IMPLEMENTATIONS OF PROCESS SYNCHRONISATION, AND THEIR ANALYSIS

by

Kevin Mitchell

Doctor of Philosophy
University of Edinburgh
1985

Abstract

The implementation problems associated with the synchronised handshaking form of process communication are analysed within a formal setting. Milner's Calculus of Communicating Systems (CCS) is used as the vehicle for these investigations.

A single processor implementation of CCS is described and its adequacy as a programming language discussed. For the more general case of a distributed implementation, a subset of CCS is identified that admits a simple synchronisation scheme. The subset consists of those programs that possess a synchronising annotation. A method for constructing these annotations is developed and an implementation based on this approach is then proved correct.

A technique for synchronising arbitrary (static) CCS programs is developed involving program transformations. In order to prove the validity of these transformations, a new equivalence relation is proposed based on the testing approach of DeNicola and Hennessy. The new equivalence incorporates the notion of strong fairness while preserving the natural connections between parallelism and non-determinism as expressed by the expansion theorem in CCS. The meaning of transformational correctness within the CCS framework is also investigated. These developments are used to prove that an implementation scheme based on program transformations is indeed correct. The results are then extended to the case where the transformation is only partially applied to the source program, leading to an efficient implementation strategy.

Acknowledgements

I would like to thank first of all my supervisor, Robin Milner, for his help and encouragement in the development of this thesis. His concern for clarity and mathematical elegance has greatly influenced my work. I would also like to thank my colleagues at Edinburgh, especially Matthew Hennessy, Rocco DeNicola and Niklas Traub, for many helpful discussions. Finally, I would like to thank my wife Shirin whose support and patience played a major part in making this thesis possible.

This work was partially supported by a research studentship from the UK Science and Engineering Research Council.

Declaration

The research presented in this thesis is all my own, and previously unpublished, with minor exceptions as indicated in the text.

Table of Contents

List of Figures

Introduction

The last decade has seen a great increase in the demand for concurrent programming languages. In part, this has been due to the continual requirement for faster machines. By exploiting the parallelism inherent in many problems, multiprocessor systems may provide a cost effective solution to the performance problem. One approach is to detect this potential for parallel execution automatically, using sophisticated compilers. While there have been some notable successes, mainly in the area of numerical computations, the work has been hampered by the continued use of imperative, sequential programming languages. The presence of side-effects in these languages greatly increases the problems associated with the automatic detection of parallelism. This has led to a greater interest being shown in the purely functional programming languages.

An alternative to the functional approach is to allow the programmer to explicitly indicate the parallelism inherent in the problem by using a concurrent programming language. This has a number of advantages. For instance, the parallelism specified by the programmer may be of a more useful form than that derived automatically. Many parallel machines consist of a comparatively small number of powerful processors, and for these machines the detection of parallelism present in the evaluation of an arithmetic expression may be unusable, due to the overhead of the process mechanism. In such cases, a more global and higher-level form of parallelism must be exploited, and this is the level at which concurrent programming languages operate. If we wish to execute a program on a dataflow machine, then the parallelism present in arithmetic expressions will assume a much greater importance. This illustrates why the choice of programming language may be influenced by the underlying target hardware. By adding concurrent features to a functional language, we may achieve the advantages of both approaches. The level at which parallelism is then exploited is left to the compiler for the target machine.

An important benefit of concurrency is that it provides a useful structuring tool, and may greatly aid the program design process in a way similar to functional abstraction. This fact has been appreciated by real-time programmers for many years, but the lack of convenient and commonly available concurrent languages has inhibited its use by the wider programming community. This situation is gradually improving as languages such as Ada [DoD 80], Edison [Brinch 81], Modula [Wirth 77], and Occam [INMOS 84a] become more widely available.

It is a sad fact that nearly all sequential programs are never proved correct in any formal sense. This is in part due to the difficulty of the task, but also because, by careful construction, it is possible to build programs that are remarkably free of errors. The need for formal verification of concurrent programs is far more acute. Even simple concurrent programs of only a few lines may contain subtle errors and there have been some notable cases where numerous versions of an algorithm have been published before the correct version was obtained [Gries 77]. When this occurs for very small programs, the seriousness of the problem should be immediately apparent.

These problems have strongly influenced the language designers, resulting, for the most part, in languages with far cleaner and elegant features than are present in their sequential counterparts. Along with the development of these languages have come the associated proof methodologies and semantic techniques necessary to form the basis of theorem provers and other verification aids. These trends are most apparent in those languages developed from a mathematical, or theoretical, background such as CSP [Hoare 78], CCS [Milner 80] and Petri Nets [Peterson 77]. These languages are not judged purely on their syntactic convenience, or even on the simplicity of the underlying semantics. Their mathematical tractability, and the ease of performing proofs, are perhaps the most crucial factors influencing the success or failure of these languages.

We have identified the need for concurrent languages, and also why they must be designed to aid the proofs of the resulting programs. The question still remains as to what concurrent primitives to provide. Perhaps the simplest approach would be to introduce processes communicating through shared variables. Unfortunately, as a program structuring device, this approach leaves much to be desired. It is also extremely difficult to perform correctness proofs in such a framework. Such deficiencies led to the introduction of more structured forms of concurrency control such as semaphores [Dijkstra 65], [Habermann 72] and monitors [Brinch 73]. These enhancements, while successful as a conceptual aid to programming, did little to aid the verification of the resulting programs, although there has been some success in applying the axiomatic approach to program correctness [Owicki 75].

Petri nets [Peterson 77] were proposed as a way of studying concurrency at its most primitive levels. The formalism was inadequate as a programming language, at least in its original form, but it did illustrate how various properties could be checked for in a program, such as the presence of deadlocks. There now exists a large body of work concerning net theory, and numerous extensions of the original proposal have been developed, such as the various forms of stochastic Petri nets [Marsan 84].

Hoare and Milner have both developed concurrent languages based on the handshake model of communication. If two processes wish to communicate, then the sending of the message and its reception form a single indivisible action. The Hoare version, known as Communicating Sequential Processes (CSP) [Hoare 78], is based on an imperative view of the world, whereas Milner's version, Calculus of Communicating Systems (CCS) [Milner 80], is more applicative in nature. Both of these languages have developed a considerable body of associated semantic descriptions, proof techniques, axiomatisations, equivalence relations, and the like, that make them eminently suitable as a vehicle for the study of concurrency. Although, superficially, the two languages appear very different, at the level of the process synchronisation mechanism they have much in common. This thesis addresses itself to the problems of implementing the handshaking view of process communication as embodied in these languages. As such, the results are of relevance to the implementation of both CCS and CSP. Ada [DoD 80] and Occam [INMOS 84a] have based their concurrent primitives on CSP, and so the work is indirectly applicable to these languages as well. The examples and proofs presented in the thesis are based on CCS, as this language has developed furthest at the theoretical level.

Concurrency may be used as a method for decomposing and structuring solutions to problems. It is therefore important to develop implementations of concurrent languages on single processor machines, even though this will not result in any performance gain, and may result in a performance loss over the sequential version. The thesis starts with a discussion of the problems of implementing CCS on a single processor. The difficulties associated with the implementation of the concurrent primitives form only part of the problem. The need to provide a user interface to the system, and the extensions required to the calculus if it is to form a complete programming environment, also raise interesting questions. An example of a programming language based on CCS is used to illustrate these problems.

Sequential implementations of concurrent languages are adequate if the languages are used purely as program structuring aids. The second aim of concurrent languages, namely the potential for performance improvements, requires the development of distributed implementations. Unfortunately, mathematical elegance does not necessarily imply ease of implementation. In particular, the implementation of the handshaking view of process synchronisation is a non-trivial problem on truly distributed systems (those that possess no shared memory). This thesis illustrates the implementation difficulties and reviews the previous algorithms aimed at solving this problem.

An alternative method of implementing languages such as CCS and

CSP is proposed. It relies on the development of program transformations that produce, as results, programs that are easier to implement in some sense. A subset of CCS is exhibited that admits a very simple implementation strategy, and an existing implementation scheme is presented in the form of a transformation that produces programs contained within this subset. By partially applying the transformation, an efficient implementation of CCS programs may be derived in many cases. A correctness proof of the transformation is complicated because, under certain circumstances, the transformed program may not terminate, even though the original program always does so. Existing definitions of process equivalence are inadequate in this respect, and so new notions of implementation and transformation equivalence must be introduced. Developing these ideas and performing the associated correctness proofs constitutes the main body of this thesis.

The remainder of the introduction is devoted to a summary of the work presented in this thesis.

In Chapter 1 the basic concepts and notations concerning CCS are introduced. The concurrent operators of CCS are defined, along with an operational semantics. The definitions of strong and observational equivalence in terms of recurrence relations are then described. An alternative to these definitions, in terms of maximal fixed points, is introduced leading to the notion of bisimulation and its use as a proof technique. Deficiencies in these equivalences are then highlighted and the testing view of process equivalence, as proposed by DeNicola and Hennessy, is introduced. No new work is presented in this chapter and those readers already familiar with the literature surrounding CCS may safely proceed directly to Chapter 2.

Chapter 2 deals with the problems associated with implementing CCS on a single processor. A particular example of a CCS implementation, the Chalmers PFL system, is described and some examples of concurrent programs written in PFL presented. This system consists of an embedding of CCS in the functional language ML, and so the chapter includes a brief introduction to those features of ML used in the PFL examples. Finally, some extensions and deficiencies of the current implementation are discussed.

Chapter 3 deals with the more general case of implementing CCS on a distributed system. The analysis is restricted to Static CCS, the subset of CCS with no dynamic process creation, as even in this case there are considerable difficulties involved in an implementation. The problems of synchronising CCS processes are discussed, and a subset of the language exhibited that can be implemented efficiently. This subset consists of those programs for which a synchronising annotation can be found. We show how a number of previously proposed implementation schemes are particular instances of this approach. An algorithm is developed for determining these annotations under certain simplifying assumptions. We then discuss the approaches that may be followed when a synchronising annotation cannot be found for the program under investigation. The algorithms previously proposed as solutions to this problem are first reviewed, and then an alternative approach is described involving the transformation of CCS programs. We show how the resulting transformed terms may be easier to synchronise than the original program. In particular, we present the synchronisation scheme due to Schwarz in the form of a transformation, and show how synchronising annotations may always be constructed for the resulting terms. The advantages of the transformational approach to process synchronisation are outlined, such as the possibility of partial application. Finally, we discuss the problems involved in proving transformations correct. The Schwarz transformation introduces the possibility of non-termination, for example, and so requires some fairness assumptions in order to prove its correctness.

The problems of transformational correctness in CCS lead naturally to the theoretical investigations of Chapter 4. The notion of implementation is first investigated, and the inadequacy of this definition in the presence of diverging processes discussed. The need for some form of fairness analysis is explained, followed by a brief review of the relevant work in this area. We argue that existing notions of fairness for CCS are inadequate because they do not preserve the expansion theorem that relates the parallel composition and non-determinism operators of CCS. An alternative approach, based on DeNicola and Hennessy's testing preorders, is proposed that captures the desired fairness properties. Various algebraic properties are exhibited for the new preorder in order to show that it is well behaved in some sense. Use of the weak testing preorder is hampered by the necessity to deal explicitly with tests, and so an investigation is carried out into the possibility of an alternative characterisation that admits a simple proof technique. The equivalence proposed by Kennaway is investigated and rejected, although a connection is shown between the two approaches under certain restrictive conditions. Two further preorders are proposed that imply the weak testing preorder, and may be proved using bisimulation techniques. Finally, the problems of transformational correctness are investigated, and a new definition presented based on the weak testing preorder.

Chapter 5 makes use of these new definitions to prove that the Schwarz synchronisation scheme is correct when presented in the form of a transformation. The structure of the problem rules out the possibility of an inductive proof, and so notation is developed to allow the bisimulation proof to be carried out on the whole program. The chapter concludes with a discussion of how the transformation may be partially applied to a program so as to minimise the synchronisation overhead.

CHAPTER 1
Preliminary Definitions

§1.1 Introduction

This chapter aims to provide a brief overview of CCS, and the theoretical work based on the language, where it is of relevance to the work that follows. CCS (Calculus of Communicating Systems) is a mathematical calculus designed to aid the specification of concurrent systems and their subsequent analysis. This desire to not only specify, but also to reason about concurrent systems has lead to a large emphasis being placed on the mathematical tractability of CCS. In particular, a number of equivalence relations have been proposed for the language, each attempting to highlight some particular aspects of the concurrent programs under investigation. The chapter starts by describing the language and a particular method for generating equivalences known as Bisimulation [Park 81]. This method is used to define strong($\sim$) and weak ($\approx$) equivalences for the language. Some deficiencies of these equivalences are identified, leading on naturally to a discussion of DeNicola and Hennessy's testing view of process equivalence ($\simeq_i$).

§1.2 CCS

We start by reviewing the definition of CCS and its operational semantics. CCS deals with systems of computing agents communicating via named ports that are connected by channels. The communication channels have no buffering capacity, and are unidirectional. Communication takes the form of a value-passing act, requiring the simultaneous co-operation of both the sender and the receiver.

The simplest agent, denoted by NIL, can perform no actions whatsoever. This agent, along with recursion, forms the basis from which all other agents are constructed. Given an arbitrary agent p, $\alpha.p$ represents the agent that may first communicate with another process through the α port,. and if successful will then evolve to the agent p. Agents may be composed through the bar operator, |, and in the resulting agent, p|q, p and q may proceed independently. However, they may also communicate with each other via *matching* ports, where the exact form of this matching will be explained shortly. To allow some degree of choice and non-determinism, the language also includes the + operator, where p+q represents either the process p or the process q. Depending on the actions offered by p and q, this choice may or may not be resolvable externally. The names of the ports used by each process are significant in that they affect the communication potential of the process (i.e. the names can be viewed as forming the channel linkage mechanism). Thus, to be able to define a generic agent, and then use it in different contexts, we need to be able to relabel the agent's ports to form the desired connections with its context. This is achieved by the postfixed renaming operator [S], where S is a port renaming function. Finally, to prevent channels from forming when not required, the language has a hiding, or scoping mechanism known as restriction. Thus, $p\backslash\alpha$ hides the label α so that the resulting agent cannot communicate to its external environment through the α port.

We now express these ideas more formally. Let us assume the existence of a fixed set of <u>names</u>, Δ, ranged over by α, β, ... The set of <u>co-names</u>, $\overline{\Delta}$, disjoint from Δ, is constructed using the bijection $\bar{\ }$, where

$$\alpha\in\Delta \supset \overline{\alpha}\in\overline{\Delta}$$

We refer to $\overline{\alpha}$ as the complement of α. We also use $\bar{\ }$ for the inverse bijection and hence $\overline{\overline{\alpha}} = \alpha$.

We define $\mathcal{A}ct$, the set of visible actions or labels, to be $\Delta\cup\overline{\Delta}$. We also introduce the label τ, where τ is a distinguished action not occurring in $\mathcal{A}ct$. We will allow μ to range over $\mathcal{A}ct\cup\{\tau\}$ and λ to range over $\mathcal{A}ct$.

We define a function, S, over $\mathcal{A}c\ell\cup\{\tau\}$ to be a <u>relabelling</u> if

i) $S(\tau) = \tau$

ii) S is a bijection

iii) S respects complements,
i.e. $\overline{S(\lambda)} = S(\bar{\lambda})$ for $\lambda,\bar{\lambda}\in\mathcal{A}c\ell$

We may now introduce the operator set for CCS. Let

$\Sigma_0 = \{\text{NIL}\}$

$\Sigma_1 = \{\mu. \mid \mu\in\mathcal{A}c\ell\cup\{\tau\}\} \cup \{[S] \mid S \text{ is a relabelling}\} \cup \{\backslash\lambda \mid \lambda\in\mathcal{A}c\ell\}$

$\Sigma_2 = \{ + , \mid \}$

$\Sigma_n = \phi,\ n\geq 3$

where Σ denotes $\bigcup\{\Sigma_k \mid k\geq 0\}$

Let X be a set of variables, ranged over by x. The set of recursive terms over Σ, ranged over by t, is defined by the following BNF-like notation:

$$t ::= x \mid op(t_1, \ldots, t_k),\ op\in\Sigma_k \mid \text{fix } x.t$$

The operator fix x._ binds occurrences of x in the subterm t of fix x.t, and introduces the usual notions of free and bound variables in a term. A term is said to be <u>closed</u> if it contains no free variables. We call such terms <u>agents</u> and will use $\mathcal{P}$ to stand for the class of agents. The terms <u>behaviour</u> and <u>process</u> will be used as alternatives for agent throughout the rest of the thesis. The operational semantics for CCS is given in terms of labelled rewrite rules over these agents([Milner 80], [DeNicola 82]). For each $\mu\in\mathcal{A}c\ell\cup\{\tau\}$, we define a binary relation $\xrightarrow{\mu}$ over $\mathcal{P}$. We interpret $p\xrightarrow{\mu}q$ as "agent p performs the action μ, and in doing so evolves to agent q".

Let $\xrightarrow{\mu}$ be the least relation over $\mathcal{P}$ that satisfies

i) $\mu.p \xrightarrow{\mu} p$

ii) $p\xrightarrow{\mu}p'$ implies $p + q \xrightarrow{\mu} p'$
$q + p \xrightarrow{\mu} p'$
$p \mid q \xrightarrow{\mu} p' \mid q$
$q \mid p \xrightarrow{\mu} q \mid p'$

iii) $p \xrightarrow{\mu} p'$ implies $p[S] \xrightarrow{S(\mu)} p'[S]$

iv) $p \xrightarrow{\lambda} p' \wedge \lambda \notin \{\gamma, \bar{\gamma}\}$ implies $p\backslash\gamma \xrightarrow{\lambda} p'\backslash\gamma$

v) $p \xrightarrow{\lambda} p'$, $q \xrightarrow{\bar{\lambda}} q'$ implies $p \mid q \xrightarrow{\tau} p' \mid q'$

vi) $t[\underline{\text{fix}}\ x.t/x] \xrightarrow{\mu} p$ implies $\underline{\text{fix}}\ x.t \xrightarrow{\mu} p$.

We extend this relation to sequences of actions in the obvious way.

It will often prove convenient to ignore the τ actions in a sequence of transitions. We define the $\overset{\mu}{\Longrightarrow}$ relation by

$p \overset{\mu}{\Longrightarrow} q$ iff there exists p', q' such that

$$p\ (\xrightarrow{\tau})^{*}\ p'\ \xrightarrow{\mu}\ q'\ (\xrightarrow{\tau})^{*}\ q$$

We extend this relation to sequences of actions in the obvious way and define ε to be the empty sequence. A question arises as to whether $p \overset{\tau}{\Longrightarrow} p$ is always true. Certainly $p \overset{\varepsilon}{\Longrightarrow} p$ is always possible, but, by the preceding definition, $\overset{\tau}{\Longrightarrow}$ is a sequence with at least one τ transition and so $p \overset{\tau}{\Longrightarrow} p$ will not be true in general. This is the approach taken by Milner [Milner 80]. Unfortunately, Hennessy and DeNicola treat τ as a special case and equate $\overset{\varepsilon}{\Longrightarrow}$ and $\overset{\tau}{\Longrightarrow}$, as this simplifies their proofs [DeNicola 82]. We will also adopt this convention for the same reason. This subtle but important difference in convention has created some confusion in the past, and the reader should be aware of this point when reviewing the current CCS literature. We use $p \overset{\lambda}{\not\longrightarrow}$ to indicate that p cannot perform a λ as its first action, and $p \overset{\lambda}{\not\Longrightarrow}$ to indicate that no sequence of silent moves will enable p to reach a state where it can perform a λ action.

We define the set of <u>derivatives</u> of p to be

$\{p' \mid \exists s.\ p \overset{s}{\Longrightarrow} p'\}$

and the set of initial moves to be

$\text{Init}(p) = \{\alpha \in \mathcal{Act} \mid p \overset{\alpha}{\Longrightarrow}\}$

A <u>sort</u> of an agent is a set of labels which contains all labels through which communication can possibly (but may not) occur. If an agent p has the sort L (written p:L) then it will prove convenient to allow p to possess all larger sorts containing L as well. The minimal sort for an arbitrary agent can be computed using an iterative closure algorithm, and it is simple to show that finitely expressible agents have finite sorts.

The version of CCS presented so far deals with pure synchronisation signals; no values are passed between the agents. It is simple to extend the calculus to the general case. We adopt the convention that $\alpha x.p$ and $\bar{\alpha}v.p$ denote agents that input and output values on the α port respectively, where in the first case the resulting input value is bound to x throughout the agent p. Let $t\{u/x\}$ denote the term which results from substituting u for every free occurrence of x in t. We extend the binary relation $\xrightarrow{\mu}$ to allow values and variables and write it as $\xrightarrow{\mu v}$. For the most part the definition of $\xrightarrow{\mu v}$ is a simple extension of the $\xrightarrow{\mu}$ definition except for the following cases.

$$\alpha x.p \xrightarrow{\alpha v} p\{v/x\}$$
$$\bar{\alpha}E.p \xrightarrow{\bar{\alpha}v} p \quad \text{where the expression E evaluates to v}$$
$$\tau.p \xrightarrow{\tau} p$$

We shall refer to a label, together with a variable or value expression, as a guard. We will sometimes use the CSP notation for value passing when this is convenient. Thus, $\alpha!3.p$ and $\beta?x.q$ are alternative ways of writing $\bar{\alpha}3.p$ and $\beta x.q$ respectively.

When mapping CCS onto finite resources, such as real processors, it may not be possible to support dynamic process creation. Furthermore, certain transformations on CCS terms may require all processes to be transformed simultaneously, thus precluding the use of dynamic process creation. Such cases are sufficiently frequent that we define a subcalculus of CCS, known as <u>Static</u> <u>CCS</u>, that only allows the use of | in a restricted form.

Let $\prod_{i\in n} p_i$ denote the CCS term $p_1 \mid p_2 \mid \ldots \mid p_n$. Furthermore, we extend the restriction notation to allow sets of actions, as in $p\backslash L$ for any $L \subseteq \mathcal{Act}$.

Definition

An agent $p \in \mathcal{P}$ is <u>static</u> if the parallel composition operator, |, is not used in its construction.

An agent $p \in \mathcal{P}$ is a member of <u>Static</u> <u>CCS</u> if it is syntactically of the form $(\prod_{i \in N} p_i)\backslash L$ for some set of static processes $\{p_i | i \in N\}$ and $L \subseteq \mathcal{A}ct$.

Note that if we only required bounded parallelism, then this could be ensured by not allowing the | operator inside the body of a recursion, which is a slightly weaker requirement than a process being a member of Static CCS.

§1.3 Strong and observational equivalences

There have been many equivalences proposed for CCS, each one accentuating some different aspects of the processes under examination. Furthermore, some of these are also congruences (i.e. equivalences that are preserved by the substitution of equivalent programs), while for others we must explicitly derive the congruence from the equivalence. We start by defining the original two equivalences for CCS, known as the strong and observational equivalences. Milner [Milner 80] describes <u>strong equivalence</u>, '$\sim$', in terms of a decreasing sequence $\sim_0, \sim_1, \ldots, \sim_k, \ldots$ of equivalence relations as follows.

$p \sim_0 q$ is always true.

$p \sim_{k+1} q$ iff for all μ, v
  (i) if $p \xrightarrow{\mu v} p'$ then for some q', $q \xrightarrow{\mu v} q'$ and $p' \sim_k q'$
  (ii) if $q \xrightarrow{\mu v} q'$ then for some p', $p \xrightarrow{\mu v} p'$ and $p' \sim_k q'$

$p \sim q$ iff $\forall k \geq 0.\ p \sim_k q$

This equivalence has a number of desirable properties, such as $p|NIL \sim p$, $p+NIL \sim p$, that ease program proofs. Using $\sim$ we may derive what

Milner describes as the Expansion Theorem [Milner 80]. This theorem provides a connection between the parallel composition and the summation operators. Summarising briefly, if $B = (B_1| \ldots |B_m)\backslash A$, where each B_i is a choice of guards, then

$$B \sim \sum\{\mu v.((B_1| \ldots |B_i'| \ldots |B_m)\backslash A) \text{ where } \mu v.B_i' \text{ is a summand of } B_i \text{ and } \mu,\bar{\mu}\notin A\}$$

$$+ \sum\{\tau.((B_1| \ldots |B_i'\{\tilde{E}/\tilde{x}\}| \ldots |B_j'| \ldots |B_m)\backslash A) \text{ where } \alpha\tilde{x}.B_i' \text{ is a summand of } B_i, \bar{\alpha}\tilde{E}.B_j' \text{ is a summand of } B_j, i\neq j\}$$

provided that in the first term no free variable in $B_k(k\neq i)$ is bound by μv.

Unfortunately, for many problems $\sim$ is too restrictive an equivalence. For example, $\alpha.p$ and $\alpha.\tau.p$ are not equated by strong equivalence. The problem is caused by the silent τ actions, as we frequently wish to ignore them. This prompted the development of the observational equivalence, '$\approx$', so called because the τ actions are not observable by an external agent. Milner defines the equivalence as follows.

$p \approx_0 q$ is always true.

$p \approx_{k+1} q$ iff for all $s\in(\mathcal{Act} \times V)^*$

(i) if $p \overset{s}{\Longrightarrow} p'$ then for some q', $q \overset{s}{\Longrightarrow} q'$ and $p' \approx_k q'$

(ii) if $q \overset{s}{\Longrightarrow} q'$ then for some p', $p \overset{s}{\Longrightarrow} p'$ and $p' \approx_k q'$

$p \approx q$ iff $\forall k\geq 0.\ p \approx_k q$

It is simple to show that $\alpha.p \approx \alpha.\tau.p$. Furthermore $\sim\subset\approx$. Unfortunately, $\approx$ is not a congruence under the + operator. To see why, we note that

$$\text{NIL} \approx \tau.\text{NIL}$$

but

$$\text{NIL} + \alpha.\text{NIL} \not\approx_2 \tau.\text{NIL} + \alpha.\text{NIL}$$

Milner defines $\approx^c$ to be the weakest congruence stronger than (smaller than) $\approx$ and shows that $\approx^c$ and $\approx^+$ are identical, where

$$p \approx^+ q \text{ iff } \forall r.\ p+r \approx q+r$$

§1.4 Bisimulations

We may redefine the latter half of the $\approx$ definition to be of the form

$$\approx_{k+1} = E(\approx_k)$$

where

$$\begin{aligned} E(R) = \{ <p,q> \mid\ & p \xRightarrow{s} p' \supset \exists q'.\ (q \xRightarrow{s} q' \wedge <p',q'> \in R) \wedge \\ & q \xRightarrow{s} q' \supset \exists p'.\ (p \xRightarrow{s} p' \wedge <p',q'> \in R) \} \end{aligned}$$
$$\text{for } s \in \mathcal{Act}^*$$

One anomaly of the $\approx$ equivalence is that it is not a fixed point of E. No simple example exists of two behaviours, p and q, such that $p \approx q$ but $<p,q> \notin E(\approx)$. However, Milner has exhibited such a pair [Sanderson 82], although the example is rather unnatural. Park [Park 81] has suggested an alternative definition of observational equivalence by considering the maximal fixed-point of E using the partial ordering of set inclusion. We can show that the function E is monotonic and so this is sufficient [Tarski 55] to deduce that a maximal fixed point for E exists given by

$$\bigcup \{R \mid R \subseteq E(R)\}$$

This leads to the following, alternative, definition of observational equivalence.

$$\approx\ = \bigcup \{R \mid R \subseteq E(R)\}$$

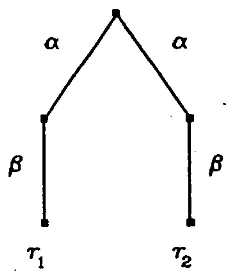
In fact, for practical purposes, these two definitions of $\approx$ appear to be identical. However, we shall show that the fixed-point version of the equivalence admits a simple yet powerful proof technique. The same fixed-point can in fact be obtained by a simpler version of E where only single actions are considered rather than arbitrary strings, i.e. we can replace the definition of E by

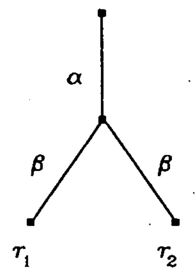
$$E(R) = \{ <x,y> \mid x \overset{\mu}{\Longrightarrow} x' \supset \exists y'.\ (y \overset{\mu}{\Longrightarrow} y' \wedge <x',y'>\in R) \wedge$$
$$y \overset{\mu}{\Longrightarrow} y' \supset \exists x'.\ (x \overset{\mu}{\Longrightarrow} x' \wedge <x',y'>\in R) \}$$
for $\mu \in \mathcal{A}ct \cup \tau$

Defining observational equivalence (and also strong equivalence) using the fixed-point approach leads to a very powerful and elegant proof technique known as bisimulation. To prove that $p \approx q$, it is sufficient to construct a relation R such that $<p,q> \in R$ and $R \subseteq E(R)$. Following Park [Park 81], we refer to such a relation R as a <u>bisimulation</u> between p and q. This technique forms one of the main proof methods for CCS as the examples in [Sanderson 82], [Backhouse 83] and [Prasad 84] illustrate. Furthermore, Sanderson [Sanderson 82] has proposed an algorithm that allows the construction of bisimulation relations to be carried out mechanically in some cases.

§1.5 Testing equivalences

For many examples the $\approx$ equivalence is too particular about when non-deterministic choices are made. Consider the two behaviours





Then $p \not\sim q$ and $p \not\approx q$. However, it is not clear why we should distinguish between these two processes. In either case an α followed by a β move is possible followed by either process r_1 or r_2. In neither case can the external environment force the choice of whether r_1 or r_2 is executed. In order to remedy this deficiency of the $\approx$ equivalence, Kennaway [Kennaway 81] developed an alternative definition of observational equivalence involving sets of processes. We discuss this approach more thoroughly in Chapter 4.

DeNicola and Hennessy [DeNicola 82] have also proposed an equivalence that equates these two processes. However, in their case, this property was merely a byproduct of their more general view of what process equivalence should mean for languages such as CCS. Their basic premise is that two processes are equivalent if they are indistinguishable when tested by another agent. A process is tested by placing it in parallel with an observer process, where the observer agent has a distinguished action $\surd$ in its sort (written ω in [DeNicola 82]). A test succeeds if the combined processes reach a state where the $\surd$ move is possible through a sequence of silent actions.

We formalise these intuitions as follows. We denote by $\mathcal{O}$ the set of agents that may be constructed from the CCS operator set augmented with the action $\surd$ (i.e. $\mathcal{P}\subset\mathcal{O}$). A term p is <u>successful</u> if it may perform a $\surd$ action, i.e. $\exists p'.\ p \xrightarrow{\surd} p'$.

A <u>computation</u> is any finite or infinite sequence of terms $\{p_n | n\geq 0\}$ such that $p_n \xrightarrow{\tau} p_{n+1}$, whenever p_{n+1} is defined, and if p_{n+1} is not defined, then $p_n \xrightarrow{\tau} p'$ for no p'.

A <u>computation</u> is <u>successful</u> if one of its states is successful, and the set of successful computations is denoted by $\mathcal{Success}$.

For any $p\in\mathcal{P}$, $o\in\mathcal{O}$, we define <u>$\mathcal{Comp}(p,o)$</u> to be the set of computations whose initial element is the term (p|o).

We can distinguish two classes of tests on the process p; those that <u>may</u> succeed (indicated by at least one successful computation in $\mathcal{Comp}(p,o)$), and those that <u>must</u> succeed (where all computations in $\mathcal{Comp}(p,o)$ are successful). This leads to the following three preorders. We first define

i) p <u>*may satisfy*</u> o iff $(p|o)\ (\xrightarrow{\tau})^*\ q$ for some q such that $q \xrightarrow{\surd}$

ii) p <u>*must satisfy*</u> o iff whenever $p|o = p_0|o_0 \xrightarrow{\tau} p_1|o_1 \xrightarrow{\tau} \ldots$ is a computation from p|o then $\exists n\geq 0$ such that $o_n \xrightarrow{\surd}$

or equivalently,

i) p <u>*may satisfy*</u> o iff $\exists c\in\mathcal{Comp}(p,o)$ s.t. $c\in\mathcal{Success}$

ii) p *must satisfy* o iff $\mathcal{C}omp(p,o) \subset \mathcal{S}uccess$

Then

$p \sqsubseteq_3 q$ if $\forall o \in \mathcal{O}$ p *may satisfy* o implies q *may satisfy* o

$p \sqsubseteq_2 q$ if $\forall o \in \mathcal{O}$ p *must satisfy* o implies q *must satisfy* o

$p \sqsubseteq_1 q$ if $p \sqsubseteq_3 q \wedge p \sqsubseteq_2 q$

Note that $\alpha.NIL|\tau^\omega$ *must satisfy* $\bar{\alpha}.\surd.NIL$ is false (written $\alpha.NIL|\tau^\omega$ *must/satisfy* $\bar{\alpha}.\surd.NIL$), because $\mathcal{C}omp(\alpha.NIL|\tau^\omega, \bar{\alpha}.\surd.NIL)$ has τ^ω as one of its computations.

The preorders may be extended to equivalences in the obvious way, i.e.

$$p \simeq_3 q \text{ if } p \sqsubseteq_3 q \wedge q \sqsubseteq_3 p$$
$$p \simeq_2 q \text{ if } p \sqsubseteq_2 q \wedge q \sqsubseteq_2 p$$
$$p \simeq_1 q \text{ if } p \sqsubseteq_1 q \wedge q \sqsubseteq_1 p$$

The definitions of $\sqsubseteq_i$ presented in [DeNicola 82] are complicated by the explicit treatment of unguarded recursion and divergence. However, the simplified definitions presented above are sufficient to give some indication of the general approach taken by DeNicola and Hennessy. It has been shown that $\simeq_3$ coincides with $\approx_1$ and $\simeq_1$ lies between $\approx_1$ and $\approx_2$. This indicates that the observational equivalence distinguishes between more terms than the testing equivalence.

The main work of this thesis now begins, starting with an investigation of how to implement CCS on a centralised machine.

CHAPTER 2
Implementing CCS on a Single Processor

§2.1 Introduction

The primary aim of this thesis is to analyse the various approaches to implementing Static CCS on a distributed system. However, for completeness, we start by discussing the difficulties associated with providing an implemention of CCS on a single processor. Part of this work will be relevant to the more general case, as we show how CCS may be embedded in a functional language to form a complete programming system.

In its intended role as a simple concurrent calculus, CCS is very successful. Each user of the calculus extends it with features appropriate to the problem domain under investigation. However, in order to implement the calculus as a programming language, we must be a lot more specific about areas such as the syntax, that tend to be neglected when the only manipulations performed on the programs are by hand. CCS is inadequate as a programming language for a number of reasons. Firstly, the syntax is very restricted. There is no facility for local declarations, for example, which may greatly improve the clarity of a program. Secondly, the calculus does not deal with the introduction and use of new data types. In fact it is not even specified what data types are provided as primitives of the language. There is no mention of how to connect a process to the external environment of printers, keyboards etc. One desirable, but not essential, facility that is omitted from the calculus is some form of static typechecking that would prevent one process from sending a value of type t_1 and the receiver expecting a value of type t_2. All of these omissions are understandable as CCS was originally designed as a minimal calculus for reasoning about concurrent systems. These points illustrate why we must extend the calculus if we wish to produce an acceptable programming language.

There are two ways to tackle this task. The first would be to develop an implementation of CCS from scratch, adding extensions as required, until an acceptable programming environment was constructed. The problem with this approach is that the sequential part of the language would almost certainly end up forming a programming language in its own right. The resulting implementation effort would therefore be considerable. The second approach would be to take an existing language as the sequential part of CCS, and embed the concurrent operators within this system. This approach has the advantage that the difficulties in implementing the concurrent operators are not obscured by decisions involving the sequential subset of the language.

Holmstrom [Holmstrom 83] tackled this problem by embedding CCS in the applicative language ML [Gordon 79]. Although the input and output primitives of CCS are imperative, the rest of the calculus has an applicative flavour due to the similarity of value binding in CCS and the Lambda Calculus. Thus a functional language was a natural choice as the embedding language. Furthermore, ML is a strongly typed language with sophisticated data abstraction facilities which make it an ideal candidate for this role. The Holmstrom system, known as PFL (Parallel Functional Language), was built on top of an existing ML system which constrained the implementation in a number of ways. Firstly, the new syntax to handle the CCS constructs was cumbersome, as the primitives were encoded in the existing ML syntax. This encoding made extensive use of the continuation style of programming to simulate call-by-name value passing in a call-by-value environment. Secondly, the processes were rescheduled only when a data transfer took place between two components. Therefore, in the worst case, the system would hang if a process entered a non-terminating computation that performed no communications.

To resolve these difficulties, the author reimplemented PFL on top of an ML-in-ML compiler [Mitchell 85]. In this implementation, the concurrent operators were built into the underlying compiler resulting in a more faithful expression of the original CCS primitives. It is this version of PFL, described in [Mitchell 84], that we use as an example of a CCS implementation in this chapter. Many of the underlying ideas are based on the original Holmstrom implementation of PFL to which we are indebted.

§2.2 A short introduction to ML

This section briefly introduces the functional language ML. It is not intended to provide a complete description of the language; for this the reader is directed to [Gordon 79], [Cardelli 82], [Milner 84]. However we hope to give some flavour of the language and sufficient detail to enable the reader to understand the PFL examples.

ML was originally intended as a metalanguage for the LCF theorem prover [Gordon 79]. However it quickly established itself as a programming language in its own right. This was in part due to certain features in the language, while originally designed to aid the LCF system, gaining wider popularity in the programming community. These included the sophisticated static typechecking, the data abstraction facilities, and the failure, or exception, mechanism.

At this point it was a natural progression to reimplement ML as a stand-alone programming language. This work was performed by Cardelli [Cardelli 82] and he took the opportunity to extend the syntax of the language, particularly in the area of environment constructors. He also introduced two new primitive data types, the labelled record and variant.

Recently there has been an attempt to rationalise the existing ML systems and the Hope language [Burstall 80] resulting in a new version of ML known as Standard ML [Milner 84]. We shall describe the Cardelli version of ML simply because this is what the ML-in-ML compiler implements, and it is this system that has served as a test bed for our PFL experiments.

First and foremost, ML is an interactive, strongly-typed language. However, unlike the typing systems in languages such as Pascal and Ada, the ML system does not require that the user specify any type information for most expressions. It is the responsibility of the type checking phase of the compiler to infer this information from the user's program. For example, consider the expression

```
- [1+2; 3];
```

The ML system on receiving this expression would perform the following analysis. Firstly, + is a binary operator requiring two integers as arguments and producing an integer as a result. 1 and 2 are both integers and so 1+2 must represent an integer. Given a list of elements $e_1, \ldots, e_n$ of type t, $[e_1; \ldots ;e_n]$ constructs a list of these elements of type t list. In this case the first element of the list is an integer and so the type checker examines the remaining elements (in this case just the element '3') to ensure that they all have the same type. This is indeed the case and so the type of the whole expression must be an integer list. It passes this information back to the user, along with the evaluated result.

```
> [3;3] : int list
```

The description given above is a simplification of the truth, as, in practice, the type checker unifies types rather than performing exact matching of types. Such details do not aid the understanding of the PFL examples that follow, and, therefore, the description of ML presented in this chapter ignores such matters.

Values and functions can be defined, and functions applied, as follows.

```
- let a = 3;
> a = 3 : int;

- let rec f(x) = if x = 0 then 1 else x*f(x-1);
> f = \ : int -> int
```

```
- f(a);
> 6 : int
```

Unnamed functions can be introduced by the \ construct (where \ is meant to represent λ). The function suc that increments its argument by 1 could be defined in either of the following two equivalent ways.

```
- let suc x = x + 1;
> suc = \ : int -> int

- let suc = \x.x + 1;
> suc = \ : int -> int
```

Consider the function definition

```
- let add(x,y) = x + y;
```

We can view this definition as stating that add takes two arguments, x and y, and returns their sum. Alternatively, we might view add as taking a single argument that is a pair. This is the view taken by ML. The comma infix operator constructs pairs or tuples, and the corresponding type constructor is denoted by #. Thus the system would respond with

```
> add = \ : int # int -> int
```

Sometimes it is desirable to specify a function that takes an argument but subsequently ignores it. This is achieved by the _ construct, as in the following function that returns the first of a pair of arguments.

```
- let fst(x,_) = x;
```

To provide a complete description of ML would take a chapter in its own right. However, the features described above should be sufficient for an understanding of the PFL programs that follow.

§2.3 PFL, an embedding of CCS in ML

CCS consists of a parallel composition operator, |, a summation operator, +, an action operator, . , a restriction operator, \, and a renaming operator, []. We ignore recursion as it does not affect this discussion. One approach to merging CCS and ML would be to simply take their union in some sense. However, this would involve duplication of some of the underlying concepts, as we shall now show.

The primary aim of the restriction operator is to limit the scope or visibility of an action. However, ML already has a static scoping mechanism, and so it would be unwise to incorporate two similar concepts in a single language. The renaming operator allows the interface to a behaviour to be relabelled. But again this is similar to a concept already existing in ML, namely functional, or lambda abstraction. Instead of constructing a behaviour and then relabelling the interface, we can construct a function that, when applied to a collection of ports, returns the appropriate behaviour. To do this we require two new types, one for ports and one for behaviours. We extend ML with the primitive data types <u>beh</u> and * <u>chan</u>, where we assume that every port is of type * <u>chan</u> for some type *. For example, a port of type <u>int</u> <u>chan</u> can only pass values of type int(eger).

