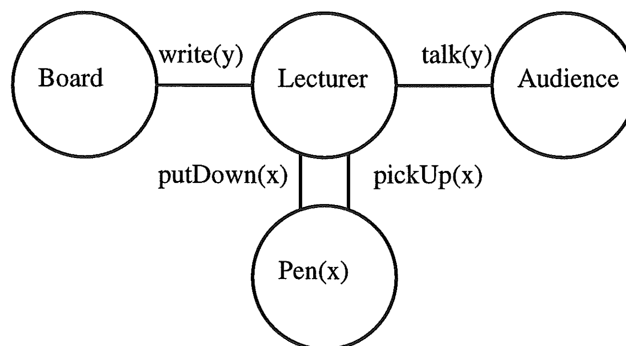
figure 291.2

In process algebras we can model an object using the value passing calculus to model the state and action sequences to model the methods. My particular line of enquiry concentrates on the communication between objects and therefore does not consider state. We can model the infamous marker pen example process interacting with a lecturer and the wipe board in the following way:

(CCS) *x = {black,red}* *y = {subjectMatter}* *Ex291.1*

Lecturer(x,y) = talk(y) → pickUp(x) → write(y) → putDown(x) → Lecturer(x,y)

P291.1



29.2 What is a Class (I)?

The simple example in the previous section (Ex291.1, *P291.1*) illustrates that certain objects share common attributes with one another. The similar objects in question are the two *Pen* objects. An implementation of *P291.1* would require two instances of *Pen*, one representing either colour. Apart from the difference in the colour parameter both objects exhibit the same behaviour. They are both from the same class of object. Detailed discussions of class structure

and object oriented concepts can be found in [12] and [19].

29.3 Why are the Last Two Sections in the Wrong Order?

In much of the object-oriented literature (e.g: [12] and [19]) the concepts of object and class are introduced *class* first, followed by *objects* taken from those classes. We can justify the order of preference for this particular text because, as individuals, we think in terms of actual **objects** first! Consider the two coloured pens that were used in *Ex291.1.* We visualise the physical attributes of each pen, we do not abstract from the individual characteristics of the pens to arrive at some general description, only then recognising the class of *Pen* before identifying each pen by its own merits (Red or Black).

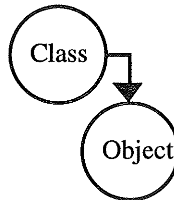
Another analogy to support object-class ordering is to imagine that we are waiting for a bus. As one approaches we look to see if it's the one that we want. We identify the individual and then place it in its appropriate class; "Oh look! it's a number 9 bus"; object before class.

Even if we do choose to refer to class then object ("a bus and its a number 6") we still encapsulate the idea of a bus, rather than the more general class of *mechanised people carrier*.

29.4 What is a Class (II)?

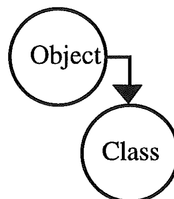
In terms of objects we classify them in class-object order when we generalise objects whilst looking for reusable attributes that can be contained within a single class.

P294.1



However, we visualise an object in the *real world* using the object-class order.

P294.2



In a process algebra the available notations do not allow us to distinguish between a class and an object. The distinction between them is lost. However, programming languages resurrect this distinction. Consider the following C++ [16] code fragment, followed by an equivalent LOTOS definition:

Ex294.1

```
(C++)
class Pen
{
      private:
            Colour      penColour;
```

```
        public:
                Pen(Colour c);
                Colour Pen::GetColour(void);
                Pen::SetColour(Colour c);

    }; //end-class

    Pen *penPointer = new Pen(Red);
```

(LOTOS) *Ex294.1*

process Pen[GetColour,SetColour](c:Colour) : noexit :=
GetColour ! c : Colour; Pen(c)
[]
SetColour ? c : Colour;Pen(c)
endproc

The pointer variable `*penPointer` represents an instance of class *Pen* (an object behaving like a pen). We could have an infinite number of instances of class pen (until we exceed allocated memory) where each instance would be a copy of all of the data fields and methods defined in the abstract class template *Pen*. Until we instantiate class *Pen* all we have is an abstract concept of a pen which has no physical presence that we can actually use. An analogy of the abstract class template is like someone describing a pen to you and describing its function without actually showing you a physical pen. If you've never seen a pen before then all you know about one is what you've been told!

30 Inheritance and Reuse

In order to make the most use of abstract classes we need to be able to reuse common elements supplied by a class to help define yet another class.

Inheritance uses class templates to build new class templates. Rudkin [15] discusses classes, templates and objects in his work with LOTOS and inheritance, referencing the work of Wegner [19] in the process. A formal definition of inheritance is based upon incremental modification. To illustrate basic inheritance (there are three main types that I shall discuss in due course) we need a more complex example of a *Pen* class.

Consider a switching unit, such as we might find controlling an alarm clock, whose behaviour is defined informally as follows:

> "Accept one of two possible signals, one *internal* signal to tell the switch to turn ON, one *external* signal to tell the switch to turn OFF. Set the internal state to reflect the nature of the signal. If the switch is left ON for a time then automatically turn it OFF."

In CCS we might define the *Switch* agent as:

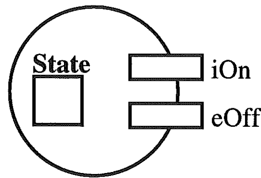
(CCS) $Switch \setminus \{on\} \stackrel{def}{=} Switch(off)$ *E30.1*

$Switch(off) \stackrel{def}{=} iOn \,.\, Switch(on)$

$Switch(on) \stackrel{def}{=} eOff \,.\, Switch(off) + \tau \,.\, Switch(off)$

Here is a pictorial view of the class *Switch*:

Object ***Switch***

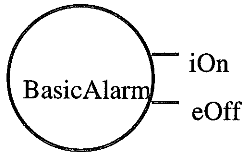


Object *Switch* has an area for storage for the 'private' state of the switch and two public methods (accessible by any process in the environment).

figure 30.1

Now suppose that we intend an object of class *Switch* to be the controlling process in a warning system for the coolant flow system in our friendly neighbourhood nuclear reprocessing plant. Coolant sensors would synchronise with the action *iOn* to turn the alarm on, thus warning the controllers of a problem. The alarm, according to the informal description of its behaviour, could then be deactivated by either an external input *eOff* or by timing out and turning itself off (using τ).