CCS uses the . operator for action prefixing and employs the overbar notation ¯ to indicate that a value is to be output. Thus $\bar{\alpha}_3.NIL$ and $\alpha_x.NIL$ denote the processes that output a value and receive a value on the α port respectively. The ¯ operator would be impractical in a programming language, and so some alternative must be sought. The CSP convention of ? for input and ! for output are attractive, but unfortunately clash with existing uses of ? and ! in ML. This illustrates one of the difficulties of embedding CCS in an existing language. The syntax eventually chosen in the Edinburgh PFL system was as follows.

a <u>inp</u> x. p	to input a value and bind it to x
a <u>out</u> v. q	to output a value v.

where p and q are of type beh.

At first glance, the binding of a value to x may look as if we have introduced a new form of value binding into ML. However a inp x. p and a out v. q are expanded by the implementation into the expressions *read*(a, \x.p) and *write*(a, v, _.q) respectively, where *read* and *write* have types

```
        read:  * chan # (* -> beh) -> beh
and     write: * chan # * # (. -> beh) -> beh
```

The function *write* requires some explanation. Firstly, () denotes the single element of type .. The reason we expand a out v. q into *write*(a,v,_.q) rather than *write*(a,v,q) is because we wish to inhibit the evaluation of q until the value v has been output. By packaging up q in a trivial function, we can delay its execution until we evaluate (\x.q)(). This technique is a standard way of simulating call-by-name in a language such as ML with a call-by-value evaluation order. Using these expansions, it becomes immediately apparent that no new variable binding mechanism has been introduced to the language.

ML has no equivalent concepts to the parallel composition or summation operators and so we must introduce these. Unfortunately, again we cannot use the CCS syntax as + and | are already used in ML. Therefore we introduce the operators ++ and || . We also add the constant NIL of type beh. Thus, for example,

- *let rec n(x,c) = c out x. n(x+1,c);*

defines a function that takes as parameters an integer, x, and a channel, c, and returns a behaviour that outputs an infinite ascending sequence of integers on channel c starting at the value x. The ML type checker would thus determine the type of n to be

```
> n = \ : int # int chan -> beh
```

Note that the function requires an int chan as an argument to ensure type consistency between behaviours.

Although we have shown how to construct functions that take channels as arguments, returning behaviours as a result, we have not shown how to construct new channels. The expression

<u>chan</u> c <u>in</u> B

binds free occurrences of c in B to a new unique port. In fact "<u>chan</u> c <u>in</u> B" is expanded by the implementation into the expression "*ch*(\c.*B*)" where *ch* is a built-in function with type (* <u>chan</u> -> <u>beh</u>) -> <u>beh</u>. We extend this notation to allow

<u>chan</u> c_1, c_2, . . . , c_n <u>in</u> B

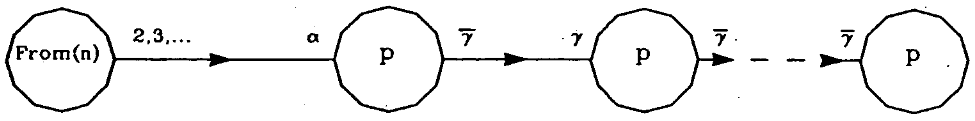
in the obvious way.

We can now construct behaviours but have no way of evaluating them. The function <u>exec</u>: <u>beh</u> -> . performs this function. Given a behaviour, b, exec(b) executes b and only terminates when all the constituent processes have either terminated, or are deadlocked.

This completes our description of PFL. The next section gives a number of PFL examples. We then discuss the problems associated with implementing a system such as PFL.

§2.4 Some PFL examples

Consider the problem of computing prime numbers using the method of Eratosthenes' sieve. Imagine constructing a process that first receives an integer, prints it out, and then passes on any further integers it receives that were not multiples of the original number. By pipelining n of these processes together and using as input a behaviour that generates the sequence 2,3,4,5,6,... we can print out the first n prime numbers.



P = αx. <output x>. P1(x)

$P1(v) = \alpha x'.$
  if $(x' \bmod v) = 0$ then $P1(v)$
  else $\bar{\beta} x'.\ P1(v)$

$PN(n) =$ if $n = 1$ then P
  else $(P[\gamma/\beta] \mid PN(n-1)[\gamma/\alpha])\backslash\gamma$

$FROM(n) = \bar{\alpha} n.\ FROM(n+1)$

To compute the first 10 primes we would use

$PN(10) \mid FROM(2)$

The equivalent PFL program is presented in Figure 2-1.

```
let rec p1(i,o,x) =
    i inp x'.
    if (x' mod x) = 0 then p1(i,o,x)
    else o out x'.p1(i,o,x)

ins p(i,o) =
    i inp x.
    (<output the value of x>; p1(i,o,x));

let rec duplicate(p,n,ic,oc) =
    if n = 0 then fail
    else if n = 1 then p(ic,oc)
    else chan c in (p(ic,c) || duplicate(p,n-1,c,oc));

let firstNprimes(n,ic,oc) = duplicate(p,n,ic,oc);

let rec from(n,oc) = oc out n.from(n+1,oc);

let first10primes =
    chan c, c' in (from(2,c) || firstNprimes(10,c,c'));

exec(first10primes);
```

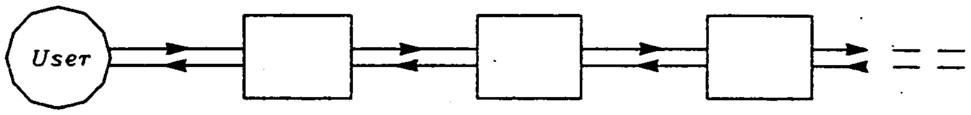
Figure 2-1: A Bounded Prime Number Program

We can modify the program to dynamically create new versions of the process p instead of having a fixed number of them. The new PFL code is illustrated in Figure 2-2.

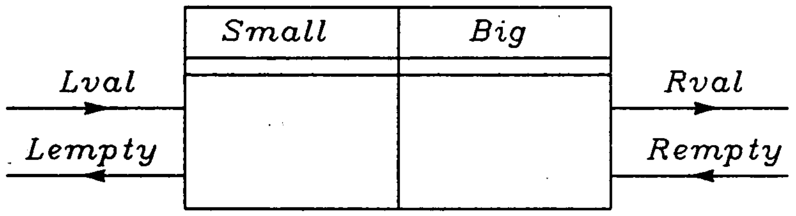
```
let rec p1(i,o,x) =
    i inp x'.
    if (x' mod x) = 0 then p1(i,o,x)
    else o out x'.p1(i,o,x)

ins rec p(i,o) =
    i inp x.
    (<output the value of x>; chan c in (p1(i,c,x) || p(c,o)));

let infiniteprimes =
    chan c, c' in (from(2,c) || p(c,c'));

exec(infiniteprimes);
```

Figure 2-2: An Unbounded Prime Number Program

As an example of a larger PFL program we show how to implement an asynchronous weavesort. The program consists of a pipeline of identical cells



each of the form



The ports Lval and Rval send and receive values between neighbouring processes. The Lempty and Rempty ports allow a process to interrogate the status of a neighbouring process. Each cell has the following behaviour characteristics.

- if the cell is empty then it may offer the Lempty action to its neighbour on the left.

- it may accept a value from the neighbour on the left, and if the cell is already full it must then wait to pass the *big* element to the process on its right.

- the value in *small* must always be kept less than or equal to the value in *big*.

- a non-empty process may transmit the *small* value to the process on its left. It must then wait for a value from the neighbour on the right unless that process is empty.

It is left to the reader to convince himself that this description constitutes a valid sorting algorithm. Further details may be found in [Hennessy 84a], where a proof of the resulting CCS program is also presented. We can implement this algorithm in CCS by the following definitions, where the ? and ! notation is used for readability.

$$AWC_0 = Lval?x.AWC_1(x) + Lempty!\,.AWC_0$$

$$AWC_1(x) = Lval?y.ASWAP(x,y) + Lval!x.AWC_0$$

$$\begin{aligned}AWC_2(x,y) = {} & Lval?z.Rval!y.ASWAP(x,z) + \\ & Lval!x.(Rval?z.ASWAP(y,z) + Rempty?\,.AWC_1(y))\end{aligned}$$

$$ASWAP(x,y) = \text{if } x > y \text{ then } AWC_2(y,x) \text{ else } AWC_2(x,y)$$

$SRW = AWC_0$ oo SRW
where oo is defined by

P oo $Q = (P[SR] \mid Q[SL])\backslash I$
where $SR(Rx) = Ix$, $SL(Lx) = Ix$
and I restricts anything of the form Ix.

The process SRW will sort integer lists of arbitrary length. Figure 2-3 contains the equivalent PFL program (including some output routines). The behaviour screen(inval,outval) prints out values sent to the port outval on the terminal, and passes user input from the terminal to the rest of the program through the channel inval. These examples should convince the reader that CCS algorithms when translated into the PFL framework lead to reasonably intelligible programs.

§2.5 The implementation of PFL

Implementing PFL on a single processor, while straightforward when compared to the distributed case, still presents some interesting problems. For example, consider the PFL expression

A ++ (B || C)

where A, B and C are PFL behaviours. In order to determine the possible initial moves of B||C we must evaluate B and C in parallel. Coping with such expressions is prohibitively expensive, because every time we evaluate a summation such as this we will have to create two new processes B and C, that are then discarded if an action in A was performed. One possibility would be to restrict the language to Static CCS where such examples cannot occur. We take this approach in the rest of the thesis as it aids in the analysis of distributed implementations by removing one extra level of complexity. It is also possible in many cases to compute the initial actions of B||C at compilation time, using the expansion theorem. However, a simpler approach for the centralised case is to prohibit the parallel composition operator at the top level of a summation (i.e. all processes in a summation must be guarded by an action). This case can be detected at compile time and an error generated. Holmstrom introduces two additional versions of the read and write primitives, and an extra behaviour type, cbeh, in order to catch this case as a type-checking error. This approach is taken to avoid performing any major changes to the underlying ML system. At the present time, the Edinburgh PFL implementation performs this check at run-time to avoid the complexity and confusion introduced by the extra types and functions, while again

```
let rec AWC(Lempty,Lval,Rempty,Rval) =
  AWC0(true)
  where rec {
    AWC0(last) =
      (Lempty out (). AWC0(last))
    ++(Lval inp x. AWC1(x,last))
  and
    AWC1(x,last) =
      (Lval inp y. ASWAP(x,y,last))
    ++(Lval out x. AWC0(last))
  and
    AWC2(x,y,last) =
      (Lval inp z. Rval out y. ASWAP(x,z,last))
    ++(Lval out x.   (Rval inp z. ASWAP(y,z,last))
                  ++ (Rempty inp _. AWC1(y,last)))
  and
    ASWAP(x,y,last) =
      if last
      then chan newRempty, newRval in
        (ASWAP(x,y,false) || AWC(Rempty,Rval,newRempty,newRval))
      else if x > y then AWC2(y,x,last) else AWC2(x,y,last) };

let rec {
  master(inval,Lval,Lempty,outval) =
    inval inp t.
    let val = intofstring(t) in
    if val < 0 then outresults(inval,Lval,Lempty,outval)
    else Lval out val.
      master(inval,Lval,Lempty,outval)
and
  outresults(inval,Lval,Lempty,outval) =
    (Lval inp x. outval out (stringofint(x)).
       outresults(inval,Lval,Lempty,outval))
  ++(Lempty inp _.
       master(inval,Lval,Lempty,outval)) };

let prog () =
  chan inval, outval, Lempty, Rempty, Lval, Rval in
  (screen(inval,outval) ||
   master(inval,Lval,Lempty,outval) ||
   AWC(Lempty,Lval,Rempty,Rval));
```

Figure 2-3: A Weavesort Program

minimising the changes to the underlying system. It is hoped that the required check for this case can be included in the compiler shortly.

One of the major problems encountered when implementing PFL arises from the need to create an acceptable impression of fairness in the system. By this we mean that every communication that is theoretically possible in a program must be possible in the implementation of the program as well. A very simple implementation might use a round-robin scheduling strategy for the processes, and the elements of a summation might be tested in a fixed order. Such an approach to the implementation of CCS would lead to an unfair system, since there may be communications between processes that could theoretically occur, but would be prevented from doing so indefinitely. As the user can only perform a finite number of tests on the system, each of a finite duration, the implementor could justify his claim that the implementation was correct. If the system was treated as a closed box, then the user would have no way of proving that some valid sequences of actions were impossible due to an unfair implementation strategy without testing the system for an infinite amount of time. Having said this, it must be appreciated that the theoretical requirements of an implementation, and the user's expectation of its behaviour, are not always in agreement. For a system to be acceptable to a user, it must be seen to be fair, preferably without having to perform prohibitively lengthy tests.

Consider the following PFL example.

<u>let</u> p(x) = x <u>out</u> 0. p(x) ++ x <u>out</u> 1. p(x)   <u>in</u>   p(outchan)

where outchan displays any values sent to it on the terminal. An infinite sequence of zeros would be a perfectly acceptable computation of this behaviour if we only took into consideration the semantics of CCS. We might add some form of fairness constraints to the language, so that in any infinite computation of this example an infinite sequence of zeros <u>and</u> ones must be printed. Unfortunately, a computation that printed zeros for a year followed by alternating ones and zeros would still be an acceptable fair computation, although it would not be very acceptable to a user of the system. The user expects to see zeros <u>and</u> ones appearing on the screen within a short space of time, and this can only be achieved with some form of random guard selection.

We also need to schedule the processes randomly, which is not such an obvious requirement at first sight. Consider the following example. A process requires exclusive access to both a card reader and a line printer in order to accomplish its task. It may request access to both of them in either order, and once allocated to the process they remain in that state until explicitly released. We would like to run two of these processes concurrently, leading to the following PFL program.

```
let rec cdr (sr,er) = sr out (). er inp x. cdr (sr,er) ;

let rec lpt (sw,ew) = sw out (). ew inp x. lpt (sw, ew) ;

let rec p (id, sr, er, sw, ew) =
   ((sw inp 1. sr inp c.
     (<output id to terminal>
      (er out (). ew out (). NIL)))
    ++
    (sr inp c. sw inp 1.
     (<output id to terminal>
      (er out (). ew out (). NIL)))) ;


let sys () = chan sr, er, sw, ew in
   (cdr (sr,er) || lpt (sw,ew) ||
    p(1,sr,er,sw,ew) || p(2,sr,er,sw,ew)) ;
```

While not being a particularly good example of a concurrent program, as it contains an obvious deadlock, it does illustrate why we require random scheduling in order to satisfy the user's expectations. Let us assume that we execute this program on a PFL implementation with round-robin scheduling. The currently executing process has exclusive use of the processor until it wishes to perform a communication. If there is a matching communication request in some waiting process then the appropriate value bindings are performed and the processes placed at the back of the scheduler queue. If no matching request is available, the process is suspended. In either case the next process in the scheduler queue is then executed. Consider how this strategy effects the above example. Process p_1 starts executing and requests exclusive access to the card reader or the line printer. Let us assume that it is granted access to the card reader. The process is then rescheduled and p_2 starts executing. It also requests access to the card reader and the line printer, but in this case only the line printer is still available. At this point the system deadlocks as both p_1 and p_2 are waiting for a resource held by the other process, and neither will release the resource it holds before it has finished its task. While this behaviour of the system is to be expected some of the time, there is another possibility that should also occasionally occur. Process p_1 might obtain access to both the card reader and the line printer before p_2, in which case it may perform its task and then release both resources. The round-robin scheduling strategy tends to inhibit this second possibility. Even if we introduce real time-slicing of the processes, the granularity of the time-slicing is typically much greater than the rate at which interprocess communications are performed, and so the problem still remains. The solution is to adopt random scheduling of the processes. However, we must also ensure that no process that can run is prohibited from doing so indefinitely. For larger examples, the deficiencies of the round-robin approach may not be so apparent, and so there is a case for providing two versions of the PFL system; a random version for demonstration purposes, and a more efficient version for larger programs.

§2.6 Extensions and restrictions

In this section we discuss extensions to PFL that would improve the language. We also propose restrictions that may be necessary in order to implement PFL on a distributed system. Our preliminary experiences with PFL lead us to believe that it could form a very powerful and useful extension to ML, as well as a practical teaching tool for concurrent programming. For these reasons we believe that the language should be developed further, and the following points investigated.

Firstly, PFL can be regarded as a superset of CCS. To see why, we note that channels and behaviours can be passed between processes as values in PFL, which is not allowed in CCS (strictly speaking, CCS allows them to be passed as values but not used). This possibility greatly increases the difficulty of reasoning about the resulting programs, which is why CCS excluded it. Restricting PFL so that structures containing channels and behaviours are not valid arguments to an output communication would result in a less elegant language due to the resulting loss of orthogonality. The view we take here is to propose the definition of a number of subsets of CCS for specific uses. These subsets could be optionally checked for in the compiler. For example, a program that is to be verified may be written in a subset that permits no channel or behaviour value-passing so as to aid the eventual proof, and the compiler can check that this was indeed the case. It might also be desirable to inhibit the passing of updatable objects between processes. Without such a restriction, we allow processes to bypass the normal CCS communication primitives by using shared variables and thus introducing all the problems of shared variable access that CCS was designed to avoid.

Another subset that is useful (and checkable) results from the observation that in a distributed implementation of PFL without any shared memory, the passing of large data structures between processes may be difficult. There is a case for only allowing primitive data types such as integers, reals, strings etc., to be passed between processes. This would allow the underlying process synchronisation mechanism to be kept as simple as possible, which is especially important when this mechanism may be implemented in hardware. The restriction may be circumvented, to some extent, if we allow channels to be passed as values between behaviours. For example, consider the problem of passing a tree between two processes. When the tree is viewed as a static (passive) data structure, this may cause problems as the components of the tree have to be passed between the processes as separate messages, the tree reassembled at the other end, and only then can the destination process continue its execution. The underlying message-passing mechanism may therefore be quite complex. However, we can assemble a tree of processes that models the original data structure, and access the elements by sending the structure requests, rather than manipulating the structure directly as in the original case. To pass this active data structure to another process, we only have to pass the channels that act as communication links to the structure. Furthermore, viewing the data structure as a collection of processes allows us to define parallel replication and maintenance functions for the structure. Of course, on a real distributed implementation of PFL this raises the question of how to manage very large numbers of processes arising from replacing some of the data objects by processes. In particular, on a distributed machine, how do we ensure that the data structures required by a process are not spread throughout the entire processor network? There have been preliminary attempts at architectures that allow processes to spread smoothly through the processor network that may be of relevance to this problem [Hewitt 80]. However, solving such problems is very difficult and is not investigated further in the thesis. This view of data as active structures in terms of processes is similar to the actor model of Hewitt [Hewitt 77] and also the object-based programming languages such as SmallTalk [Goldberg 83].

The current PFL implementations allow the user to interact with the system via an input and output channel to the terminal. Once the <u>exec</u> function has been applied to a behaviour, the input and output channels provide the only means of interaction with the system until the behaviour has terminated or deadlocked. This style of interaction has a number of deficiencies. Firstly, the need to explicitly apply the function <u>exec</u> to the behaviour in order to evaluate it appears inelegant; it would be better if this function could be implicitly applied in some way. A more serious criticism of the system becomes apparent when the program is to be tested. In order to test a behaviour, a testing process must be constructed that takes commands from the terminal and converts them into the required communication requests. The construction of such testing processes may become very tedious.

An alternative implementation strategy is to enter an interactive question and answer mode when the exec function is applied. The system would indicate the possible actions at each point, and the user would select the required choice. Such an interface is similar to that provided in proof checkers for CCS. By allowing the system to proceed automatically for a controlled number of steps, or until specified actions are possible, complicated systems may be debugged more easily than with the current implementation.

If we imagine the system under test to be composed of a tree of processes (where the tree structure is derived from the restrictions and renamings) then we can view the first approach as placing the user within, or interacting with, a special process with a limited set of communication possibilities with the rest of the system (namely just an input and output channel of strings). The second approach can be viewed as placing the user outside the system, viewing what is going on from a distance, and controlling it at the metalevel. A third approach again places the user within the system. However, in this case we allow the user complete freedom to perform any communications he desires. In practice this would mean that the evaluation of a behaviour would return control back to the user immediately, running the behaviour asynchronously in the background. The user would then be able to type simple actions, summations etc., that would communicate with the behaviour. This type of interaction emphasises the need to implicitly perform execs when required so as to provide a natural and convenient interface to the background tasks. Such an approach is similar to the use of the & operator in the Unix operating system. Using such an interface has many advantages. Behaviours may be evaluated asynchronously while the user continues with some other task. Furthermore, at any point in time the user may communicate with the behaviour through any ports that are accessible to both the user and the background behaviour. We hope to develop such an interface for the Edinburgh PFL system in the near future.

There are a number of concurrent calculi that have been influenced by CCS. The synchronous version of CCS, SCCS [Milner 83], a similar calculus developed by Austry and Boudol called MEIJE [Austry 84], and a calculus developed by Milne for hardware description and verification called CIRCAL [Milne 85], are good examples of such languages. This raises the question of whether to incorporate any of the novel features of these languages into PFL, in order to extend its power and applicability as a concurrent programming language. One extension to the language that could be considered is the ability to perform more than one communication <u>simultaneously</u>. This allows the notion of clocked systems to be conveniently specified for example. The extensions to the syntax necessary to accommodate this feature would be simple. However, even a centralised implementation of this feature would be quite a complex task.

One extension that is explicitly present in CIRCAL, and can be treated as a derived operator in SCCS, is the ability to broadcast a message on a channel to all processes with access to the channel. In fact, this is the only form of communication present in CIRCAL. This extension would be simple to add to PFL in the centralised case. The difficulty of implementation in the distributed case would depend on the underlying computer architecture. An Ethernet based implementation might support broadcasting very efficiently, whereas a message-passing network between the processes might make broadcasting impractical.

The asynchronous nature of the current PFL implementation makes debugging of behaviours difficult. One possibility would be to create a pseudo-random version of PFL that takes a seed as a parameter to <u>exec</u>. Such an approach would be useful when testing systems, although genuinely asynchronous behaviours, such as interrupt handlers, would still cause problems.

§2.7 Conclusions

In this chapter we have presented a version of CCS called PFL that transforms the calculus into a usable programming language. We have given some illustrative examples of its use, and discussed a few of the implementation problems associated with the language. Finally, we have pointed out some areas in which the system is still deficient and some areas for future work and development.

We believe the embedding of CCS in a functional language is the most desirable way of constructing a usable CCS programming environment. There are obviously difficulties associated with such an approach, such as syntax constraints and the possible need to constrain the types of values used in value passing. However, experience with the system has lead us to conclude that there are no obvious features of a language such as ML that are a hindrance to CCS. Furthermore, although we could construct a system with restriction and relabelling operations, we believe that the features of a general purpose functional language would still be required leading to the redundancy described at the start of this chapter. Therefore, we believe that even if an implementation of CCS was carried out directly, the resulting system would be very similar to PFL.

The next step is to consider the implementation of CCS in a distributed environment, a much more difficult task.

CHAPTER 3
Implementing Static CCS on a Distributed System

§3.1 Introduction

The previous chapter described how we might implement a concurrent language such as CCS on a single processor. There are no problems in synchronising such a system, because all the information pertaining to each of the processes is readily available, and can be used by a centralised scheduler. However, it is difficult to impose a degree of randomness on the system without an associated loss in performance. Without some form of randomisation, the user's expectations of the likely behaviour of a system will differ from the actual behaviour, even though the implementation may be technically correct. We now wish to examine the case where CCS is implemented on a distributed network of processors. Such implementations require the development of protocols, or interaction strategies, for synchronising CCS agents efficiently in a distributed environment. One possibility would be to implement a centralised scheduler, as for the single processor case. However, such a scheduler is undesirable because the overhead in keeping a centralised record of the state of each of the processes may be significant. This contrasts with the single processor approach where the scheduler can ascertain the global state merely by examining shared memory locations. Furthermore, the centralised scheduler creates a bottleneck on the performance of the system, as all synchronisations in the system are managed by a single process. For these reasons, we do not consider the centralised approach any further. The problems encountered when implementing CCS on a distributed system, with a distributed scheduler, are opposite to those of a single processor system. The fluctuations in the relative speeds of processes on different processors creates a degree of randomisation without any additional overhead. However, the task of synchronising communications is not trivial in the distributed case.

The goal of this chapter is to present a number of different protocols, or interaction strategies, for synchronising processes in a distributed framework. We start by examining the problems encountered when synchronising Static CCS processes on a distributed system. We show that for certain programs, those where a *synchronising annotation* can be constructed, there is a simple algorithm for synchronising the processes. The algorithm requires a minimum of unidirectional control messages to be exchanged between processes for the establishment of each bidirectional handshake. We then show how various restrictions that have been proposed for CSP can be viewed as methods for guaranteeing the existence of synchronising annotations. One of these schemes has been used as the basis for a derivative of CSP, called Occam [INMOS 84a], designed to run on a special purpose processor called a Transputer [INMOS 84b]. We argue that by using synchronising annotations explicitly, rather than just one particular scheme for constructing them, we would obtain a more flexible language.

The second part of the chapter deals with those situations where we wish to implement a program that does not possess a synchronising annotation. Such situations are common, especially as we may not wish to force an unnatural structure on a program purely to aid its efficient execution. One approach in these situations would be to use a more complicated synchronisation scheme, that placed fewer or no constraints on the program. Section 3.6 briefly reviews the work in this area. Some of these schemes rely on an underlying synchronous message-passing mechanism, or can be modified to do so. Such algorithms can often be expressed within Static CCS, and in these cases we can view the schemes as program transformations, rather than an implementation of Static CCS on top of some lower-level protocol. By using such transformations, it is possible to replace a program that uses the full power of Static CCS by an equivalent one that only requires a subset of the language. Implementation schemes may then be developed for these subsets that are more efficient than implementations of the full language. We illustrate this technique by describing an algorithm due to Schwarz [Schwarz 78] in the form of a transformation. We show that by using this scheme it is possible to construct a synchronising annotation for the transformed version of any Static CCS program. We also discuss why a transformational approach may be preferable to using the more traditional implementation techniques. Finally, we illustrate the difficulties involved in reasoning about the correctness of such transformations. For example, we must show that our transformations do not change the overall visible behaviour of the system. This last section illustrates the inadequacies of the existing equivalences for CCS when applied to these problems and leads us into the theoretical investigations of Chapter 4.

§3.2 Synchronising processes in a distributed environment

A discussion of the problems of implementing CCS in a distributed environment can only take place once the exact nature of the environment has been specified. Furthermore, some method of describing the possible process synchronisations in a program, in a form amenable to analysing the complexity of the communication requests, must also be developed.

A collection of processes communicating via shared memory may be referred to as a distributed system. The term may also be used to describe a collection of machines spread across a continent, communicating via a satellite network. The diversity of systems covered by the term is sufficiently vast that any general discussion of the problems involved in the distributed implementation of CCS would be of little practical use. We therefore restrict our attention to a particular class of distributed systems, namely those where processes communicate via asynchronous, unidirectional, point-to-point messages. Such systems are important for a number of reasons. They are relatively easy to implement, and therefore form one of the more common classes of distributed system. The message-passing model may also allow us to view systems employing a variety of inter-processor communications methods in a uniform framework. For example, processors may be connected to their nearest neighbours using shared memory. A broadcasting method, such as an Ethernet, may be used for medium distance communications, while a satellite link may be used for long distance traffic. The type of interaction possible between any group of processors may therefore depend on the communications link(s) between them. The use of a message-passing protocol for inter-process communications may allow us to reason about the system uniformly, at the expense of not using the full communications capabilities of some of the interconnections.

Informally, we may view a CCS agent as evolving by packaging up the actions it may potentially perform in its current state into a request that is then passed to an underlying subsystem whose task it is to find matching requests. The agent is then suspended until the subsystem finds a request containing a complementary label, at which point it performs the corresponding action, possibly involving an exchange of values with the matching agent. It then evolves to the continuation agent associated with this action, and the cycle repeats. A non-deterministic choice may be required in this last step, as in the agent $p = \alpha.p_1 + \alpha.p_2$. The mapping of CCS onto a distributed system must therefore describe how agents are mapped onto the processes provided by the underlying system, the form of a communication request, and the mechanism by which matching requests are found.

There are undoubtedly many ways of implementing CCS on a distributed system of the form described above. CCS agents communicate with each other by exchanging messages through complementary labels, and so one possibility would be to assign a process to each CCS agent, and also to each pair of complementary labels in the program. We refer to these label processes as <u>ports</u>. Agents may then express their desire to communicate using any one of a set of labels by sending messages to the corresponding port processes. Using such an approach, an agent need not be aware of the identities of the processes that may synchronise with it by issuing a complementary request. This property may be especially important when processes are created dynamically, and so the number of communicating partners cannot be determined in advance. If CCS possessed no choice operator, then a very simple synchronisation protocol would suffice. In the general case, however, the request sent to a port must contain the identities of the other ports the agent is also willing to communicate with. The ports may then communicate amongst themselves in order to establish a synchronisation.

If we restrict our attention to Static CCS, then the synchronisation task becomes simpler. In this case, the possible recipients for each action can be statically enumerated, or at least an upper bound established, and so the label mechanism can be viewed as a convenient notation for naming an explicit set of processes. In such cases, we may be able to map each agent into one or more processes communicating directly with the processes representing the other agents. The implementation of Static CCS in such a framework raises many interesting issues, and so the rest of the thesis will limit itself to this case. However, while the specific protocols developed for synchronising agents in such a framework may not be directly applicable to the general case of CCS, the techniques developed in the thesis for analysing these protocols may also be of use in the analysis of implementation strategies for the full language.

If we wish to map a Static CCS program onto a system where the processes representing agents communicate directly, rather than through ports, then a request will have the form of a set of explicitly named processes, as well as the corresponding labels. In Static CCS, each program is syntactically of the form $(\prod_{i \in N} p_i)\backslash L$, and so each constituent agent, p_i, may be assigned a unique number i, its process index, corresponding to its syntactic position in the product representing the program. We can therefore represent a request by a set of pairs of the form <process index, label>, called a request set. To construct a request set, we must replace a set of labels, formed by the guards in a summation, with the process indices of all the processes that may potentially offer a complementary action.

The use of request sets is similar to the original development of CSP, where all communication requests had to explicitly name the destination process. However, in this case, the translation from labels to process indices is quite subtle. In general, it will depend on the current state of all the agents in the system. It is technically sufficient to map a label to the set of all process indices, as requesting to communicate with an agent, using a label whose complement is not contained in the sort of that agent, cannot result in any unwanted synchronisations. However, it is reasonable to assume that the complexity of synchronising a system is in some way related to the number of simultaneous requests issued by an agent, and so increasing the number of requests unnecessarily may lead to implementation difficulties.

At any point in a computation there will be a minimum request set. Consider a state P = $\prod_{i\in N} \mathbf{p}_i$, where the implementation of process $\mathbf{p}_i$ has translated a set of labels L to a set of process indices PI. This translation is <u>safe</u> if the identities of all potential communicating partners for the summation corresponding to the actions in L are contained in the set PI. More formally, the translation is safe if for all derivatives of P that require no participation from $\mathbf{p}_i$ in their derivation from P, if the jth component, $\mathbf{p}_j'$ $(i \neq j)$, in the resulting derivative can synchronise with an action in L, then $j \in$PI.

For a process to produce an optimal safe request set from a given set of labels, the global state of the system must normally be accessible. This is because the ability of an agent to respond to an action will typically depend on its current state, and hence to determine an optimal safe request set, the current state of the other processes must be known. Even if the global state is accessible by each process, the analysis may still be computationally infeasible in most cases. In practise, the situation is further complicated because only local state information will be accessible to a distributed process. A careful analysis of the state information that is available may reveal some indications as to the current state of the other processes, and hence potentially reduce the size of the request set. For example, in a network of processes communicating by message-passing, the local state information might consist of the values of the local state variables and the history of previous communications performed by this process (and possibly the identities of the recipients). Using this information, the possible states of the other processes may be deduced, or at least partially constrained.

A simple mapping between labels and process indices may be constructed using the sorts of the processes in a program. Its simplicity makes it a suitable candidate for practical implementations. For a given program $P = \prod_{i\in N} p_i$, where each process p_i has sort S_i, we define the function

$$RS(L) = \{\ j \mid \lambda\in L \wedge \bar{\lambda}\in S_j\ \}$$

where $L\subseteq Act$.

For any derivative of P, say $P' = \prod_{i\in N} p_i'$, the request set offered by p_i' may be generated by $RS(Init(p_i'))$. It is simple to show that such request sets are safe.

Other methods for mapping between labels and process indices may require additional arguments to the mapping function, containing local state information. A discussion based on the general notion of a request set mapping is complicated because of this flexibility in the definition of the mapping. We will therefore assume that RS is used as the mapping function throughout the rest of this thesis. Generalising to other, more elaborate, functions is not difficult, but notationally cumbersome.

Given a program P, and a request set mapping, RS, we can associate with each derivative of P a labelled directed graph, called a <u>request graph</u>, that describes the currently outstanding communication requests in the network. The complexity of synchronising the communication requests of a program is reflected in the complexity of the corresponding request graphs.

Definition

For a given program P, request set mapping RS, and derivative of P, $P' = \prod_{i\in\mathcal{N}} p_i'$ the labelled directed graph $G_R=(\mathcal{N},\mathcal{E}_R)$ is the <u>request graph</u> of P' iff the set $\mathcal{N}$ corresponds to the process indices of P, and $\langle i,\lambda,j\rangle$ is an element of $\mathcal{E}_R$ if $\lambda\in \mathrm{Init}(p_i')$ and $j\in RS(\{\lambda\})$, where $i\neq j$.

One approach to analysing the complexity of synchronising a program, P, would be to examine the request graphs generated by all the derivatives of P. A process will pass a request to the synchronising subsystem asynchronously from the rest of the processes, and will receive its replies asynchronously as well. Checking the request graphs corresponding to each derivative of the program assumes that all of the requests are synchronised. For example, if processes p_i and p_j synchronise, then the only request graphs that are examined after this transaction assume that both p_i and p_j have issued their next requests immediately after the synchronisation. We do not examine a request graph where p_i has issued its request, but p_j is still computing its request set. Such a distinction does not affect the analysis if delaying a request does not alter the value of the request set. This is obviously true for the function RS, but would not be true if a process had access to fragments of communications histories involving other processes, such as could be obtained by eavesdroping on an Ethernet. In such cases, delaying a request may mean that the eventual request set is smaller. Indeed, we could imagine constructing protocols that deliberately waited until other communications had taken place, in order to minimise the size of some request sets. The analysis of such an approach, and its consequences, are outside the scope of this thesis.

Corresponding to each request graph there is a <u>synchronisation graph</u>, where $\langle i,j\rangle$ is an edge in the graph if it is possible for p_i' and p_j' to synchronise.

Definition

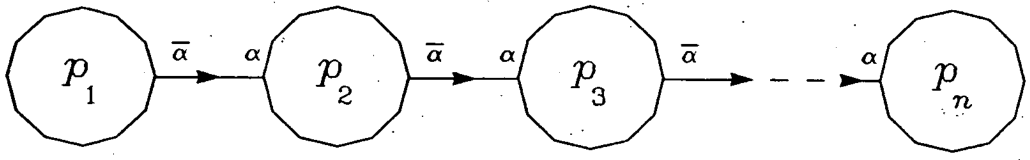
The <u>synchronisation graph</u> corresponding to the request graph $G_R=(\mathcal{N},\mathcal{E}_R)$, for a derivative $P' = \prod_{i\in\mathcal{N}} p_i'$, is defined to be the undirected graph $G_S=(\mathcal{N},\mathcal{E}_S)$, where <i,j> is an element of $\mathcal{E}_S$ if there exists a label λ such that $\langle i,\lambda,j\rangle$ and $\langle j,\bar{\lambda},i\rangle$ are both elements of $\mathcal{E}_R$.

The synchronisation graph summarises the possible communications that can take place in the current state. The complexity of these communications is reflected in the complexity of the graph.

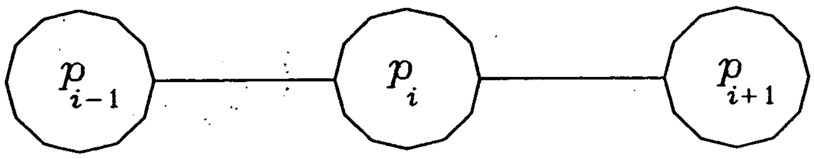
§3.3 Synchronisation graphs for some simple examples

This section informally examines the request graphs and synchronisation graphs for some simple examples, in order to characterise which programs are simple to synchronise. A method for synchronising programs with simple synchronisation graphs will then be presented in the next section. The problem of synchronising more complex Static CCS programs will be treated later in the chapter.

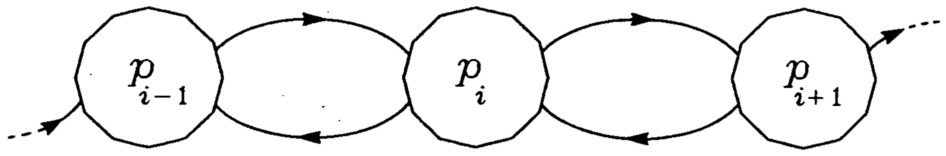
Consider a network of processes p_1, p_2, . . . , p_n statically connected as follows.



The behaviours represented by p_i, . . . ,p_n are unimportant, except that we assume that p_1 can always output a value and p_n can always input a value. Furthermore, any process p_i, $1<i<n$, can either input a value from p_{i-1} or output a value to p_{i+1}, but cannot offer both possibilities simultaneously. All synchronisation graphs resulting from such a system have the property that the maximum path length in the graph is one. To see that this is the case, suppose there existed a graph such that for some process p_i ,

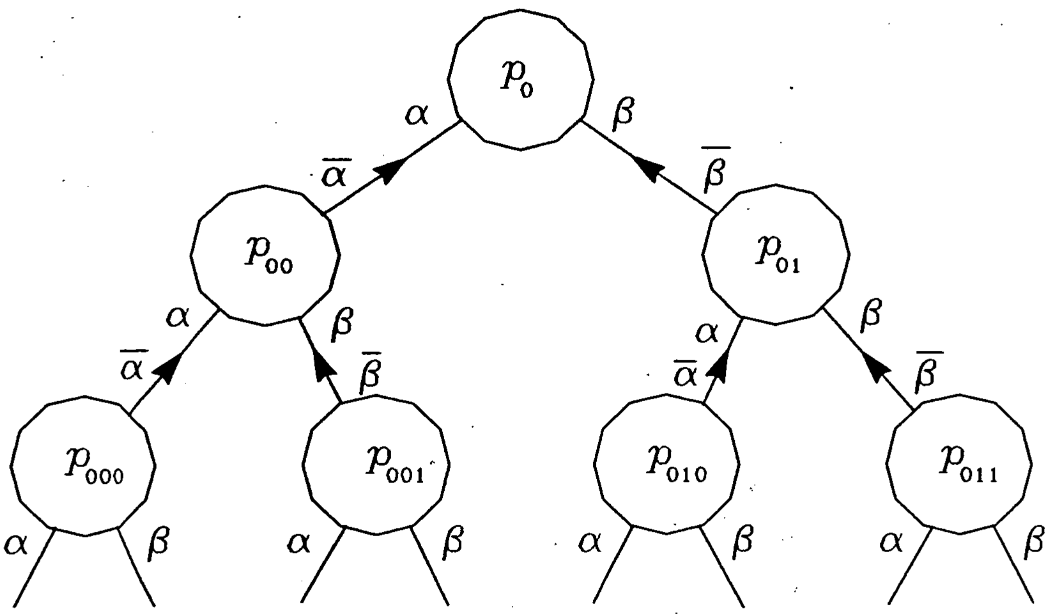


Then the corresponding request graph must be of the form

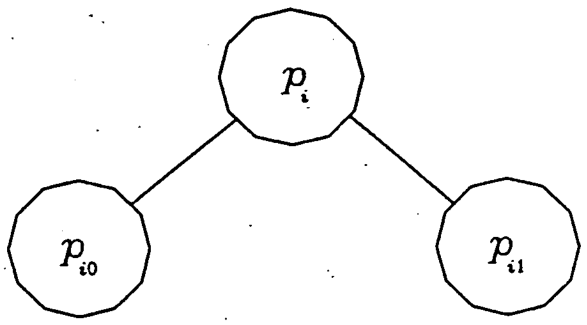


which is impossible from the definition of the behaviours. Such systems are simple to synchronise because whenever a process $\mathbf{p}_i$ can potentially communicate with another process $\mathbf{p}_j$ then this is the <u>only</u> process it can communicate with. Suppose we impose some arbitrary ordering on the processes. For example we might chose $\mathbf{p}_i<\mathbf{p}_j$ if i<j. Then to synchronise $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ in our simple example it is sufficient to always make $\mathbf{p}_i$ wait for a request from $\mathbf{p}_{i+1}$, or in general to make the smaller of the two processes, with respect to the ordering, wait for the larger.

Let us now consider a slightly more complicated example. We construct a tree of processes that can be viewed statically as shown below.