So, we have the basic description for an alarm controller class *BasicAlarm*.




```
process BasicAlarm[iOn,eOff](s :State) : noexit
        [isOn(s)] → (eOff;BasicAlarm(iOn,eOff)(off)
                              []
                     i;BasicAlarm[iOn,eOff](off))
        []
        [not isOn(s)] → iOn;BasicAlarm[iOn,eOff](on)
endproc
```

figure 30.1

What would happen to our specification if we were to include a new action *eOn* into *BasicAlarm*? The new action models a panic button for the warning system allowing external activation of the alarm (i.e: the controllers pressing a large red button marked "Panic!").

The new process *PanicAlarm* extends the behaviour of *BasicAlarm* thus: (by the way, notice how 'effortlessly' we move between the notations of CSP, CCS and LOTOS...?).

```
(LOTOS)     process PanicAlarm[iOn,eOff,eOn](s : State) : noexit :=        E30.2
                BasicAlarm[iOn,eOff](s)
                ⊕
                [not isOn(s)] → eOn;PanicAlarm[iOn,eOff,eOn](on)
            endproc

            where PanicAlarm written out in full is specified thus:

            process PanicAlarm[iOn,eOff,eOn](s : State) : noexit :=        E30.3
                hide iOn in
                    [isOn(s)] → (eOff;PanicAlarm[iOn,eOff,eOn](off)
                                  []
                                  i ;PanicAlarm[iOn,eOff,eOn](off))
                    []
                    [not isOn(s)] → (iOn;PanicAlarm[iOn,eOff,eOn](on)
                    ⊕
                    eOn;PanicAlarm[iOn,eOff,eOn](on))
            endproc
```

Note that the ***hide*** *operator is not used in the definition of the PanicAlarm in E30.2 because it is inherited from the definition of the BasicAlarm's behaviour introduced in figure 30.1.*

For reference, other work on object-oriented specification using LOTOS has been carried out, evaluating the feasibility of the object-oriented concepts within the language [5], [11], [15]. The inheritance that exists between the *BasicAlarm* class and the new derived *PanicAlarm* class can be illustrated in the following diagram:

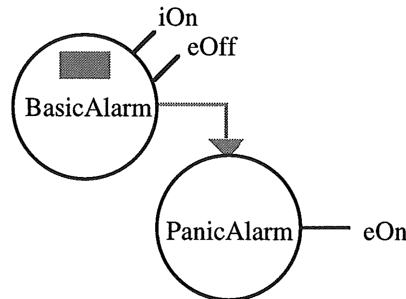


figure 30.2

Note that only one of each method and data storage is offered in the definition of the new class *PanicAlarm*. From an external observer's point of view all that can be seen are the three methods (interface actions) *iOn*, *eOff* and *eOn*.

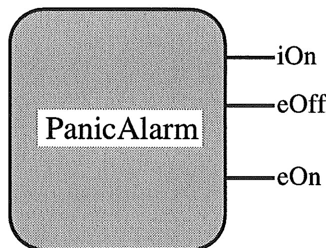


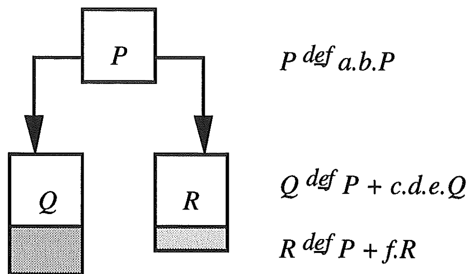
figure 30.3

As observers we have no idea that *PanicAlarm* is constructed from *BasicAlarm* with extra action extensions.

Before we proceed let's take a step back to the ⊕ symbol that crept into the definition of *PanicAlarm* in *E30.2*. What does ⊕ stand for and what are its operational semantics? Many readers will have seen this particular symbol before in the context of set theory. In some sense ⊕ is used exactly as it is for function overriding. In the context of inheritance we use ⊕ to modify existing behaviour and to extend a process with new behaviour.

30.1 Three Types of Inheritance

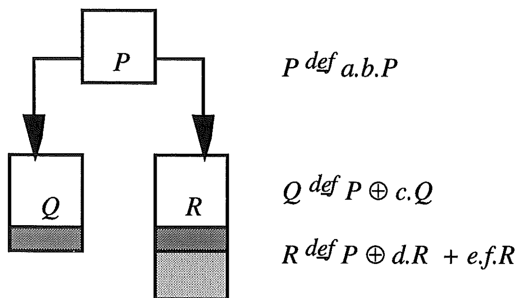
At this point in the text I am ready to explicitly define the different types of inheritance that I consider important, taken from my own previous work in this area [18]. These types of inheritance are strict inheritance, non-strict inheritance (which I refer to as casual inheritance) and transitive inheritance. There now follows a short graphical and formal description of each of these three types of inheritance.



i) Strict inheritance (Copies state and behaviour and can extend both attributes)

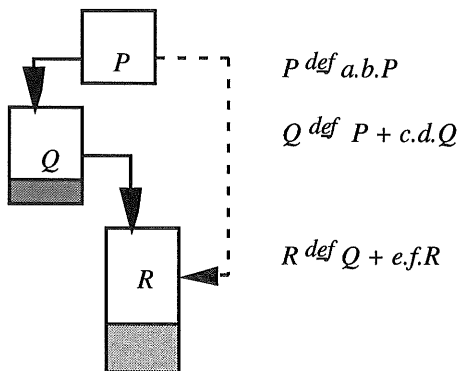
*Note: No modification at this stage.

figure 301.1



ii) Non-strict inheritance (a.k.a casual inheritance, copies modifies and extends existing attributes).

figure 301.2



iii) Transitive inheritance (Inheritance across a hierarchy greater than one ancestor. R inherits from Q which in turn inherits from P. Consequently R transitively inherits from P).

figure 301.3

The work so far has only covered strict inheritance via the on-going example using *BasicAlarm* and *PanicAlarm*. Let us now move a step further and consider how we can illustrate the remaining types of casual and transitive inheritance. Again, I shall use *BasicAlarm* as the foundation for this work.

As we have seen when I covered the inner workings of processes, in section 7 onwards, the behaviour of a process is defined as an ordered sequence of actions. In our friendly neighbourhood nuclear reprocessing plant example we may wish to include new actions within the defined sequence of actions already present in the system. Perhaps we want to introduce a warning light into the system which flashes when the alarm bell sounds or whenever there is a less serious problem to be reported by our monitoring system (that would help, don't you think?).

Written in full (in CSP, why not?) the new sequence of actions, including a signal to the warning light telling it flash, would be defined as follows:

(CSP) *E301.1*

$$
\begin{aligned}
&WarningAlarm(s) = \mathit{if}\ isOn(s)\ \mathit{then}\ (eOff \rightarrow Warning\ Alarm(off) \sqcap \\
&\qquad\qquad i \rightarrow WarningAlarm(off)) \setminus \{i\} \\
&\qquad \mathit{else\ if} \neg\ isOn(s)\ \mathit{then} \\
&\qquad\qquad (iOn \rightarrow flashLight \rightarrow WarningAlarm(on)\ \square \\
&\qquad\qquad\qquad eOn \rightarrow flashLight \rightarrow WarningLight(on))
\end{aligned}
$$

Note: the more readable if...then...else syntax for CSP is used, rather than the less readable notation incorporating ◁ and ▷.

We note that the signal to start the warning light flashing has been inserted into the sequence of existing actions inherited from *PanicAlarm*. A CCS derivation tree for *WarningAlarm* would be drawn as:

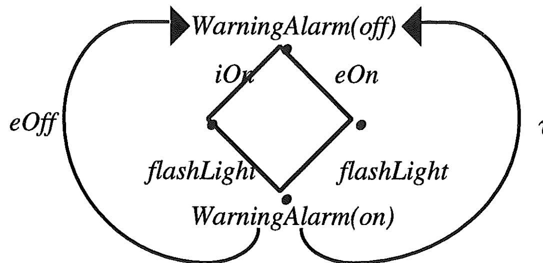


figure 301.4

Taken from the CCS expression:

(CCS)	*WarningAlarm \ {on}* $\stackrel{def}{=}$	*WarningAlarm(off)*	*E301.2*
	WarningAlarm(off) $\stackrel{def}{=}$	*iOn . WarningAlarm(on) + eOn . WarningAlarm(on)*	
	WarningAlarm(on) $\stackrel{def}{=}$	*eOff . WarningAlarm(off) +* τ *. WarningAlarm(off)*	

The casual inheritance that has been shown in the definition of *WarningAlarm* is evident in the behavioural modification that *PanicAlarm* has been subjected to. The ports *iOn* and *eOn* in *WarningAlarm* no longer offer the same behaviour as their *PanicAlarm* ancestors. Notice how easily the biological meaning of inheritance slipped into the text to illustrate the relationship between *PanicAlarm* and *WarningAlarm.* It is sometimes easier to think of inheritance between classes in this way, although care must be taken as biological inheritance is not quite the same as class inheritance (re: mutations of a parental template as opposed to direct copies of one's parents). Meanwhile, back to the *WarningAlarm* example. What is the state of our current inheritance hierarchy for *WarningAlarm*? Here is a diagram depicting the chain of inheritance:

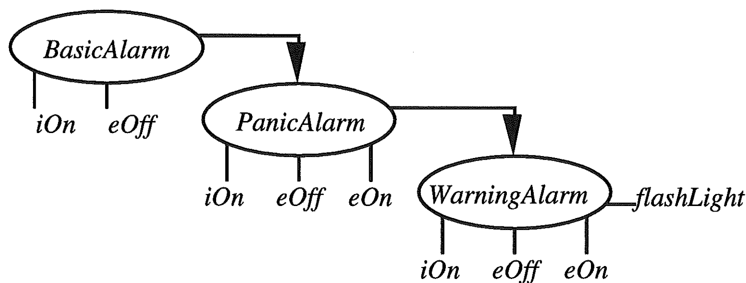


figure 301.5

Note that the original actions *iOn* and *eOn* in *BasicAlarm* and *PanicAlarm* respectively, have been superseded by modified equivalent action sequences due to the casual inheritance enforced by the definition of *WarningAlarm.* In LOTOS the class *WarningAlarm* would be defined as a process.

(LOTOS) *E301.3*

```
process WarningAlarm[iOn,eOn,eOff,flashLight](s:State) : noexit :=
        PanicAlarm[iOn,eOn,eOff](s)
        ⊕
```

[not isOn(s)] → (iOn;flashLight, WarningAlarm[iOn,eOn,eOff,flashLight](on)
[]
eOn;flashLight; WarningAlarm[iOn,eOn,eOff,flashLight](on))
endproc

From the simplicity of *WarningAlarm* in *E301.3* we get a feel for the benefits of reuse via the power of inheritance. Once again, the original occurrences of actions which are guarded under *[not isOn(s)]* are replaced with new versions of the action sequences that the original guard covered.