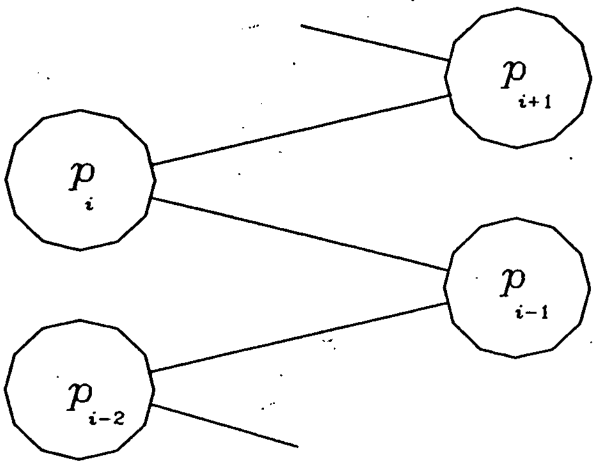


Each process can input values from either of its sons (if any) or transmit a value to its father (except p_0). However it cannot attempt both simultaneously. If no path exists between two nodes, p_i and p_j, in a synchronisation graph, G, then at that point in the computation the synchronisation of p_i is independent of the synchronisation of p_j. Therefore the network of processes containing p_i may be synchronised separately from the network containing p_j. Of course, p_j may communicate with some other process, and then evolve to a state where it can communicate with p_i. This case will manifest itself as a more complicated network in some other synchronisation graph corresponding to a different derivative of the program. We may therefore analyse the complexity of the synchronisation graphs by analysing each connected component of each graph separately. The most complicated connected component that can occur in a synchronisation graph resulting from our example is of the form



We can exploit this property of the synchronisation graphs by allowing one process in any potential synchronisation to wait for the other process. However, whereas in our first example the process that waited could be chosen arbitrarily, in this case we must force the process that contains the sum (p_i) to wait for either of the other processes (p_{i0},p_{i1}) to send a request.

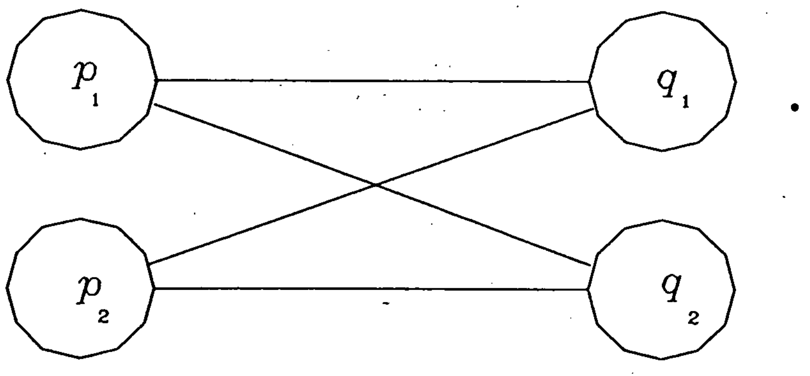
The synchronisation of processes was simplified in these two examples due to the restricted forms of the possible synchronisation graphs. In both cases, one partner in every potential communication had no other partners. We could therefore choose a rule whereby this process would send a message to its partner and the partner would wait for the first matching request. Unfortunately, it is not always natural to express an algorithm in such a way that the resulting synchronisation graphs always have this property. As an example of such a case, consider our first example where each process p_i is now allowed to simultaneously attempt to communicate with p_{i-1} and p_{i+1}. The resulting synchronisation graphs may now have connected components with path lengths greater than two, as is illustrated in the following example.



If a network of processes can produce such graphs, then the approach we used for the first two examples is obviously not applicable, as there is no process that can safely wait for a request without the possibility of deadlock being introduced. Any scheme for synchronising such a network must take care to avoid the introduction of deadlock or livelock. As an example of these possibilities, consider the four processes

$$\begin{array}{ll} p_1 = \alpha.p_1 + \beta.p_1 & q_1 = \overline{\alpha}.q_1 + \overline{\gamma}.q_1 \\ p_2 = \gamma.p_2 + \delta.p_2 & q_2 = \overline{\beta}.q_2 + \overline{\delta}.q_2 \end{array}$$

Synchronising such a network is difficult because one of the resulting synchronisation graphs may be of the form



If we try to synchronise this system in a distributed environment, the following two scenarios might take place.

1. Lazy or timid behaviour.

   In this scenario a process will attempt to communicate with each of its partners in turn and will reject all requests from other processes until it has received a reply. We call the process lazy or timid because it only attempts one possibility at a time. This scenario may lead to a livelock. Suppose p_1 sends a message to q_1, q_1 sends a message to p_2, p_2 to q_2 and q_2 to p_1. When the target processes receive their requests they will reject them because each process will have an outstanding query. These rejections will eventually arrive back at the source processes and each process will then try one of the other possibilities. However,the new set of requests may also be rejected for the same reason and this sequence of events may continue indefinitely. If the relative speeds of the processes can fluctuate then we might use probabilistic arguments to show that a successful communication will eventually occur. However,there will be no upper bound on the number of messages that may have to be exchanged before a successful synchronisation is achieved.

2. Eager behaviour.

   We might try the opposite approach where a process hoards requests while waiting for a reply from its own query in the hope that, if its query is rejected, then it can positively acknowledge one of the waiting requests. This scenario may introduce a deadlock as the following sequence of messages illustrates. Suppose p_1 sends a request to q_1. While waiting for an acknowledgement it receives a request from q_2 which it queues. p_2 now sends a request to q_2 which q_2 queues as it is waiting for a reply from p_1. q_1 sends a message to p_2 and this message is also queued. This completes our deadlock as

   p_1 is waiting for q_1

   q_1 is waiting for p_2

p_2 is waiting for q_2
q_2 is waiting for p_1

§3.4 Synchronising annotations

We have seen that for some Static CCS programs a very simple synchronisation strategy will suffice, whereas in the general case, a much more sophisticated protocol is necessary. We now consider the simple case in more detail. The second part of this chapter will then deal with how to synchronise arbitrary Static CCS programs.

A simple synchronisation mechanism was possible for the first two examples because in both cases there was an asymmetry that could be imposed on the system. Each request issued by a process could either be flagged as passive, in which case the process was suspended until a matching request was received, or else active, in which case the process transmitted its request to the only process that could synchronise with this request. We call the process that waits the *slave*, and the process that issues the request the *master*. If we wish to use such a protocol, then, in addition to translating each summation into a set of process indices, we must also indicate whether the process is to perform an active (*master*) or passive (*slave*) role in any potential communication.

If all requests are annotated with either a *slave* or *master* flag, then a request graph $G_R=(\mathcal{N},\mathcal{E}_R)$ may also be annotated by constructing the corresponding function **MS** from the set $\mathcal{N}$ of process indices to the set $\{master, slave\}$ of annotations. The resulting annotated graph is represented by the pair $<G_R,MS>$. Synchronisation graphs may also be annotated in a similar way.

Definition

The *master/slave* function used to produce these annotations is safe if for every annotated synchronisation graph $<(\mathcal{N},\mathcal{E}_S),MS>$ corresponding to a derivative of P;

1. $<i,j>\in\mathcal{E}_S \supset MS(i) \neq MS(j)$

2. $<i,j>\in\mathcal{E}_S \wedge <i,k>\in\mathcal{E}_S,\ j\neq k \supset \mathrm{MS}(i) = \mathit{slave}$

The intuition behind the definition is that if two processes can potentially communicate, then one will be the *master* and the other the *slave*, and if a process wishes to communicate with more than one partner, then it must play a passive role in this communication.

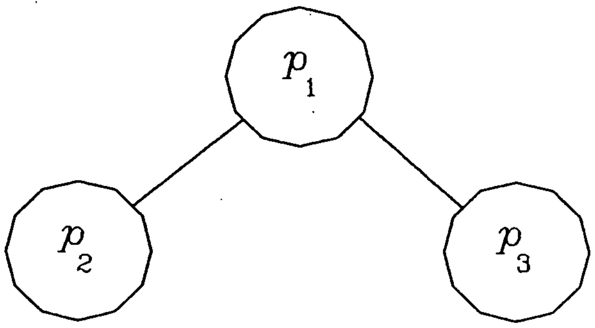
Consider the program

$(\prod_{i\in\{1,2,3\}} p_i)\backslash\alpha,$

where

$p_1 = \alpha.p_1,\quad p_2 = \bar{\alpha}.p_2,\quad p_3 = \bar{\alpha}.p_3$

There is only one synchronisation graph for this program, namely



A safe annotation function for this example would be

MS(1) = *slave*

MS(2) = MS(3) = *master*

A program will have an efficient implementation if it is possible to construct a safe annotation function MS for the program. In such cases we say that the function MS safely annotates the program. If this function is static, i.e. it remains constant as the program evolves, then this is equivalent to annotating each summation in the program with either the *master* or *slave* flag.

There will be many cases where it is impossible to construct a safe *master/slave* annotation function for a given Static CCS program. This will obviously be the case when the program contains patterns of communications that cannot be synchronised using the *master/slave* approach. However, even when all communications are amenable to this approach, the restricted local state information available to each process may make it impossible to construct a *master/slave* function that only bases its decision on this restricted knowledge.

If it is not possible to construct a function that safely annotates a program with the limited state information available, then it may still be useful to know which communications can use this approach. The simple communications could still be synchronised using this method, while a more sophisticated scheme could be used to synchronise the remaining communications. Chapter 5 takes this approach further by transforming those communications that are difficult to synchronise into 'equivalent' sequences of communications that can use the *master/slave* method.

To allow the partial annotation of a program, we extend the possible values of an annotation to include the *unknown* flag. An annotation function is then safe if the communications that use the *master/slave* approach are disjoint from the communications where the requests are flagged with *unknown*, and the *master/slave* communications are safe in the sense defined earlier.

Definition

An annotation function is safe if for every annotated graph $<(\mathcal{N},\mathcal{E}_s),MS>$ corresponding to a derivative of P,

1. $<i,j>\in\mathcal{E}_s \wedge MS(i) = \textit{unknown} \supset MS(j) = \textit{unknown}$
2. $<i,j>\in\mathcal{E}_s \supset MS(i) \neq MS(j) \vee MS(i) = \textit{unknown}$
3. $<i,j>\in\mathcal{E}_s \wedge <i,k>\in\mathcal{E}_s,\ j\neq k \supset MS(i) \neq \textit{master}$

Every program has a trivial safe annotation function that flags each request with the *unknown* value. If there exists a safe annotation function, MS, for a program P that produces no *unknown* flags, then we say that MS is a synchronising annotation for P. Similarly, we may say that P possesses a synchronising annotation under specified constraints on the local state information.

§3.5 Generating synchronising annotations

One simple technique that can be used to automatically generate synchronising annotations is to place some restrictions on the source language. For instance, Hoare's original proposal for CSP [Hoare 78] did not allow output guards within summations. If we adopt this restriction, we could then annotate every input summation as a slave and every output action as a master and this would produce a simple synchronising annotation. The same restriction was used in the Occam language [INMOS 84a], and in both cases a very efficient synchronisation scheme is possible, but at the expense of imposing an asymmetry on the source language. This approach has the advantage that each process can be annotated independently of the rest of the system, whereas in the more general case the context influences the annotation of a process.

There are many cases where a problem cannot be naturally expressed in the asymmetric subsets of CSP or Static CCS. We would like to be able to automatically annotate all Static CCS programs, while keeping the number of *unknown* requests to a minimum. The function RS may be used as the basis of a simple annotation function. It is certainly not optimal, as it avoids examining the state space of a program by assuming that all combinations of process states are possible. The advantage of this approach is that it is computationally efficient, whereas a more detailed analysis would probably be too expensive for a practical implementation. The main disadvantage is that we may have to synchronise some of the communications using a more complicated protocol than is theoretically necessary.

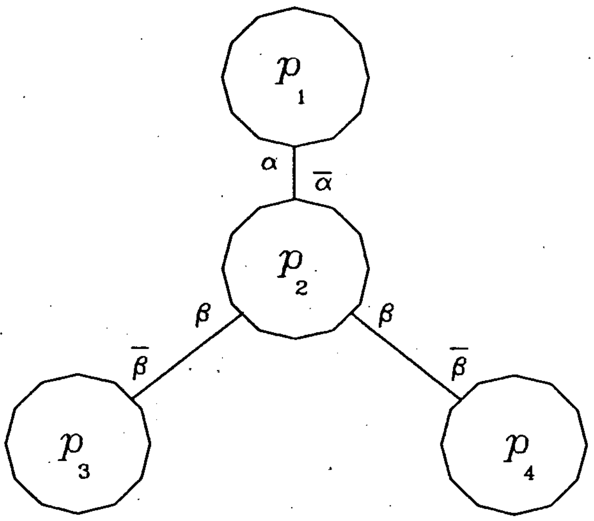
An algorithm for automatically annotating Static CCS programs is presented below. It avoids examining the state space of the system by using function RS to map between labels and process indices. For this reason it is far from optimal, although a more detailed analysis of the program is probably not feasible if the algorithm is to form part of a compiler. We show that under this simplifying assumption, the problem of annotating the program is equivalent to constructing an acyclic dominance relation between the processes with certain properties. We present an algorithm for constructing this dominance relation based on computing the connected components of graphs.

The first step in the algorithm is to construct for each process p_i, in a program $P = \prod_{i \in N} p_i$, a set PC_i containing all request sets that may be issued by this process. We define PC_i by

$$PC_i = \{ \text{RS}(\text{Init}(p_i')) \mid p_i' \in \textit{derivatives}(p_i) \}$$

Note that some of the derivatives of p_i may not be reachable when p_i is placed in the program P. A more careful analysis might detect this, and the definition of PC_i could be modified accordingly.

As an example of the construction of the PC_i sets, consider the following network of processes.



where $p_1 = \alpha.p_1$ $p_2 = \beta.\overline{\alpha}.p_2$ $p_3 = \overline{\beta}.p_3$ and $p_4 = \overline{\beta}.p_4$

Then $PC_1 = PC_3 = PC_4 = \{\{2\}\}$ and $PC_2 = \{\{1\}, \{3,4\}\}$

In order to construct an annotation, we must assign to each element of every set PC_i either the *master*, *slave* or *unknown* flag. We adopt the convention that the annotation assigned to an element s of PC_i refers to the role that process *i* will assume when communicating with the processes in the summation represented by s. Because of our underlying assumption about the accessibility of process states, annotating the sets can be shown to be equivalent to imposing an acyclic dominance relation $<$ on the processes. In any communication between p_i and p_j, if $p_i<p_j$ then p_i is the *slave* and p_j the *master*, and if they are incomparable, then this is equivalent to annotating both terms with the *unknown* flag. For example, consider the annotated set PC_i. Let us suppose that $\{j, \ldots\}$ is an element of the set and has been annotated with the *slave* flag. Then it is simple to show that every other possible communication between p_i and p_j must also be annotated so that p_i is the slave. Suppose that this were not the case, i.e. there was an element of PC_j of the form $\{i, \ldots\}$ that was annotated as a slave. Our simplifying assumption would then imply that process p_i and process p_j may reach a state where they wish to communicate with each other and both of them are slaves in the communication, which is not possible if the sets have been annotated correctly. Similarly, if an element of PC_i mentions j and is flagged as a master then in all communications involving p_i and p_j, p_i will always be the master. Finally, if a communication of p_i involving p_j is flagged as *unknown* then all communications of p_i involving p_j will be flagged as *unknown*.

The previous analysis implies that the task of annotating the elements of the sets PC_i is equivalent to constructing an acyclic dominance relation $<$ between processes such that

i) if $s \in PC_i$ and $j \in s$ then $p_i < p_j$ or $p_j < p_i$
or they are incomparable (written $p_i \# p_j$)

ii) if $s \in PC_i \wedge |s|>1$
then either $\forall j \in s \; p_i < p_j$
or $\forall j \in s \; p_i \# p_j$ •

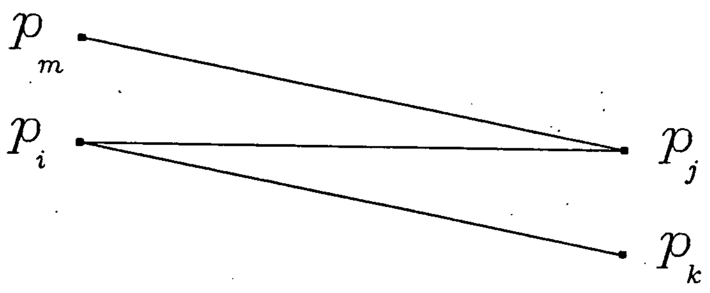
Note that the relation will not be a partial order in general as $p_i<p_j$ and $p_j<p_k$ does not necessarily imply $p_i<p_k$, i.e. $<$ is antisymmetric but not transitive. It is simple to show that such an ordering generates safe annotations.

Silberschatz [Silberschatz 79] proposed a scheme whereby a dominance relation was provided by the user along with his program. If two processes p_i and p_j could communicate then either $p_i<p_j$ or $p_j<p_i$. Furthermore, the relation was constrained so that if p_i has a summation where p_j and p_k are communicands, for example, then $p_i<p_j$ and $p_i<p_k$. This scheme can therefore be considered as a method of producing a synchronising annotation, and implicitly uses our simplifying assumption. Silberschatz extended this work by introducing *communication ports* [Silberschatz 81]. These can be viewed as a mechanism for implicitly generating the process dominance relation. Processes communicate via ports but in this proposal each port is owned by one, and only one, process. However, there may be several users of the port. Silberschatz imposes the restriction that summations can only involve ports <u>owned</u> by the process. This restriction provides the asymmetry necessary to construct a dominance relation automatically and hence a synchronising annotation can be determined for any program using communication ports. In both of these schemes, the onus is on the user to provide the dominance relation, either explicitly in the first case, or implicitly in the second. We now show how to generate these orderings mechanically, although in many cases we will not achieve a synchronising annotation as there may be incomparable processes.

We wish to detect all pairs of sets of the form

$$\{j,k, \ldots\} \in PC_i \quad , \quad \{i,m, \ldots\} \in PC_j$$

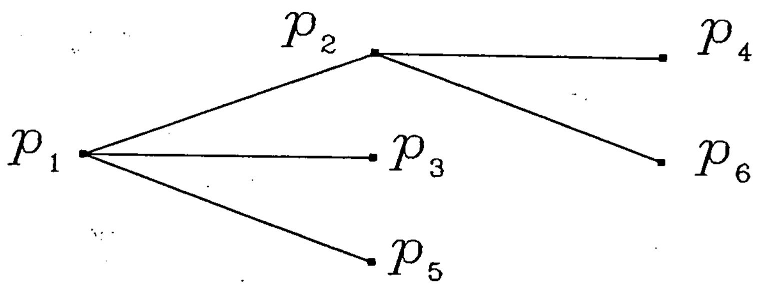
as these may lead to the following connected component in the synchronisation graph.



If we encounter such a case then we must make p_i and p_j incomparable ($p_i \# p_j$). This means that in any communication between p_i and p_j the waiting strategy is inapplicable. As p_k is also involved in this communication, we must set $p_i \# p_k$ so that p_k does not use the waiting strategy when communicating with p_i. Similarly $p_m \# p_i$. Suppose that {k,n} was also an element of PC_i. Now $p_i \# p_k$ so we must make $p_i \# p_n$. In other words, the incomparability may propagate.

We start by grouping together all those processes in PC_i that are affected by setting p_i incomparable to one of them. For each set PC_i, we define an undirected graph $G_i = <N_i, E_i>$, where N_i is the set of communicating partners of p_i and $<j,k> \in E_i$ iff $\{j,k\} \subseteq s$ for some $s \in PC_i$. If we compute the connected components of G_i, CC_i, then it is simple to show that if $p_i \# p_j$ then $p_i \# p_k$ for all processes p_k that are in the same connected component as p_j and these are the only processes that are affected by setting p_i incomparable to p_j. Thus we have a convenient way of propagating # to other processes. Furthermore, we may use the CC_i sets as substitutes for the original PC_i sets, as any clashes between the PC_i sets will also occur between the CC_i sets, and these sets will not introduce any additional clashes.

For example, consider the network



with the following PC_i sets

PC	1	2	3	4	5	6
	$\left\{\begin{matrix}\{2,3\}\\\{3,5\}\end{matrix}\right\}$	$\left\{\begin{matrix}\{1,4\}\\\{6\}\end{matrix}\right\}$	{{1}}	{{2}}	{{1}}	{{2}}

This leads to the following CC_i sets

CC	1	2	3	4	5	6
	$\{\{2,3,5\}\}$	$\left\{\begin{matrix}\{1,4\} \\ \{6\}\end{matrix}\right\}$	$\{\{1\}\}$	$\{\{2\}\}$	$\{\{1\}\}$	$\{\{2\}\}$

The next step in determining a safe annotation is to look for synchronisation clashes in the CC_i sets. One approach is to construct a graph where i_c is a node if c is a connected component of p_i and $<i_c, j_{c'}>$ is an edge of the graph if $i \in c'$ and $j \in c$. Returning to the previous example, this would produce the following graph.

$$\begin{tikzcd}
1_{\{2,3,5\}} \arrow[rr, dash] \arrow[d, dash] \arrow[dr, dash] & & 2_{\{1,4\}} \arrow[d, dash] & 2_{\{6\}} \arrow[d, dash] \\
3_{\{1\}} & 5_{\{1\}} & 4_{\{2\}} & 6_{\{2\}}
\end{tikzcd}$$

If we now examine all of the connected components in the resulting graph, then any component containing a path of length greater than two is a potential cause of a clash. Therefore if $<i_c, j_{c'}>$ is an edge in this component then we set i#j. These are the only instances of incomparable pairs and any other connected components are either of the form

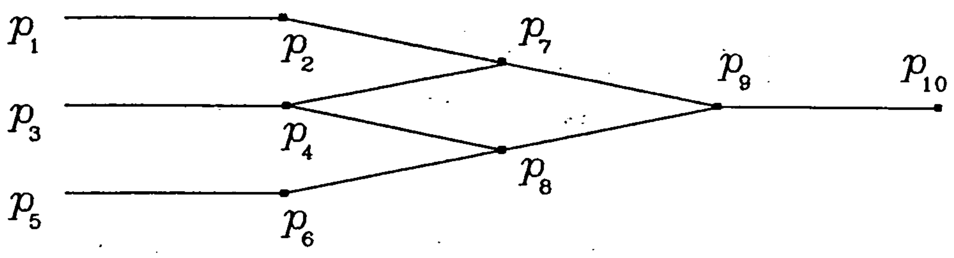
$$\begin{tikzcd}
i_c \arrow[r, dash] \arrow[dr, dash] \arrow[ddr, dash, dashed] & j_{c'} \\
& k_{c''} \\
& {}
\end{tikzcd}$$

in which case we set i<j, i<k etc, or the component is of the form

$$\begin{tikzcd}
i_c \arrow[r, dash] & j_{c'}
\end{tikzcd}$$

in which case we can set i<j or j<i. Thus we may deduce $p_1 \# p_2$, $p_2 \# p_4$, $p_1 \# p_3$, $p_1 \# p_5$ and $p_2 < p_6$ from our previous example.

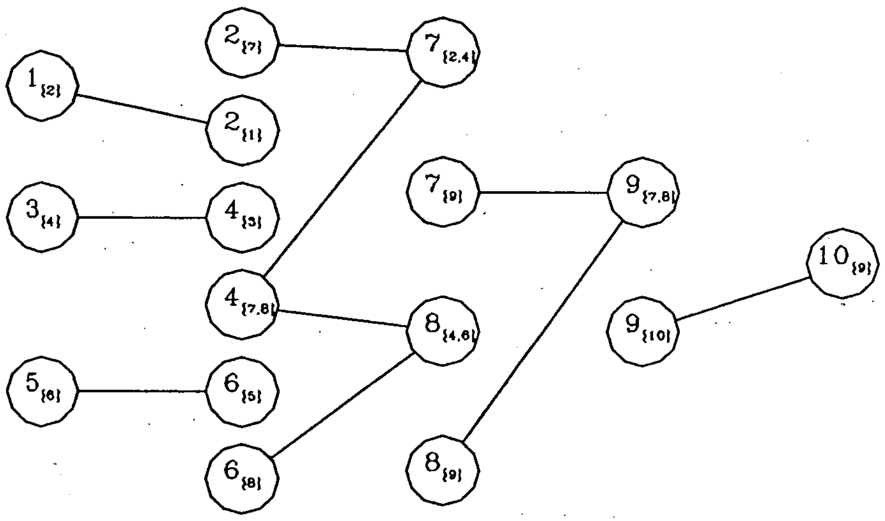
For a slightly more encouraging example, one where there are more comparable pairs, consider the following example.



where the *PC* sets (and the *CC* sets) are

PC	1	2	3	4	5	6	7	8	9	10
	$\{\{2\}\}$	$\left\{\begin{matrix}\{7\}\\ \{1\}\end{matrix}\right\}$	$\{\{4\}\}$	$\left\{\begin{matrix}\{3\}\\ \{7,8\}\end{matrix}\right\}$	$\{\{6\}\}$	$\left\{\begin{matrix}\{5\}\\ \{8\}\end{matrix}\right\}$	$\left\{\begin{matrix}\{2,4\}\\ \{9\}\end{matrix}\right\}$	$\left\{\begin{matrix}\{4,6\}\\ \{9\}\end{matrix}\right\}$	$\left\{\begin{matrix}\{7,8\}\\ \{10\}\end{matrix}\right\}$	$\{\{9\}\}$

This leads to the following graph



from which we can deduce

$$\begin{array}{lll}
P_1 < P_2 & P_2 \# P_7 & P_9 < P_7 \\
P_3 < P_4 & P_4 \# P_7 & P_9 < P_8 \\
P_5 < P_6 & P_4 \# P_8 & P_9 < P_{10} \\
 & P_6 \# P_8 &
\end{array}$$

Given a graph G = $<N,E>$ it takes $O(\text{MAX}(N,E))$ time to compute the connected components, and so this algorithm should be of practical use in constructing safe annotations.

This concludes our analysis of synchronising annotations. We have shown that by annotating a Static CCS program we may obtain an efficient implementation scheme in some cases. This can be done automatically if we make some simplifying assumptions, although the algorithm may produce an annotated program involving *unknown* flags. In such cases, we cannot use the simple synchronisation scheme directly, and the techniques of Chapter 5 must be employed. These involve transforming the program to simplify those communications that are annotated with the *unknown* flag.

§3.6 A review of synchronisation schemes for Static CCS

We now consider the case where the synchronising annotation approach is not applicable. We start by reviewing some of the synchronisation schemes that have been proposed for Static CCS. In fact they were all originally proposed as solutions to the synchronisation problems of CSP, but the two languages are sufficiently similar at the process synchronisation level for the algorithms to be applicable to Static CCS as well. This section is not meant to provide an exhaustive review of the literature in this area although it does cover the major published papers. The aim of the section is to give some idea of the types of implementation strategy that are possible when a synchronising annotation cannot be found for a program. We postpone the discussion of one synchronisation scheme, the polling algorithm due to Schwarz [Schwarz 78], until later in this chapter.

There are a number of ways of classifying process synchronisation schemes. For example, some are designed for real-time applications while others are designed to work on broadcasting networks such as an Ethernet. Some use the natural hierarchies present in the source program to aid synchronisation while others attempt a probabilistic approach where it is possible for two processes to wait indefinitely to synchronise although this can only happen with a vanishingly small probability. Figure 3-1 summarises the synchronisation algorithms that are discussed in this chapter.

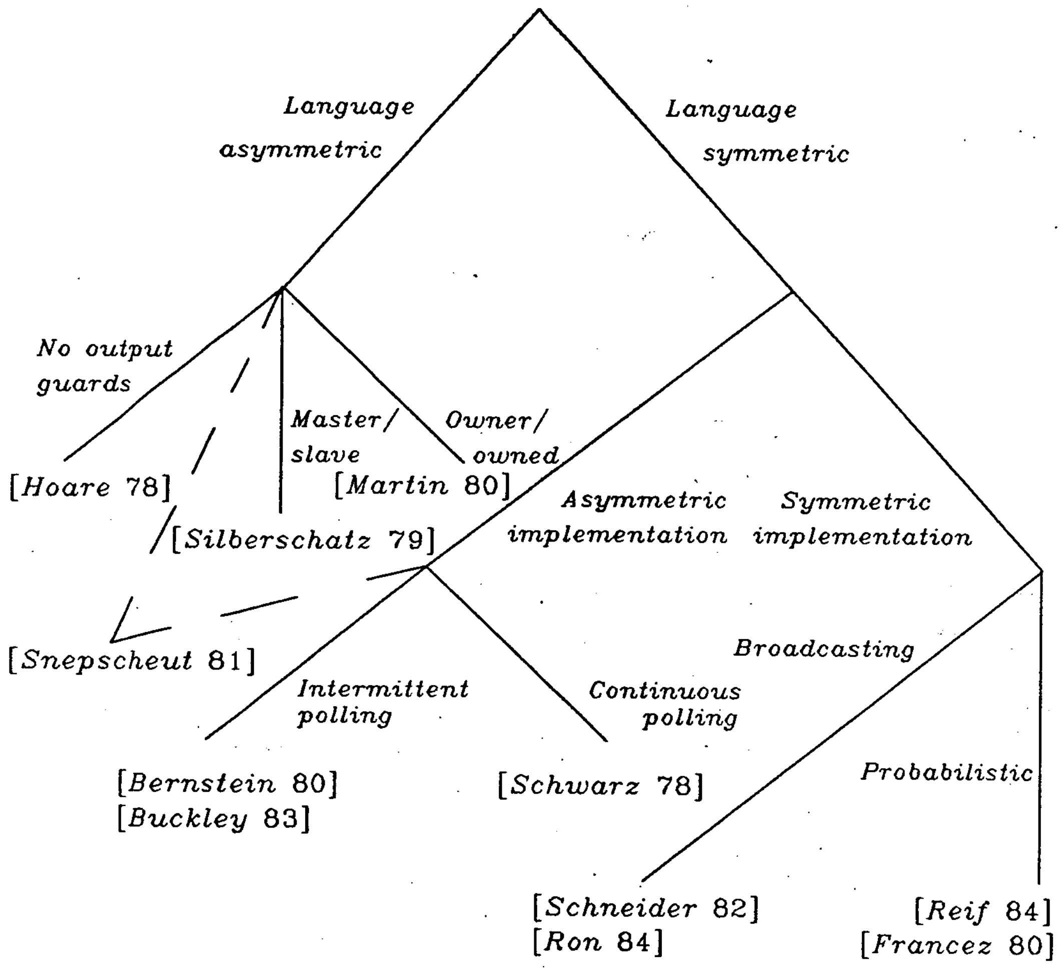


Figure 3-1: Synchronisation schemes for Static CCS

One approach to synchronising Static CCS is to break the symmetry of the system in some way. This can be done either at the source level ([Hoare 78], [Silberschatz 79]), or at the implementation level. We have already seen some examples of the first possibility earlier in this chapter. One other scheme, due to Snepscheut [Snepscheut 81], also falls into this category. Snepscheut argues that it is natural to restrict oneself to hierarchically composed systems. In such a framework, a process p_i may either communicate with its parent p_j, its siblings (children of p_j), or its children. It is the task of process p_j to synchronise all communication requests of p_i naming either p_j or a sibling of p_i. p_i synchronises all communication requests naming one of its children and it is also responsible for synchronising pairs of children with matching requests. Although there are no technical limitations on the programs that may be synchronised using this approach because a suitable communication tree may always be constructed, there are practical limitations. For example, if a particular program requires a communication tree of height two, then the algorithm degenerates to a global scheduler synchronising all communications, which is obviously undesirable. This is why we classify this scheme as one requiring a restriction of the source language.

We now discuss those schemes that place no constraints on the source language and will start by describing some algorithms that produce asymmetric implementations. An implementation is asymmetric if the code executed by process p_i depends in some way on its syntactic position in the program. The purpose of the following descriptions is to illustrate the scope of the possible strategies for implementing a process synchronisation mechanism. For more detailed descriptions of these algorithms, the reader is referred to the relevant references.

The first scheme we consider is the one due to Bernstein [Bernstein 80]. A process may be in one of three states, called *active*, *query* and *wait*. When a process p_i does not wish to perform a communication it is in the active state. Eventually p_i will reach a point where it needs to communicate with some other processes and at that point it enters the query state. In this state it queries each of its possible communicands to ascertain if they are waiting to communicate with p_i. A communicand p_j may respond positively with the message *YES*, in which case the connection is established. It may also respond negatively with the message *NO*, in which case this indicates that process p_j is not interested in a communication with p_i at the current time. Process p_j may also respond with the message *BUSY* in which case p_i may try to query it again at some later point. If all communicands respond negatively then the process enters the wait state where it will agree to the first matching request that is presented to it. If process p_i sends a query to p_j, and while waiting for a reply a matching query from p_k arrives, then two things may happen. If $k > i$ then p_i sends a *BUSY* message back to p_k and otherwise it delays responding to p_k until it has received its reply from p_j.

Buckley and Silberschatz [Buckley 83] show that with certain schedulers there may be no bound on the amount of time or number of messages needed to establish a communication between two processes using Berstein's algorithm. They remedy this defect by proposing a more sophisticated retry mechanism. The first part of the algorithm is identical to the Bernstein proposal. However, if p_i receives a *BUSY* response from p_j then it attempts no further communications with that process until p_j has finished its initial queries. At this point p_j sends a *RES* message to all processes it had sent a *BUSY* reply to earlier. Process p_i may then resolve the noncommittal answer it received from p_j by sending one *RETRY* message. Process p_j will either respond with *NO* in the case where it has returned to the active state since the *BUSY* message was sent, or *YES* if it has unsuccessfully queried all of its communicands. Process p_j may also be currently resolving a *BUSY* communication with some other process p_k and in this case it delays replying to the *RETRY* message (or any new *QUERY* messages). This delay does not introduce a potential deadlock as the chain of *RETRY* messages is acyclic. The algorithm has the desirable property that if two processes can communicate, and one of them does not establish a communication with a third process, then they will communicate with each other within a time bounded statically by the program text.

It is possible to construct synchronisation schemes that have a symmetric implementation. The ones we shall describe either require a broadcasting network as their underlying communication mechanism, or they rely on probabilistic arguments to justify their correctness. Ron, Rosemberg and Pnueli [Ron 84] present a synchronisation scheme which relies on the ability of processes in a carrier-sense based network, such as an Ethernet, to "eavesdrop" on messages not directly addressed to them. A process p_i starts by sending communication requests to all of its communicands. If a matching request is sent back from one of the processes then the other communicands recognise this event by eavesdropping on the line and discard the request from p_i as it is no longer valid. This obviates the need to send retraction messages to the other partners when a successful communication has been established.

The synchronisation scheme due to Schneider [Schneider 82] makes use of a buffered communications network with broadcasting facilities. Each message is tagged with a timestamp obtained from a distributed clock. Lamport [Lamport 78] shows how such clocks may be implemented without using a centralised control mechanism. This timestamp is used to order the requests received by each process. The queue represents the complete state of the system as far as process synchronisation is concerned and so to ensure that process selection operates on a consistent queue, every process broadcasts an acknowledgement to all other processes when it receives a request. Schneider's algorithm deals with fault-tolerant issues not addressed in the other synchronisation schemes but this advantage has to be weighed against the large number of control messages that may have to be transferred before a synchronisation is achieved. Banino, Kaiser and Zimmerman [Banino 79] have also developed a synchronisation scheme based on the use of a shared broadcast channel.

Another approach to obtaining symmetrical implementations where a broadcasting mechanism is not possible, or would be prohibitively expensive to implement, is to make use of probabilistic methods. This approach does not guarantee that two processes will communicate within a finite time but the probability of them not doing so can be made vanishingly small. Francez and Rodeh [Francez 80] have developed such a scheme. We assume that each pair of processes that potentially may wish to communicate has access to a private shared variable. For simplicity, we will also assume that for each pair of processes there is a unique label that is used for their communication. Suppose process p_i wishes to communicate with process p_j. It indicates its willingness to do so by setting the shared variable between them. We assume that this flag is initially cleared. If after a certain 'timeout' period the flag is still set then the process assumes that p_j is not interested in performing a communication. In this case p_i clears the flag shared with p_j and sets the flag connecting it to one of the other communicands. This action is known as a *retraction*. Process p_i will continue this polling until it finds that the flag has been cleared when it is examined after a timeout. Process p_i takes this as an indication that the process that shares the variable wishes to communicate and so the connection is established. Similarly if p_j wishes to communicate with p_i and finds that the shared variable has already been set, it clears the variable and waits for the connection to be established. The algorithm assumes that there is some form of mutual exclusion mechanism that prevents p_i and p_j from setting their shared variable simultaneously.

The probabilistic scheme introduces the possibility that a pair of processes may repeatedly set the shared variable but, due to an unfortunate scheduling strategy, may continually miss each other. Francez and Rodeh therefore assume that the underlying implementation uses a fair random scheduler. With such a scheduler they argue that although an infinite time may elapse while two processes try to establish a successful connection, this can only happen with zero probability.

Reif and Spirakis [Reif 84] also present a probabilistic solution to the synchronisation problem. Their approach can be viewed as a real-time solution in the sense that they can place a limit on the time taken to establish a communication, and the chance that this bound is exceeded can be made vanishingly small.

While our review of these synchronisation schemes covers the major work in this area, there are other algorithms which, although not specifically addressed to the problems of process synchronisation, can be applied to such a case. The research published on resource synchronisation problems is a good example of such work ([Lynch 80]).

§3.7 Synchronisation schemes considered as program transformations

Some synchronisation schemes for Static CCS are based on a synchronous message-passing mechanism, and so may be expressible in Static CCS. Other schemes may be placed in such a framework even though they may be initially presented in terms of shared variables for example. We might argue that to transform a program using a particular synchronisation scheme is no better than implementing the program directly using the scheme. We counter this remark in the following ways. Firstly, we may not be able to use a particular scheme directly because we have no control over the underlying implementation. More importantly, it is very wasteful to synchronise an entire network of processes using a complicated synchronisation scheme when only a small number of the communications may require its generality. The transformational approach has the advantage that it can be applied only to those parts of the system where a synchronising annotation cannot be found. While it would obviously be possible to mix strategies at the implementation level, this would be more difficult and because of this we argue that the transformational approach is more flexible. The algorithm we use may depend on our particular problem; we may want a transformation with real-time properties for example. It would be unreasonable to expect the underlying implementation to present us with such a wide range of choices.

§3.8 Schwarz' synchronisation scheme

This section describes the synchronisation scheme due to Schwarz [Schwarz 78]. The next section will then show how it may be expressed as a program transformation. Furthermore, it is shown that a synchronising annotation may always be constructed for the transformed program.

Many of the schemes for synchronising Static CCS employ the technique of imposing some form of asymmetry on the system either at the source or implementation level. The scheme due to Schwarz also follows this approach. We assume that an acyclic dominance relation $>$ has been imposed on the processes. The choice of whether p dominates **q** or vice versa is independent of the direction of any possible communication between these two processes and can be chosen arbitrarily. Schwarz has shown that the choice of dominance relation can affect the performance of the algorithm but not its correctness.

In order for process p_i to establish communication with p_j, Schwarz proposes that they perform a "question and answer exchange". One process is permanently designated the asker and the other the answerer. If $p_i > p_j$ then p_i is the asker and let us assume that this is the case. These two processes synchronise through the two variables Q_{ij} and A_{ji}. We will allow Q_{ij} to be set by p_i and be read by p_j. Similarly we will allow p_j to set A_{ji} and it can be read by p_i. We assume that both variables are initially set to Λ. If p_i wishes to communicate with p_j it starts by setting Q_{ij} to the value "R". This is sensed by p_j and the process responds by setting A_{ji} to either "*Y*" if it wishes to communicate with p_i, or "*N*" otherwise. When p_i senses the setting of A_{ji} it resets Q_{ij} and after this action p_j clears A_{ji}. In order for a connection to be successfully established between p_i and p_j, p_i must ask if p_j wishes to communicate and p_j must reply positively. Process p_i is suspended until it receives an answer from p_j which implies that p_j must be monitoring Q_{ij} even when it has no desire to communicate with p_i. Because p_j must respond to questions even when no communication is to take place, Schwarz assumes that each process contains a "poller" subprocess which is responsible for asking and answering questions. Each process must have a means of communicating with its subprocess, or poller, and so Schwarz provides a set of variables C_{ij} for process p_i such that setting C_{ij} to true implies that the main process p_i is willing to communicate with p_j. How the poller indicates the successful establishment of a communication is left undefined.

Schwarz shows that to avoid deadlocks it is necessary to choose the $>$ relationship between processes such that there are no infinite sequences

$$\mathbf{p}_i > \mathbf{p}_j > \mathbf{p}_k > \ \ldots$$

This is why we stated that the dominance relation should be acyclic. If we assume that the processes are indexed by positive integers in some arbitrary way, we may take $>$ to be "greater than" (>). In order for a poller to check for communication requests from other pollers, it must ascertain which processes can potentially communicate with it. We assume that the ith process $\mathbf{p}_i$ can potentially communicate with the N_i processes $connect_i[0]$, $connect_i[1]$, . . . , $connect_i[N_i-1]$.

We give an outline of the algorithm executed by the ith poller in Figure 3-2.