The careful reader will note that even at this stage of our journey nothing has been said about the detail or operational semantics of ⊕. We have not seen how it works or what the implications of its use entail. It so happens that the mystery surrounding ⊕ has not yet been solved. However, I recognise its importance in implementing inheritance using process algebras. At present I regard the override operator as a meta-operator that seems to provide the results I'm looking for in a modifier for existing process behaviour. The work that is being done in this area has not yet been formalised regarding how ⊕ actually works! Let us keep in mind its shortcomings but let us not ignore its potential either.

At this point I should move on to consider the mechanics of transitive inheritance, but it so happens that I do not need to expand our example process *WarningAlarm* further to encapsulate transitive inheritance, it is already present in *WarningAlarm*. If you're not sure then refer to figures *301.5* and *301.3*. Process *WarningAlarm* does two things for us. It provides us with an example of casual inheritance and it also has transitive inheritance links with our original foundation process (the grandparent process) *BasicAlarm*.

31 Sub-Type Relationships (Basic)

The relationship that is defined between *BasicAlarm* and *PanicAlarm* can be presented as a sub-type relationship. To be a valid sub-type of some class an inherited class must be able to guarantee that it provides (at least) the same behaviour to the environment as its parent class (the class it inherited from). If a process can guarantee this behaviour then it can substitute its parent in all instances where a copy of the parent was expected. We show sub-types as $Q \leq P$, where *"Q is a valid sub-type of P"* [11].

Notice how *WarningAlarm* was excluded from the opening statement in this section. The process *WarningAlarm* is not a valid sub-type of either *BasicAlarm* or *PanicAlarm*. Why is this the case? After all *WarningAlarm* did inherit from *PanicAlarm*, which in turn inherits from *BasicAlarm*. The answer is simple, although undesirable and is explained as follows.

The behaviour of *WarningAlarm* is radically different from either *BasicAlarm* or *PanicAlarm*. If *WarningAlarm* extended the choice of either of its parent classes then that would be a different case and we could say that *WarningAlarm* was a valid sub-type of *PanicAlarm* (using transitivity in the case of *BasicAlarm*).

I could have constructed WarningAlarm in two different ways; the casual inheritance version we have already seen (that's one way!). I have not yet discussed the strict inheritance version (that's the other way!) but will do in the next section.

31.1 Sub-Type Relationships (CSP)

This section covers the strict inheritance version of *WarningAlarm*, making use of the behaviour of *PanicAlarm*, defined using CSP syntax.

(CSP) *E311.1*

$$WarningAlarm(s) = if\ \neg isOn(s)\ then\ (PanicAlarm\ \square\ flashLight \rightarrow WarningAlarm(on))$$

The derivation tree for this type of *WarningAlarm* process is shown in figure 311.1.

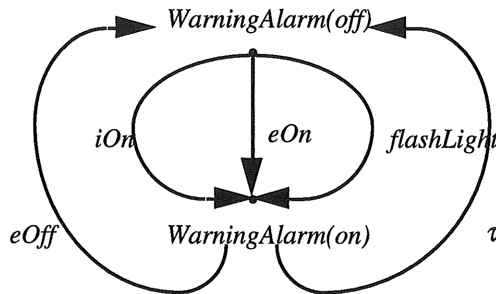
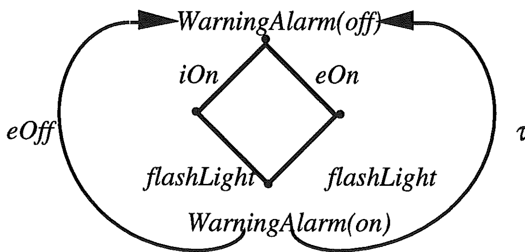


figure 311.1

The extended action choice *flashLight* (note that it is deterministic choice, which is environmentally influenced) is another possible choice attached to the same Boolean guard $\neg isOn(s)$ when the alarm is OFF. Recall that casual inheritance produced an entirely different derivation tree (reference figure 301.4). Closer examination of the two versions of *WarningAlarm* (*E301.3 (Casual-WA) and E311.1 (Strict-WA)*) shows differences using Hoare's failure sets.

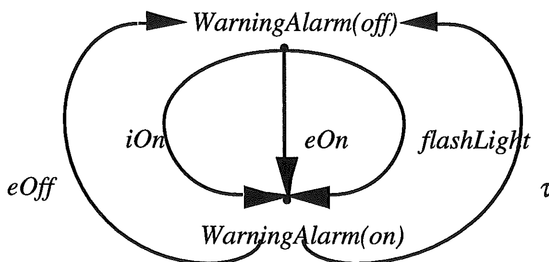
(CSP) The failure sets for Casual-WA are: *E311.2*



{ Ø, {eOff}, {t}, {flashLight}, {eOff,t}, {eOff,flashLight}, {t,flashLight}, {eOff,t,flashLight} }

{ Ø, {eOn}, {iOn}, {flashLight}, {eOn,iOn}, {eOn,flashLight},{iOn,flashLight} }

whereas the failure sets for Strict-WA are defined as:



{ {eOff, τ}, Ø, {eOff}, {τ} }

{ Ø, {eOn}, {iOn}, {flashLight}, {eOn,iOn}, {eOn,flashLight},{iOn,flashLight} }

Therefore, the result of a union of *failures(Strict-WA)* is not the same as a union of *failures(Casual-WA)*. Different failures sets, consequently different process behaviour.

CCS bisimulation equivalence would also reveal differences between *Strict-WA* and *Casual-WA*, which seems intuitive as the behaviour of each process at *first glance* seems quite different. Using CCS bisimulation equivalence the process *Strict-WA* would perform, say, the action *iOn* and be in a position to offer an *eOff* action or a τ action; *Casual-WA* could not comply as its next action after an *iOn* or *eOn* is the *flashLight* action. Hence the bisimulation set *B* for

Strict-WA ≈ *Casual-WA* could not be closed, therefore no equivalence is present.