The commands "lock x" and "unlock x" respectively freeze and unfreeze the variable x to prevent the variable changing while in a critical section. We assume that these primitives can only be used on variables that are shared by local processes. This is true in the poller definition as C_{ij} is shared by the poller and its controlling process and these are local to each other. "await C" is an abbreviation for the busy-waiting loop

"while ¬C do od"

§3.9 A transformation version of Schwarz' scheme

In an experimental implementation of CSP, Shrira and Francez [Shrira] have transformed Schwarz' synchronisation scheme into a version that uses message-passing rather than shared variables to communicate between processes. The approach is quite general, and could be applied to other algorithms that use shared variables. Instead of a variable being set by one process and read by another, we modify the algorithm so that a message is sent by one and received by the other. We must also modify the poller algorithm so that it continually offers the message "I do not

```
Poller i:

begin n,j,a,q; n:=0;

    while true do
        n:=(n+1) mod $N_i$;  j:=connect$_i$[n];
        lock $C_{ij}$;

        if i>j ∧ $C_{ij}$
            then $Q_{ij}$:="R"; await $A_{ji}$≠Λ;
                    a:=$A_{ji}$;
                    $Q_{ij}$:=Λ; await $A_{ji}$=Λ;
                    if a = "Y" then "establish channel i j"
                    □ a = "N" then {offer rejected} fi
        □ i > j ∧ ¬$C_{ij}$
            then {do nothing}
        □ j > i
            then q:=$Q_{ji}$;
                    if q≠Λ ∧ $C_{ij}$
                        then $A_{ij}$:="Y"; await $Q_{ji}$=Λ; $A_{ij}$:=Λ;
                                "establish channel i j"
                    □ q≠Λ ∧ ¬$C_{ij}$
                        then $A_{ij}$:="N"; await $Q_{ji}$=Λ; $A_{ij}$:=Λ;
                    □ q=Λ
                        then {do nothing}
                    fi
        fi

        unlock $C_{ij}$;
    od;
end;
```

Figure 3-2: The Schwarz Poller Algorithm

want to communicate" rather than doing nothing and letting the other process deduce this fact by looking at the shared variable.

The substitution of message-passing for the shared variables allows us to express Schwarz' scheme in Static CCS, and hence to view it as a program transformation. However, if this approach is to be meaningful, we must avoid introducing any additional communication possibilities that are difficult to synchronise as otherwise we would be reintroducing the problem that the transformation was designed to solve.

We introduce the transformation as a function over syntactic terms representing Static CCS expressions. Because of the syntactic nature of the transformation function we would not necessarily expect the transformation of p|q to be identical to the transformation of q|p although the two resulting terms would hopefully behave in an identical fashion to an external observer. The syntactic treatment of the arguments to the mapping function, coupled with the fact that all Static CCS arguments will consist of a parallel composition of one or more simple processes, allows us to associate a natural ordering on the source processes based on their relative positions in the parallel composition.

In order to transform a Static CCS program, we must translate each summation into an explicit request set. We use a variant of the request set function, RS, to perform this task. We define

$$C(\lambda) = \{ j \mid \bar{\lambda} \in S_j \} \text{ for any } \lambda \in \mathcal{Act},$$

and will assume that the source programs are such that the corresponding function C also satisfies the following restriction

$$C(\lambda) \cap C(\lambda') \neq \phi \supset \lambda = \lambda' \text{ whenever } \lambda, \lambda' \in S_i \text{ for any i.}$$

This condition guarantees that if process p_i wishes to communicate with process p_j then there is no confusion over which label they wish to use in the communication. The restriction simplifies the presentation of the algorithm, although the analysis can be extended to cover the general case without difficulty.

Our first step in the conversion of Schwarz' scheme to a CCS version is to decide how a master process interacts with its poller. In the original algorithm the two processes interacted via the C_{ij} variables. The poller prevented its master changing a variable when it was examining it, and at all other times the master was free to change the variable. We must replace this mechanism by one which involves message-passing. There are a number of possibilities open to us at this point. For example, the poller may refuse to communicate with any other process until it has received a request from its master, after which it ignores the master until a communication has been established.

Another alternative would be for the poller process to be always willing to accept a request from its master. We might choose to poll the master process along with all the other pollers. Allowing the poller to always be able to accept a message from the master process creates a synchronisation problem because when two pollers wish to communicate there will be a many-to-many communication request at this point. We therefore reject this alternative.

Allowing a poller to poll its master means that a translation of a process must continually be willing to offer its request until a successful communication has been established. This approach does allow a master process to retract a request at any point which may be an advantage if we wish to allow τ moves in our source program. Such an extension does not provide any extra insight into the problem and adds an extra degree of complexity. For this reason we will choose the first alternative which does not allow retractions but allows a simple presentation of the algorithm.

It is not sufficient to leave the *NIL* process unchanged in the transformation using this approach because the poller will interrogate its master when it has no outstanding requests, and this will deadlock the poller and possibly lead to a total deadlock of the system. However, if process p_i reaches a state where it is equivalent to *NIL*, then the transformed version can send a message to the poller requesting to communicate with itself. Such a request can never be satisfied and so the poller does not require any additional communications with its master. At this point the master can safely evolve to the *NIL* process. To avoid similar problems when a process wishes to synchronise on communications that can never occur, due to the inverse labels not appearing in the sorts of any other processes, we add i to all request sets of process p_i.

We assume that each poller, $Poller_i$ say, receives requests from its master in the form of a set of process identifiers via the port $offer_i$. $Poller_i$ indicates a successful synchronisation by passing back the identity of the communicating partner using the port $select_i$. Given a Static CCS term of the form $\prod_{i \in N} p_i$, we translate each process p_i using the function tr_i as follows.

Each process p_i can syntactically be viewed as the process $\sum_{j \in m} a_j.p_{ij}$ with a suitable choice of variables. Then

$$tr_i\left[\!\left[\sum_{j \in m} a_j.p_{ij} \right]\!\right] = \underline{\text{let}}\ partners = \bigcup_{j \in m} C(a_j)\ \underline{\text{in}}$$
$$\left(\begin{array}{l} \overline{offer_i}\left(partners \cup \{i\}\right). \\ \displaystyle\sum_{\pi \in partners} \left(select_i(\lambda).a_k.tr_i[\![p_{ik}]\!]\ \underline{\text{where}}\ C(a_k) = \pi\right) \end{array} \right)$$

We should really treat the label a_j as being composed of two parts; a label and an optional value or variable if we are using value passing. The function C should then be defined so as to ignore the value part, or alternatively we should supply a projection function from a_j to the label of a_j. However, as there is no scope for confusion if this coercion is performed implicitly, we will use the variable a_j for both purposes.

Instead of using an indexed family of transformation functions, we could equally well have used a single transformation function and then applied an indexed family of renamings to the transformed processes to achieve the same result. Each master process has its own local poller process. We introduce a second local process, $Buffer_i$, to obtain the effect of the Q_{ij} shared variables. The process could be incorporated into the poller definition but this would unduly complicate matters. The setting of the Q_{ij} variable by $Poller_i$ is achieved by sending a set_i(j) message to $Buffer_i$. $Poller_j$ can interrogate the status of this variable by using the Q_{ij} port. This port returns *YES* if $Poller_i$ currently wishes to query $Poller_j$ and *NO* otherwise. A typical transformed component is illustrated in Figure 3-3.

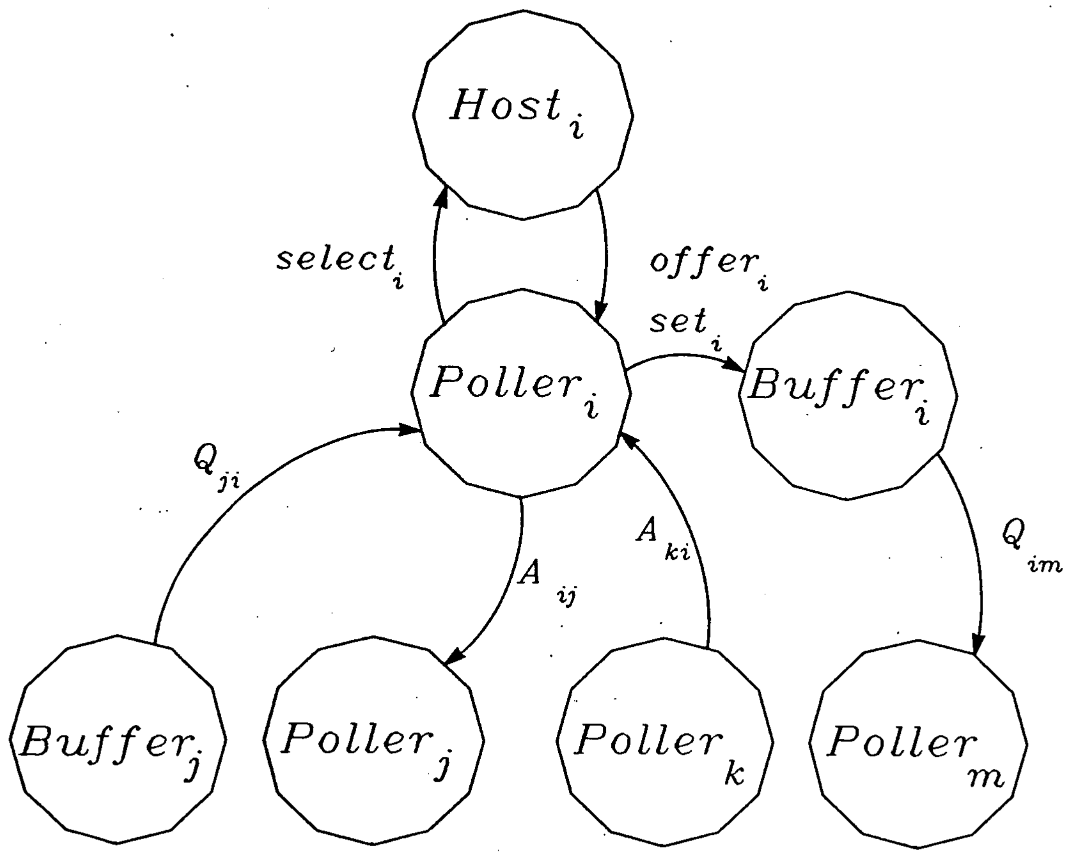


Figure 3-3: A Poller Component

We define PC_i to be $\bigcup_{\lambda \in S_i} C(\lambda) - \{i\}$. Then $Buffer_i$ can be defined by

$$Buffer_i = \sum_{j \in PC_i} \left(\begin{array}{c} \overline{Q_{ij}}(NO).\ Buffer_i \\ +\ set_i(j).\ Buffer'_i(j) \end{array} \right)$$

$$Buffer'_i(k) = \left(\sum_{j \in PC_i - \{k\}} \overline{Q_{ij}}(NO).\ Buffer'_i(k) \right) + \overline{Q_{ik}}(YES).\ Buffer_i$$

All that remains to complete the transformation is a description of the poller subprocesses. In order for $Poller_i$ to perform its task it must know the identities of all the remote pollers that may wish to communicate with it. The set PC_i contains the identities of all such processes. To allow $Poller_i$ to interrogate the members of this set fairly we assume that some arbitrary ordering has been imposed on the set such that $PC_i[n]$ denotes the nth element.

A description of $Poller_i$ is presented in Figure 3-4.

We assume that addition is (modulo $|PC_i|$)+1 so that n ranges over 1 to $|PC_i|$.

To transform a Static CCS program $\prod_{i \in N} \mathbf{p}_i$, we first determine the sorts S_i and then apply the transformation function Tr where

$$Tr\left[\!\!\left[\prod_{i \in N} \mathbf{p}_i \right]\!\!\right] = \prod_{i \in N} \left(tr_i[\![\mathbf{p}_i]\!] \mid Poller_i(1,\phi) \mid Buffer_i \right)$$

In fact this definition illustrates one of the problems that arises when dealing with transformations in CCS. It would be natural to expect that some restrictions should appear in the above expression. However, if we added the relevant restrictions we would find that <u>all</u> the actions of the system would be affected by the restrictions. There would be no externally visible actions, and this raises the question of how to prove that the transformation is correct. We deal with such problems in the next chapter.

```
$Poller_i(n,k)$ = let $j = PC_i[n]$ in

    $offer_i(k'). Poller_i(n,k')$                 if $k = \phi$

  + $\overline{set}_i(j).$                         if $i>j \wedge j \in k$
    $A_{ji}(r).$
    if $r = YES$
    then $\overline{select}_i(j). Poller_i(n+1,\phi)$
    else $Poller_i(n+1,k)$

  + $\tau. Poller_i(n+1,k)$                       if $i>j \wedge j \notin k \wedge k \neq \phi$

  + $Q_{ji}(r).$                                   if $i<j \wedge j \in k$
    if $r = YES$
    then $\overline{A}_{ij}(YES). \overline{select}_i(j). Poller_i(n+1,\phi)$
    else $Poller_i(n+1,k)$

  + $Q_{ji}(r).$                                   if $i<j \wedge j \notin k \wedge k \neq \phi$
    if $r = YES$
    then $\overline{A}_{ij}(NO). Poller_i(n+1,k)$
    else $Poller_i(n+1,k)$
```

Figure 3–4: The Schwarz Poller in Static CCS

§3.10 A synchronising annotation for Schwarz' transformation

In order to justify the transformation, at this point we must ask ourselves two questions. Firstly, are the transformed terms any easier to implement than the original program, and secondly, does the transformed program behave identically to the original program, i.e. is the transformation correct in some sense? A third question, how does the transformation affect the performance of the program, will be left until later when we describe how the transformation may be partially applied. This section deals with the first question. We show how a synchronising annotation can be constructed for any transformed program, even when the original program does not possess one. The first part of this chapter has already described how we can efficiently implement programs that have a synchronising annotation, and so we may deduce that the transformation does indeed aid the implementation process.

The first property we observe about the transformation is that all the conditional cases in the poller definition are mutually exclusive. The $offer_i$ message between $Poller_i$ and its master is a one-to-one communication and so either may be the "master". The $\overline{Set}_i(j)$ message matches with a sum in $Buffer_i$ so $Poller_i$ must be the "master" and $Buffer_i$ the "slave" for this communication. $A_{ji}(r)$ is a one-to-one communication between $Poller_i$ and $Poller_j$ so either can be the "master". The $\overline{select}_i(j)$ message interacts with a sum in the master process so $Poller_i$ must be the "master" in this case. $Q_{j,r}(r)$ interacts with a sum in $Buffer_j$ so $Poller_i$ must be the "master" in this case as well. Finally, the messages that are exchanged between masters are of the one-to-one form so any "master/slave" relationship is adequate. This completes our analysis and shows how a synchronising annotation may be constructed for this transformation.

Both the pollers and buffer processes contain input and output actions within a summation, and so the Hoare restriction of forbidding output guards in summations, to obtain an efficient implementation, would be inapplicable in this case.

§3.11 Transformation correctness

If the same observer process examined both the program and its transformation, then it would obviously notice a difference. This is because the interfaces to the environment are different in the two programs. Therefore, if we tried to apply the testing equivalence $\simeq_2$ to these programs we would deduce that they were not equal. In order to compare the programs the observer must interact with the transformed system like all the other processes, via a poller. Suppose we are presented with an observer $\prod_{i\in M} o_i$ and a source program $\prod_{i\in N} p_i$. Then we must translate both components simultaneously, i.e. we first construct a new term

$$\prod_{i\in M+N} q_i \text{ where } o_i = q_i \ \forall i\in M \text{ and } p_i = q_{i+M} \ \forall i\in N.$$

We can then determine the sorts and apply the *Tr* function to obtain a transformed version of both the observer and the observed. Such an approach to transformational correctness is dealt with in more detail in the next chapter. At this point it is sufficient to note that some modification of our notion of equivalence is necessary when a transformation changes the externally visible interface to a system.

Another perhaps more serious problem that confronts us when trying to reason about the correctness of the transformation is due to the introduction of non-termination caused by the transformation even when the original processes terminate. For example, consider the process $Poller_1$. If the master process reaches a *NIL* state then $Poller_1$ will have the variable k set to {1}. The poller will continually check all other pollers in PC_1, and if none of them wish to communicate with $Poller_1$, then there can be an infinite sequence of communications between $Poller_1$ and remote Buffer processes with no other processes progressing.

To avoid such problems in the Schwarz transformation, we could try to modify the polling algorithm, or restrict the class of programs that were transformed, in an attempt to eliminate all infinite τ sequences that may be introduced by an unfair implementation. If we view the collection of PC_i sets as specifying a connection graph for the system, then there may be more than one connected component in the graph, and the lowest element of every connected component may perform an infinite sequence of polling communications leading to a diverging computation. We might argue that it only makes sense to have one connected component in any connection graph, because either the observer must exist in only one of the connected components, in which case the other connected components cannot influence the success of its testing, or else the observer is part of a number of connected components, in which case this is equivalent to running several tests on several separate components simultaneously, and this could equally well be done separately. Even if we restrict ourselves to connection graphs with only one connected component, there is still the possibility of non-termination. To try to remove the possibility of non-termination from the process with the lowest index, we might force this process to be part of the observer and assume that the system is always willing to communicate with the observer. However, this is unrealistic as in general the observed system will need to evolve internally between each communication with the observer. There appear to be no other reasonable changes or restrictions we might make to the system so that it will function correctly on an unfair implementation.

Before we can prove the correctness of Schwarz' transformation scheme, we must examine in more detail those preorders and equivalences that treat the introduction of some forms of non-termination as being benign. We must also describe how to perform transformation correctness proofs when the visible interface to the system is altered by the transformation. This work forms the core of the next chapter.

CHAPTER 4

A Mathematical Framework for the Notion of "Implementation"

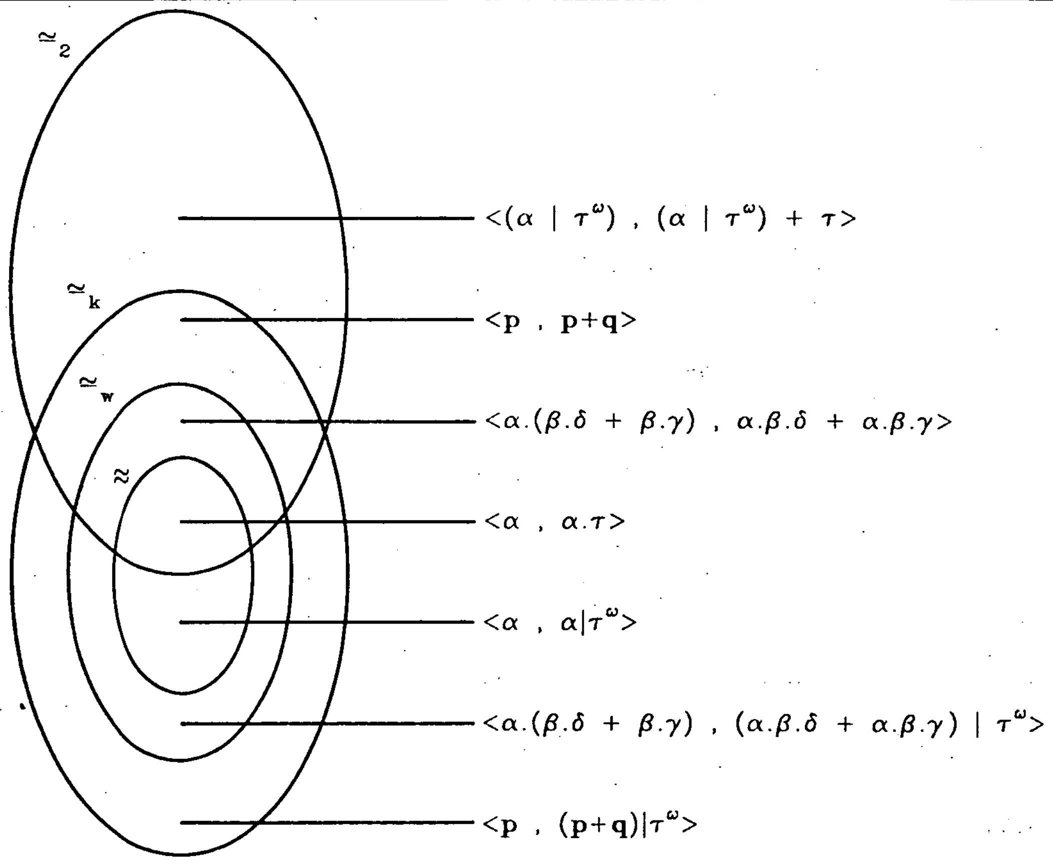
§4.1 Introduction

An agent defining the intended behaviour of a component is usually called a specification in CCS. There is no formal distinction between specifications and other agents, although a specification will typically define the desired behaviour in as clear a way as possible, with little regard for efficiency. An implementation of the specification then consists of a behaviour that is equivalent to it, but that also satisfies other constraints, such as being more efficient, or containing a fixed number of processes. There is some flexibility in these informal definitions, as the equivalence used may depend on the particular agents under investigation, but the specification/implementation relationship is symmetric. The first part of the chapter argues that there is a case for making the relationship asymmetric. The aim is to develop an ordering that places less constraints on what constitutes an implementation, while retaining the ability to be observably indistinguishable from the specification. We do not deal with context-dependent proofs in this thesis, and so any proposed definition of implementation should preserved the ordering under all CCS contexts. These constraints limit the degree to which an implementation can differ from being simply equivalent to its specification. A candidate for this ordering is proposed that corresponds to the intersection of $\sqsubseteq_2$ and the converse of $\sqsubseteq_3$. Proving that a behaviour implements a specification using the new definition is not only simpler than proving an equivalence, but is also sufficient for many applications, including the proof of the Schwarz transformation.

The second part of the chapter develops a fair equivalence for CCS. The starting point for this analysis is the testing equivalence approach of DeNicola and Hennessy [DeNicola 82]. We argue that their approach agrees with our intuitions concerning process equivalence, except where fairness arguments influence these intuitions. In particular, we argue that for any behaviour p, $p|\tau^\omega$ should be a valid implementation of p, if we are only interested in fair versions of CCS. Previous attempts at applying fairness arguments to CCS have been too restrictive, and do not interact properly with the expansion theorem. A new preorder, known as the weak-must testing preorder, $\sqsubseteq_w$, is developed as a fair replacement for the must testing preorder, $\sqsubseteq_2$. A weak-must equivalence, $\simeq_w$, is also defined. We show that the new definition behaves in a similar way to the $\sqsubseteq_2$ preorder, except for certain infinite computations where we argue that the new treatment of the behaviours is more natural when reasoning in a fair framework.

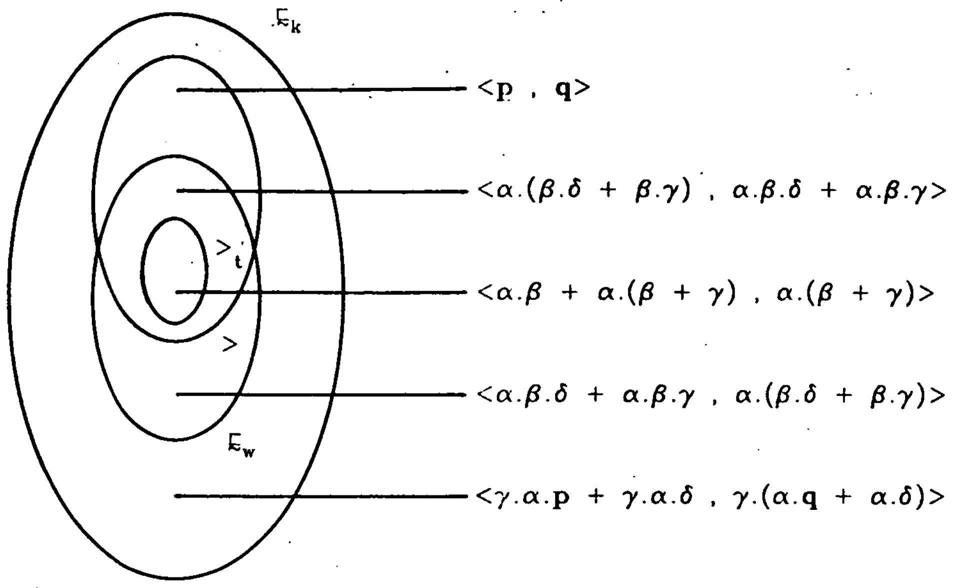
Proofs that deal explicitly with observers are difficult in general, and so it is desirable to find an alternative characterisation of the weak-must preorder that does not involve observers. Kennaway [Kennaway 81] defines an equivalence, $\simeq_k$, whose treatment of certain infinite terms is identical to that required by $\simeq_w$, and so provides a suitable starting point for this search. We define the corresponding preorder, $\sqsubseteq_k$, and prove that $\sqsubseteq_w$ is contained within $\sqsubseteq_k$, although the converse is not true. We argue that the treatment of behaviours when the two preorders differ is more natural using the weak must preorder. We show that a sufficient, but not necessary, condition for the two preorders to agree is when the class of processes is restricted to those that are determinate in some sense. The Kennaway preorder involves sets of processes and so, while proofs may be simpler than with the weak must preorder, they are by no means straightforward. A preorder whose definition is amenable to bisimulation style proof techniques [Park 81] is obviously more desirable, and this leads to the definition of the > preorder. The new preorder directly implies $\sqsubseteq_k$, but not $\sqsubseteq_w$. It is also more particular about when non-deterministic choices are made in the two processes. Finally, to obtain a preorder that does directly imply the $\sqsubseteq_w$ preorder, and also has a bisimulation proof technique, the $>_t$ preorder is proposed. This preorder is even more particular about when non-deterministic choices are made. There is therefore a trade-off between ease of proof, and the class of programs to which these techniques are applicable. This last point will become important when we prove the correctness of the Schwarz transformation. Figures 4-1 and 4-2 summarise the relationships between the preorders and equivalences presented in this chapter.

The final part of the chapter discusses how a CCS transformation function can be proved correct. A definition of transformation correctness, due to Millington [Millington 82], is first presented. This definition is then generalised to cover the type of functions typified by the Schwarz transformation.


$\simeq_2$
$\simeq_k$
$\simeq_w$
$\approx$
$<(\alpha \mid \tau^\omega)\ ,\ (\alpha \mid \tau^\omega) + \tau>$
$<p\ ,\ p+q>$
$<\alpha.(\beta.\delta + \beta.\gamma)\ ,\ \alpha.\beta.\delta + \alpha.\beta.\gamma>$
$<\alpha\ ,\ \alpha.\tau>$
$<\alpha\ ,\ \alpha|\tau^\omega>$
$<\alpha.(\beta.\delta + \beta.\gamma)\ ,\ (\alpha.\beta.\delta + \alpha.\beta.\gamma) \mid \tau^\omega>$
$<p\ ,\ (p+q)|\tau^\omega>$

where $\mathbf{p} = \alpha.\mathbf{p} + \alpha.\beta.NIL$ and $\mathbf{q} = \alpha.\mathbf{q}$

Figure 4-1: The relationship between various equivalences


$\sqsubseteq_k$
$<p \, , \, q>$
$<\alpha.(\beta.\delta + \beta.\gamma) \, , \, \alpha.\beta.\delta + \alpha.\beta.\gamma>$
$>_t$
$<\alpha.\beta + \alpha.(\beta + \gamma) \, , \, \alpha.(\beta + \gamma)>$
$>$
$<\alpha.\beta.\delta + \alpha.\beta.\gamma \, , \, \alpha.(\beta.\delta + \beta.\gamma)>$
$\sqsubseteq_w$
$<\gamma.\alpha.p + \gamma.\alpha.\delta \, , \, \gamma.(\alpha.q + \alpha.\delta)>$

where $p = \alpha.p + \alpha.\beta.NIL$ and $q = \alpha.q$

Figure 4-2: The relationship between various preorders

§4.2 Implementations

Let us consider a specification process **s** and another process **i** that is supposed to "implement" **s** in all contexts (see Larsen [Larsen 85] for a discussion on context-dependent proofs). Following the *testing* approach of Hennessy and DeNicola, what relations would we expect to hold between processes **s** and **i**? Suppose **s** *must satisfy* **o**, or in other words, when observing **s** with observer **o**, the combined system always succeeds. Process **s** therefore always performs in such a way that **o** can eventually signal √, success. What would we expect to happen if **i** was placed in parallel with **o**? If there was a computation of **i|o** where success could not be reported we would be rather unhappy. We might argue that part of the specification of **s** demanded that it must always satisfy **o**, and so any process that claims to implement the specification must also satisfy **o**. This requirement is equivalent to stating that s _must satisfy_ o $\supset$ i _must satisfy_ o

Suppose that i _may satisfy_ o. It would seem reasonable to demand that an implementation has no more possible actions than its specification. If we did not demand this then placing i in a context previously occupied by s might result in totally unforseen behaviour, due to the implementation communicating with the environment through labels that were not used in the specification. We must also ensure that the implementation does not contain sequences of actions that are not present in the specification for the same reason. Thus we will demand that

i _may satisfy_ o $\supset$ s _may satisfy_ o

What about the other two possibilities? Suppose s _may satisfy_ o. If s always satisfied o then we would have s _must satisfy_ o and hence i _must satisfy_ o. If this is not the case then it may be that the implementation has chosen to implement a different non-deterministic branch of the specification, and so may never satisfy o.

Suppose that i _must satisfy_ o. Then we do not want to demand that s _must satisfy_ o because s may specify a choice of actions of which the implementation only choses one of them. As i _must satisfy_ o implies i _may satisfy_ o, we know that it is possible for the specification to satisfy o some of the time. To illustrate these points, let us take a simple example. A change making machine might be specified and implemented as

$$\begin{aligned} CD &= pound?.\\ &\quad \big(\tau.pence!(100).CD + \tau.shilling!(20).CD + \tau.fiftypence(2).CD\big) \end{aligned}$$

$$ICD = pound?.(pence!(100).ICD)$$

where CD is the specification and ICD a possible implementation.

CD must be able to accept a pound note and then deliver some change. There is no way of forcing a particular form of change as this depends on what resources are left in the machine. Therefore the only tests that <u>must</u> succeed are those which are prepared to accept any form of change. But then ICD is always willing to give back one acceptable form of change, and hence ICD is a reasonable implementation of CD. Similarly, we had better check that any change given back by our implementation of the cash dispenser was mentioned in the specification. For example, a cash dispenser that returned dollars and cents would not be an acceptable implementation of CD.

What of the other two cases? CD may return shillings but as we cannot demand that it does, we have no way of checking whether our implementation has this capability, and so we do not demand it. Similarly, our implementation must return pence but again any observer would be happy with our implementation as it would always be performing an acceptable part of the specification; it is not necessary that the specification <u>must</u> return pence and so we do not demand it.

To summarise, we define the notion of implementation as follows.

i <u>implements</u> s or i is an <u>implementation</u> of s iff

$$\forall\, o \in \mathcal{O}.\ i\ \underline{\textit{may satisfy}}\ o \supset s\ \underline{\textit{may satisfy}}\ o$$

$$s\ \underline{\textit{must satisfy}}\ o \supset i\ \underline{\textit{must satisfy}}\ o$$

It might be argued that even this notion of implementation is too restrictive. For example consider the hardware device known as a flip-flop. The specification of this device is normally very naive in that it only specifies what happens if the device is used sensibly. By this we mean that it is possible to drive the flip-flop in a way that is not covered by the specification. In these cases, the behaviour of the hardware may be non-deterministic and so we have a situation where the implementation has more possibilities than the specification. We might argue that the specification should be strengthened to cover these cases but this would complicate the specification, especially as the component is not supposed to be used in such an environment. The more elegant solution would be to prove that the implementation and the specification, when placed in a certain context, or within a certain environment restriction, behaved correctly. This type of context-dependent proof in CCS is currently under investigation [Larsen 85]. However the simplest approach would be to allow the implementation to have more capabilities than the specification. We might desire something of the form

$$\text{i } \underline{\text{implements}} \text{ s} \Leftrightarrow \text{s } \underline{\text{after}}\ s\ \underline{\text{must}} \text{ L} \supset \text{i } \underline{\text{after}}\ s\ \underline{\text{must}} \text{ L}$$

for all visible sequences of actions *s*, and sets of visible actions L (the definitions of after and must are presented on page 106). However, although this definition may be easier to use and prove, the implementation may be able to perform all kinds of actions not mentioned in the specification. We would therefore have to be very careful about the contexts in which we placed the implementation. In general, the behaviour of $\mathscr{C}[\![i]\!]$ would be very different from $\mathscr{C}[\![s]\!]$ for an arbitrary context $\mathscr{C}$. Placing constraints on the sorts, such as $Sort(i) \subseteq Sort(s)$, would not help, as i may still possess sequences of actions not present in s. We might add further constraints to cover this case as well, but as we strengthen the constraints, we are inescapably drawn to the stage where we demand that i and s are observably indistinguishable. This is what our previous definition of implementation was intended to formalise. To prove the flip-flop example correct in CCS, some form of context-dependent proof therefore seems unavoidable.

§4.3 The introduction of non-termination and fairness

Let us imagine that we have been presented with a CCS expression that represents the specification of some problem. Our task is to write an implementation in CCS that in some sense agrees with this specification. Assuming that s is the specification and i the resulting implementation, then by our previous analysis this amounts to showing that i implements s. Part of this task involves demonstrating that for any observer o, if s *must satisfy* o then so must i. This would appear to be a reasonable requirement of any implementation, and in many problems this is indeed the case. However, suppose that our implementation introduces auxiliary behaviours that may 'chatter' amongst themselves indefinitely. As an implementor we may find this addition perfectly acceptable. We might appeal to fairness arguments, for example, to justify the correctness of the implementation. Although we are introducing the possibility of divergence, it is of a restricted form in that at any point in a computation it is always possible to continue with the desired execution sequence. This contrasts with the introduction of an infinite τ chain with no other possibilities along its length, which is obviously harmful.

Perhaps the simplest example of the sort of process we have in mind is where we wish to view $\mathbf{p}|\tau^{\omega}$ as an implementation of **p**. If we assumed that both **p** and τ^{ω} were scheduled fairly, i.e. neither behaviour was allowed to monopolise the processor indefinitely, then $\mathbf{p}|\tau^{\omega}$ would appear to be a reasonable implementation of **p** (although we would expect it to be slower). Unfortunately, CCS does not have any fairness assumptions built into it and so, for example, the *must testing* equivalence for CCS, $\simeq_2$, differentiates between these terms. The reason for this is clear; $\mathbf{p}|\tau^{\omega}$ *must satisfy* **o** only when **o** may report success immediately. This follows from the definition of *must satisfy*, where we demand that every computation passes through a state where a √ move is possible. This includes the infinite τ^{ω} computation, and so √ must form one of the initial actions of **o**. **p** *must satisfy* **o**, on the other hand, may be true because of cooperation between **p** and **o**. Therefore

$$\mathbf{p} \ \underline{\textit{must satisfy}}\ \mathbf{o} \not\Rightarrow \mathbf{p}|\tau^{\omega}\ \underline{\textit{must satisfy}}\ \mathbf{o}$$

If our view of the world is such that we wish to treat $\mathbf{p}|\tau^{\omega}$ as a valid implementation of **p**, how can we modify the system to permit this? We need to introduce some form of fairness assumption into the system. Apt and Olderog [Olderog 84] define four different notions of fairness; impartiality, justice, weak fairness and strong fairness. Their definitions assume a static language with a fixed number of processes so that there is no ambiguity about what is meant by component i, for example. We introduce the different notions of fairness in this framework, and then discuss what needs to be altered for a language with dynamic process creation such as CCS.

A computation is <u>impartial</u> if it is either finite or else every concurrent component in the system participates in an infinite number of communications. Extending this idea to CCS requires some care in the definition of what constitutes a component when processes may be created, and terminate, dynamically [Hennessy 84b]. However, this simple notion of fairness leads to many undesirable anomalies. For example, p|*NIL* has no infinite impartial computations as the second component cannot participate in any communications. The second notion of fairness, <u>justice</u>, attempts to remedy this deficiency by distinguishing between terminated and running components of a parallel program. Even this notion of fairness is not adequate for languages such as Static CCS because a component may not have terminated but may still be unable to proceed because of no matching requests. This leads to the definition of weak fairness where the concept of an enabled component is introduced. A component is enabled if it can potentially communicate with another process. Then a computation is <u>weakly fair</u> if it is either finite or else the following holds for each component: if for all but a finite number of steps component i is enabled then the component participates in an infinite number of communications. Thus a weakly fair computation of

$$(\alpha.NIL \mid (\text{fix X. } \overline{\alpha}.NIL + \beta.\text{X}))\backslash\alpha$$

is guaranteed to terminate whereas

$$(\alpha.NIL \mid (\text{fix X. } \overline{\alpha}.NIL + \beta.\gamma.\text{X}))\backslash\alpha$$

is not, since an infinite computation of the second example may have an infinite number of steps where the first component is not enabled.

Weak fairness guarantees that components which are continuously enabled are not indefinitely prevented from progressing. Such a condition may not be sufficient in mutual exclusion algorithms, for example, where a component is waiting to enter a critical section. In such a case there may be an infinite number of steps where the critical section is occupied by some other component, and therefore weak fairness will not be sufficient to guarantee eventual entry to the section. Such problems prompted the development of strong fairness. A computation is <u>strongly fair</u> if it is either finite or the following holds for each component i: if for infinitely many steps component i is enabled then this component participates in an infinite number of communications. Returning to our previous example,

$$(\alpha.NIL \mid (\text{fix } X.\ \bar{\alpha}.NIL + \beta.\gamma.X))\backslash\alpha$$

has no infinite strongly fair computations.

The strong form of fairness, while being a desirable attribute of a system, places some additional constraints on the implementation techniques that may be employed in addition to those required for weak fairness. For example, some forms of round robin schedulers are inapplicable if a strongly fair implementation is required. Chapter 2 has already shown how such schedulers do not provide an acceptable implementation of Static CCS even though they technically agree with the semantics of the language. Therefore the techniques required to implement a strongly fair system may be required to give an acceptable view of non-determinism anyway.

The application of these definitions to CCS presents special problems due to the dynamic nature of the language. In such a framework, the notion of concurrent component is inadequate because new processes may be created and old processes may terminate within the span of a computation. Costa and Stirling [Costa 84] develop techniques that deal with this problem and give a set of finite rules for generating all and only the admissible execution sequences when fairness is assumed.

A possible objection to the concurrent component view of fairness arises because the expansion theorem can no longer be used. To see why, consider the following two systems.

- $p = (\text{fix } X.\ \alpha.X) \mid (\text{fix } X.\ \beta.X) \qquad q = (\text{fix } X.\ \alpha.X + \beta.X)$

We would traditionally treat **q** to be equivalent to **p** as they are observably indistinguishable, which is why **q** can be derived from **p** by applying the expansion theorem. A strong or weak fair computation of **p** will contain an infinite number of α and β actions as it is composed of two separate concurrent components. However, an infinite α sequence is a valid fair computation of **q** as it consists of only one concurrent component, and the forms of fairness treated so far do not concern themselves with the choice operator. This leads us to conclude that the expansion theorem is no longer appropriate when dealing with these notions of fairness.

The stand we take in this thesis is to argue that it is not the expansion theorem that is at fault when reasoning about fairness, but rather the decision to consider the fairness of the | operator (| fairness) and to exclude considering the fairness of the + operator (+ fairness). We believe that <u>both</u> operators must have fair implementations to provide an acceptable system. This point has already been touched on in Chapter 2 where we argued for the necessity of random guard selection in an implementation. If the fairness of both operators is considered then the expansion theorem still holds, which is important as we consider it to aid considerably the understanding and usefulness of CCS. If we wish to take such a decision then definitions of fairness based on the concurrent component view of the world are no longer sufficient.

Parrow and Gustavsson [Parrow 84] consider a version of CCS where agents may be tagged with temporal logic expressions that filter out the unfair sequences of actions. Such an approach allows the implementor to specify exactly what fairness constraints are required by a particular algorithm. Furthermore, because these constraints are expressed at the level of sequences of actions, the distinction between | fairness and + fairness is not relevant. While this scheme is appropriate for particular algorithms, we would also like to be able to express the fairness properties guaranteed by a particular implementation. It would be an advantage if the fairness assumptions could be built into an existing equivalence so as to retain a familiar environment when reasoning about the equivalence of processes. The *testing* principle of DeNicola and Hennessy for the most part agrees very well with our intuitions of process equivalence, except for its handling of certain infinite computations where fairness assumptions play a part in influencing these intuitions.

One possibility would be to develop a weaker notion of *must satisfy* that ignores certain infinite computations. However, we must be careful to distinguish between $p|\tau^\omega$ and $p + \tau^\omega$. Both processes have the possibility of an infinite τ sequence but the first process always has the option of continuing normally. This gives us a hint as to how to define a new, *weak*, form of the *must satisfy* predicate. The original definition of *must satisfy* specified that every computation of p|o must be successful. If we only demand that at any point in a computation it is always possible to continue successfully, i.e. every finite prefix of a computation forms the initial part of a successful computation, then we have the basis of a weak form of *must satisfy*.

§4.4 The *weak–must* form of testing

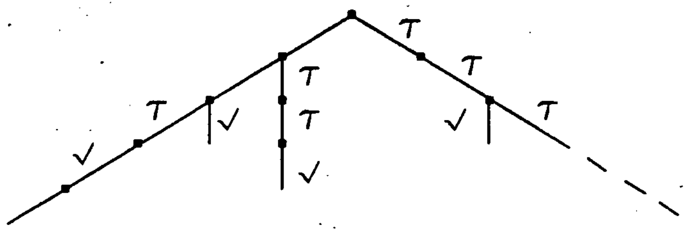
Chapter 1 defined the set of computations obtained from p|o as $\mathcal{C}omp(p,o)$. Let us extend this notation to represent the set of prefixes of $\mathcal{C}omp(p,o)$ by $\mathcal{PC}omp(p,o)$. We may then give an alternative definition of the *must satisfy* predicate as follows.

Definition $\forall p \in \mathcal{P},\ o \in \mathcal{O}.$

$$p\ \underline{must\ satisfy}\ o \iff \forall pc \in \mathcal{PC}omp(p,o).\ \exists c \in \mathcal{C}omp(p,o) \text{ s.t. } c \in \mathcal{S}uccess \wedge pc < c$$

We use the notation $pc<pc'$ to indicate that pc is a prefix of pc'. To see that this is equivalent to the original definition of *must satisfy* we only have to note that $\mathcal{C}omp(p,o) \subseteq \mathcal{PC}omp(p,o)$.

If p *must satisfy* o then every path through the derivation tree of p|o, including the infinite ones, must pass through a node where a $\sqrt{}$ move is possible as is illustrated below.