To conclude this section I state that inheritance with the process algebras of CSP, CCS and LOTOS has not been defined formally. I have discussed a mechanism for introducing inheritance at the meta-language level, using $\oplus$. I have also underlined the types of extension and modification that I am trying to achieve with inheritance but as yet the work needs to be done to actually formalise these ideas. That's the state of play so far; the work continues.

Already we start to see that the darker side of modifying a process behaviour and trying to substitute one process for another is fraught with dangers to the unwary. I hope to have highlighted some of those dangers here. The remainder of this text will try to highlight further problem areas. Rather than get embroiled in the obvious deeper issues surrounding inheritance, particularly casual inheritance, I shall move swiftly on to uncover the intricacies of method invocation. From there I shall discuss recursion. Up until now I have managed to avoid recursion in too much detail, certainly since I have started to discuss objects. Well, not for much longer as you, the reader, shall find out.

32 Method Invocation

Firstly, let us explicitly define what I mean by *method invocation*, as it is a rather grand title for a section after all. Each class in the inheritance hierarchy (such as appears in figure 301.5) is responsible for adding something to the behaviour of the classes that follow it, together with additions to the data storage within a class (part of the state information). Of course, we could simply inherit from a class and leave the structure and state alone although then we would only have a straight copy of the parent class under a different name.

Each method belonging to a class will need to be referenced when that method is required to be executed. In the *WarningAlarm* inheritance hierarchy (figure 301.5) we might consider invoking the method *eOff*. If we invoke *eOff* from within an instance of the *WarningAlarm* class then we begin a search for the method in that class. If *eOff* cannot be found in *WarningAlarm* (as indeed it cannot!) then we start to search in the parent of the class for *eOff*. And so on up the inheritance hierarchy until we find the method we're looking for. In the example in figure 301.5 (using the definition of *E301.3*) we get a 'hit' in the class *BasicAlarm*. It is the *BasicAlarm* action *eOff* that will be executed.

Incidentally, if I were using an object-oriented programming language as the basis for discussion then only one class would exist (*WarningAlarm*) and it would be a resident method from that class that was executed rather than one (supposedly) belonging to another class. This is the difference between how object-oriented programming languages and the mathematical notations that we are using model this eventuality. In one domain each parent has some presence, in another only instances of the classes carry any influence. The process algebra view is the former (in case you're not sure).

Consider the following inheritance hierarchy, taken from the CSP strict inheritance expression in *E311.1*.

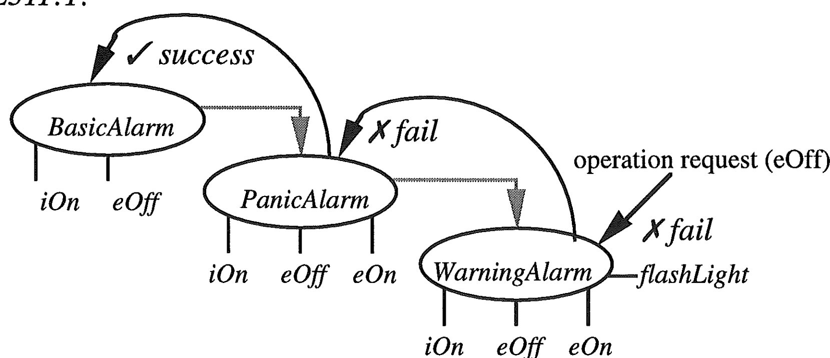


figure 32.1

From the diagram in figure 32.1 we can see that the control thread (shown as the solid arrow line) has stopped in *BasicAlarm* to execute *eOff*, which is where that operation resides. There is one point to consider before we continue. From an observer's point of view we know nothing about the inner workings of *WarningAlarm* or its parents. The original processes are absorbed into the latest child and we (as outsiders) cannot distinguish between each individual process under the cloak of one interface. As far as we're concerned there are four ports connecting *WarningAlarm* to the environment.

As an introduction to the next section let us consider how we get the control thread back from its current place of execution in *BasicAlarm*. We require control of the system to return to *WarningAlarm* so that **all** of the available behaviour from each process is once again available for execution by the environment. Welcome to the topic of recursion, objects and inheritance!

33 Recursion with Inheritance

Towards the end of the previous section I 'rattled the tiger's cage' by daring to ask the question "How do I define recursion within an inheritance hierarchy?". Recall that each process in the chain *WarningAlarm* ≤ *PanicAlarm* ≤ *BasicAlarm* is defined recursively thus:

(CSP) *E33.1*

$x = \{on, off\}$

$BasicAlarm(x) \setminus \{iOn\} \stackrel{def}{=} if\ (x == off)\ then$
$\quad iOn\ .\ BasicAlarm(on)$
$else$
$\quad (eOff\ .\ BasicAlarm(off) + \tau\ .\ BasicAlarm(off))$

$PanicAlarm(x) \stackrel{def}{=} BasicAlarm + if\ (x == off)\ then$
$\quad eOn\ .\ PanicAlarm(on)$

$WarningAlarm(x) \stackrel{def}{=} PanicAlarm + if\ (x == off)\ then$
$\quad flashLight\ .\ WarningAlarm(on)$

The important aspect of the three processes defined in *E33.1* is the recursion defined at the end of each action sequence. Because each process explicitly names the process to pass control to (namely itself) then as soon as we enter the domain of that process control is trapped there forever. What is required is a general recursive term which can redirect control back towards the process where the call originated.

The general term I am seeking is *self* [15], which is quite powerful as it knows where to redirect control, based on the origin of the method call. It may help to think of *self* as a generic pointer capable of storing the address of a class and using that address to direct the recursion. Initially *self* will hold the address of the class where it is resident. That is, it will always point back to the class where it is situated if no external invocations of that process's methods arrive.

Upon arrival of an external call to a method *self* will store the origin address of that call. An illustration of the alarm system inheritance hierarchy follows, showing recursion with and without *self*. To simplify the diagram *BA*, *PA* and *WA* have been used to shorten the process names *BasicAlarm*, *PanicAlarm* and *WarningAlarm* respectively.

Without ***Self*** **With** ***Self***

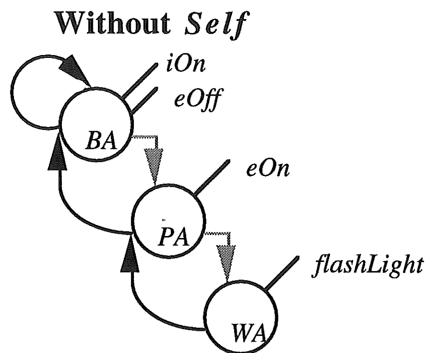


figure 33.1

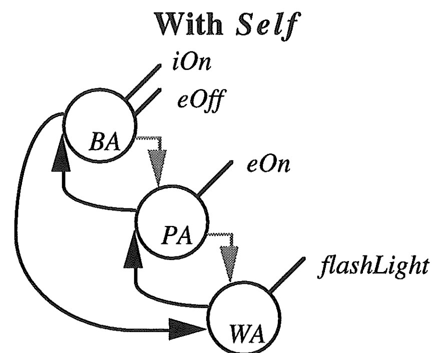


figure 33.2

The difference between figures 33.1 and 33.2 is clear. The primitive *self* allows us to offer the complete behaviour of *WarningAlarm* (its entire inherited behaviour) after a method invocation to one of *WarningAlarm's* parent classes.

Other methods of recursion have been introduced using the LOTOS process enabling (>>) operator. The ROOA method [14,§3.3.9] defines a parent class as having *exit* functionality, which passes a state of the class to some other process upon successful termination (refer to section 16: Sequential Composition and section 17: Value Passing Between Processes). An example of LOTOS recursion using process enabling follows. *BasicAlarm* would require *exit* functionality as it is a parent class (a super class). Super classes are required to exit on termination as they need to pass control to their sub-types.

(LOTOS) *E33.2*

```
process BasicAlarm[iOn,eOff](s: State) : exit(State) :=
        ...
endproc
```

Process *PanicAlarm*, as a child of *BasicAlarm*, could then be defined using *noexit* functionality (remember that *PanicAlarm* was at the end of the inheritance chain before *WarningAlarm* was introduced).

(LOTOS) *E33.3*

```
process PanicAlarm[iOn,eOff,eOn](s: State) : noexit :=
        (BasicAlarm[iOn,eOff] >> accept s :State in exit(s)
        []
                ... (* PanicAlarm extended behaviour *)
        ) >> accept s : State in PanicAlarm[iOn,eOff,eOn](s)
endproc
```

The output exit value of *BasicAlarm* is fed into the >> *accept PanicAlarm[...]...* statement, passing control back to *PanicAlarm* after some method call to *BasicAlarm*.

Now I uncover a problem. I introduced *WarningAlarm* into the system as a child process of *PanicAlarm*. The rules of *exit* functionality state that to pass control (values) to another process requires the successful termination of the current process, enabling the next process. It is here that I enter a dilemma!

BasicAlarm is the grandparent of all processes in the alarm system. Therefore, *BasicAlarm* is a super class and, as such, has *exit* functionality. What about *PanicAlarm*, which is stuck in the middle of the inheritance chain? Is it a parent with *exit* functionality, or a child with *noexit* functionality? From its position in the inheritance hierarchy *PanicAlarm* is both! Where does that leave us in terms of *PanicAlarm's* defined functionality? Obviously, *PanicAlarm* cannot hold both types of functionality simultaneously (LOTOS syntax does not allow such luxury). Here I highlight an explicit problem! Process enabling, when used as part of recursion to aid the definition of inheritance, cannot cope with an inheritance chain of greater

than one layer [18]. If we introduce further layers of inheritance the rules governing the use of process enabling (>>) causes recursion to break down. Bearing in mind the restriction of process enabling (>>) I hope that the reader will come to recognise the subtle power of *self.*

Coping with the recursion of processes is important as we are required to model processes that repeat indefinitely in order to capture *real world* behaviour. It would not prove to be a very useful alarm system if, after activating the klaxon and warning light the control process driving the system stopped executing. If the system only executed once then we might not get warned about a potential disaster at our friendly neighbourhood nuclear reprocessing plant. Recursion allows for infinite execution of a process, unless one of the choices within it is defined as STOP. After such a choice the process ceases to function and no longer engages in activities with the rest of the system.

Now that we have the model for a process that continues indefinitely we must consider how other processes interact with each other. I am referring to the communication between objects (modelled as processes). The next section is a continuation of section 9 (Process Interaction), which now uses our current view of the world that I have built up during these past few sections.

34 Object Communication

To show how objects communicate I shall introduce some new objects into the world of our model (the world of alarm systems within friendly neighbourhood nuclear reprocessing plants). I shall then connect these objects together via synchronising actions.