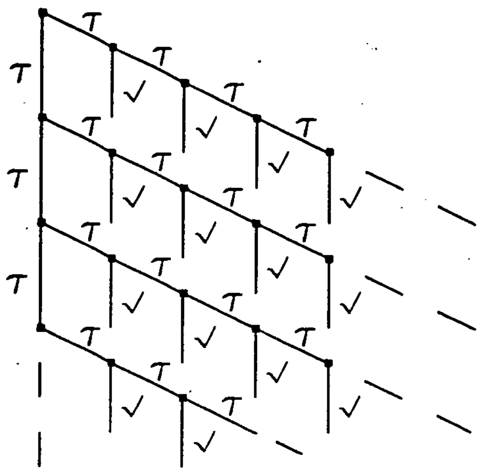
We now alter the definition subtly to produce what we shall call the *weak* form of _must satisfy_.

Definition $\forall \mathbf{p} \in \mathcal{P}, \mathbf{o} \in \mathcal{O}$.

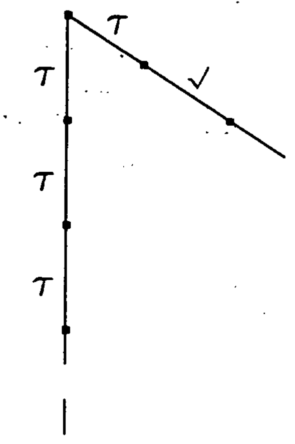
p _w-must satisfy_ o $\Leftrightarrow \forall$ _finite_ $pc \in \mathcal{PC}omp(\mathbf{p},\mathbf{o})$.
$\exists c \in \mathcal{C}omp(\mathbf{p},\mathbf{o})$ s.t. $c \in \mathcal{S}uccess \wedge pc < c$

The only change we have made is the addition of a constraint that the only prefixes we are interested in are finite. We can view the statement that p _w-must satisfy_ o as an assertion that at any point in the derivation tree of p|o it is possible to pick a path down the tree that passes through a successful state.

Consider the derivation trees of $(\alpha|\tau^{\omega})$ and $\alpha + \tau^{\omega}$ when observed by $\bar{\alpha}.\surd.NIL$



$(\alpha \mid \tau^{\omega} \mid \bar{\alpha}\checkmark) \upharpoonright \checkmark$



$((\alpha + \tau^{\omega}) \mid \bar{\alpha}\checkmark) \upharpoonright \checkmark$

In the first case, wherever we get to in the derivation tree it is always possible to find a continuation of the path that forms part of a successful computation. Therefore

$$\alpha|\tau^{\omega} \ \underline{w\text{-}must\ satisfy}\ \bar{\alpha}.\checkmark.NIL$$

However there is an infinite computation that is not successful. This corresponds to always taking the leftmost branch in our tree. Therefore

$$\alpha|\tau^{\omega} \ \underline{must\!\!/\ satisfy}\ \bar{\alpha}.\checkmark.NIL$$

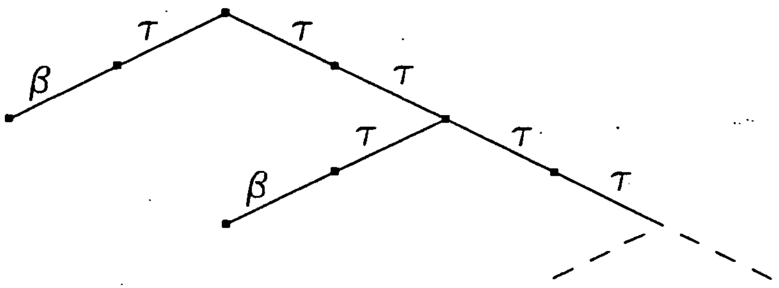
In our second example, there is also an infinite τ computation. However this computation is harmful in that once we have started down this path all other possibilities are lost. Therefore

$$\alpha + \tau^{\omega} \ \underline{w\text{-}must\!\!/\ satisfy}\ \bar{\alpha}.\checkmark.NIL$$

The first example illustrates why our intuitions are not always in agreement with the original definition of <u>*must satisfy*</u>. When we look at the derivation tree of $\alpha|\tau^{\omega}|\bar{\alpha}.\checkmark.NIL$, we might argue that if we ran the definition on any 'reasonable' implementation of CCS, one of the τ branches leading to a $\checkmark$ possibility would eventually be taken. However, CCS has no such fairness constraints built into its definition and so one might counter this argument by exhibiting an extremely malicious scheduler that carefully picked a path through the derivation tree so as to avoid reaching a state that may perform a $\checkmark$ move. In our particular example this would correspond to taking the leftmost path of the tree.

One obvious question we might ask at this point concerns the relationship between the new form of testing and the notions of fairness previously discussed. Consider the process

$$p = (\alpha.\beta.NIL \mid (\text{rec } X.\ \bar{\alpha}.NIL + \tau.\tau.X))\backslash\alpha$$



Then under strong fairness assumptions the first process will eventually perform the α action and hence a β action will eventually be offered to the environment. Note that no such guarantee could be made only assuming weak fairness of the system. Consider now what would happen if we observed the system with the observer $o = \bar{\beta}.\surd.NIL$. Then

p <u>*w-must satisfy*</u> o

because at any point in a computation of p|o it is always possible to extend the prefix to a successful state. Thus the new notion of weak testing captures the flavour of strong fairness. Because it is defined in terms of derivation trees, the new definition makes no distinction between | fairness and + fairness and so the expansion theorem is still applicable.

Proposition 4.1 If all the elements of $\mathscr{C}omp$(p,o) are finite then

p <u>*must satisfy*</u> o $\Leftrightarrow$ p <u>*w-must satisfy*</u> o

Proof:

Trivial, as all elements of $\mathscr{PC}omp$(p,o) are finite in this case. □

Proposition 4.2 p <u>*w-must satisfy*</u> o $\supset$ p <u>*may satisfy*</u> o

Proof:

If p <u>*w-must satisfy*</u> o then there must exist at least one successful computation in $\mathscr{C}omp$(p,o) and hence p <u>*may satisfy*</u> o. □

We define the weak equivalents of $\sqsubseteq_2$ and $\simeq_2$ as follows.

Definition

$$p \sqsubseteq_w q \iff \forall o \in \mathcal{O}.\ p\ \underline{\textit{w-must satisfy}}\ o \ \supset\ q\ \underline{\textit{w-must satisfy}}\ o$$

$$p \simeq_w q \iff p \sqsubseteq_w q \wedge q \sqsubseteq_w p$$

Our motivation for introducing the weak form of *must satisfy* was to allow an implementor some freedom of choice. Thus we could have defined an asymmetric version of $\sqsubseteq_w$ where we assumed that the specification was not divergent, for example. We might define an alternative version of $\sqsubseteq_w$, $\sqsubseteq_w'$, as follows.

$$p \sqsubseteq_w' q \iff \forall o \in \mathcal{O}.\ p\ \underline{\textit{must satisfy}}\ o \ \supset\ q\ \underline{\textit{w-must satisfy}}\ o$$

However, such a definition does not lead to a transitive relation. For example,

$$p \sqsubseteq_w' p|\tau^\omega \sqsubseteq_w' \tau^\omega$$

but

$$p \not\sqsubseteq_w' \tau^\omega$$

Therefore this possibility was rejected.

§4.5 Some properties of $\sqsubseteq_w$ and $\simeq_w$

We will be primarily interested in properties of the preorder $\sqsubseteq_w$ as this is what we will use in the final definition of implementation. However, for completeness, the $\simeq_w$ equivalence is also investigated.

One of the most important properties of $\sqsubseteq_w$ is its ability to be preserved by most of the CCS operators. Before showing this, two auxiliary lemmas are first proved.

Lemma 4.3

If p *w-must satisfy* o and

$$p|o = p_0|o_0 \xrightarrow{\tau} p_1|o_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} p_n|o_n$$

for some n s.t. $\nexists m, 0 \leq m < n.\ o_m \xrightarrow{\surd}$ then p_n *w-must satisfy* o_n

Proof:

Suppose false. Then there exists a prefix π of a computation c from $p_n|o_n$ which cannot be extended to a successful computation. But the computation obtained by prefixing $<p_0|o_0>, <p_1|o_1>, \ldots, <p_{n-1}|o_{n-1}>$ to c is a computation of p|o and so $<p_0|o_0>, <p_1|o_1>, \ldots, <p_{n-1}|o_{n-1}>$ prefixed to π can therefore be extended to a successful computation. The only way that p *w-must satisfy* o can be true is if for some $m<n.\ o_m \xrightarrow{\surd}$. But this is impossible due to our assumptions. □

Lemma 4.4

If $\alpha.p$ *w-must satisfy* o then $\exists n$ s.t.
<u>either</u> $o \xrightarrow{\tau^n} o' \xrightarrow{\surd}$ for some o'
<u>or</u> $o \xrightarrow{\tau^n} o' \xrightarrow{\bar{\alpha}}$ for some o'

Proof:

Follows from the definition of a successful computation. □

Theorem 4.5 If $p \sqsubseteq_w q$ then $\forall \mu \in \mathcal{A}ct \cup \{\tau\}$, $\forall \lambda \in \mathcal{A}ct$, $\forall r \in \mathcal{P}$ and relabeling S,

1. $\mu.p \sqsubseteq_w \mu.q$
2. $p|r \sqsubseteq_w q|r$
3. $p\backslash\lambda \sqsubseteq_w q\backslash\lambda$
4. $p[S] \sqsubseteq_w q[S]$

Proof:

1. Assume $\mu.p$ *w-must satisfy* o.
   Take a computation of $\mu.q|o$, and any finite prefix of the computation

   $$\mu.q|o = q_0|o_0 \xrightarrow{\tau} q_1|o_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n|o_n$$

   If there exists an $i \leq n$ such that $o_i \xrightarrow{\surd}$ then the computation is successful so we will assume that there is no such i. If there exists an $i \leq n$ such that $q_i = q$ then we have $\mu.q|o \xrightarrow{\tau^i} q|o_i$ and so $\mu.p|o \xrightarrow{\tau^i} p|o_i$. But p *w-must satisfy* o_t and so q *w-must satisfy* o_t which implies that the prefix can be extended successfully. Otherwise $\mu.p|o \xrightarrow{\tau^n} p|o_n$ and by Lemma 4.3, $\mu.p$ *w-must satisfy* o_n. Then by Lemma 4.4, either $o_n \xrightarrow{\tau^m} o' \xrightarrow{\surd}$ in which case $q_n|o_n \xrightarrow{\tau^m} q_n|o' \xrightarrow{\surd}$, or $o_n \xrightarrow{\tau^m} o' \xrightarrow{\bar{\mu}} o''$ so $\mu.p|o_n \xrightarrow{\tau^{m+1}} p|o''$ where p *w-must satisfy* o''. But then $q_n|o_n \xrightarrow{\tau^{m+1}} q|o''$ and q *w-must satisfy* o'' so again the prefix can be extended successfully.

2. Assume p|r *w-must satisfy* o.
   Then p *w-must satisfy* r|o which implies that q *w-must satisfy* r|o and hence q|r *w-must satisfy* o.

3. $p\backslash\lambda$ *w-must satisfy* o $\Leftrightarrow$ p *w-must satisfy* $o\backslash\bar{\lambda}$ and so the result follows trivially.

4. Assume p[S] *w-must satisfy* o.
   We first extend S to deal with the label $\surd$, i.e. $S(\surd)=\surd$.
   We then define a complement renaming $\bar{S}$ by $\bar{S}(\lambda) = \bar{\lambda}$ if $S(\bar{\lambda}) = \lambda$.
   Then p[S] *w-must satisfy* o $\Leftrightarrow$ p *w-must satisfy* $o[\bar{S}]$ and so the result follows trivially.

□

The behaviour of $\sqsubseteq_w$ under the fixpoint operator remains an open question. Milner [Milner 83] shows that his equivalence $\approx$ is preserved by **fix** using a bisimulation. Hennessy and DeNicola [DeNicola 82] do not prove that their equivalence $\simeq_2$ is preserved by **fix** directly, but rely on their induction results to deduce this fact. Neither of these approaches are open to us for $\sqsubseteq_w$ but fortunately we do not require the preservation of $\sqsubseteq_w$ by **fix** for the work in this thesis. We therefore postpone this investigation for the time being.

Unfortunately, $\sqsubseteq_w$ is not preserved by + as this simple example illustrates.

$\alpha.NIL \sqsubseteq_w \tau.\alpha.NIL$ but $\lambda.NIL + \alpha.NIL \not\sqsubseteq_w \lambda.NIL + \tau.\alpha.NIL$

because if $o = \bar{\lambda}.\surd.NIL$ then

$\lambda.NIL + \alpha.NIL$ <u>*w-must satisfy*</u> $\bar{\lambda}.\surd.NIL$

whereas

$\lambda.NIL + \tau.\alpha.NIL$ <u>*w-must/satisfy*</u> $\bar{\lambda}.\surd.NIL$

as we cannot prevent the τ branch from being taken leading to an unsuccessful computation. This result is not too surprising as $\sqsubseteq_2$ is not preserved by + either. Furthermore, we have the following theorem which is applicable in many cases.

Theorem 4.6 If $p \sqsubseteq_w q$ then $\forall \mu \in \mathcal{A}ct \cup \{\tau\}$, $\forall r \in \mathcal{P}$. $\mu.p + r \sqsubseteq_w \mu.q + r$

Proof:

Suppose this is false. Then there is a computation of $(\mu.q + r)|o$ such that a prefix of it cannot be successfully extended, i.e.

$$(\mu.q + r)|o = q_0|o_0 \xrightarrow{\tau} q_1|o_1 \xrightarrow{\tau} \ \ldots \ \xrightarrow{\tau} q_n|o_n \overset{\surd}{\not\Longrightarrow}$$

and $\forall j<n.\ o_j \overset{\surd}{\not\longrightarrow}$. Let q_i be the first point in the sequence where $\mu.q + r$ participates in the computation either by moving silently by itself or by synchronising with the observer. If no such i exists in the prefix we can always extend the prefix until it does as otherwise this would imply that the observer must be able to reach a successful state by itself which would cause a contradiction.

Suppose the move at q_i is due to r, i.e. $r \xrightarrow{\nu_i} r' = q_{i+1}$.
Then $(\mu.p + r)|o \xrightarrow{\tau} p_i|o_i \xrightarrow{\tau} r'|o_{i+1}$ and so r' <u>*w-must satisfy*</u> o_{i+1} which leads to a contradiction.

Suppose instead that the μ move takes place.
Then we have $(\mu.q + r)|o_i \xrightarrow{\tau} q|o_{i+1}$. But then $\mu.p|o_i \xrightarrow{\tau} p|o_{i+1}$ and p <u>*w-must satisfy*</u> o_{i+1} so q <u>*w-must satisfy*</u> o_{i+1} leading to a contradiction. □

Although we have shown that $\sqsubseteq_w \neq \sqsubseteq_2$, this does not necessarily imply that $\simeq_2 \neq \simeq_w$. If we only consider finite processes, then all computations are finite, and so trivially $\simeq_2 = \simeq_w$. However, in the more general case, $\simeq_2$ and $\simeq_w$ differ in subtle but significant ways as the following examples illustrate.

Let $p_1 = \alpha.NIL|\tau^\omega \quad q_1 = (\alpha.NIL|\tau^\omega) + \tau.NIL$

$p_2 = \alpha.NIL \quad q_2 = \alpha.NIL|\tau^\omega$

Proposition 4.7

1. $p_1 \simeq_2 q_1$ but $p_1 \not\simeq_w q_1$

2. $p_2 \not\simeq_2 q_2$ but $p_2 \simeq_w q_2$

Proof:

1. We first show that $p_1 \simeq_2 q_1$.
Let $F=\tau.F$, $S_1(x) = \alpha.NIL|x$ and $S_2(x) = (\alpha.NIL|x) + \tau.NIL$. We must show that $S_1(F) \simeq_2 S_2(F)$ and we prove this by Scott Induction whose use is justified for $\simeq_2$ in [DeNicola 82].

<u>Inductive base</u>, $S_1(\Omega) \simeq_2 S_2(\Omega)$

$\alpha.NIL|\Omega \simeq_2 \alpha.\Omega + \Omega$

by the expansion theorem

$\sqsubseteq_2 \alpha.\Omega + \Omega + \tau.NIL$

as $\Omega \sqsubseteq X$

$\simeq_2 (\alpha.NIL|\Omega) + \tau.NIL$

by the expansion theorem

i.e. $S_1(\Omega) \sqsubseteq_2 S_2(\Omega)$

$\sqsubseteq_2 \alpha.\Omega + \tau(\Omega+NIL)$

by the expansion theorem

and $X + \tau.Y \sqsubseteq \tau.(X + Y)$

$\simeq_2 \alpha.\Omega + \tau.\Omega$

as $X + NIL = X$

$\sqsubseteq_2 \tau(\alpha.\Omega + \Omega)$

as $X + \tau.Y \sqsubseteq \tau.(X + Y)$

$\sqsubseteq_2 \alpha.\Omega + \Omega$

as $\tau.X \sqsubseteq X$

$\simeq_2 \alpha.NIL|\Omega$

i.e. $S_2(\Omega) \sqsubseteq_2 S_1(\Omega)$

Hence $S_1(\Omega) \simeq_2 S_2(\Omega)$.

<u>Induction step</u> Assume that $S_1(F) \simeq_2 S_2(F)$ and show that $S_1(\tau.F) \simeq_2 S_2(\tau.F)$

$\alpha.NIL|\tau.F \simeq_2 \alpha.F + \tau.(\alpha.NIL|F)$

by the expansion theorem

$\sqsubseteq_2 \alpha.F + \tau.(\alpha.NIL|F + \tau.NIL)$

by the inductive hypothesis

$\simeq_2 \alpha.F + \tau(\tau(\alpha.NIL|F) + \tau.NIL)$

as $X + \tau.Y = \tau.(X + Y) + \tau.Y$

and $X + NIL = X$

$\sqsubseteq_2 \tau(\alpha.F + \tau(\alpha.NIL|F) + \tau.NIL)$

as $X + \tau.Y \sqsubseteq \tau.(X + Y)$

$\sqsubseteq_2 \alpha.F + \tau(\alpha.NIL|F) + \tau.NIL$

as $\tau.X \sqsubseteq X$

$\simeq_2 \alpha.NIL|\tau.F + \tau.NIL$

i.e. $S_1(\tau.F) \sqsubseteq_2 S_2(\tau.F)$

$\simeq_2 \alpha.F + \tau(\tau(\alpha.NIL|F) + \tau.NIL)$

by the expansion theorem

and $\mu.X + \mu.Y = \mu.(\tau.X + \tau.Y)$

$\simeq_2 \alpha.F + \tau(\alpha.NIL|F) + \tau.NIL)$

as $X + NIL = X$

and $X + \tau.Y = \tau.(X + Y) + \tau.Y$

$\sqsubseteq_2 \alpha.F + \tau(\alpha.NIL|F)$

by the inductive hypothesis

$= \alpha.NIL|\tau.F$

i.e. $S_2(\tau.F) \sqsubseteq_2 S_1(\tau.F)$

Hence $S_1(\tau.F) \simeq_2 S_2(\tau.F)$ and so $p_1 \simeq_2 q_1$.

To see that $p_1 \not\simeq_w q_1$, consider the test $\bar{\alpha}.\surd.NIL$
Then

p_1 <u>*w-must satisfy*</u> $\bar{\alpha}.\surd.NIL$

whereas

q_1 <u>*w-must/satisfy*</u> $\bar{\alpha}.\surd.NIL$

2. p_2 <u>*must satisfy*</u> $\bar{\alpha}.\surd.NIL$ whereas q_2 <u>*must/satisfy*</u> $\bar{\alpha}.\surd.NIL$. Therefore $q_2 \not\simeq_2 q_2$.

   Suppose p <u>*w-must satisfy*</u> o but $p|\tau^\omega$ <u>*w-must/satisfy*</u> o for some process p and observer o. Then there is a prefix of a computation of $p|\tau^\omega|o$ that cannot be successfully extended.

   $$p|\tau^\omega|o \xRightarrow{\varepsilon} p'|\tau^\omega|o' \overset{\surd}{\not\Longrightarrow}.$$

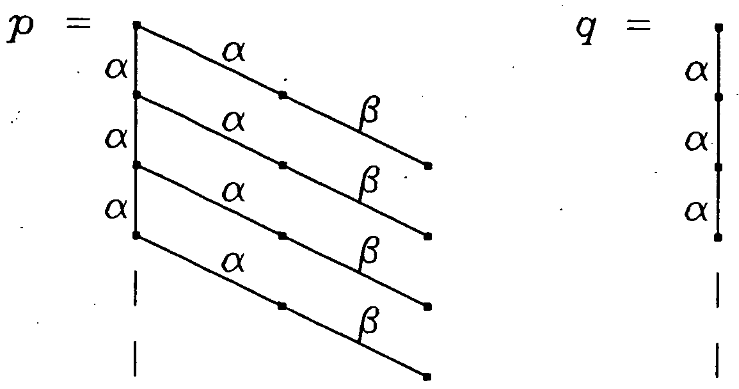
   where o has passed through no successful states enroute to o'.

   Then $p|o \xRightarrow{\varepsilon} p'|o'$ and as p <u>*w-must satisfy*</u> o, $p'|o' \xRightarrow{\surd}$ and so $p'|\tau^\omega|o' \xRightarrow{\surd}$ which is a contradiction. Hence $p \sqsubseteq_w p|\tau^\omega$.

   Suppose $p|\tau^\omega$ <u>*w-must satisfy*</u> o but p <u>*w-must/satisfy*</u> o. Then $p|o \xRightarrow{\varepsilon} p'|o' \overset{\surd}{\not\Longrightarrow}$ for some p', o'.
   But $p|\tau^\omega|o \xRightarrow{\varepsilon} p'|\tau^\omega|o' \xRightarrow{\surd}$ and therefore p' and o' must be able to communicate in such a way that eventually the observer may perform a $\surd$ action. Therefore $p'|o' \xRightarrow{\surd}$ which leads to a contradiction.
   Hence $p|\tau^\omega \sqsubseteq_w p$ and so $p \simeq_w p|\tau^\omega$. □

This last proof illustrates another important difference between $\simeq_2$ and $\simeq_w$ (and also between $\sqsubseteq_2$ and $\sqsubseteq_w$). To prove that $p \sqsubseteq_w q$ is difficult in general because we have to work with tests. Although $\sqsubseteq_2$ is also defined in terms of tests there exists an alternative characterisation of $\sqsubseteq_2$ that avoids the use of these tests. Furthermore, DeNicola and Hennessy [DeNicola 82] have shown that for $\sqsubseteq_2$ it is only necessary to consider <u>finite</u> tests. If $p \not\sqsubseteq_2 q$ then there will always be a finite observer that can distinguish between them. This is not true for $\sqsubseteq_w$ as the following example shows.

Let $p = \alpha.\beta.NIL + \alpha.p$ and $q = \alpha.q$
i.e.



Then we can construct an infinite observer $\mathbf{o} = \bar{\alpha}.(\bar{\beta}.\surd.NIL + \mathbf{o})$ that can differentiate between them, i.e.

p <u>*w-must satisfy*</u> o but q <u>*w-must/satisfy*</u> o.

However, there is no finite observer that can differentiate between these processes.

We would like an alternative characterisation of $\sqsubseteq_w$ that avoids the use of tests. It would also be highly desirable if it allowed us to perform bisimulation style proofs. At the very least we would like an alternative, simpler preorder that treats divergent terms in a similar way to $\sqsubseteq_w$ and also implies $\sqsubseteq_w$. In [Kennaway 81], Kennaway develops an equivalence that is very similar to the 'weak-must' equivalence in that it treats some of the diverging terms we are concerned about in a similar way. Furthermore, Kennaway's equivalence can be expressed in a form that is amenable to proofs using the bisimulation technique. It would therefore seem prudent to investigate the relationship between $\sqsubseteq_w$ and the preorder version of Kennaway's equivalence.

§4.6 Kennaway's preorder $\sqsubseteq_k$

The version of Kennaway's preorder we shall use is in fact based on the definition given in [DeNicola 82] by DeNicola and Hennessy. It differs from the original in a number of subtle but important ways. Appendix A describes the original version of Kennaway's equivalence and shows why that version is undesirable because it is not an observational equivalence.

We start with some definitions that will be used frequently in the rest of this chapter.

Definition

We use p, q to range over $\mathcal{P}$ and P, Q to range over subsets of $\mathcal{P}$. Let

$$\text{Init}(p) = \{\alpha \in \mathcal{A}ct \mid p \overset{\alpha}{\Longrightarrow}\}$$
$$\text{Traces}(p) = \{s \in \mathcal{A}ct^* \mid p \overset{s}{\Longrightarrow}\}$$

For any sequence $s \in \mathcal{A}ct^*$, p <u>after</u> s and P <u>after</u> s are defined by

$$p \ \underline{\text{after}}\ \varepsilon = p$$
$$p \ \underline{\text{after}}\ \alpha = \{p' \mid p \overset{\alpha}{\Longrightarrow} p'\}$$
$$p \ \underline{\text{after}}\ \alpha.s = (p \ \underline{\text{after}}\ \alpha)\ \underline{\text{after}}\ s$$

$$P \ \underline{\text{after}}\ s = \bigcup \{p \ \underline{\text{after}}\ s \mid p \in P\}$$

For any set $L \subseteq \mathcal{A}ct$, p <u>must</u> L and P <u>must</u> L are defined by

$$p \ \underline{\text{must}}\ L \Leftrightarrow \forall p' \text{ s.t. } p \overset{\varepsilon}{\Longrightarrow} p'.\ \exists \lambda \in L.\ p' \overset{\lambda}{\Longrightarrow}$$
$$P \ \underline{\text{must}}\ L \Leftrightarrow \forall p \in P.\ p \ \underline{\text{must}}\ L$$

Hennessy and DeNicola presented their version of Kennaway's equivalence directly. However, as we are trying to find an alternative characterisation of $\sqsubseteq_w$, it is necessary to present it in the form of a preorder.

We define a set of approximations to the desired relation. The preorder is then obtained by taking the limit of this series.

$P \sqsubseteq_k^0 Q$ is always true.

$P \sqsubseteq_k^{n+1} Q \Leftrightarrow$ i) $\forall$finite $L \subseteq \mathcal{A}ct$. P <u>must</u> L $\supset$ Q <u>must</u> L

ii) $\forall \lambda \in \mathcal{A}ct$. P <u>after</u> $\lambda \sqsubseteq_k^n$ Q <u>after</u> λ

$$P \sqsubseteq_k Q \Leftrightarrow \forall n \geq 0.\ P \sqsubseteq_k^n Q.$$

We extend this definition to single processes and the equivalence in the obvious way.

$$\begin{array}{lcl} p \sqsubseteq_k q & \Leftrightarrow & \{p\} \sqsubseteq_k \{q\} \\ P \simeq_k Q & \Leftrightarrow & P \sqsubseteq_k Q \ \wedge \ Q \sqsubseteq_k P \\ p \simeq_k q & \Leftrightarrow & p \sqsubseteq_k q \ \wedge q \sqsubseteq_k p \end{array}$$

We can give an alternative characterisation of $\sqsubseteq_k$ which does not involve a recurrence. This will be useful when reasoning about the preorder. The proof is based on an equivalent proof for $\simeq_k$ presented in [DeNicola 82].

Theorem 4.8 $P \sqsubseteq_k Q \Leftrightarrow \ \forall s \in \mathcal{Act}^*, \forall \text{finite } L \subseteq \mathcal{Act}.$

$(P \ \underline{\text{after}} \ s) \ \underline{\text{must}} \ L \supset (Q \ \underline{\text{after}} \ s) \ \underline{\text{must}} \ L$

Proof:

1. $(\Leftarrow)$

We prove that $P \not\sqsubseteq_k Q$ implies $\exists s \in \mathcal{Act}^*, L \subseteq \mathcal{Act}$ s.t. $(P \ \underline{\text{after}} \ s) \ \underline{\text{must}} \ L$ and $(Q \ \underline{\text{after}} \ s) \ \underline{\text{m\cancel{u}st}} \ L$

If $P \not\sqsubseteq_k Q$ then $\exists n$ s.t. $P \not\sqsubseteq_k^n Q$. We use induction on n.

<u>Inductive basis</u>, n = 1

$P \not\sqsubseteq_k^1 Q$ implies $\exists L$ s.t. $P \ \underline{\text{must}} \ L$ and $Q \ \underline{\text{m\cancel{u}st}} \ L$.

Therefore $(P \ \underline{\text{after}} \ \varepsilon) \ \underline{\text{must}} \ L$ and $(Q \ \underline{\text{after}} \ \varepsilon) \ \underline{\text{m\cancel{u}st}} \ L$.

<u>Inductive step</u>

$P \not\sqsubseteq_k^{n+1} Q$ iff either i) $P \not\sqsubseteq_k^1 Q$

or ii) $\exists \lambda \in \mathcal{Act}$ s.t. $P \ \underline{\text{after}} \ \lambda \not\sqsubseteq_k^n Q \ \underline{\text{after}} \ \lambda$.

In case i) the claim follows from the base case.

In case ii) the inductive hypothesis states that for some $s \in \mathcal{Act}^*$, $L \subseteq \mathcal{Act}$,

$((P \ \underline{\text{after}} \ \lambda) \ \underline{\text{after}} \ s) \ \underline{\text{must}} \ L$ and

$((Q \ \underline{\text{after}} \ \lambda) \ \underline{\text{after}} \ s) \ \underline{\text{m\cancel{u}st}} \ L$

But then we may deduce

$(P \ \underline{\text{after}} \ \lambda.s) \ \underline{\text{must}} \ L$ and $(Q \ \underline{\text{after}} \ \lambda.s) \ \underline{\text{m\cancel{u}st}} \ L$.

2. $(\Rightarrow)$

Suppose $\exists s \in \mathcal{Act}^*$ and finite $L \subseteq \mathcal{Act}$ such that

(P <u>after</u> s) <u>must</u> L and (Q <u>after</u> s) <u>m~~u~~st</u> L.

We show that $P \not\sqsubseteq_k Q$ by induction on s.

<u>Inductive basis</u>, $s = \varepsilon$

P <u>after</u> ε = P so $P \not\sqsubseteq_k^1 Q$, i.e. $P \not\sqsubseteq_k Q$

<u>Inductive step</u>, $s = \alpha s'$

Then ((P <u>after</u> α) <u>after</u> s') <u>must</u> L whereas

((Q <u>after</u> α) <u>after</u> s') <u>m~~u~~st</u> L

By induction P <u>after</u> $\alpha \not\sqsubseteq_k$ Q <u>after</u> α and so $P \not\sqsubseteq_k Q$

□

Corollary 4.9 For all finitely expressible agents **p**, **q**

$\mathbf{p} \sqsubseteq_k \mathbf{q} \supset \text{Traces}(\mathbf{q}) \subseteq \text{Traces}(\mathbf{p})$

Proof:

Suppose $\exists s$ s.t. $s \in \text{Traces}(\mathbf{q})$ and $s \notin \text{Traces}(\mathbf{p})$. Let α be such that

$$\mathbf{q} \overset{s\alpha}{\not\Longrightarrow}.$$

α exists because **q** is finitely expressible and hence has a finite sort. Then (**q** <u>after</u> s) <u>m~~u~~st</u> $\{\alpha\}$ whereas trivially (**p** <u>after</u> s) <u>must</u> $\{\alpha\}$. Therefore $\mathbf{p} \not\sqsubseteq_k \mathbf{q}$. □

We can prove a similar result for $\sqsubseteq_w$, i.e.

Proposition 4.10 $\mathbf{p} \sqsubseteq_w \mathbf{q} \supset \text{Traces}(\mathbf{q}) \subseteq \text{Traces}(\mathbf{p})$

Proof:

Suppose $\exists s$ s.t. $s \in \text{Traces}(\mathbf{q})$ and $s \notin \text{Traces}(\mathbf{p})$. If s denotes $\alpha_1\alpha_2 \ldots \alpha_n$, then we construct an observer **o** as follows,

$$\mathbf{o} = \tau.\checkmark + \bar{\alpha}_1(\tau.\checkmark + (\ldots \tau.\checkmark + \bar{\alpha}_n) \ldots)$$

Then **p** *<u>w-must satisfy</u>* **o** because at any point in a computation from **p**|**o**, the observer can get to a position where it may perform a $\checkmark$ move by itself. However **q** *<u>w-must/ satisfy</u>* **o** because

$$q|o \xrightarrow{\tau} \ . \ . \ . \ \xrightarrow{\tau} q'|NIL$$

which cannot be extended successfully. Therefore $p \not\sqsubseteq_w q$. □

These two results are useful because they allow us to deduce $q \sqsubseteq_3 p$ from $p \sqsubseteq_w q$ or $p \sqsubseteq_k q$ because Hennessy has shown in [DeNicola 82] that

$$Traces(p) \subseteq Traces(q) \Leftrightarrow p \sqsubseteq_3 q$$

What is the relationship between $\sqsubseteq_w$ and $\sqsubseteq_k$? The following theorem answers part of this question.

Theorem 4.11 $p \sqsubseteq_w q \ \supset \ p \sqsubseteq_k q$

Proof:

We prove that $p \not\sqsubseteq_k q$ implies $p \not\sqsubseteq_w q$.

If $p \not\sqsubseteq_k q$ then $\exists s,L$ s.t.

(p <u>after</u> s) <u>must</u> L and (q <u>after</u> s) <u>m~~u~~st</u> L,

i.e. $p \overset{s}{\Longrightarrow} p'$ implies $\exists \lambda \in L$ s.t. $p' \overset{\lambda}{\Longrightarrow}$ while

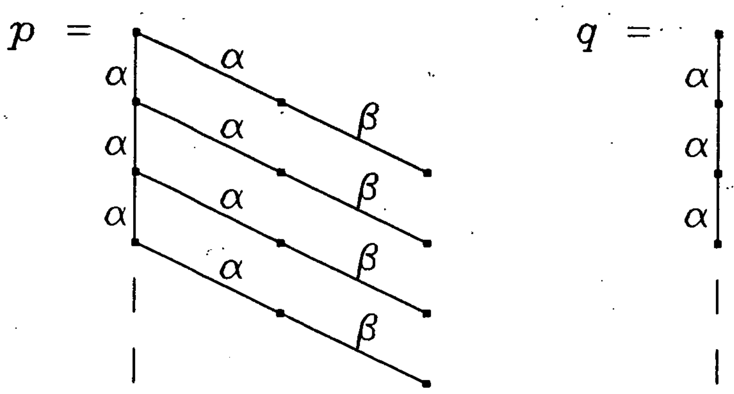
$\exists q'$, $q \overset{s}{\Longrightarrow} q'$ and $q' \overset{\lambda}{\Longrightarrow}$ for no $\lambda \in L$.

If $s = \alpha_1 \alpha_2 \ . \ . \ . \ \alpha_n$ then we define an observer o as follows

$$o = \tau.\surd + \bar{\alpha}_1.(\tau.\surd + \bar{\alpha}_2(\ . \ . \ . \ \bar{\alpha}_{n-1}(\tau.\surd + \bar{\alpha}_n(\sum_{a \in L} \bar{a}.\surd) \ . \ . \ . \ )$$

Then p <u>*w-must satisfy*</u> o whereas q <u>*w-must/satisfy*</u> o and so $p \not\sqsubseteq_w q$. □

Unfortunately, $p \sqsubseteq_k q \not\supset p \sqsubseteq_w q$ as the following example shows. Consider the processes $p = \alpha.\beta.NIL + \alpha.p$ and $q = \alpha.q$.



Then **p** $\sqsubseteq_k$ **q** because (**p** <u>after</u> s) <u>must</u> L $\supset$ (**q** <u>after</u> s) <u>must</u> L $\forall$s,L.
To see this we perform a case analysis on s.

i) s = α^n, so (**p** <u>after</u> s) = {**p**, $\beta.NIL$}.
If (**p** <u>after</u> s) <u>must</u> L then $\{\alpha, \beta\} \subseteq$ L.
But **q** <u>after</u> s = **q** and **q** <u>must</u> $\{\alpha, \beta\}$.

ii) s = $\alpha^n\beta$ so **p** <u>after</u> s = {NIL} which <u>must</u> L for no L.

iii) s = something else, in which case **p** <u>after</u> s = {},
q <u>after</u> s = {} and both of these <u>must</u> L for any L.

However the observer **o** = $\bar{\alpha}$.**o** + $\bar{\beta}.\surd$ can distinguish between **p** and **q** as **p** <u>*w-must satisfy*</u> **o** but **q** <u>*w-must/satisfy*</u> **o**, i.e. **p**$\not\sqsubseteq_w$**q**.

We can trivially show that **p** $\simeq_w$ **q** $\supset$ **p** $\simeq_k$ **q** but again the converse is not true, i.e. **p** $\simeq_k$ **q** $\not\supset$ **p** $\simeq_w$ **q**. As an example of why this is not the case, consider **p** and **o** as defined previously and **q**' = **q** + **p**. Then **p** $\simeq_k$ **q**, i.e.
(**p** <u>after</u> s) <u>must</u> L $\Leftrightarrow$ (**q**' <u>after</u> s) <u>must</u> L $\forall$s,L.
To see this we again perform a case analysis on s.

i) s = α^n.
Then (**p** <u>after</u> s) = {**p**, $\beta.NIL$}
and (**q**' <u>after</u> s) = {**q**, **p**, $\beta.NIL$}
Both sets <u>must</u> L only for any L where $\{\alpha,\beta\} \subseteq$L.

ii) $s = \alpha^n\beta$

Then (p <u>after</u> s) $=$ $\{NIL\}$ $=$ (q' <u>after</u> s)

iii) s = something else.

Then (p <u>after</u> s) $=$ $\{\}$ $=$ (q' <u>after</u> s)

However p <u>*w-must satisfy*</u> o but q' <u>*w-must/satisfy*</u> o so p $\not\leq_w$ q' and hence p $\not\simeq_w$ q'.

This is an example where $\simeq_k$ and $\simeq_2$ agree as we can show that p $\simeq_2$ q'. Let $S_1(p,q) = p$ and $S_2(p,q) = p + q$. Then we will show that $S_1(p,q) \simeq_2 S_2(p,q)$

<u>Inductive basis</u>, $S_1(\Omega,\Omega) \simeq_2 S_2(\Omega,\Omega)$ trivially.

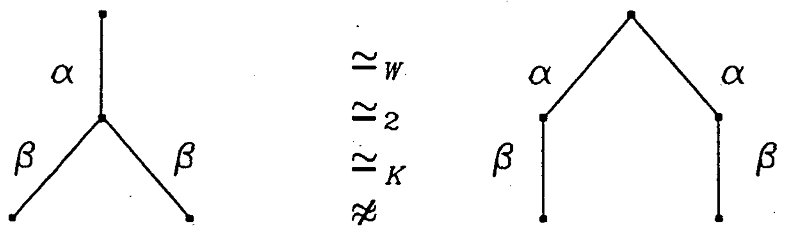
<u>Inductive step</u>,

Assuming $S_1(p,q) \simeq_2 S_2(p,q)$,

show $S_1(\alpha.p + \alpha.\beta,\ \alpha.q) \simeq_2 S_2(\alpha.p + \alpha.\beta,\ \alpha.q)$.

$$\begin{aligned}
S_1(\alpha.p + \alpha.\beta,\ \alpha.q) &\simeq_2 \alpha.p + \alpha.\beta \\
&\simeq_2 \alpha.(p + q) + \alpha.\beta \quad \text{by inductive hypothesis} \\
&\simeq_2 \alpha.(\alpha.p + \alpha.\beta + \alpha.q) + \alpha.\beta \quad \text{by expansion} \\
&\simeq_2 \alpha.(\alpha.p + \alpha.\beta) + \alpha.\alpha.q + \alpha.\beta \\
&\simeq_2 \alpha.p + \alpha.\beta + \alpha.q \\
&\simeq_2 S_2(\alpha.p + \alpha.\beta,\ \alpha.q).
\end{aligned}$$

Before summarising our results we show how these equivalences relate to Milner's observational equivalence $\approx$ [Milner 80]. None of the equivalences imply $\approx$ as we can show that



It is simple to show that $\approx$ implies $\simeq_w$ and $\simeq_k$.

Proposition 4.12

$$p \approx q \supset \text{i) } p \simeq_w q \qquad \text{ii) } p \simeq_k q$$

Proof:

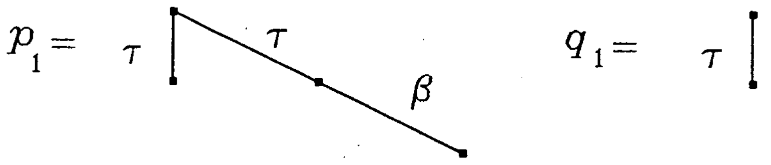
i) Suppose w.l.g. that p *w-must satisfy* o but q *w-must/satisfy* o.
Then $q|o \overset{\varepsilon}{\Longrightarrow} q'|o' \overset{\surd}{\not\Longrightarrow}$, i.e. $q \overset{s}{\Longrightarrow} q'$, $o \overset{\bar{s}}{\Longrightarrow} o'$.
But $p \approx q$ so $\exists p' \approx q'$ s.t. $p \overset{s}{\Longrightarrow} p'$.
As p *w-must satisfy* o then $p'|o' \overset{\surd}{\Longrightarrow}$,
i.e. $p' \overset{s'}{\Longrightarrow} p''$, $o' \overset{\bar{s}'}{\Longrightarrow} o'' \overset{\surd}{\longrightarrow}$.
This implies $q' \overset{s'}{\Longrightarrow}$ and so $q'|o' \overset{\surd}{\Longrightarrow}$, a contradiction.

ii) If $p \not\simeq_k q$ then w.l.g. assume that
(p after s) must L but (q after s) m~~u~~st L.
Then $q \overset{s}{\Longrightarrow} q'$ and for all $\lambda \in L$. $q' \overset{\lambda}{\not\Longrightarrow}$.
But $\exists p'$ s.t. $p \overset{s}{\Longrightarrow} p'$ and $p' \approx q'$.
Furthermore $\exists \lambda \in L$ s.t. $p' \overset{\lambda}{\Longrightarrow}$
and this implies $q' \overset{\lambda}{\Longrightarrow}$, a contradiction. □

It is simple to show that $p \approx q$ does not imply $p \simeq_2 q$ as $\alpha \approx \alpha|\tau^\omega$ but $\alpha \not\simeq_2 \alpha|\tau^\omega$.