Informally, I shall describe the alarm system in the context of its new expanded environment. I place the *PanicAlarm* process into a subsystem within the reprocessing plant's control system. The *PanicAlarm* process forms part of some larger warning and control system. The contents of the subsystem that we are interested in are:

> i) Coolant Flow Valve (CFV), which opens and closes to a preset maximum and minimum setting. Commands to the CFV are sent from the Coolant Heat Sensor (HS). The warning light is activated from the CFV if it is tasked with opening or closing beyond its maximum and minimum range.
>
> ii) Heat Sensor (HS), which tests the temperature of the reactor core coolant. HS issues commands to the CFV to regulate the reactor core temperature. HS sounds the alarm klaxon and flashes the warning light if the core temperature gets too high or too low.
>
> iii) Panic Alarm (PA). This process should be familiar to the reader. It is activated via signals from the heat sensor (HS) or the panic button situated on the operator's console. After a time (which is not defined) the alarm deactivates, returning to an OFF state.
>
> iv) Warning Light (WL) is similar in operation to the *BasicAlarm* process discussed earlier (figure 30.1). We can see no need to include a panic button on the warning light.

Here is a diagram showing all of the control system's processes interconnected (with one obvious exception). It includes an external link to some temperature sensor via *eTemp* and an external switch to turn off the warning light via *eWLOff.*

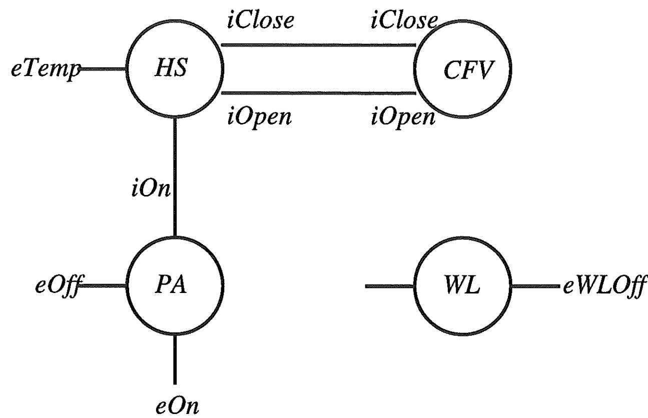


figure 34.1

Notice how the *WarningLight* process (WL) floats free of connections with other processes in the system. Why is this so?

34.1 Why is the System Diagram Partially Connected?

From the system diagram in figure 34.1, in the previous section, we can see that *WarningLight* is disjoint from the other three processes in the system. Before we connect *WarningLight* to the system we must consider how we shall connect it. We could link *WL* to both *HS* and *PA* using the same line of communication (i.e: via port *iOn*). If we perform this multi-way connection then shall we decide to use the *flashLight* port or some other (as yet undefined) port?

If we name the missing port on process *WL* flash*Light* then both *HS* and *CFV* could connect to it using multiple synchronisation between *HS*, *CFV* and *WL*. Remember that only CSP and LOTOS could be used to model this eventuality as they do not hide their inter-process communications, allowing them to cope with multiple synchronisations. The hiding of synchronising actions in CCS precludes it from being able to synchronise between more than two processes at any one time. A new structure for the control system using multiple synchronisation is show below:

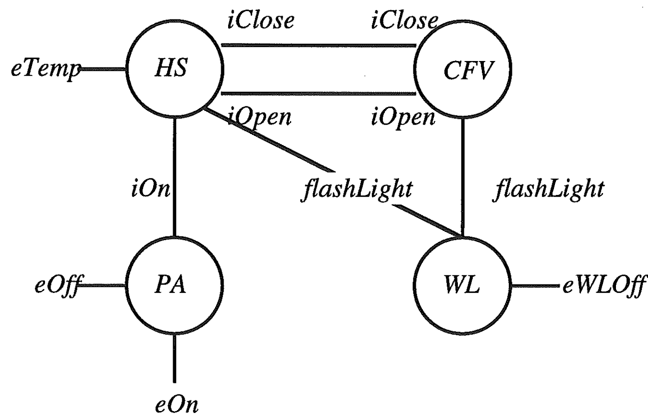


figure 341.1

The structure of the system appears to be fine except now we have a redefined interface to *HS* (assuming *CFV* and *WL* already have a port named *flashLight*). So, a potential problem already begins to show itself, the redefinition of the interface to a process simply by including a new process into an established system [18]. Another hazard is that now each of the three synchronising processes must be in a position to communicate with *WL* on port *flashLight* at the

same time before either process can continue with its own action sequence. Put simply, if only *CFV* and *WL* wish to synchronise (presumably because an attempt has been made to set the valve beyond its limits) then *HS* will get involved in the synchronisation, whether we want it to or not. The three processes are explicitly tied together via the multiple synchronisation.

Ideally, we would seek to keep the *HS* ↔ *WL* and *CFV* ↔ *WL* communication separate. To do this we could choose an alternative way of composing the processes together (via interleaving). Alternatively, we could consider synchronising *HS* with *WL* and *PA* on the same port and then connecting *CFV* and *WL* on a different port. However, the same problems occur, again due to multiple synchronisation. The focus of the problem simply shifts towards *HS*, *PA* and *WL* instead of *HS*, *CFV* and *WL*.

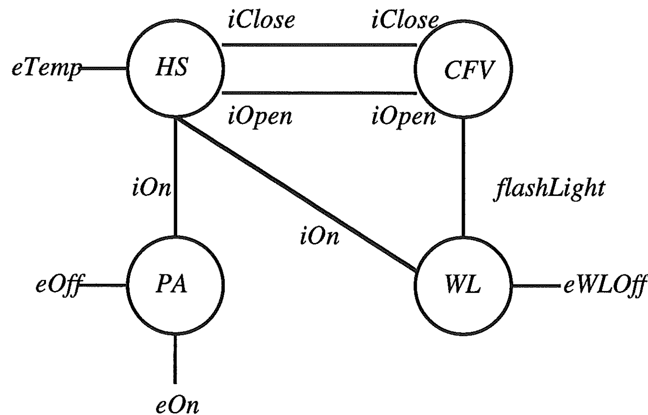


figure 341.2

Yet another possible solution to our ongoing problem would involve communications between *HS* and *WL* on one port, *HS* and *PA* on another port and *CFV* and *WL* on yet another port. This strategy would require a lot of redefinition of processes already in the system as the number of interface ports would increase per process. The resulting system would resemble figure 341.3.

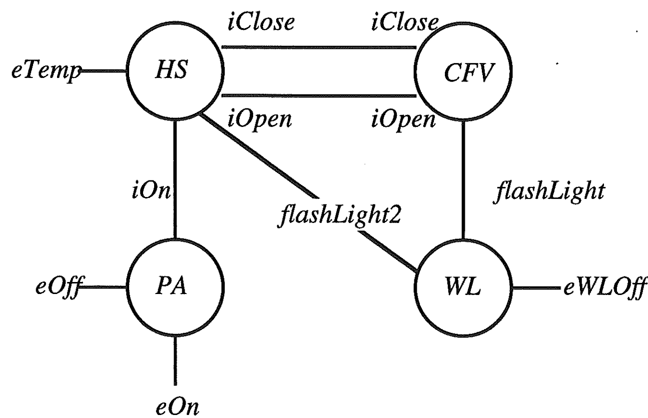


figure 341.3

Note the difference between the two *flashLight* communications (*flashLight* and *flashLight2*). These stop the name clash that would normally enforce multiple synchronisation, causing the earlier problem of all three processes synchronising together and holding one another up.

For much of this work involving communication and synchronisation it might be useful to think in terms of computer network architecture. One idea that we can experiment with is that of a shared common port. Simple bus architecture is elegant as it provides a flexible way of extending the number of processes in a system without the need to alter the interface to each

process. With a bus network we have the potential to communicate with every node sharing the same communication medium. It is also a flexible architecture in terms of expansion and modification [18]. The following diagram illustrates a new version of the nuclear reprocessing control system using a shared communications medium; namely *g*. All processes within the system communicate via this common port *g*.

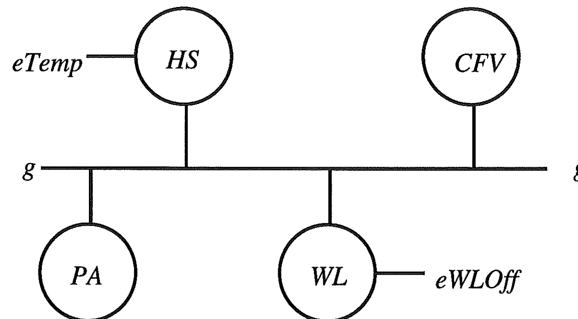


figure 341.4

But (and here is another problem) the power that a process has over other processes communicating on the same port is quite substantial. Any process on the same communications medium has the power to halt all of the other processes on that medium. If any one process crashes and the other processes are expecting to synchronise with that (recently deceased) process then the whole system is in trouble. Recall that in the current system, taking into account the semantics of both CSP and LOTOS, if one process fails then they all fail [18]!

To throw yet another iron in the fire let us now consider modifying the Heat Sensor process *HS* using casual inheritance. We now want *HS* to activate both the alarm (*PA*) and the warning light (*WL*) without any user interaction. The change in the system does not require an additional process but a new behaviour embedded within an existing process (namely *HS*). This modification is similar to the work carried out in section 30.1, *E301.3*.

For the modification I refer the reader to the semantics of atomic actions which state that an action must be allowed to complete before being interrupted by another process (first introduced in section 6). Remember that neither of the process algebras presented here offer true concurrency All processes execute a little at a time rather than all at once; akin to multi-programming rather than multi-processing. Recall that if **we** walk through an action sequence ourselves then **we're** the processor as well as being part of the environment!

I can give a grand name to the insertion of extra actions into an existing process action sequence; temporal behavioural modification. The task of getting the Heat Sensor (*HS*) to communicate with both *PA* and *WL* is a good example of this type of modification. Problems associated with temporal modification and its potential influence over synchronising processes have been discussed. It is possible to deadlock a system simply by adding some new process which then deadlocks, causing the entire system to do likewise. The domino effect occurs, bringing down the entire system. For this bus architecture method to work more formalisation of the process algebras is required to ensure that processes only have a limited influence on other processes.

—*end of section on Process Algebras with Object application.*

35 Conclusions

From the discussion in the last few sections we can see how easily the integrity of a system can be damaged by modifying the existing components or adding new components to the system

which then communicate with any existing processes. Indeed, what use would a new process have if it could not integrate into an existing system?

Throughout this lengthy text it has been my intention to create an interconnected view of the world which contains objects, as viewed from the modelling capabilities of three established process algebras. The languages of CSP and CCS are the least similar of the three we have studied, with LOTOS being derived from elements of both languages (although CSP having by far the greater influence over the way LOTOS looks and behaves).

For many of the examples contained within this text either language can be used as a formal representation. Only when we compose processes together does it become apparent that certain differences arise from the semantics of communication and action restriction.

With luck, if this text has done its job, you, the reader, can begin to see how the different process algebras relate to one another and to the complex task of modelling (what has become the growth area of) object-oriented systems. From this point onwards I am in a position to discuss the problems highlighted in earlier sections and concentrate on the issues surrounding the modification and extension of communicating systems.

Future papers and discussion of work in this area will shed more light on the world of process algebras and objects. With reference to the future, let's see what's out there and figure out how to formalise it!

References

[1] 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.

[2] Bergstra, J.A. and Klop, J.W. (Jan—Mar 1984). Process Algebras for Synchronous Communication. *Information and Control.* 60(1—3): 109—137.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] Hawking, S.W. (1988). *A Brief History of Time: from the Big Bang to Black Holes*. Bantam Press.

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

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

[17] 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.

[18] Taylor, P.N (July 1995). *The Analysis of Formal Models of Communication for the Specification of Reusable Systems*. **TR229**, Ph.D. Transfer Document, Division of Computer Science, University of Hertfordshire. U.K.

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