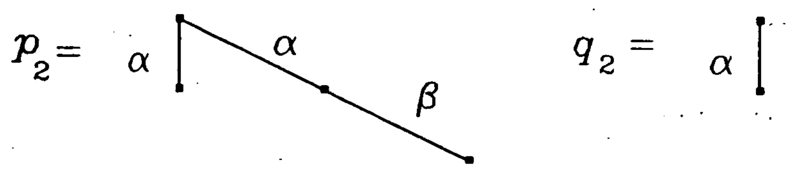
§4.7 An analysis of the differences between $\sqsubseteq_w$ and $\sqsubseteq_k$

Consider the processes p_1 and q_1 defined by



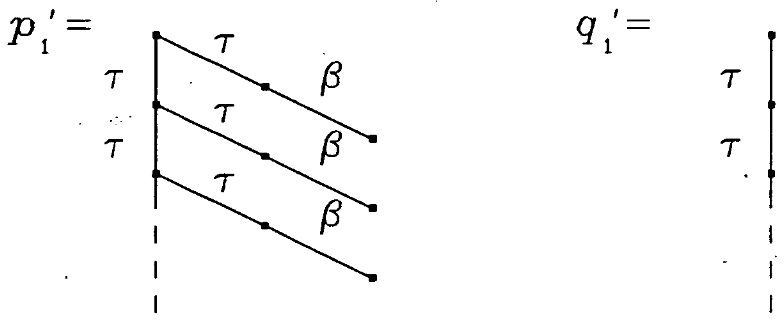
Both orderings agree, i.e. $p_1 \sqsubseteq_w q_1$, $p_1 \sqsubseteq_k q_1$,
$q_1 \not\sqsubseteq_w p_1$, $q_1 \not\sqsubseteq_k p_1$.

Similarly, if we define p_2 and q_2 by

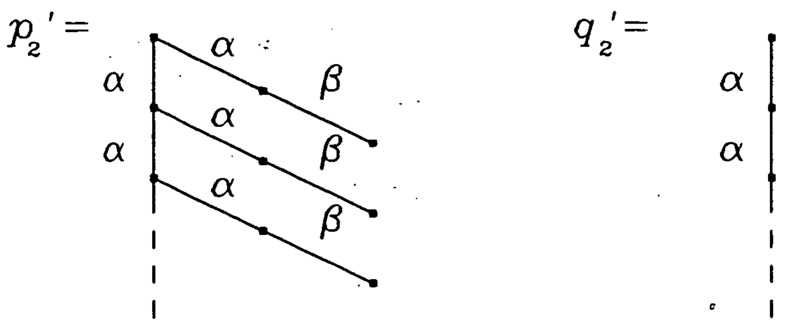


then again they both agree, i.e. $p_2 \sqsubseteq_w q_2$, $p_2 \sqsubseteq_k q_2$,
$q_2 \not\sqsubseteq_w p_2$, $q_2 \not\sqsubseteq_k p_2$.

Let us now extend p_1 and q_1 to the infinite case, i.e. we define



Then both orderings take the view that because it is possible for $p_1{}'$ to reach a state that can perform a β, and because it will always have this option, then it will eventually be allowed to take place. A β move can never happen in $q_1{}'$ and so $p_1{}' \not\sqsubseteq_w q_1{}'$ and $p_1{}' \not\sqsubseteq_k q_1{}'$. Unfortunately, the two orderings disagree on what should happen when p_2 and q_2 are extended to the infinite case. Let $p_2{}'$ and $q_2{}'$ be defined by



Then following the argument for $p_1{}'$ and $q_1{}'$, the weak preorder distinguishes between these two terms because it assumes that a β action will eventually be allowed to happen in $p_2{}'$ whereas it cannot in $q_2{}'$, i.e. $p_2{}' \not\sqsubseteq_w q_2{}'$.

However $p_2{}' \sqsubseteq_k q_2{}'$ which is unfortunate as it implies that $\sqsubseteq_k$ is not preserved by the bar operator. To see this we note that

$$p_1{}' = (p_2{}'|\bar{\alpha}^\omega)\backslash\alpha \quad \text{and} \quad q_1{}' = (q_2{}'|\bar{\alpha}^\omega)\backslash\alpha$$

To summarise, these results imply that Kennaway's preorder does not form the basis of a characterisation of $\sqsubseteq_w$, and furthermore, it would be very difficult to use the $\sqsubseteq_k$ preorder in its own right as it is not preserved by the parallel composition operator.

In order to precede from this point there appear to be a number of choices. We could try to find another characterisation of $\sqsubseteq_w$. The Kennaway preorder encounters difficulties because the definition of <u>must</u> captures our intuitions where τ moves are involved but when we replace the τ actions by visible actions then the preorder cannot 'see through' these actions. We have experimented with some alternative definitions of <u>must</u> that try to remedy this problem. For example, we might define

$$p \ \underline{\text{must}}^{\mathcal{M}} \ L \text{ iff } \forall p' \text{ s.t. } p \overset{s}{\Longrightarrow} p', \quad p' \overset{s'\alpha}{\Longrightarrow} \text{ for some } \alpha \in L$$
$$\text{where } s, s' \in \mathcal{M}^*$$

This would allow us to differentiate between p_2' and q_2' in our previous example as

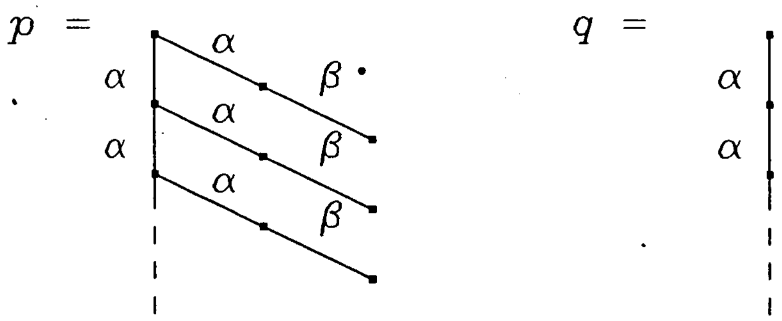
$$p_2' \ \underline{\text{must}}^{\{\alpha\}}\{\beta\} \quad \text{whereas} \quad q_2' \ \underline{\text{m\cancel{u}st}}^{\{\alpha\}}\{\beta\}.$$

Unfortunately, all attempts to build such definitions into the Kennaway preorder have so far proved unsuccessful in faithfully characterising $\sqsubseteq_w$.

Another possibility is to find some additional constraints on processes p and q such that $p \sqsubseteq_k q$ does imply $p \sqsubseteq_w q$. The next section investigates such a constraint called *Controllability*. Demanding that a process is controllable is rather a strong requirement and hence the work, while providing a connection between $\sqsubseteq_w$ and $\sqsubseteq_k$, is of limited applicability. Section 4.9 extends this work by relating the notion of controllability to *Determinacy* [Milner 80, Engelfriet 84]. Section 4.10 introduces a simple preorder that implies the Kennaway preorder. We use this preorder in Chapter 5 to reason about the Schwarz synchronisation scheme. However it does not directly imply $\sqsubseteq_w$ and so suffers from the same deficiency as $\sqsubseteq_k$ in that it currently relies on determinacy to establish a connection with the weak testing preorder. Because of the limited applicability of these results (although an example of their use appears later in the chapter and also in Chapter 5), the next three sections may be viewed as a digression from the main results of this chapter which continue in Section 4.11.

§4.8 Controllable processes

Let us start by considering again the example where $\sqsubseteq_k$ and $\sqsubseteq_w$ disagreed.



One constraint that might allow us to deduce $p \sqsubseteq_w q$ from $p \sqsubseteq_k q$ would be to filter out uncontrollable processes such as p in the above example. Informally, a process is controllable if its evolution can be controlled by means of the actions offered to it by the environment. Process p in the example above is uncontrollable because there is no way for the external environment to guide the process to a state where a β action is possible.

The formal definition of controllability uses sets of agents. A set P is controllable if, whenever one of the processes in the set can perform an action, then this action must be an unavoidable choice for all the agents (i.e. a τ transition cannot remove this possibility). Furthermore, the set of processes obtained by performing this action must also be controllable. A process p is controllable if the singleton set containing p is controllable. The use of sets of processes in the definition of controllability allows non-deterministic choices to be present in an agent, but ensures that such choices do not affect the externally visible behaviour of the process.

Definition

P is <u>controllable</u> iff

$(\exists p \in P,\ \lambda \in \mathcal{A}ct.\ p \overset{\lambda}{\Longrightarrow}) \supset$ P <u>must</u> $\{\lambda\}$, and (P <u>after</u> λ) is controllable

p is controllable iff $\{p\}$ is controllable.

Proposition 4.13 P controllable $\supset$ (P <u>after</u> s) controllable for any $s \in \mathcal{A}ct^*$

Proof:

By induction on s.

<u>Induction Basis</u>, $s = \varepsilon$

P <u>after</u> ε = P so the proposition follows immediately.

<u>Inductive step</u>, $s = \alpha s'$

P <u>after</u> s = (P <u>after</u> α) <u>after</u> s'

Then if P <u>after</u> α = {} then P <u>after</u> s = {} and

is trivially controllable. Otherwise P <u>after</u> α is controllable by definition and so by the inductive hypothesis, (P <u>after</u> α) <u>after</u> s' is therefore controllable. □

Corollary 4.14 If $p' \in p$ <u>after</u> s and $p' \overset{\lambda}{\Longrightarrow} p''$ then (p <u>after</u> s) <u>must</u> $\{\lambda\}$

Proof: Immediate. □

We now show that controllability of p is sufficient to deduce $p \sqsubseteq_w q$ from $p \sqsubseteq_k q$.

Theorem 4.15 $p \sqsubseteq_k q \ \wedge$ p controllable $\supset \ p \sqsubseteq_w q$

Proof:

Suppose $p \not\sqsubseteq_w q$ so that there exists an observer o where p *<u>w-must satisfy</u>* o and q *<u>w-must/satisfy</u>* o. In other words there exists a computation

$$q|o = q_0|o_0 \xrightarrow{\tau} q_1|o_1 \xrightarrow{\tau} \ . \ . \ . \xrightarrow{\tau} q_n|o_n \xrightarrow{\tau} \ . \ . \ .$$

and a prefix $q_0|o_0, \ . \ . \ . \ ,q_n|o_n$ that cannot be extended to a successful computation. Let s be the sequence of actions performed by q between q_0 and q_n. Either the computation is finite or infinite; we treat the two cases separately.

1. The finite case

   We take $q_n|o_n$ to be the final state of the computation. $q \overset{s}{\Longrightarrow} q_n$ and $p \sqsubseteq_k q$ so $p \overset{s}{\Longrightarrow} p_n$ for some p_n. Now if a computation from p|o is to be successful then either the computation has passed through a successful state before reaching $p_n|o_n$, in which case the computation from $q_0|o_0$ would also have been successful, or

   $$o_n \overset{s'}{\Longrightarrow} o_n' \xrightarrow{\checkmark} \quad , \quad p_n \overset{\bar{s}'}{\Longrightarrow} p_n' \qquad \text{for some } o_n', p_n'$$

   and so p_n <u>must</u> $\overline{\text{Init}(o_n)}$ (with rather loose notation). This is true of any such p_n, i.e.

   (p <u>after</u> s) <u>must</u> $\overline{\text{Init}(o_n)}$

   whereas

(q <u>after</u> s) $\underline{\text{m}\cancel{\text{u}}\text{st}}$ $\overline{\text{Init}(o_n)}$

and so we have derived a contradiction.

2. The infinite case.

If $q \overset{s}{\Longrightarrow} q_n$ then $p \overset{s}{\Longrightarrow} p_n$ for some p_n. Furthermore, there exists an $s' = s_1.a.s_2$ such that

$$o_n \overset{\bar{s}'}{\Longrightarrow} o_m \xrightarrow{\checkmark},$$
$$p_n \overset{s_1}{\Longrightarrow} p_n' \overset{a}{\Longrightarrow} p_n'' \overset{s_2}{\Longrightarrow} p_m \text{ and}$$
$$q_n \overset{s_1}{\Longrightarrow} q_n' \overset{a}{\not\Longrightarrow}.$$

Now $p_n' \in p$ <u>after</u> $s.s_1$ and $p_n' \overset{a}{\Longrightarrow}$ so (p <u>after</u> $s.s_1$) <u>must</u> $\{a\}$ as p is controllable. However (q <u>after</u> $s.s_1$) $\underline{\text{m}\cancel{\text{u}}\text{st}}$ $\{a\}$ which leads to a contradiction. □

Corollary 4.16 If p and q are finite and $p \underset{\sim}{\sqsubset}_k q$ then $p \underset{\sim}{\sqsubset}_w q$

Proof:

For finite processes we only need to use finite observers and so all computations are finite. The first part of the last proof will therefore hold which makes no assumptions about controllability. □

§4.9 $\simeq_k$-determinacy

Suppose we have a process p with the property that if p can perform an α action to become the process p_1 then any other process p_2 that is also reachable from p via an α move is related to p_1 in some way. If $p_1 \sim p_2$ then Milner [Milner 80] calls this property of a process *strong determinacy*, i.e.

Definition

p is <u>strongly determinate</u> iff $\forall \lambda \in \mathcal{Act}$

i) $p \xrightarrow{\lambda} p_1$ and $p \xrightarrow{\lambda} p_2 \supset p_1 \sim p_2$

ii) $p \xrightarrow{\lambda} p' \supset p'$ is strongly determinate.

The intuition behind this definition is that if a process is strongly determinate, and contains a non-deterministic choice involving visible actions, then this choice has no observable significance, i.e. the behaviour appears deterministic. Unfortunately, the definition does not constrain non-determinism due to the τ action, and so this aim is only partially successful. The requirement that members of the set of agents reachable from a visible non-deterministic choice must be strongly congruent to each other is also an unnecessarily strict requirement in many cases.

Engelfriet [Engelfriet 84] extends this idea by defining $\equiv$-determinacy for any equivalence relation $\equiv$. His definition is observational, or weak, in the sense that he deals with sequences of actions $\overset{s}{\Longrightarrow}$ rather than the single actions of strong determinacy.

Definition Let $\equiv$ be an equivalence relation over processes.
Then a process $\mathbf{p}$ is $\equiv$-determinate iff for any $s \in \mathcal{A}ct^*$
$\mathbf{p} \overset{s}{\Longrightarrow} \mathbf{p}_1$ and $\mathbf{p} \overset{s}{\Longrightarrow} \mathbf{p}_2 \supset \mathbf{p}_1 \equiv \mathbf{p}_2$

Engelfriet goes on to prove that $\approx$-determinacy and $\simeq_3$-determinacy are the same and calls this property determinacy. He then shows that for determinate processes $\approx$ and $\simeq_3$ coincide. We may obviously extend this result to our weak equivalence as

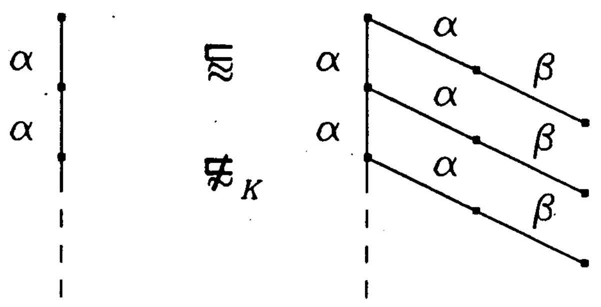
$$\approx \;\subset\; \begin{matrix}\simeq_w \\ \simeq_k\end{matrix} \;\subset\; \simeq_3$$

Therefore, if we are working with equivalences rather than preorders, showing that the processes are determinate is sufficient to deduce $\mathbf{p} \simeq_w \mathbf{q}$ from $\mathbf{p} \simeq_k \mathbf{q}$.

The situation for the preorders is more complicated. If we define a preorder version of $\approx$,

$\mathbf{p} \sqsubseteq_{\approx} \mathbf{q}$ iff $\forall s \in \mathcal{A}ct^*.\ \mathbf{p} \overset{s}{\Longrightarrow} \mathbf{p}' \supset \exists \mathbf{q}'.\ \mathbf{q} \overset{s}{\Longrightarrow} \mathbf{q}' \wedge \mathbf{p}' \sqsubseteq_{\approx} \mathbf{q}'$

then $\mathbf{p} \sqsubseteq_{\approx} \mathbf{q} \not\supset \mathbf{p} \sqsubseteq_w \mathbf{q}$, for example, as



A result for the other preorders along the lines of [Engelfriet 84] may be possible but it is outwith the scope of this thesis. However, we will show that $\simeq_k$-determinacy is equivalent to controllability for a process and so either constraint on p will allow us to deduce $p\underset{\approx}{\sqsubset}_w q$ from $p\underset{\approx}{\sqsubset}_k q$. We start by presenting some lemmas about controllable processes.

Lemma 4.17

If P controllable then P' controllable for all $P' \subseteq P$.

Proof:

Suppose false. There must exist a sequence s such that for $p' \in P'$ after s, $p' \overset{\lambda}{\Longrightarrow} p''$ but (P' after s) m̸ust $\{\lambda\}$. But $p' \in P$ after s and so (P after s) must $\{\lambda\}$ which leads to a contradiction. □

Lemma 4.18

If P controllable and P must L then $\exists \lambda \in L$ s.t. P must $\{\lambda\}$

Proof:

If P must L then either P is empty in which case trivially P must $\{\lambda\}$ for any λ, or P has at least one element p. Furthermore, as P must L, $\exists \lambda \in L$ s.t. $p \overset{\lambda}{\Longrightarrow}$. But then P must $\{\lambda\}$ as P is controllable. □

Lemma 4.19

If P is controllable then for any $P' \subseteq P$,
if (P' after s) must L for some s, L where (P' after s) $\neq \phi$,
then (P after s) must L.

Proof:

P' after s is controllable as it is a subset of P after s by Lemma

4.17. Therefore $\exists\lambda\in L$ s.t. (P' <u>after</u> s) <u>must</u> $\{\lambda\}$ by Lemma 4.18. As P' <u>after</u> s is non-empty, $\exists p'\in$P' <u>after</u> s s.t. $p' \overset{\lambda}{\Longrightarrow} p''$. But $p'\in$P <u>after</u> s and so (P <u>after</u> s) <u>must</u> $\{\lambda\}$ and hence (P <u>after</u> s) <u>must</u> L. □

Lemma 4.20

If P is controllable and $P\times P\subseteq\simeq_k$ then (P <u>after</u> s)$\times$(P <u>after</u> s)$\subseteq\simeq_k$ for any sequence $s\in\mathcal{Act}^*$

Proof:

Suppose it is false, i.e. $\exists$s s.t $p_1\not\simeq_k p_2$ for some $p_1,p_2\in$ P <u>after</u> s. Then let s' be such that for any s'' where $|s''| < |s'|$

(p_1 <u>after</u> s'') <u>must</u> L' $\Leftrightarrow$ (p_2 <u>after</u> s'') <u>must</u> L' for any L'

and w.l.g.

(p_1 <u>after</u> s') <u>must</u> L but (p_2 <u>after</u> s') <u>~~must~~</u> L for some L.

There are two cases to consider.

1. p_1 <u>after</u> s' is non-empty.
   In this case (P <u>after</u> s) <u>after</u> s' <u>must</u> L by Lemma 4.19 as $\{p_1\}\subseteq$P <u>after</u> s. But p_2 <u>after</u> s' $\subseteq$ (P <u>after</u> s) <u>after</u> s' so we have a contradiction.

2. p_1 <u>after</u> s' = ϕ. Now s' cannot equal ε so $\exists$a,s'' such that s' = s''a. Furthermore, p_2 <u>after</u> s' $\neq\phi$ and so

   $p_2 \overset{s''}{\Longrightarrow} p_2' \overset{a}{\Longrightarrow} p_2''$ for some p_2', p_2''.

   Now p_2 <u>after</u> s'' is controllable and so (p_2 <u>after</u> s'') <u>must</u> $\{a\}$ but (p_1 <u>after</u> s'') <u>~~must~~</u> $\{a\}$. But then by Lemma 4.19, ((P <u>after</u> s) <u>after</u> s'') <u>must</u> $\{a\}$ and (p_1 <u>after</u> s'') $\subseteq$ ((P <u>after</u> s) <u>after</u> s'') so we have obtained a contradiction again. □

Lemma 4.21

If $p\overset{\lambda}{\Longrightarrow}$ and p is $\simeq_k$-determinate, then p <u>must</u> $\{\lambda\}$

Proof:

Suppose false, i.e. $p\overset{\varepsilon}{\Longrightarrow}p_1\overset{\lambda}{\longrightarrow}p_1'$ and $p\overset{\varepsilon}{\Longrightarrow}p_2\overset{\lambda}{\not\Longrightarrow}$. Now because of $\simeq_k$-determinacy, $p_1\simeq_k p_2$. Let γ be such that $p_1'\overset{\gamma}{\not\Longrightarrow}$. Then

(p_2 <u>after</u> λ) <u>must</u> $\{\gamma\}$ whereas (p_1 <u>after</u> λ) <u>$\cancel{\text{must}}$</u> $\{\gamma\}$ and hence $p_1 \not\simeq_k p_2$, a contradiction. □

Corollary 4.22 If $p \overset{\lambda}{\Longrightarrow}$ and p is $\simeq_k$-determinate then (p <u>after</u> ε) <u>must</u> $\{\lambda\}$

We are now in a position to establish a connection between $\simeq_k$-determinacy and controllability.

Theorem 4.23 p controllable $\Leftrightarrow$ p $\simeq_k$-determinate

Proof:

1. ($\Longrightarrow$)
   Suppose false, i.e. $p \overset{s}{\Longrightarrow} p'$, $p \overset{s}{\Longrightarrow} p''$, $p' \not\simeq_k p''$. But p' and p'' are members of $\{p\}$ <u>after</u> s and so by Lemma 4.20, $p' \simeq_k p''$, a contradiction.

2. ($\Longleftarrow$)
   Suppose false, i.e. $\exists s$ and $p_1 \in p$ <u>after</u> s s.t. $p_1 \overset{\lambda}{\Longrightarrow} p_1'$ but (p <u>after</u> s) <u>$\cancel{\text{must}}$</u> $\{\lambda\}$. In other words $\exists p_2 \in$ p <u>after</u> s s.t. $p_2 \overset{\varepsilon}{\Longrightarrow} p_2' \overset{\lambda}{\not\Longrightarrow}$. Now $p_1 \simeq_k p_2$ and so by the previous lemma, p_1 <u>must</u> $\{\lambda\}$. But this leads to a contradiction as (p_1 <u>after</u> ε) <u>must</u> $\{\lambda\}$ but (p_2 <u>after</u> ε) <u>$\cancel{\text{must}}$</u> $\{\lambda\}$. □

§4.10 The > preorder

Although Kennaway's preorder is easy to work with because it does not involve observers, it does require the manipulation of sets of processes. Motivated by what we require of an implementation, we now develop a very simple preorder that will imply the Kennaway preorder. It involves no sets of processes or tests, but the relation will be quite restrictive. However, we will argue that it is applicable to many real situations.

Let us suppose that i was designed to be an implementation of the specification s. What relationship would we expect between i and s?

One reasonable requirement would be that if the implementation i can perform a visible action λ to become the process i' then the specification s must also allow this to happen, and i' must be an implementation of the resulting process s'. Furthermore, if the specification cannot avoid performing a particular action then the implementation must also be unable to avoid it. Formally we can express these requirements as follows.

Definition

i is a refinement of s or i refines s (written s > i) iff

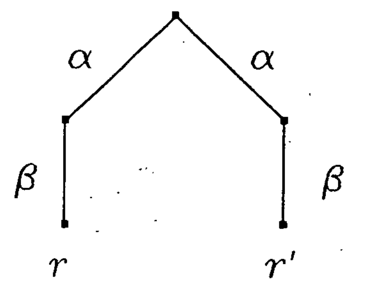
i) $i \overset{\lambda}{\Longrightarrow} i' \supset \exists s'$ s.t. $s \overset{\lambda}{\Longrightarrow} s' \wedge s' > i'$ for all $\lambda \in \mathcal{A}ct$

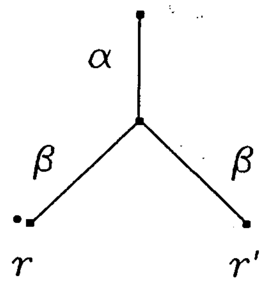
ii) s must L ⊃ i must L $\forall L \subseteq \mathcal{A}ct$

Hennessy has investigated a similar preorder in [Hennessy 84c] called the *must-testing* preorder. His definition differs slightly from the one presented here because in his framework all divergence is considered harmful and so it is explicitly dealt with in the definition.

What is the relationship between the refinement preorder and Kennaway's preorder? The definition of > does not involve sets of processes and so we would expect that the refinement ordering is more particular about when non-deterministic choices are made. However, the situation is a little more subtle than this. Consider the processes

$$p = \alpha.\beta.r + \alpha.\beta.r' \qquad q = \alpha.(\beta.r + \beta.r')$$





Kennaway's ordering equates these terms, i.e. $p \sqsubseteq_k q$ and $q \sqsubseteq_k p$. Although we can show that q>p, which we can view as saying that p is a valid implementation of the specification q, we cannot show that p>q. In other words, our definition allows p to be an implementation of q if, amongst other things, p makes non-deterministic choices <u>no later</u> than q. This may not be too restrictive in practise as a specification will typically keep its options open for as long as possible whereas an implementation may commit itself to a particular non-deterministic choice (because of silent internal communications) much earlier on in the execution sequence.

Before showing that $p>q$ implies $p \sqsubseteq_k q$ we will need to prove the following lemma.

Lemma 4.24 If $p>q$ and $q \overset{s}{\Longrightarrow} q'$, $s \subseteq \mathcal{Act}^*$ then $\exists p'$ s.t. $p \overset{s}{\Longrightarrow} p' \wedge p'>q'$

Proof:

If $s = \varepsilon$ then we show that $p>q'$.
Suppose $q' \overset{\lambda}{\Longrightarrow} q''$. Then $q \overset{\lambda}{\Longrightarrow} q''$ and so $p \overset{\lambda}{\Longrightarrow} p'$ for some p' where $p'>q''$. If p <u>must</u> L then q <u>must</u> L and so q' <u>must</u> L. Therefore $p>q'$.

If $s \neq \varepsilon$ then

$$q=q_0 \overset{a_1}{\Longrightarrow} q_1 \overset{a_2}{\Longrightarrow} q_2 \; . \; . \; . \; \overset{a_n}{\Longrightarrow} q_n=q'$$

where $s=a_1a_2 \ldots a_n$. Therefore $\exists p_1$ s.t. $p \overset{a_1}{\Longrightarrow} p_1$ and $p_1>q_1$, and similarly for the other q_i processes so $\exists p_n$ s.t. ,

$$p=p_0 \overset{a_1}{\Longrightarrow} p_1 \overset{a_2}{\Longrightarrow} p_2 \; . \; . \; . \; \overset{a_n}{\Longrightarrow} p_n \text{ and } p_n>q_n. \qquad \Box$$

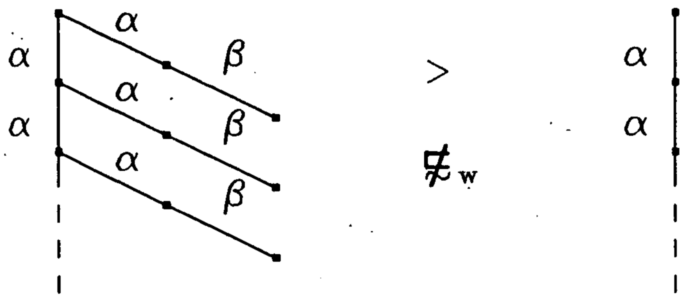
Theorem 4.25

$p>q \supset p \sqsubseteq_k q$

Proof:

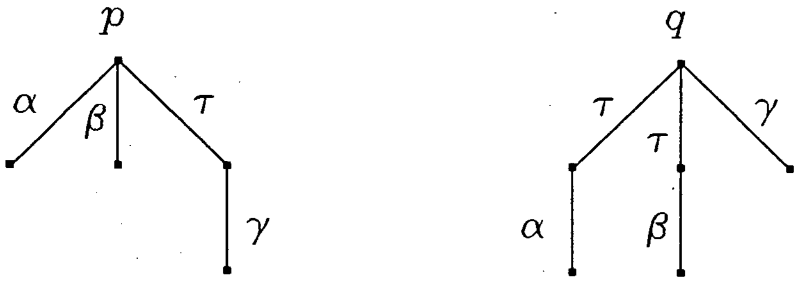
Suppose $p \not\sqsubseteq_k q$. then $\exists$s,L s.t. (p <u>after</u> s) <u>must</u> L and (q <u>after</u> s) <u>m~~u~~st</u> L, i.e. $\forall p'$ s.t. $p \overset{s}{\Longrightarrow} p'$. p' <u>must</u> L and $\exists q'$ s.t. $q \overset{s}{\Longrightarrow} q'$ and q' <u>m~~u~~st</u> L. If $q \overset{s}{\Longrightarrow} q'$ then $\exists p'$ s.t. $p \overset{s}{\Longrightarrow} p'$ and $p'>q'$ which leads to a contradiction as p' <u>must</u> L and q' <u>m~~u~~st</u> L. $\Box$

It is not true that $p>q \supset p \sqsubseteq_w q$ as the following example illustrates.



This result is not surprising as > uses <u>must</u> in its definition which is what caused the problems for Kennaway's preorder.

Can we simplify > even further? Suppose we used the version of <u>must</u> defined in [Milner 80]. What would be the consequences? Consider the processes



With process **p** we can choose between performing an α or a β action, but we don't always get a chance and may have to perform a γ action. With **q** we always have a chance to perform an α or a β, but only one of them will be offered, the choice being non-deterministic. Neither process can really be viewed as an implementation of the other. (**p** <u>after</u> ε) <u>must</u> $\{\gamma\}$ whereas (**q** <u>after</u> ε) <u>must</u> $\{\alpha, \beta\}$ so $\mathbf{p} \not\lesssim_k \mathbf{q}$ and $\mathbf{q} \not\lesssim_k \mathbf{p}$. Furthermore, $\mathbf{p} \not> \mathbf{q}$ and $\mathbf{q} \not> \mathbf{p}$.

Let us use $>^s$ to denote the refinement ordering using the single action <u>must</u>, defined in [Milner 80] as

$$\mathbf{p}\ \underline{\text{must}}\ \lambda \quad \text{iff} \quad \forall \mathbf{p}' \text{ s.t. } \mathbf{p} \overset{\tau}{\Longrightarrow} \mathbf{p}'.\ \ \mathbf{p}' \overset{\lambda}{\Longrightarrow}$$

Then $\mathbf{p} \not>^s \mathbf{q}$ but $\mathbf{q} >^s \mathbf{p}$ which is undesirable.

The other simplification we might make would be to replace $p \overset{\lambda}{\Longrightarrow} p'$ in the definition of $>$ by $p \xrightarrow{\mu} p'$ or even $p \xrightarrow{\lambda} p'$ (where $\mu \in \mathcal{Act} \cup \{\tau\}$ and $\lambda \in \mathcal{Act}$). This would simplify bisimulation style proofs as we would only have to examine the immediate actions that can be performed by p rather than examining all sequences that p might perform. In fact we could define four variants of $>$ as follows.

$q >_1 p$ iff i) $p \overset{\mu}{\Longrightarrow} p' \supset q \overset{\mu}{\Longrightarrow} q' \wedge q' >_1 p'$ and ii) q <u>must</u> L $\supset$ p <u>must</u> L
$q >_2 p$ iff i) $p \overset{\lambda}{\Longrightarrow} p' \supset q \overset{\lambda}{\Longrightarrow} q' \wedge q' >_2 p'$ and ii) q <u>must</u> L $\supset$ p <u>must</u> L
$q >_3 p$ iff i) $p \xrightarrow{\mu} p' \supset q \overset{\mu}{\Longrightarrow} q' \wedge q' >_3 p'$ and ii) q <u>must</u> L $\supset$ p <u>must</u> L
$q >_4 p$ iff i) $p \xrightarrow{\lambda} p' \supset q \overset{\lambda}{\Longrightarrow} q' \wedge q' >_4 p'$ and ii) q <u>must</u> L $\supset$ p <u>must</u> L

What is the relationship between this family of orderings. $>_1$ is the most difficult to use and $>_4$ the simplest. However, the following example forces us to rule out $>_4$. Consider

$$p = \tau.\alpha.p' + \alpha.q' \qquad q = \alpha.q'$$

Then $q >_4 p$ but $q \not>_{1,2,3} p$. Moreover p is not a very reasonable realisation of q intuitively and so we will reject $>_4$ as a possible candidate.

We can trivially show that $>_1 \subseteq >_2$ and $>_1 \subseteq >_3$. We will show that $>_2 \subseteq >_1$ and $>_3 \subseteq <_1$, which will be sufficient to show that all three orderings are equivalent.

Let $\mathfrak{R}_1$ be the defining relation for $>_1$ when presented in its simulation form (where a simulation is one half of a bisimulation).

Let $\mathcal{R} = >_2$. Consider any pair $<q, p> \in \mathcal{R}$.
If $p \overset{\mu}{\Longrightarrow} p'$ then either $\mu = \tau$, in which case $q \overset{\varepsilon}{\Longrightarrow} q$ and $q >_2 p'$, or $\mu = \lambda$ in which case trivially $q \overset{\lambda}{\Longrightarrow} q'$ where $q' > p'$.
If q <u>must</u> L then p <u>must</u> L as $q >_2 p$.
This shows that $\mathcal{R} \subseteq \mathfrak{R}_1(\mathcal{R})$ and hence $q >_2 p \supset q >_1 p$.

Let $\mathcal{R} = >_3$. Consider any pair $<q, p> \in \mathcal{R}$.
If $p \overset{\mu}{\Longrightarrow} p'$ then $\exists p_1, \ldots, p_n$ s.t.

$$p = p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} \ldots p_i \xrightarrow{\mu} p_{i+1} \xrightarrow{\tau} \ldots \xrightarrow{\tau} p_n = p'$$

But $q >_3 p$ so $\exists\, q_1, \ldots, q_n$ s.t.

$$q=q_0 \overset{\tau}{\Longrightarrow} q_1 \overset{\tau}{\Longrightarrow} \ldots q_i \overset{\mu}{\Longrightarrow} q_{i+1} \overset{\tau}{\Longrightarrow} \ldots \overset{\tau}{\Longrightarrow} q_n=q'$$

where $q_j>p_j$, $0\leq j\leq n$. Therefore $q \overset{\mu}{\Longrightarrow} q'$ where $q'>_3p'$.
If q <u>must</u> L then p <u>must</u> L as $q>_3p$.
In other words, $\mathcal{R}\subseteq\mathfrak{R}_1(\mathcal{R})$ and so $q>_3p \supset q>_1p$.

This proves that $>_1 \; = \; >_2 \; = \; >_3$. As $>_3$ is the simplest of the definitions to use, we will assume that we are referring to $>_3$ when we write $q>p$.

§4.11 The $>_t$ preorder

So far, our only connection between $\sqsubseteq_k$ (and hence >) and $\sqsubseteq_w$ is if the processes under investigation are controllable or $\simeq_k$-determinate. This is quite a strong condition to demand of a system, and in particular, we will see that it is not true for the Schwarz transformation. As the final result of this part of the chapter we will develop a preorder along the lines of > that does imply $\sqsubseteq_w$. Although it is more restrictive than >, we will show in Chapter 5 how to express Schwarz' scheme in a way that makes the new preorder applicable.

Definition

$p >_t q$ <u>iff</u> i) $q \xrightarrow{\mu} q' \supset \exists p'.\ p \overset{\mu}{\Longrightarrow} p' \wedge p'>_t q'$

ii) $\mathrm{Traces}(p) \subseteq \mathrm{Traces}(q)$

where $p \overset{\tau}{\Longrightarrow} \; \equiv \; p \overset{\varepsilon}{\Longrightarrow}$

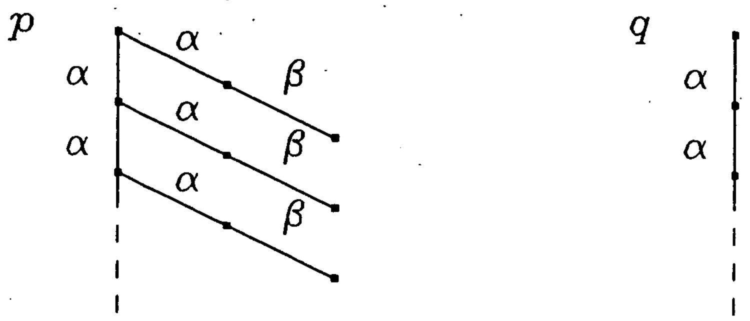
If $p>_tq$ then we know that if $q \overset{s}{\Longrightarrow}$ then $p \overset{s}{\Longrightarrow}$ and so $\mathrm{Traces}(p) = \mathrm{Traces}(q)$.

Proposition 4.26 If $p>_tq$ then $\forall L\subseteq\mathcal{A}ct.$ p <u>must</u> L $\supset$ q <u>must</u> L

Proof:

Assume false. Then $q \overset{\varepsilon}{\Longrightarrow} q'$ s.t. $\not\exists\lambda\in L.\ q' \overset{\lambda}{\Longrightarrow}$. But $\exists p'$ s.t. $p \overset{\varepsilon}{\Longrightarrow} p' \wedge p'>_tq'$ and $p' \overset{\lambda}{\Longrightarrow}$ for some $\lambda\in L$. Therefore $q' \overset{\lambda}{\Longrightarrow}$ as $\mathrm{Traces}(p')\subseteq\mathrm{Traces}(q')$ and so we have a contradiction. □

This implies that $>_t \subseteq >$ as whenever $p >_t q$ it satisfies the conditions for $p > q$. They are not equal, however, as our favorite example shows.



Here $p > q$ but $p \not>_t q$. This example illustrates why $>$ did not imply the <u>*weak must*</u> preorder. We now show that $>_t$ does imply $\sqsubseteq_w$.

Theorem 4.27 $p >_t q$ implies $p \sqsubseteq_w q$

Proof:

Suppose $p >_t q$ but $p \not\sqsubseteq_w q$, i.e. there exists an observer $o \in \mathcal{O}$ such that p <u>*w-must satisfy*</u> o and q <u>*w-must/satisfy*</u> o. In other words there exists a computation

$$q|o = q_0|o_0 \xrightarrow{\tau} q_1|o_1 \xrightarrow{\tau} \ . \ . \ . \xrightarrow{\tau} q_n|o_n \xrightarrow{\tau} \ . \ . \ .$$

with a prefix $q_0|o_0, . . . , q_n|o_n$ that cannot be extended to a successful computation. Let s be the sequence of visible actions performed by q between q_0 and q_n. Then $p \overset{s}{\Longrightarrow} p_n$ where $p_n > q_n$. Furthermore $p_n \overset{s'}{\Longrightarrow} p_m$ and $o_n \overset{\bar{s}'}{\Longrightarrow} o_m \xrightarrow{\surd}$ for some s' as all computations of $p|o$ have successfully extendable prefixes. But if $p_n \overset{s'}{\Longrightarrow} p_m$ then $\exists q_m$ such that $q_n \overset{s'}{\Longrightarrow} q_m$ and so $q_n|o_n \overset{\varepsilon}{\Longrightarrow} q_m|o_m \xrightarrow{\surd}$ which is a contradiction. □

Theorem 4.28 If $p >_t q$ then $\forall \mu \in \mathcal{A}ct \cup \{\tau\}$, $\forall \lambda \in \mathcal{A}ct$, $\forall r \in \mathcal{P}$, S a relabelling,

1. $\mu.p >_t \mu.q$
2. $p|r >_t q|r$
3. $p \backslash \lambda >_t q \backslash \lambda$
4. $p[S] >_t q[S]$

Proof:

1. If $\mu.q \xrightarrow{\mu} q$ then $\mu.p \overset{\mu}{\Longrightarrow} p$ and $p >_t q$.
   If $s \in Traces(\mu.p)$ then either $\mu = \tau$, in which case $s \in Traces(q)$, or else $s = \mu.s'$ and so $s' \in Traces(p)$ which implies $s' \in Traces(q)$ and hence $\mu.s' \in Traces(\mu.q)$.

2. Let $\mathfrak{R}_t$ be the defining relation for $>_t$ when presented in its simulation form.
   Let $\mathcal{R} = \{<p|r, q|r> \mid p >_t q\}$. If $q|r \xrightarrow{\mu} q'|r'$ then there are three possibilities.

   a. $r \xrightarrow{\mu} r'$, $q = q'$. Then $p|r \xrightarrow{\mu} p|r'$ where $<p|r', q|r'> \in \mathcal{R}$
   b. $q \xrightarrow{\mu} q'$, $r = r'$. Then $p >_t q$ so $p \overset{\mu}{\Longrightarrow} p'$ where $p' >_t q'$ and hence $<p'|r, q'|r> \in \mathcal{R}$
   c. $q \xrightarrow{\lambda} q'$, $r \xrightarrow{\bar{\lambda}} r'$, $\mu = \tau$.
      Then $p \overset{\lambda}{\Longrightarrow} p'$ and hence $p|r \overset{\mu}{\Longrightarrow} p'|r'$ where $<p'|r', q'|r'> \in \mathcal{R}$

   Suppose $s \in Traces(p|r)$, i.e. $p|r \overset{s}{\Longrightarrow} p'|r'$. Then there exists s_1, s_2 such that $p \overset{s_1}{\Longrightarrow} p'$, $r \overset{s_2}{\Longrightarrow} r'$, where s_1 and s_2 can be merged to form s (possibly with some actions cancelling to form τ moves). But then $s_1 \in Traces(q)$ as $p >_t q$ and hence $q \overset{s_1}{\Longrightarrow} q'$ for some q' which implies $q|r \overset{s}{\Longrightarrow} q'|r'$, i.e. $s \in Traces(q|r)$. This proves that $\mathcal{R} \subseteq \mathfrak{R}(\mathcal{R})$ and hence $p|r >_t q|r$.

3. Let $\mathcal{R} = \{<p\backslash\lambda, q\backslash\lambda> \mid p >_t q\}$
   If $q\backslash\lambda \xrightarrow{\mu} q'\backslash\lambda$ then $q \xrightarrow{\mu} q'$, $p \overset{\mu}{\Longrightarrow} p'$ and hence $p\backslash\lambda \overset{\mu}{\Longrightarrow} p\backslash\lambda$ where $<p'\backslash\lambda, q'\backslash\lambda> \in \mathcal{R}$.
   If $s \in Traces(p\backslash\lambda)$ then $s \in Traces(p)$, $s \in Traces(q)$ and hence $s \in Traces(q\backslash\lambda)$.
   Thus $\mathcal{R} \subseteq \mathfrak{R}(\mathcal{R})$ and hence $p\backslash\lambda >_t q\backslash\lambda$.

4. Let $\mathcal{R} = \{<p[S], q[S]> \mid p >_t q\}$
   If $q[S] \xrightarrow{\mu} q'[S]$ then $q \xrightarrow{\nu} q'$ for some ν such that $S(\nu) = \mu$. But then $p \overset{\nu}{\Longrightarrow} p'$ where $p' >_t q'$ and hence $p[S] \overset{\mu}{\Longrightarrow} p[S]$ where $<p'[S], q'[S]> \in \mathcal{R}$.

If $s \in Traces(p[S])$ then $\exists s'$ such that $s' \in Traces(p)$ and $S(s')=s$ where the morphism S is extended to sequences in the obvious way. But $s' \in Traces(q)$ and hence $s \in Traces(q[S])$.

Thus $\mathcal{R} \subseteq \mathfrak{R}(\mathcal{R})$ and hence $p[S] >_t q[S]$.

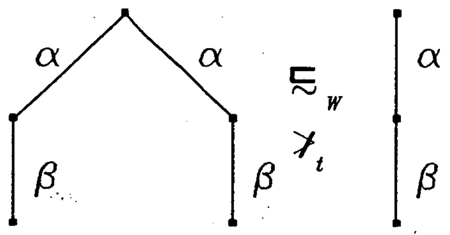
□

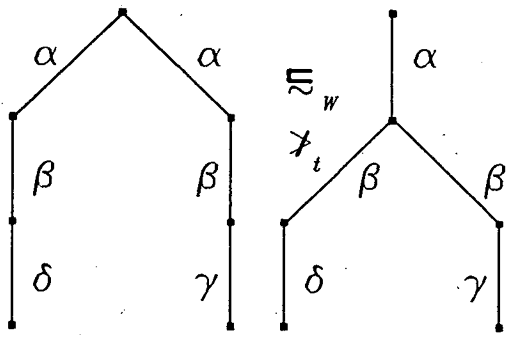
$>_t$ is not preserved by + as the normal example for this case illustrates, i.e.

$\alpha.p >_t \tau.\alpha.p$

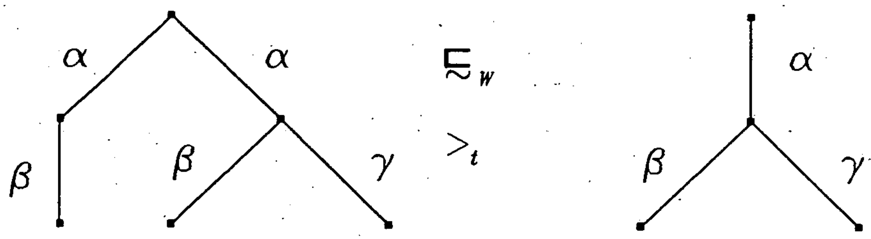
but $\alpha.p + r \not>_t \tau.\alpha.p + r$ in general.

Trying to prove that $p \sqsubseteq\!\!\approx_w q$ by using $>_t$ removes some of the freedom of the implementer. For example,





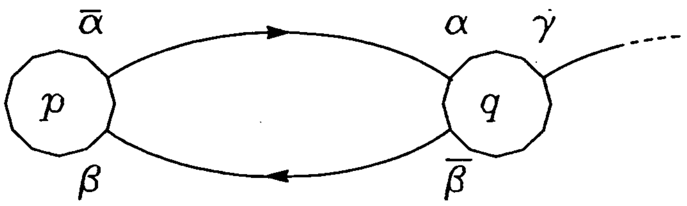
However, some freedom of choice is left as the following example shows.



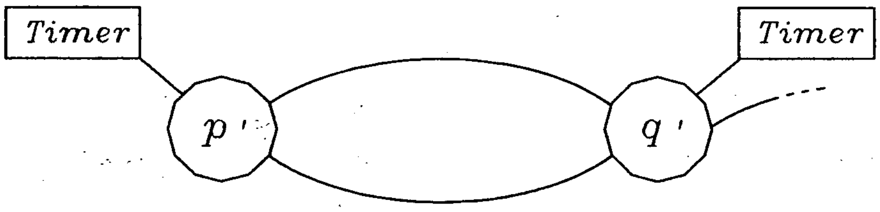
§4.12 A simple example

Consider the simple synchronisation problem involving processes p and q defined by

$$p = \bar{\alpha}.p + \beta.p \quad q = \alpha.q + \bar{\beta}.q + \gamma.NIL$$



In an implementation of this program we might try to replace the summations in p and q by simpler ones involving a timeout agent. Each process would offer one or other of its actions for a while but would then timeout and try the other one if unsuccessful.



If we assume that the timers are not synchronised, we may model the timeout processes implicitly using τ actions, i.e.

$p' = p_1$ and $q' = q_1$

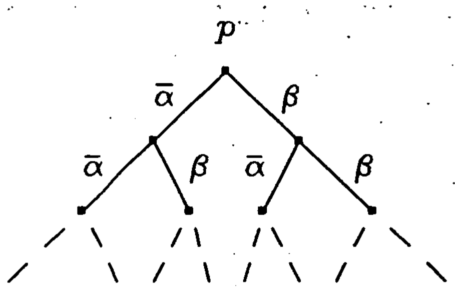
where

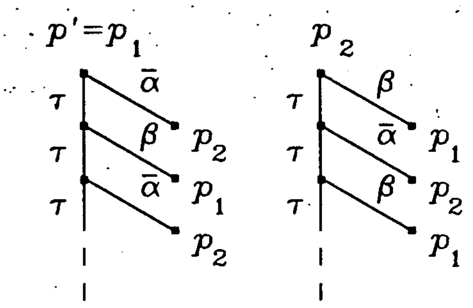
$$\begin{array}{ll} p_1 = \bar{\alpha}.p_2 + \tau.p_2 & q_1 = \alpha.q_2 + \tau.q_2 \\ p_2 = \beta.p_1 + \tau.p_1 & q_2 = \bar{\beta}.q_2 + \tau.q_3 \\ & q_1 = \gamma.NIL + \tau.q_1 \end{array}$$

It is simple to show that $p \not\approx_2 p'$ as p' introduces the possibility of an infinite τ sequence that was not present in p. The observer $\alpha.\surd.NIL$ can differentiate between the two processes as

p <u>*must satisfy*</u> $\alpha.\surd.NIL$ whereas p' <u>*must/ satisfy*</u> $\alpha.\surd.NIL$

The derivation trees for p and p' make this clearer.





In this particular case we can apply either of our approaches to show that $p \simeq_w p'$. Of course with such a simple example it would be easy to prove the equivalence directly but on larger examples this would be much more difficult.

We start by showing that $p > p'$ and $p' > p$. Rather than performing two separate simulation proofs, we combine them as follows.

Let $\mathcal{R} = \{<p_1,p>, <p_2,p>, <p,p_1>, <p,p_2>\}$

$<p_1,p>$.	if $p_1 \xrightarrow{\bar{\alpha}} p_2$ then $p \overset{\bar{\alpha}}{\Longrightarrow} p$, $<p_2,p> \in \mathcal{R}$
	if $p_1 \xrightarrow{\tau} p_2$ then $p \overset{\varepsilon}{\Longrightarrow} p$, $<p_2,p> \in \mathcal{R}$
$<p_2,p>$.	if $p_2 \xrightarrow{\beta} p_1$ then $p \overset{\beta}{\Longrightarrow} p$, $<p_1,p> \in \mathcal{R}$
	if $p_2 \xrightarrow{\tau} p_1$ then $p \overset{\varepsilon}{\Longrightarrow} p$, $<p_1,p> \in \mathcal{R}$
$<p,p_1>$.	if $p \xrightarrow{\bar{\alpha}} p$ then $p_1 \overset{\bar{\alpha}}{\Longrightarrow} p_2$, $<p,p_2> \in \mathcal{R}$
	if $p \xrightarrow{\beta} p$ then $p_1 \overset{\beta}{\Longrightarrow} p_1$, $<p,p_1> \in \mathcal{R}$
$<p,p_2>$.	if $p \xrightarrow{\bar{\alpha}} p$ then $p_2 \overset{\bar{\alpha}}{\Longrightarrow} p_2$, $<p,p_2> \in \mathcal{R}$
	if $p \xrightarrow{\beta} p$ then $p_2 \overset{\beta}{\Longrightarrow} p_1$, $<p,p_1> \in \mathcal{R}$

It is easy to show that $\forall L \subseteq \mathcal{A}c\ell$, p <u>must</u> L implies $\{\bar{\alpha},\beta\} \subseteq L$. This is true for p_1 and p_2 as well and so the second part of the conditions for > are satisfied. We have therefore shown that $p > p_1$ and $p_1 > p$ (similarly for p_2). We may therefore deduce that $p \simeq_k p_1$. Both p and p_1 are trivially controllable and hence we may deduce that $p \simeq_w p_1$.

In this particular example we may prove that $p \simeq_w p_1$ more directly by using $>_t$. Using the same relation $\mathcal{R}$ as before we must show that for each pair $<p_i, p_j>$ in $\mathcal{R}$, $\text{Traces}(p_i) \subseteq \text{Traces}(p_j)$. This follows immediately from the fact that

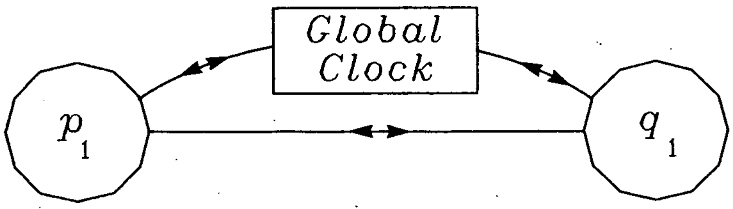
$$\text{Traces}(p) \quad = \quad \text{Traces}(p_1) \quad = \quad \text{Traces}(p_2)$$

Therefore we can prove that $p >_t p_1$ and $p_1 >_t p$. This allows us to deduce that $p \simeq_w p_1$.

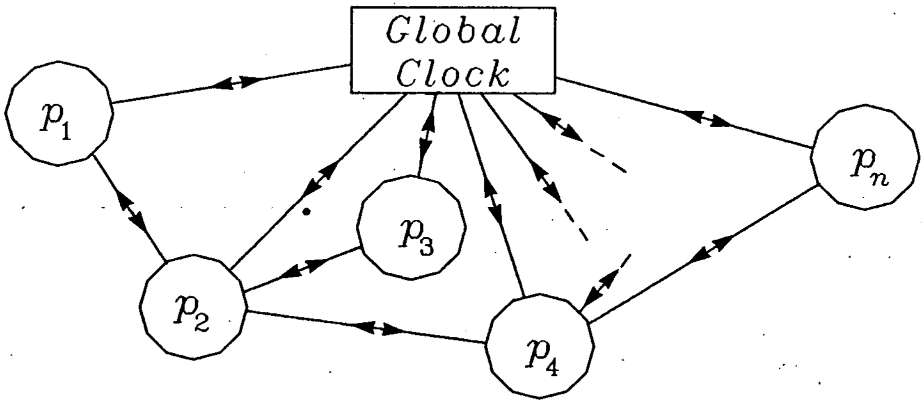
4.12.1 Decomposing problems with global dependencies

Our assumption that the timers for each process were unconnected allowed us to replace them by internal τ moves. The interfaces to the original process **p** and its replacement p' were therefore identical; they both just had $\{\bar{\alpha}, \beta\}$ in their sort. This allowed us to reason about the transformation applied to **p** separately from the rest of the system.

Suppose we wished to model a variant of the above system where there was a global clock that generated the timeout signals, e.g.



We might generalise the problem further and assume that all the original communications took place via this transformation, i.e.



The new variant suffers from two complications not present in the original problem. Firstly, we cannot prove anything about the individual processes as they no longer have an identical interface to the original process they represent. This is because they will have timeout actions appearing in their sort. We therefore have to reason about the system as a whole. However, this introduces our second problem. If all of the communications in the original system are transformed so that they take place via the new mechanism, then either there will be no externally visible actions in either system, or the visible actions will be different because of the different protocols involved in the two systems. This raises the question of how to tell that we have constructed a valid transformation. These problems form the motivation behind the rest of this chapter.

§4.13 Implementation and translation transformations

In this section we develop further the notion of implementation and then present a definition of transformation correctness for CCS processes.

4.13.1 The weak-must form of implementation

While the definition of implementation presented in the first part of this chapter characterises most of our intuitions about what constitutes an implementation, there are some deficiencies that are now discussed.

Firstly, consider an arbitrary process **p**. If we placed it in parallel with a process that idled continuously, would we view the resulting system as a valid implementation of **p**? Certainly the current definition would not view $\mathbf{p}|\tau^{\omega}$ as a valid implementation of **p**. This is because for all observers **o**, $\mathbf{p}|\tau^{\omega}$ *must satisfy* **o** is false due to an infinite τ path in the derivation tree of $\mathbf{p}|\tau^{\omega}|\mathbf{o}$. However with anything other than the most malicious of schedulers, we would view $\mathbf{p}|\tau^{\omega}$ as an implementation of **p**. It might run (a lot) slower than the original but it would still behave eventually in a similar fashion to **p**.

Suppose that we have a process $p|\tau^{\omega}$. Would we expect $(p|\tau^{\omega}) + \tau.NIL$ to be an implementation of this process? Our current definition of implementation says that this is the case, but our intuitions might say that $p|\tau^{\omega}$ is a slower version of p whereas $(p|\tau^{\omega}) + \tau.NIL$ might stop completely. At this point it should be pointed out that if we make no assumptions about the scheduler used to implement these examples then the existing definition of an implementation may be adequate. The reason our intuitions may differ from this definition is because we would like to assume that a fair scheduler is used to run these processes, or at the very least a scheduler that is not designed to select the worst possible path through a derivation tree. The development of the weak form of _*must satisfy*_ was prompted by these intuitions and so the definition of implementation is changed accordingly to

i _implements_ s or i is an _implementation_ of s iff
$\forall$ o$\in\mathcal{O}$. i _*may satisfy*_ o $\supset$ s _*may satisfy*_ o
s _*w-must satisfy*_ o $\supset$ i _*w-must satisfy*_ o

This definition may be simplified by noting that $p \sqsubseteq_w q \supset q \sqsubseteq_3 p$, and hence proving that i implements s is equivalent to showing that $s \sqsubseteq_w i$, i.e.

i _implements_ s or i is an _implementation_ of s iff $s \sqsubseteq_w i$

Incorporating the *weak must* preorder into the definition of implementation results in a simpler definition. Unfortunately, it also means that ICD is no longer an implementation of CD in the change machine example presented at the beginning of this chapter. The original definition allows an implementation to provide only part of the non-deterministic choices offered by the specification. The new definition requires, in addition, that if a particular action will be eventually offered by the specification, due to fairness arguments, then it will also be eventually offered by the implementation as well. By fairness arguments, CD must eventually offer shillings as change, whereas ICD never has this possibilty. It is, however, possible to define an unfair version of CD such that ICD is a valid implementation.

The development of the original definition of implementation, and the weak must preorder, are essentially independent. For the proof of the Schwarz scheme it will be convenient to merge these ideas into the new, simplified, definition of implementation. However, for other purposes, it may be more convenient to work with the original definition of implementation.

With our new definition of implementation, it is easy to show that $\mathbf{p}|\tau^\omega$ implements **p** whereas $(\mathbf{p}|\tau^\omega) + \tau$.NIL does not implement $(\mathbf{p}|\tau^\omega)$.

Proposition 4.29 Implementation is a transitive relation

Proof:

Let us suppose that **p** implements **q** and **q** implements **r**.
Then for all observers $\mathbf{o} \in \mathcal{O}$,
if **r** *w-must satisfy* **o** then **q** *w-must satisfy* **o** and so
p *w-must satisfy* **o**. Therefore **p** implements **r**. □

4.13.2 Transformations

In general, when $\mathbf{p} \sqsubseteq_2 \mathbf{q}$ or **p** implements **q**, the syntactic structure of **p** is not related to the syntactic structure of **q**. However, sometimes we wish to exhibit a transformation function *tr* such that the expression produced by *tr*(p) is related to the expression p in some way. Often the relationship is independent of a particular process p. These transformations are purely syntactic; they take as arguments expressions representing CCS terms and syntactically manipulate them to produce new expressions representing CCS terms. The transformations do not depend on the semantic meaning of their arguments; thus, for example, the transformation of p|q is not necessarily the same as the transformation of q|p. This point will become important later when we introduce functions that apply a different transformation to each process in a product depending on its relative position.

The synchronisation scheme presented in Chapter 3 is an obvious example of such a transformation. These purely syntactic transformations also occur when reasoning about different variants of CCS. For example, rather than using a single message to synchronise between two processes, we may wish to model the synchronisation by a start message and a finish message. This allows a communication to take a finite amount of time rather than being instantaneous. This variant of CCS may be expressed as a syntactic transformation. Motivated by this example and other similar problems, Millington [Millington 82] developed a notion of transformation correctness for CCS based on the testing approach to process equivalence of DeNicola and Hennessy. The key observation was to point out that if we replace a process **p** by the transformed process $tr(\mathbf{p})$ then it will not usually be valid to examine both processes with the same observer. This is due to the fact that a transformation may introduce observable differences. If we take a process **p** and transform it into its 'start finish' form, then there would obviously be an observable difference between the original and the transformed system. However, if we introduce a pair of transformations, one for the process **p** and one for the observer **o**, then Millington showed that we can develop a notion of correctness for such transformations.

Prompted by the sort of reasoning that influenced our definition of implementation, Millington developed a similar notion for transformations. Furthermore, he introduced the concept of a translation which can be viewed as the transformation equivalent of the $\simeq_1$ equivalence.

We start by presenting Millington's original definitions of these concepts and then develop them further based on our previous discussions of the *weak must* and also motivated by our intended usage.

Millington views a transformation as being composed of a pair of transformation functions. One of these functions is applied to the system under investigation, **p**, and the other to the observer of the system, **o**.

Definition A transformation $tr = <tr_{proc}, tr_{obs}>$ is <u>implementation correct</u>

iff $\forall p \in \mathcal{P}.\ \forall o \in \mathcal{O}.$

1. p _must satisfy_ o implies tr_{proc}(p) _must satisfy_ tr_{obs}(o)

2. tr_{proc}(p) _may satisfy_ tr_{obs}(o) implies p _may satisfy_ o.

A transformation $tr = \langle tr_{proc}, tr_{obs} \rangle$ is translation correct iff the reverse implications also hold, i.e. $\forall p \in \mathcal{P}.\ \forall o \in \mathcal{O}.$

1. p _may satisfy_ o iff tr_{proc}(p) _may satisfy_ tr_{obs}(o)

2. p _must satisfy_ o iff tr_{proc}(p) _must satisfy_ tr_{obs}(o).

In Millington's paper, a transformation was called a translation which led to the possibility of a translation being translation correct, for example. We prefer to keep the notion of transformation distinct from that of a particular form of transformation called a translation.

We may view the concept of implementation correctness as the transformation equivalent of our original notion of implementation, and similarly, translation correctness may be viewed as corresponding to the $\simeq_1$ equivalence.

When we introduced the notion of implementation earlier in this chapter, it was eventually defined using the weak form of _must satisfy_. It therefore seems natural to apply the same sort of reasoning to the transformation case. A more serious limitation of the previous definitions involves the choice of two separate transformation functions. This is adequate when the transformation applied to the observer is independent of that applied to the observed process. However this is not always the case. For example, the transformation applied to the observer may depend on the number of processes in the observed component. Apt and Olderog [Olderog 84] define the adjective $\|$-preserving for transformations that preserve the parallel structure of programs. In such a case, the only information the transformation function applied to each component may use about the structure of the system, S, is the total number of components in S and the index of the currently transformed component.

For ||-preserving transformations, the transformations applied to the observer and the observed processes must be linked in some way. One approach to the problem would be to treat the transformation as a function that accepted a pair of processes (i.e. a process and an observer) and returned a pair of transformed processes. However, this is more general than we need as the definitions of _w-must satisfy_ and _may satisfy_ immediately place the resulting pair in parallel again. The only difference between the pair of processes is that the translated observer process may have √ in its sort. Therefore we take the view that the transformation need only return a single expression which can be viewed as the parallel composition of the transformed process **p** and the transformed observer process **o**. Frequently, the same transformation is applied to both the observer and the observed processes. As this simplifies the presentation, we will assume that this will always be the case, although it is not essential to our work. In other words, we will assume that a transformation also takes a single expression as its argument, formed from the parallel composition of **p** and **o**.

In order to express these ideas in practice, we must modify our definition of _w-must satisfy_, because the current definition is in the form of an infix binary predicate. We introduce an equivalent postfixed predicate as follows.

Definition

$(p|o)$ _w-must succeed_ $\equiv$ p _w-must satisfy_ o •

It will also prove convenient to be able to reason about the correctness of a transformation relative to a second transformation. The simpler case then follows by taking the second transformation to be the identity function. Finally, as $p \sqsubseteq_w q \supset q \sqsubseteq_3 p$, we may omit the _may satisfy_ case as it is implied by the _w-must satisfy_ case. Summarising all of these developments, we might define

Definition

$\mathbf{tr}_1 \sqsubseteq_w \mathbf{tr}_2$ (read "$\mathbf{tr}_1$ implements $\mathbf{tr}_2$") iff $\forall p \in \mathcal{P}.\ \forall o \in \mathcal{O}.$

$\mathbf{tr}_2(p|o)$ *__w-must succeed__* implies $\mathbf{tr}_1(p|o)$ *__w-must succeed__*

By merging the observer and the observed processes, we see that the distinction between **p** and **o** in the definition has become irrelevant. In Millington's case, the distinction may be useful because different transformations may be applied to the two processes. Because there is no longer a need for this distinction, we define implementation and translation transformations as follows.

Definition $\mathbf{tr}_1 \sqsubseteq_w \mathbf{tr}_2$ (read "$\mathbf{tr}_1$ implements $\mathbf{tr}_2$") iff $\forall q \in \mathcal{O}.$

$\mathbf{tr}_2(q)$ *__w-must succeed__* $\supset$ $\mathbf{tr}_1(q)$ *__w-must succeed__*

$\mathbf{tr}_1 \simeq_w \mathbf{tr}_2$ (read "$\mathbf{tr}_1$ translates $\mathbf{tr}_2$") iff $\forall q \in \mathcal{O}.$

$\mathbf{tr}_1(q)$ *__w-must succeed__* $\Leftrightarrow$ $\mathbf{tr}_2(q)$ *__w-must succeed__*

If we take $\mathbf{tr}_2$ to be the identity function then we say that $\mathbf{tr}_1$ is an implementation transform, or $\mathbf{tr}_1$ is a translation transform. We may also omit the word transform and talk about $\mathbf{tr}_1$ being an implementation if the context makes it clear that we are referring to a transformation function.

Proposition 4.30 If the pair $<tr,tr>$ is translation correct by Millington's definition and $tr(p|o) \equiv tr(p)|tr(o)$ then tr is a translation transform.

Proof: Follows from the definitions.

Proposition 4.31 $\sqsubseteq_w$ and $\simeq_w$ are transitive for transformations, i.e. for any $\mathbf{tr}_1$, $\mathbf{tr}_2$, $\mathbf{tr}_3$,

1. if $\mathbf{tr}_1 \sqsubseteq_w \mathbf{tr}_2$ and $\mathbf{tr}_2 \sqsubseteq_w \mathbf{tr}_3$ then $\mathbf{tr}_1 \sqsubseteq_w \mathbf{tr}_3$

2. if $\mathbf{tr}_1 \simeq_w \mathbf{tr}_2$ and $\mathbf{tr}_2 \simeq_w \mathbf{tr}_3$ then $\mathbf{tr}_1 \simeq_w \mathbf{tr}_3$

Proof: Follows immediately from the definitions.

We may also compose the transformation functions $\mathbf{tr}_1$ and $\mathbf{tr}_2$ to form $\mathbf{tr}_1 \circ \mathbf{tr}_2$. This composed transformation preserves $\sqsubseteq_w$ and $\simeq_1$ in the following sense.

Proposition 4.32

1. if $\mathbf{tr}_1$ is an implementation and $\mathbf{tr}_2$ is an implementation then $\mathbf{tr}_1 \circ \mathbf{tr}_2$ is an implementation

2. if $\mathbf{tr}_1$ is an translation and $\mathbf{tr}_2$ is a translation then $\mathbf{tr}_1 \circ \mathbf{tr}_2$ is a translation

Proof:

1. If q *w-must succeed* then $\mathbf{tr}_2(q)$ *w-must succeed* and so $\mathbf{tr}_1(\mathbf{tr}_2(q))$ *w-must succeed*.

2. The proof is similar to the implementation case. □

Li [Li 83] has also investigated the concept of the correctness of a translation in an operational framework. The task of finding sufficient conditions for proving translation correctness is called the adequacy problem. Li presents an adequate set of conditions for his notion of correctness. However, we prefer to work with the testing view of translation correctness as it is incorporated more naturally with the weak-must testing preorder.

Note that, as with the definition of implementation, it is not essential for the defintion of transformation to be based on the weak-must preorder. However, if this is not desirable for a particular application, then we must ensure that the *may succeed* case is retained in the definitions.

This completes our discussion of transformations and also concludes this chapter. To summarise, we have introduced the notion of implementation which led to a discussion about fairness. This prompted the development of the *weak-must* testing preorder $\sqsubseteq_w$. In an attempt to develop a proof technique for this preorder, we introduced Kennaway's preorder $\sqsubseteq_k$ and the > preorder. A connection was established between $\sqsubseteq_w$ and $\sqsubseteq_k$ using the notion of controllability or $\simeq_k$-determinacy. A simpler preorder, $>_t$, was then introduced that directly implied the weak-must preorder. We then turned our attention again to the definition of implementation where it was redefined to reflect the work on $\sqsubseteq_w$. Finally, Millington's work on transformations was introduced and extended to prepare the ground for Chapter 5 which now follows.

CHAPTER 5

A Rigorous Validation of an Implementation

§5.1 Introduction

Chapter 3 developed a transformation that may be applied to arbitrary Static CCS expressions in order to facilitate their efficient execution. However, as we saw in the latter part of that chapter, it was not clear in what sense the transformed system was equivalent to the original program. This prompted the developments outlined in Chapter 4. We are now in a position to use these more precise notions of implementation and transformation to show that our transformation is indeed a valid implementation of the original system.

The proof of the Schwarz transformation is split into two parts. We first show that the subnetwork consisting of the pollers is an implementation of a simpler network of processes called synchronisers. We then show that a simple transformation involving the synchronisers is correct, from which the correctness of the Schwarz transformation follows almost immediately. The proofs are complicated by the unstructured nature of the program which prevents an inductive style of proof from being used. We therefore develop some notation to conveniently represent the states of the systems.

In order to prove that the poller network P is an implementation of the network of simple synchronisers, S, we first use the refinement ordering, which is sufficient to prove that $S \sqsubseteq_k P$. Unfortunately, S is not $\simeq_k$-determinate, and so we cannot prove that $S \sqsubseteq_w P$ using this approach. This prevents us from completing the proof that P implements S, although we believe this to be the case, and illustrates the need for a less restrictive property that allows us to deduce $p \sqsubseteq_w q$ from $p \sqsubseteq_k q$.

The only other indirect technique for proving $S \sqsubseteq_w P$ is to use the $>_t$ ordering. Unfortunately, we can show that $S \not>_t P$, although a simple modification of the poller algorithm does allow this ordering to be used, and hence produces a proof that the modified poller network is an implementation of the synchroniser network. We motivate why the modification to the algorithm is reasonable, and the proof that $S > P$ may be adapted to this case, involving only a small amount of additional work.

The final part of the chapter discusses how to partially apply the Schwarz transformation, and the consequences of this on the correctness proof.

We start by briefly summarising the transformation described in Chapter 3. Given a Static CCS term of the form $\prod_{i\in N} \mathbf{p}_i$, we syntactically translate the processes using the function $\mathbf{Tr}_{\mathbf{Poll}}$ defined by

$$\mathbf{Tr}_{\mathbf{Poll}}\left[\!\!\left[\prod_{i\in N} \mathbf{p}_i \right]\!\!\right] = \prod_{i\in N} \left(tr_i[\![\mathbf{p}_i]\!] \mid Poller_i(1,\phi) \mid Buffer_i \right)$$

where the transformation function applied to each process, tr_i, is defined by

$$tr_i\left[\!\!\left[\sum_{j\in m} a_j.p_{ij} \right]\!\!\right] = \underline{\text{let}}\ partners = \bigcup_{j\in m} C(a_j)\ \underline{\text{in}}$$
$$\overline{offer_i}\big(partners \cup \{i\}\big).$$
$$\left(\sum_{\pi\in partners} \big(select_i(\lambda).a_k.tr_i[\![p_{ik}]\!]\ \underline{\text{where}}\ C(a_k) = \pi\big) \atop \{+\ \surd.NIL\ \text{if}\ \exists a_j = \surd\} \right)$$

This definition is identical to the one presented in Chapter 3 except that we now deal with the case where $\surd$ may appear in the sort of the processes. There is no partner for such an action and so it is left unchanged.

Each translated process is placed in parallel with a poller and buffer process, the definitions of which are presented in Figure 5-1.

Our problem then is to show that $\mathbf{Tr}_{\mathbf{Poll}}\left[\!\!\left[\prod_{i \in N} \mathbf{p}_i \right]\!\!\right]$ is a valid implementation of $\prod_{i \in N} \mathbf{p}_i$.

5.1.1 Structuring the proof

The transformed processes in $\mathbf{Tr}_{\mathbf{Poll}}[\![\prod_{i \in N} \mathbf{p}_i]\!]$ communicate with each other via pollers, and so the observable behaviour will be different from $\prod_{i \in N} \mathbf{p}_i$. Therefore the conventional equivalence relations and preorders cannot be used to prove the correctness of $\mathbf{Tr}_{\mathbf{Poll}}$. The approach we take here is to show that $\mathbf{Tr}_{\mathbf{Poll}}$ is an implementation transformation in the sense of the previous chapter. $\mathbf{Tr}_{\mathbf{Poll}}[\![\prod_{i \in N} \mathbf{p}_i]\!]$ is a rather complicated object to work with as it involves both a change of interface and also reduced behaviour potential when compared with $\prod_{i \in N} \mathbf{p}_i$. The interface is changed because of the use of pollers to synchronise the inter-process communications. The behaviour potential is reduced because the sequencing of the pollers may mean that certain non-deterministic branches of the computation are not always possible. One way of reasoning about the translated system would be to analyse it inductively. We can view a translated system pictorially as

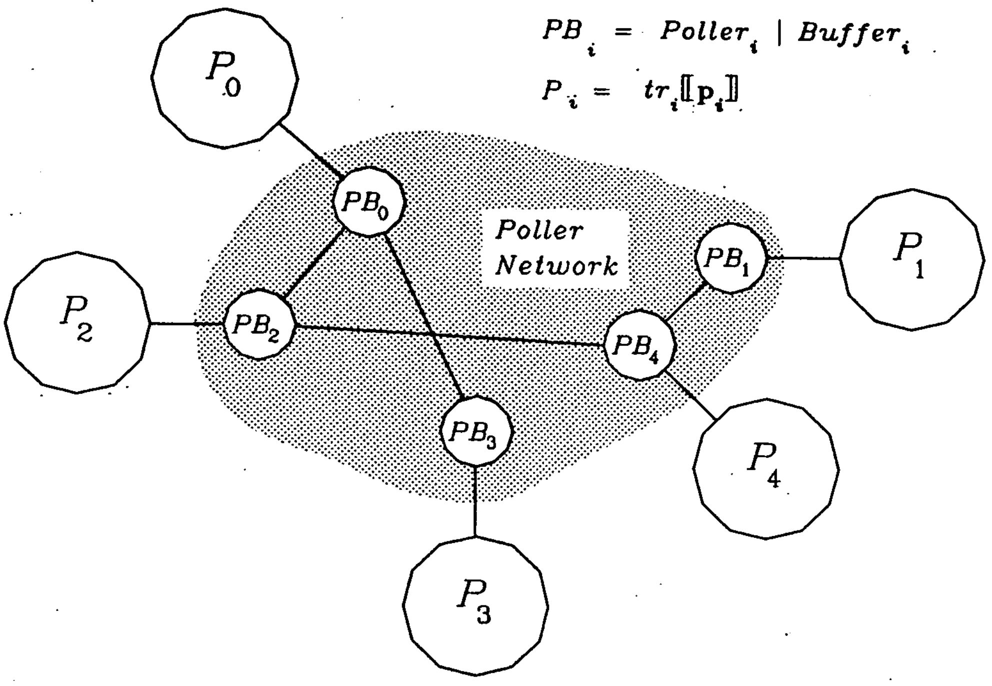
$$Poller_i(n,k) \;=\; \underline{\text{let}}\; j = PC_i[n]\; \underline{\text{in}}$$

$$\begin{array}{lll}
 & offer_i(k').\; Poller_i(n,k') & \underline{\text{if}}\; k = \phi \\
 & & \\
+ & \overline{set}_i(j). & \underline{\text{if}}\; i>j \wedge j\in k \\
 & A_{ji}(r). & \\
 & \underline{\text{if}}\; r = YES & \\
 & \underline{\text{then}}\; \overline{select}_i(j).\; Poller_i(n+1,\phi) & \\
 & \underline{\text{else}}\; Poller_i(n+1,k) & \\
 & & \\
+ & \tau.\; Poller_i(n+1,k) & \underline{\text{if}}\; i>j \wedge j\notin k \wedge k\neq\phi \\
 & & \\
+ & Q_{ji}(r). & \underline{\text{if}}\; i<j \wedge j\in k \\
 & \underline{\text{if}}\; r = YES & \\
 & \underline{\text{then}}\; \overline{A}_{ij}(YES).\; \overline{select}_i(j).\; Poller_i(n+1,\phi) & \\
 & \underline{\text{else}}\; Poller_i(n+1,k) & \\
 & & \\
+ & Q_{ji}(r). & \underline{\text{if}}\; i<j \wedge j\notin k \wedge k\neq\phi \\
 & \underline{\text{if}}\; r = YES & \\
 & \underline{\text{then}}\; \overline{A}_{ij}(NO).\; Poller_i(n+1,k) & \\
 & \underline{\text{else}}\; Poller_i(n+1,k) &
\end{array}$$

$$Buffer_i = \sum_{j\in PC_i} \left(\begin{array}{l} \overline{Q}_{ij}(NO).\; Buffer_i \\ \quad +\; set_i(j).\; Buffer'_i(j) \end{array} \right)$$

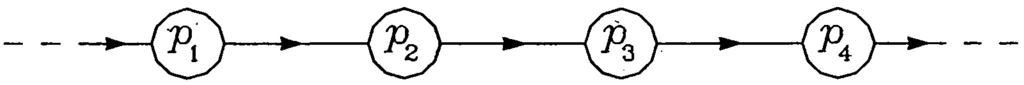
$$\begin{array}{l}
Buffer'_i(k) = \left(\displaystyle\sum_{j\in PC_i-\{k\}} \overline{Q}_{ij}(NO).\; Buffer'_i(k) \right) \\
\qquad\qquad +\; \overline{Q_{ik}}(YES).\; Buffer_i
\end{array}$$

Figure 5-1: The Schwarz Poller and Buffer in Static CCS

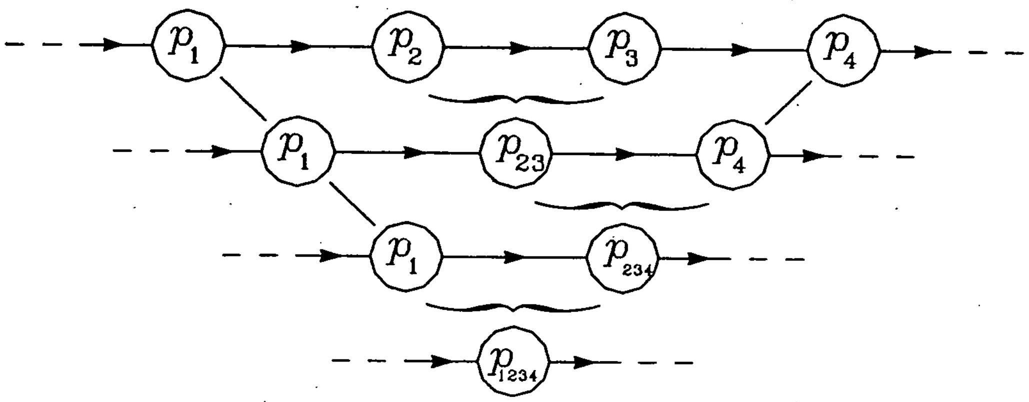


There is a link between PB_i and PB_j if at some point in a computation of the program, P_i may request to communicate with P_j, or vice versa.

We could attempt to prove something about the subnetwork of PB_i nodes by showing inductively that a single PB_i node was equivalent to some simpler expression and then showing that n+1 nodes were equivalent to the expression assuming that n were. However, the inductive approach is not really applicable in this case. The reason becomes clear if we analyse the connection pattern between the PB_i cells. The connections depend on the particular program under investigation, and so there is no general structure that we can exploit inductively. In contrast, consider the case where we have a pipeline of processes



If we extract $\mathbf{p}_2$ and $\mathbf{p}_3$, and hide away their internal interfaces, then we can often replace them by some simpler definition that has the same interface as one of the constituent parts. Thus, by repeatedly applying this technique, we gradually reduce the number of processes without increasing the complexity of the overall term.



If the same technique was applied to the poller subnetwork, we would find that although the number of processes decreased, their complexity increased. It is only when the internal details are hidden from the complete poller subnetwork that a simpler definition can be found. It is because all the network needs to be present before it can be simplified that prevents us from using the inductive approach.

Another way of simplifying the system inductively would be to show, loosely speaking, that

$$\mathbf{Tr}_{\mathbf{Poll}}[\![\, p \,]\!] \mid \mathbf{Tr}_{\mathbf{Poll}}[\![\, q \,]\!] \equiv \mathbf{Tr}_{\mathbf{Poll}}[\![\, p|q \,]\!]$$

Such an approach would allow us to reduce the number of processes but unfortunately, $\mathbf{Tr}_{\mathbf{Poll}}[\![p]\!] \mid \mathbf{Tr}_{\mathbf{Poll}}[\![q]\!]$ and $\mathbf{Tr}_{\mathbf{Poll}}[\![p|q]\!]$ have different behaviours, and so this technique is not directly applicable either.

§5.2 Simple synchronisers

Our analysis of the problem seems to imply that we must reason about the network without using an inductive approach. We start by showing the the complete poller subnetwork is equivalent to a simpler network composed of simple processes called *synchronisers*.

The transformation function tr_i that we apply to process p_i is quite general. It converts a communication that would normally take place implicitly into one where a request is passed to some synchronising agent whose task it is to find a partner for one of the offered communications. The synchronising agent may be composed of a set of pollers, but there will be many other possible definitions of processes that perform this task. We would be one step nearer to our goal if we could show that the complicated synchronising agent in terms of pollers was equivalent to some simpler agent. The simplest agent we might consider would be a single centralised process that took requests from all over the network and selected matching requests. However, the definition of such a process would be very cumbersome, and a more structured approach is to define a network of simple synchronising agents, one for each host process, that perform this task.

The simple synchronising agent is defined as follows.

$$\begin{aligned} SSync_i = {} & offer_i(\mathrm{k}). \\ & \underline{\text{if}}\ \mathrm{k}=\phi\ \underline{\text{then}}\ SSync_i \\ & \underline{\text{else}}\ \Big(\sum_{j\in k \wedge j<i} m_{ji}.\ \overline{select_i}(\mathrm{j}).\ SSync_i \\ & \qquad + \sum_{j\in k \wedge j>i} \overline{m_{ij}}.\ \overline{select_i}(\mathrm{j}).\ SSync_i \Big) \end{aligned}$$

A simple synchronising agent accepts offers from its host until it receives a non-empty request set. The synchroniser then attempts to communicate with the synchronisers mentioned in the request set, using the m_{ij} actions. To ensure that these actions complement each other correctly, the processes are ordered, and the convention adopted that $SSync_i$ offers an m_{ji} to any $SSync_j$ such that j < i, and an m_{ik} to any other synchroniser, $SSync_{ik}$. When one of the m_{ij} requests succeeds, this fact is reported back to the host via the select action. The m_{ij} communications reflect the communications that would have happened in the original system except that no values are passed, and the 'direction' of the message may be switched.

Let us suppose that we can show that

$$\prod_{i \in N} \left(Poller_i(1,\phi) \mid Buffer_i \right)$$

is an implementation of

$$\prod_{i \in N} \left(SSync_i \right)$$

when both collections of processes are surrounded by a restriction so that the only visible actions are *offers* and *selects*. Then we can define a simple transformation function $\mathbf{Tr}_{\mathbf{Sync}}$ that treats the host processes as in $\mathbf{Tr}_{\mathbf{Poll}}$ but places them in parallel with simple synchronisers rather than pollers. If we can show that $\mathbf{Tr}_{\mathbf{Sync}}$ is an implementation transformation in the sense of Chapter 4, then we can also deduce that $\mathbf{Tr}_{\mathbf{Poll}}$ is an implementation transformation. There is the additional benefit that if we wished to analyse a different synchronising agent then we would only need to show that it was an implementation of our simple synchronising agent.

Unfortunately, with our current definitions,

$$\prod_{i \in N} \left(Poller_i(1,\phi) \mid Buffer_i \right)$$

is not a valid implementation of

$$\prod_{i \in N} \left(SSync_i \right).$$

To see why, consider what happens when $Poller_i$ and $Poller_j$ can potentially communicate, and $Poller_j$ has no other summands. Furthermore, assume that no other process wishes to communicate with $Poller_i$. Eventually we would expect the system to output a $select_i$(j) message to the external environment. Further let us suppose that there exists another poller, $Poller_k$, that is in a position where it wishes to output a *select* message. The system may be in a state where i>k and k is a possible communicand of $Poller_i$. Under these circumstances,

$Poller_i$ may have to check for a communication with $Poller_k$ before it reaches a state where it can communicate with $Poller_j$. In this case, $Poller_i$ will have to wait until the environment accepts the *select* message from $Poller_k$ which then frees $Poller_k$ to reply negatively to $Poller_i$. With an arbitrary environment this causes problems because the environment may wish to accept a message from $Poller_i$ before accepting one from $Poller_k$ and so this part of the system will deadlock. No such problems exist with the simple synchroniser network.

The difficulty arises only when we consider environments which may perform arbitrary interactions with the system. In a closed system, one where all environment and source program communications take place via the translation, such problems cannot arise. Thus we are only requiring the two subnetworks to be equivalent in a limited class of environments. There are two courses of action open to us at this point. One would be to conduct some form of context-dependent proof [Larsen 85] assuming that all of the original communications take place via the transformation. The second approach would be to modify the poller algorithm so as to remove the potential deadlock when placed in an arbitrary environment. One of the eventual aims of this work is to enable the transformation to be used selectively on those communications that are difficult to synchronise. Therefore there may be communications that take place without the aid of the transformation. To avoid potential problems at a later stage, and also to avoid performing a context-dependent proof, it would be desirable to develop a version of the poller network that matched our simple synchroniser specification more exactly.

One way of solving the problem is to allow the poller to satisfy its commitments to other pollers while waiting to output a *select* message or input an *offer* message. But in order for the transformation to be meaningful, we must take steps to avoid introducing any new communications that may be difficult to synchronise. If we did not do this, we would be reintroducing the problem that the pollers were designed to solve. As an example of this potential pitfall, suppose we removed the constraint that $k \neq \phi$ in the definition of the poller when

$j \notin k$. This would allow the poller either to accept an offer from its master or to accept a Q_{ji} message from a remote poller. Unfortunately, the remote process offering this message is also offering similar messages to other processes and so we have unwittingly constructed a communication net which is difficult to synchronise.

The approach we take here is to introduce an extra τ move before the Q_{ji} message when $i<j$ and $j \notin k$. In this case, when we are in a situation where $i<j$ and $j \notin k$ for some non-empty set k, then although we still have a choice of actions, the sources of these choices do not themselves have other possible communications. This removes the synchronisation difficulty. In practice, it is easy to implement summands with τ moves as follows. When we encounter a summation, some of whose summands are τ moves, we first check to see if any matching communication requests are waiting and if so we pick one of these; otherwise we randomly choose one of the τ branches. What this would imply for our particular example is that if the master process was currently waiting to send a request then the poller would accept it, and if the master was busy then the poller would carry on polling the rest of the system.

In order to avoid a similar deadlock problem with the *select* message, we treat it in the same way as the *offer* message. This allows the poller to process other messages while waiting for the master to respond. The modified version of the poller process is presented in Figure 5-2. It will prove convenient to be able to specify which part of the definition a process is currently executing. We therefore label the different locations in the algorithm by means of integers enclosed in braces.

In order that we may compare the states a poller can reach with those of the simple synchroniser, we label the *SSync* processes with numeric labels as well.

```
$Poller_i(n,k,s)$ = {1} let j = $PC_i[n]$ in

        $offer_i(k').\ Poller_i(n,k',s)$                if $k = \phi$

    +   $\overline{select}_i(s).\ Poller_i(n+1,\phi,\perp)$          if $s \neq \perp$

    +   $\overline{set}_i(j).$                          if $i>j \wedge j \in k \wedge s=\perp$
    {2} $A_{ji}(r).$
        if r = $YES$
        then $Poller_i(n,k,j)$
        else $Poller_i(n+1,k,s)$

    +   $Q_{ji}(r).$                            if $i<j \wedge j \in k \wedge s=\perp$
        if r = $YES$
        then {3} $\overline{A}_{ij}(YES).\ Poller_i(n,k,j)$
        else $Poller_i(n+1,k,s)$

    +   $\tau.\ Poller_i(n+1,k,s)$                  if $i>j \wedge (j \notin k \vee s \neq \perp)$

    +   $\tau.$ {4}$Q_{ji}(r).$                      if $i<j \wedge (j \notin k \vee s \neq \perp)$
        if r = $YES$
        then {5}$\overline{A}_{ij}(NO).\ Poller_i(n+1,k,s)$
        else $Poller_i(n+1,k,s)$
```

We assume that each poller starts with $n=1$, $k=\phi$ and $s=\perp$.

Figure 5-2: The Modified Schwarz Poller

$SSync_i$ = {1}$offer_i(k).$

  if $k=\phi$ then $SSync_i$

  else {2}$\left(\sum_{j \in k \wedge j<i} m_{ji}.\ \{3\}\overline{select_i}(j).\ SSync_i \right.$

  $\left. + \sum_{j \in k \wedge j>i} \overline{m_{ij}}.\ \{3\}\overline{select_i}(j).\ SSync_i \right)$

Note that we have two labels with the value 3 as they are effectively the same state.

§5.3 Process state representations

In order to reason about the relationships between the states of the individual pollers, a convenient representation for a poller's state must be developed. A similar representation for a simple synchroniser's state is also needed to ease the expression of the simulation relation which we develop shortly. The notation developed so far only allows us to denote a poller in its initial state. We would like to be able to write an expression denoting a poller in any of its possible states. We could have made the locations explicit by breaking the poller definition into a number of small parts and then either giving them all unique names or, more conveniently, introducing an extra variable to distinguish between the states. However, such approaches would obscure the structure of the algorithm, and so we will define a new behaviour, p_i, that denotes a poller and its associated buffer, and will include the state information as part of its definition.

Definition $p_i(n_i,k_i,s_i,l_i,t_i)$

denotes the derivative of $(Poller_i(1,\phi)|Buffer_i)\backslash set_i$ where the current value of n, k and s in $Poller_i$ are n_i, k_i and s_i. $Poller_i$ is at location l_i and t_i is either equal to $\perp$, in which case the buffer is in state $Buffer_i$, or else t_i is equal to some j in which case the buffer is in state $Buffer_i'(j)$.

Thus p_i allows us to denote any possible derivative of the component

$$(Poller_i(1,\phi) \mid Buffer_i)\backslash set_i.$$

We extend this notation to allow us to denote any derivative of the complete system as follows.

Definition $\mathrm{PSys}(\tilde{n},\tilde{k},\tilde{s},\tilde{l},\tilde{t}) \equiv \ldots \mid p_i(n_i,k_i,s_i,l_i,t_i) \mid \ldots$

The state of each component of the network will be dependent on the states of one or more of the other components in the system. We will express these constraints by means of predicates on the component states. Given a state of the system, $PSys(\tilde{n},\tilde{k},\tilde{s},\tilde{l},\tilde{t})$, we can define a predicate $P_i(n,k,s,l,t)$ on the ith component as follows.

Definition $PSys(\tilde{n},\tilde{k},\tilde{s},\tilde{l},\tilde{t}) \models P_i(n,k,s,l,t)$

holds for any i iff $n = n_i$, $k = k_i$, $s = s_i$, $l = l_i$ and $t = t_i$

Note that if $PSys(\tilde{n},\tilde{k},\tilde{s},\tilde{l},\tilde{t}) \models P_i(n,k,s,l,t)$ holds then the ith component of the system must be $p_i(n,k,s,l,t)$.
Often we shall write $P_i(n,k,s,l,t)$ in place of $PSys(\tilde{n},\tilde{k},\tilde{s},\tilde{l},\tilde{t}) \models P_i(n,k,s,l,t)$ when the parameters of the intended system are clear from the context.

We introduce a similar set of definitions for the simple synchroniser system, i.e.

Definition $s_i(k_i,l_i,j_i)$

denotes the derivative of $SSync_i$ where the current value of k in $SSync_i$ is k_i, $SSync_i$ is at location l_i and j_i is either equal to $\perp$ when the synchroniser is not at location 3, or else j_i is equal to the value currently bound to j.

$$SSys(\tilde{k},\tilde{l},\tilde{j}) \equiv \ \ldots \ | \ s_i(k_i,l_i,j_i) \ | \ \ldots$$

$SSys(\tilde{k},\tilde{l},\tilde{j}) \models S_i(k,l,j)$ holds iff $k = k_i$, $l = l_i$ and $j = j_i$

§5.4 The relationship between individual pollers

Before exploring the relationship between the poller network and the simple synchroniser network, a closer look at how the individual pollers interact with each other would be advisable. We study this interaction by means of the following two theorems.

Definition $PSys(\tilde{n},\tilde{k},\tilde{s},\tilde{l},\tilde{t})$ is an <u>accessible state</u>

iff it is a derivative of the initial state $PSys(\tilde{1},\tilde{\emptyset},\tilde{\perp},\tilde{1},\tilde{\perp})$

Theorem 5.1

Every accessible state $\mathrm{PSys}(\tilde{n},\tilde{k},\tilde{s},\tilde{l},\tilde{t})$
satisfies the following implications,
i.e. they are invariant properties of the system.

i) $P_i(n,k,s,1,t) \supset t=\bot \wedge \forall j.\neg \mathit{partner}_j(i) \wedge \neg(k=\phi \wedge s\neq\bot)$

iia) $P_i(n,k,s,2,PC_i[n]) \supset s=\bot \wedge \forall j.\neg \mathit{partner}_j(i)$

iib) $P_i(n,k,s,2,\bot) \supset s=\bot \wedge \forall j\neq m.\neg \mathit{partner}_j(i) \wedge PC_m[n']=i \wedge$
$\left(P_m(n',k',\bot,3,\bot) \vee P_m(n',k',s',5,\bot)\right)$
$\underline{\text{where}}\ m=PC_i[n]$

iii) $P_i(n,k,s,3,t) \supset t=\bot \wedge s=\bot \wedge \forall j\neq m.\neg \mathit{partner}_j(i) \wedge$
$P_m(n',k',\bot,2,\bot) \wedge PC_m[n']=i\ \underline{\text{where}}\ m=PC_i[n]$

iv) $P_i(n,k,s,4,t) \supset t=\bot \wedge \forall j.\neg \mathit{partner}_j(i)$

v) $P_i(n,k,s,5,t) \supset t=\bot \wedge \forall j\neq m.\neg \mathit{partner}_j(i) \wedge$
$P_m(n',k',\bot,2,\bot) \wedge PC_m[n']=i\ \underline{\text{where}}\ m=PC_i[n]$

where $\mathit{partner}_i(j) \equiv \left(\exists n,k,s,l,t.\ P_i(n,k,s,l,t) \wedge (l=3 \vee l=5)\right) \supset PC_i[n]=j$

Proof:

To prove the theorem we need to show that the initial state satisfies the invariants and every transition preserves the invariants.

Initially every component is of the form $p_i(1,\phi,\bot,1,\bot)$ and therefore $P_i(1,\phi,\bot,1,\bot)$ is true. The consequences of i) are trivially true and none of the other invariants are applicable so this satisfies the first part of the proof.

For every possible transition of one of the pollers we must check that in the new state the relevant consequences will be true and additionally, we must make sure that none of the other consequences have been falsified without their antecedents also being falsified.

We consider all the transitions of an arbitrary component $p_i(n,k,s,l,t)$.

Case 1, $P_i(n,k,s,1,t)$ is true.

We know that $t=\perp$. Suppose $k=\phi$. Then the system may perform an *offer*$_i$(k') action and evolve to a state where $P_i(n,k',s,1,\perp)$ is true. The consequences of i) remain unchanged for component i and the transition does not effect any of the other processes and hence the invariants still hold in the new state.

Suppose $s\neq\perp$. Then we may output a *select*$_i$(s) message which results in a state satisfying $P_i(n+1,\phi,\perp,1,\perp)$. Again it is easy to verify that the invariants hold in the new state.

If $k\neq\phi\wedge s=\perp\wedge j\in k$ then there are two possibilities depending on whether i>j or i<j. If i>j then we may perform a τ move which internally sends a *set*$_i$(j) to *Buffer*$_i$ resulting in a state satisfying $P_i(n,k,\perp,2,PC_i[n])$. The consequences of iia) are true and the transition effects nothing else.

If i<j then we may receive a Q_{ji} message with either the value *YES* or the value *NO*. If *NO* is received the system evolves to a state satisfying $P_i(n+1,k,\perp,1,\perp)$ which still preserves the invariants. If we receive a reply with the value *YES*, the system evolves to a state satisfying $P_i(n,k,\perp,3,\perp)$. For this last transition to occur, $P_j(n',k',\perp,2,PC_j[n'])$ must have been true before the transition where $PC_j[n']=i$. Therefore, after the transition, $P_j(n',k',\perp,2,\perp)$ will hold and it is easy to verify that the invariants iib) and iii) hold for j and i respectively.

If $j\notin k$ or $s\neq i$ then there are two more possibilities depending on the relative values of i and j. If i>j then we may perform a τ move to a state satisfying $P_i(n+1,k,s,1,\perp)$ and if i<j then we may perform a τ move to a state satisfying $P_i(n,k,s,4,\perp)$ which satisfies the consequences of iv). •

Finally, as $t=\perp$, we may always issue a Q_{ij} message with the value *NO* to any poller that requests it. This transition leaves the process unchanged. However, the remote poller may either request it from a state satisfying $P_j(n',k',\perp,1,\perp)$ which has already been dealt with, or it may request it from a state satisfying $P_j(n',k',s,4,\perp)$. In this case the remote process evolves to a state satisfying $P_j(n'+1,k',s,1,\perp)$ which preserves the invariants.

This exhaustively deals with all transitions that involve the ith component of the system when $P_i(n,k,s,1,t)$ holds for some n,k,s and t.

Case 2a, $P_i(n,k,s,2,PC_i[n])$ is true.

We know that $s=\perp$. A component that satisfies this predicate cannot receive an A_{ji} message as this would imply that p_j (where $j=PC_i[n]$) satisfied $P_j(n',k',s,3,t)$ or $P_j(n',k',s,5,t)$ where $PC_j[n']=i$. However, these possibilities are ruled out by the invariance conditions. Therefore the only possible transition that involves this component is a response to a Q_{ij} query.
If $j \neq PC_i[n]$ then we respond with the value *NO* and the situation is identical to the analysis in Case 1. In fact, in every state there may be the possibility of sending a negative response to a Q_{ij} query. In each case the analysis is identical, and so when examining the rest of the states we will ignore this possibility.
If $j=PC_i[n]$ then we respond with *YES* and progress to a state satisfying $P_i(n,k,\perp,2,\perp)$. p_j may have previously been in a state satisfying either $P_j(n',k',\perp,1,\perp)$, in which case the new state will satisfy $P_j(n',k',\perp,3,\perp)$, or $P_j(n',k',s',4,\perp)$, in which case the new state will satisfy $P_j(n',k',s',5,\perp)$. In either case, $i=PC_j[n']$ and so the invariants are preserved.

Case 2b, $P_i(n,k,s,2,\perp)$ is true.

We can deduce from the invariants that $s=\perp$ and p_j satisfies either $P_j(n',k',\perp,3,\perp)$ or $P_j(n',k',s',5,\perp)$, where $j=PC_i[n]$ and $PC_j[n']=i$. In the first case, the two pollers may communicate evolving to states satisfying $P_i(n,k,PC_i[n],1,\perp)$ and $P_j(n',k',PC_j[n'],1,\perp)$. In the second case, the two processes evolve to states satisfying $P_i(n+1,k,\perp,1,\perp)$ and $P_j(n'+1,k',\perp,1,\perp)$. In both cases it is simple to verify that i) holds for i and j and that no other invariants have been effected.

Case 3, $P_i(n,k,s,3,t)$ is true.

We may deduce from the invariants that $t=\perp$, $s=\perp$ and p_j is in a state satisfying $P_j(n',k',\perp,2,\perp)$ where $j=PC_i[n]$ and $i=PC_j[n']$. This possibility was analysed in 2b).

Case 4, $P_i(n,k,s,4,t)$ is true.

We may deduce that $t=\perp$ from the invariants. The case where we receive a Q_{ji} message with the value *NO* has already been dealt with. If we receive a *YES* response then p_j must have been in a state satisfying $P_j(n',k',\perp,2,i)$ where $j=PC_i[n]$ and this possibility was analysed in 2a).

Case 5, $P_i(n,k,s,5,t)$ is true.

From the invariants we may deduce that $t=\perp$ and p_j is in a state satisfying $P_j(n',k',\perp,2,\perp)$ where $j=PC_i[n]$ and $i=PC_j[n']$. This possibility was analysed in 2b).

This completes our case analysis and proves that the invariants are preserved by all transitions. □

The next theorem formalises our intuitions that a poller will always be able to cycle around its communication partners without becoming deadlocked waiting for a response from the environment. This theorem was not true for our original presentation of the system as a poller may have been prevented from interacting with its partners because it was waiting to output a *select* message.

Theorem 5.2

Let $PSys(\tilde{n},\tilde{k},\tilde{s},\tilde{r},\tilde{t})$ represent a possible derivative of the system where $PSys(\tilde{n},\tilde{k},\tilde{s},\tilde{r},\tilde{t}) \models P_i(n,k,s,1,t)$ holds.

Then for all $1 \leq m \leq |PC_i|$,

$$PSys(\tilde{n},\tilde{k},\tilde{s},\tilde{r},\tilde{t}) \xRightarrow{\varepsilon} PSys(\tilde{n}',\tilde{k}',\tilde{s}',\tilde{r}',\tilde{t}')$$

such that $PSys(\tilde{n}',\tilde{k}',\tilde{s}',\tilde{r}',\tilde{t}') \models P_i(m,k,s',1,\perp)$ holds for some s'.

Proof:

By inspection of the text of a poller in conjunction with Theorem 5.1, we see that the only communication that can stop a poller progressing from a state satisfying $P_i(n,k,s,l,t)$ to one satisfying $P_i(n+1,k,s',1,\perp)$ is when it is waiting for an A_{ji} message that never arrives. All the other communications are either optional (the *offer* and *select* messages), or involve synchronising with processes that are guaranteed to be able to reply.

We prove the theorem by using induction on the index i.

Induction basis, i=1
In this case, there exists no j such that i>j and so $Poller_i$ never waits for an A_{ji} message to be output from another poller. Therefore the theorem holds.

Inductive step, we assume the theorem is true for all j, $1 \le j < i$.
The only case that might cause problems is when $Poller_i$ is waiting at position {2} for an A_{ji} message from $Poller_j$, where i>j. But then, by the inductive hypothesis, $Poller_j$ can eventually move to a state satisfying $P_j(n',k',s',1,\perp)$ where $PC_j[n']=i$. At this point, the two processes have the opportunity of communicating, thus freeing $Poller_i$ to move onto the next state. □

§5.5 The relationship between pollers and simple synchronisers

We are now in a position where we can attempt to show that the poller network is an implementation of the simple synchroniser network. We could approach this task in a number of ways. The most direct scheme would be to prove that $SSys \sqsubseteq_w PSys$ using the definition of $\sqsubseteq_w$. However, this task is made difficult by the need to quantify over all tests. We could try to show that $SSys \sqsubseteq_k PSys$, which is perhaps an easier task as some form of simulation could be used. Finally, we could try to show that SSys>PSys or $SSys >_t PSys$. Unfortunately, for reasons that will become clearer later in the chapter, it is not true that $SSys >_t PSys$. If we try to use $>$ or $\sqsubseteq_k$ we risk the possibility of not being able to relate the result to $\sqsubseteq_w$. The technique that we have currently proposed to relate $\sqsubseteq_w$ and $\sqsubseteq_k$, namely controllability or $\sqsubseteq_k$-determinacy, is too strong a requirement for our example, although as controllability is a sufficient, but not essential condition for establishing a connection, there may be other ways of completing the proof. Fortunately, a small alteration to the transformation function is sufficient to allow $>_t$ to be successfully applied and this result will directly imply the result for $\sqsubseteq_w$. This proof is contained in Section 5.6. Although it appears highly probable that the current version of the poller algorithm is a valid implementation of the simple synchroniser network, this cannot be proved until a less restrictive condition than controllability is found that still allows us to deduce $p \sqsubseteq_w q$ form $p \sqsubseteq_k q$. However, we start by proving that SSys>PSys, and hence SSys$\sqsubseteq_k$PSys, in the hope that such a connection will eventually be established.

5.5.1 A proof that SSys > PSys

The first step is to show that the prooler network is a refinement of the simple synchroniser network.

Theorem 5.3 SSys($\tilde{\phi},\tilde{1},\tilde{\perp}$) > PSys($\tilde{1},\tilde{\phi},\tilde{\perp},\tilde{1},\tilde{\perp}$)

Proof:

We construct a set $\mathcal{R}$ and then show that it forms a valid simulation relation for $>$.

$$\begin{array}{ll}
\mathcal{R} = & \{ <\mathrm{SSys}(\tilde{k}_s,\tilde{1}_s,\tilde{j}_s),\mathrm{PSys}(\tilde{n}_p,\tilde{k}_p,\tilde{s}_p,\tilde{1}_p,\tilde{t}_p)> \mid \\
\text{(c1a)} & \forall i.\big[\ (P_i(n,\phi,\perp,1,\perp) \wedge S_i(\phi,1,\perp)) \\
\text{(c1b)} & \quad \vee\ (P_i(n,k,\perp,1,\perp) \wedge k \neq \phi \wedge S_i(k,2,\perp)) \\
\text{(c1c)} & \quad \vee\ (P_i(n,k,s,1,\perp) \wedge s \neq \perp \wedge S_i(k,3,s)) \\
\text{(c2a)} & \quad \vee\ (P_i(n,k,\perp,2,t) \wedge ((t \neq \perp) \vee P_j(n',k',s',5,\perp)\ \underline{\text{where}}\ j=PC_i[n]) \wedge \\
 & \quad\quad S_i(k,2,\perp)) \\
\text{(c2b)} & \quad \vee\ (P_i(n,k,\perp,2,\perp) \wedge P_j(n',k',\perp,3,\perp) \wedge S_i(k,3,j)\ \underline{\text{where}}\ j=PC_i[n]) \\
\text{(c3a)} & \quad \vee\ (P_i(n,k,\perp,3,\perp) \wedge S_i(k,3,PC_i[n])) \\
\text{(c4a)} & \quad \vee\ (P_i(n,k,\perp,4,\perp) \wedge k \neq \phi \wedge S_i(k,2,\perp)) \\
\text{(c4b)} & \quad \vee\ (P_i(n,k,s,4,\perp) \wedge s \neq \perp \wedge S_i(k,3,s))
\end{array}$$

(c4c) $\vee\ (P_i(n,\phi,\perp,4,\perp) \wedge S_i(\phi,1,\perp))$

(c5a) $\vee\ (P_i(n,k,\perp,5,\perp) \wedge k\neq\phi \wedge S_i(k,2,\perp))$

(c5b) $\vee\ (P_i(n,k,s,5,\perp) \wedge s\neq\perp \wedge S_i(k,3,s))$

(c5c) $\vee\ (P_i(n,\phi,\perp,5,\perp) \wedge S_i(\phi,1,\perp))\ ]\}$

We factor the proof into two parts. We first show that for any pair <SSys,PSys> in $\mathcal{R}$, if PSys $\xrightarrow{\mu}$ PSys' then SSys $\overset{\mu}{\Longrightarrow}$ SSys', where <SSys',PSys'> is also an element of $\mathcal{R}$. We then show that for each pair <SSys,PSys> in $\mathcal{R}$, if SSys <u>must</u> L for some L then PSys <u>must</u> L as well. This allows us to concentrate on one aspect of the preorder at a time, and in addition, it will allow us to use the first part of the proof as the basis of a proof that SSys$>_t$PSys', where PSys' is the modified version of PSys mentioned at the beginning of this section.

Initially, we must check that for any pair <SSys,PSys> in $\mathcal{R}$, if PSys $\xrightarrow{\mu}$ PSys' then SSys $\overset{\mu}{\Longrightarrow}$ SSys' where <SSys',PSys'>$\in\mathcal{R}$. To simplify the analysis of internal transitions, we need only look at those transitions that alter the state of a process. For example, if a *Buffer* process outputs a *NO* value, the state of the associated component does not change. We therefore rely on the change of state at the destination process to trigger any analysis we may have to perform. If the state of the destination process remains unaltered as well then this is still acceptable as SSys $\overset{\tau}{\Longrightarrow}$ SSys trivially and the resulting pair is obviously in $\mathcal{R}$. To reduce the case analysis further, for the internal τ moves we need only check one side of the communication since at that point we examine all the possible configurations of the remote process. We start by performing a case analysis on each possible transition of an arbitrary component that involves a change in its state.

1a) $P_i(n,\phi,\perp,1,\perp) \wedge S_i(\phi,1,\perp)$

Suppose $p_i(n,\phi,\perp,1,\perp) \xrightarrow{offer_i(k)} p_i(n,k,\perp,1,\perp)$.

There are two possibilities.

If $k=\phi$ then $s_i(\phi,1,\perp) \overset{offer_i(k)}{\Longrightarrow} s_i(\phi,1,\perp)$

and if $k\neq\phi$ then $s_i(\phi,1,\perp) \overset{offer_i(k)}{\Longrightarrow} s_i(k,2,\perp)$.

In either case, the resulting pairs are in $\mathcal{R}$.

We may also perform a τ move to $p_i(n+1,\phi,\perp,1,\perp)$

or to $p_i(n,\phi,\perp,4,\perp)$.

In either of these cases component s_i can remain unchanged.

1b) $P_i(n,k,\perp,1,\perp) \wedge k\neq\phi \wedge S_i(k,2,\perp)$

There are four possible moves we are interested in depending on whether $i<j$ and $j\in k$.

a) $i>j \wedge j\in k$

$p_i(n,k,\perp,1,\perp) \xrightarrow{\tau} p_i(n,k,\perp,2,PC_i[n])$

which satisfies (c2a)

b) $i<j \wedge j\in k$

There are two possibilities;

either $p_i(n,k,\perp,1,\perp) \mid p_j(n',k',s',l',t') \xrightarrow{\tau}$

$p_i(n+1,k,\perp,1,\perp) \mid p_j(n',k',s',l',t')$ where $j=PC_i[n]\wedge t\neq i$

or $p_i(n,k,\perp,1,\perp) \mid p_j(n',k',s',l',i) \xrightarrow{\tau}$

$p_i(n,k,\perp,3,\perp) \mid p_j(n',k',s',l',\perp)$ where again $j=PC_i[n]$

In the first case, the state effectively remains unchanged.

In the second case, Theorem 5.1 implies

that $l'=2$ and $s'=\perp$.

We can therefore deduce from $\mathcal{R}$ that $S_j(k',2,\perp)$ was true, where $i\in k'$. But

$$s_i(k,2,\perp) \mid s_j(k',2,\perp) \overset{\tau}{\Longrightarrow} s_i(k,3,j) \mid s_j(k',3,i)$$

and the resulting pair satisfies the requirements for membership of $\mathcal{R}$.

c) $i>j \wedge j\notin k$

$p_i(n,k,\perp,1,\perp) \xrightarrow{\tau} p_i(n+1,k,\perp,4,\perp)$ and so

the state effectively remains unchanged.

d) $i<j \wedge j\notin k$

$p_i(n,k,\perp,1,\perp) \xrightarrow{\tau} p_i(n,k,\perp,4,\perp)$ which satisfies (c4a).

1c) $P_i(n,k,s,1,\perp) \wedge s\neq\perp \wedge S_i(k,3,s)$

From Theorem 5.1, we know that $k\neq\phi$.

There are three possibilities

a) $p_i(n,k,s,1,\perp) \xrightarrow{\overline{select}_i(s)} p_i(n+1,\phi,\perp,1,\perp)$

But $s_i(k,3,s) \xRightarrow{\overline{select}_i(s)} s_i(\phi,1,\perp)$.

b) $p_i(n,k,s,1,\perp) \xrightarrow{\tau} p_i(n+1,k,s,1,\perp)$ if $i>j$

c) $p_i(n,k,s,1,\perp) \xrightarrow{\tau} p_i(n,k,s,4,\perp)$ if $i<j$.

2a) $P_i(n,k,\perp,2,t) \wedge ((t\neq\perp) \vee P_j(n',k',s',5,\perp)$ $\underline{\text{where}}$ $j=PC_i[n]) \wedge S_i(k,2,\perp)$

Suppose $t\neq\perp$. Then by Theorem 5.1, the only transition that we are interested in is if p_j reads the Q message where $j = PC_i[n]$.

There are two possibilities depending on which branch of $Poller_j$ requests the communication.

a) $j<i$, $P_j(n',k',\perp,1,\perp) \wedge i\in k'$

This case has been dealt with in 1b) part b).

b) $j<i$, $P_j(n',k',s',4,\perp) \wedge i\notin k'$

Then $p_i(n,k,\perp,2,j) \mid p_j(n',k',s',4,\perp) \xrightarrow{\tau}$

$p_i(n,k,\perp,2,\perp) \mid p_j(n',k',s',5,\perp)$.

But before the transition $S_j(k',3,s')$, $S_j(k',\perp,2)$ or $S_j(\phi,1,\perp)$ must have been true depending on the value of s' and k, and therefore after the transition the states will still match with $P_j(n',k',s',5,\perp)$

Let us now assume that $t=\perp$ so $P_j(n',k',s',5,\perp)$ is true where

$j=PC_i[n]$.

Then $p_i(n,k,\perp,2,\perp) \mid p_j(n',k',s',5,\perp) \xrightarrow{\tau}$

$p_i(n+1,k,\perp,1,\perp) \mid p_j(n'+1,k',s',1,\perp)$

2b) $P_i(n,k,\perp,2,\perp) \wedge P_j(n',k',\perp,3,\perp) \wedge S_i(k,3,j)$ <u>**where**</u> $j=PC_i[n]$

Then we can use Theorem 5.1 and $\mathcal{R}$ to deduce that $S_j(k',3,PC_j[n'])$ and $PC_j[n']=i$ must be true.

$p_i(n,k,\perp,2,\perp)|p_j(n',k',\perp,3,\perp) \xrightarrow{\tau} p_i(n,k,j,1,\perp)|p_j(n',k',i,1,\perp)$

and the resulting pairs are in $\mathcal{R}$

3a) $P_i(n,k,\perp,3,\perp) \wedge S_i(k,3,PC_i[n])$

Then by Theorem 5.1, $P_j(n',k',\perp,2,\perp)$ and $PC_j[n']=i$ must be true. This case has been covered in 2b)

4a) $P_i(n,k,\perp,4,\perp) \wedge k\neq\phi \wedge S_i(k,2,\perp)$

There are two possibilities depending on the value received with the Q_{ji} communication.

If the value received is *NO* then the following transition must have taken place

$p_i(n,k,\perp,4,\perp) \mid p_j(n',k',s',l',t') \xrightarrow{\tau} p_i(n+1,k,\perp,1,\perp) \mid p_j(n',k',s',l',t')$

where $j=PC_i[n] \wedge t'\neq i$.

If a *YES* value is received then by Theorem 5.1, $P_j(n',k',\perp,2,i)$ must be true. This case has been covered in 2a).

4b) $P_i(n,k,s,4,\perp) \wedge s\neq\perp \wedge S_i(k,3,s)$

Again there are two possible transitions and the analysis is similar to 4a).

4c) $P_i(n,\phi,\perp,4,\perp) \wedge S_i(\phi,1,\perp)$

This case is essentially the same as 4a)

5a) $P_i(n,k,\perp,5,\perp) \wedge k\neq\phi \wedge S_i(k,2,\perp)$
Then by Theorem 5.1, $P_j(n',k',\perp,2,\perp) \wedge S_j(k',2,\perp) \wedge k'\neq\phi$ must be true where $j=PC_i[n]$.
This case has been dealt with in 2a).

5b) $P_i(n,k,s,5,\perp) \wedge s\neq\perp \wedge S_i(k,3,s)$
and
5c) $P_i(n,\phi,\perp,5,\perp) \wedge S_i(\phi,1,\perp)$
can be treated as in 5a).

This completes our case analysis and hence the first stage of the proof.

We must now check all the pairs in $\mathcal{R}$ to make sure that if $<SSys,PSys> \in\mathcal{R}$ and SSys <u>must</u> L for some L then PSys <u>must</u> L also. W.l.g. we may assume that L is a minimal set, i.e. $\nexists L'\subset L$ s.t. SSys <u>must</u> L'. We perform an induction based on the cardinality of L. If L is empty then SSys <u>must</u> ϕ is not possible and this forms our base case. For each λ in L we show that if $PSys \overset{\varepsilon}{\Longrightarrow} PSys'$ then either $PSys' \overset{\lambda}{\Longrightarrow}$, or else $SSys \overset{\varepsilon}{\Longrightarrow} SSys'$ such that $<SSys',PSys'>\in\mathcal{R}$ and $SSys' \overset{\lambda}{\not\Longrightarrow}$. In this case SSys' <u>must</u> L' for some $L'\subset L$ where $\lambda\notin L'$. We assume inductively that PSys' <u>must</u> L' and so $\exists\gamma\in L'$ s.t. $PSys' \overset{\gamma}{\Longrightarrow}$ and $\gamma\in L$. The induction relies on the fact that each τ derivative must eventually be able to perform one of the actions in L as otherwise we would reach the base case which is not possible.

If L is a minimal set it can only be composed of *offer* and *select* actions. Furthermore, SSys must be in a state where it can potentially perform these actions. We treat each possibility in turn, and for each action consider the possible states the synchroniser components, and hence the poller components, may be in.

1. $offer_i(j)\in L$ for some i,j.
   Then for this action to be possible, $S_i(\phi,1,\perp)$ must be true. Using the relation $\mathcal{R}$, there are three cases where this is

possible; when $P_i(n,\phi,\perp,1,\perp)$, $P_i(n,\phi,\perp,4,\perp)$ or $P_i(n,\phi,\perp,5,\perp)$ is true. Using Theorem 5.2 and the fact that k remains empty until an offer is selected, we know that all τ sequences starting from PSys can be extended to a PSys' such that PSys' $\models P_i(n',\phi,\perp,1,\perp)$. But $p_i(n',\phi,\perp,1,\perp) \xrightarrow{offer_i(j)} p_i(n',j,\perp,1,\perp)$.

2. $\overline{select_i}(j) \in L$ for some i,j.
   Then $S_i(k,2,\perp)$ or $S_i(k,3,j)$ is true where $j \in k$.
   We treat the two cases separately starting with the simpler case.

   a. $S_i(k,3,j)$ is true
      Let us enumerate the possible states of p_i.

      i. $P_i(n,k,j,1,\perp)$
      ii. $P_i(n,k,\perp,2,\perp) \wedge P_j(n',k',\perp,3,\perp)$ where $j=PC_i[n]$
      iii. $P_i(n,k,\perp,3,\perp) \wedge j=PC_i[n]$
      iv. $P_i(n,k,j,4,\perp)$
      v. $P_i(n,k,j,5,\perp)$

      We know that irrespective of what the rest of the system does, if ii) or iii) is true then p_i can progress to a state where $P_i(n,k,j,1,\perp)$, i.e. i) is true. If iv),v) or i) is true then by Theorem 5.2 and the fact that the s field is only cleared after a select message, we know that eventually we can get to a state where i) is true. Finally, it is immediately apparent that

$$p_i(n,k,j,1,\perp) \xrightarrow{\overline{select_i}(j)} p_i(n+1,\phi,\perp,1,\perp)$$

   b. $S_i(k,2,\perp)$ is true where $j \in k$
      This is the case where the induction is required because at the point where $S_i(k,2,\perp)$ is true, SSys is not committed to the select action. It must be possible for the action to occur but the system may also non-deterministically choose another alternative if one is available. Let us start by enumerating the possible states of p_i.

i. $P_i(n,k,\perp,1,\perp)$
ii. $P_i(n,k,\perp,2,t) \wedge ((t \neq \perp) \vee (P_m(n',k',s',5,\perp))$ where $m=PC_i[n]$
iii. $P_i(n,k,\perp,4,\perp)$
iv. $P_i(n,k,\perp,5,\perp)$

By Theorem 5.2 we know that after any sequence of τ moves it will always be possible to get to a state where $P_i(n',k',s',1,\perp)$ is true where $PC_i[n']=j$.
Furthermore, we know that k cannot change until we have output a *select*$_i$ message and so k'=k. Suppose $s' \neq \perp$. If s'=j then we are able to output a *select*$_i$(j) message. What if $s' \neq j$? Then at some point in our sequence of τ either p_i was in a state satisfying $P_i(m,k,\perp,2,\perp)$ and $p_{s'}$ was in a state satisfying $P_{s'}(m',k'',\perp,3,\perp)$, or vice versa, where $i=PC_{s'}[m']$ and $s'=PC_i[m]$. In either case this implies that $S_{s'}(k'',2,\perp)$ must have been true in SSys. But then

$$s_i(k,2,\perp) \mid s_{s'}(k'',2,\perp) \xrightarrow{\tau} s_i(k,3,s') \mid s_{s'}(k'',3,i)$$

and the resulting SSys' cannot output a *select*$_i$(j) action. Furthermore, $p_{s'}$ must now be in a state satisfying $P_{s'}(m'',k'',i,l,\perp)$ where l=1, 4 or 5, and in all of these cases the resulting pairs of new states are in $\mathcal{R}$. Therefore our inductive hypothesis allows us to deduce that this case is correct.
If $s'=\perp$ then there are two cases to be considered depending on whether i is less than or greater than j.

Case i<j

In this case the system can wait until it receives a Q_{ji} message. If the value *YES* is returned then the system will eventually reach a state which satisfies $P_i(n'',k,j,1,\perp)$ which will then be able to output a *select*$_i$(j) message. If a *NO* message is received then it may be because *Poller*$_j$ has not yet reached a state where it wishes to communicate with *Poller*$_i$. In this case, the poller can continue until either a *YES* answer is returned to the Q_{ji} query, or a *NO* value is returned due to p_j being in a state such that the system satisfies $P_j(m,k'',s',l,\perp)$ where $s' \neq \perp$. In the second case we may reason that p_j must have arranged this communication during the sequence of τ moves as otherwise the *select*$_i$(j) message would not have been possible right from the start. By a similar analysis to the case where $s' \neq \bot$, we may reason that s_i also had the potential to communicate with $s_{s'}$ and so we may employ the inductive hypothesis.

Case i>j

The analysis is similar to the previous case. Either the poller receives a positive A_{ji} reply, in which case it may then output a *select*$_i$(j) message, or else a negative response is received which implies that *Poller*$_j$ has negotiated a communication with some other process in which case we can show that this is also a possibility in SSys.

This completes our proof and shows that $\mathcal{R}$ is a valid simulation relation and hence that SSys($\tilde{\phi},\tilde{\top},\tilde{\bot}$) > PSys($\tilde{\top},\tilde{\phi},\tilde{\bot},\tilde{\top},\tilde{\bot}$). □

From the previous result, and Theorem 4.25, we may deduce that

$$\mathrm{SSys}(\tilde{\phi},\tilde{\top},\tilde{\bot}) \sqsubseteq_k \mathrm{PSys}(\tilde{\top},\tilde{\phi},\tilde{\bot},\tilde{\top},\tilde{\bot})$$

We must now show how to extend this result to the $\sqsubseteq_w$ preorder. Unfortunately, our efforts in this direction have so far proved to be unsuccessful. Chapter 4 showed one way of deducing $p \sqsubseteq_w q$ from $p \sqsubseteq_k q$ if p is $\simeq_k$-determinate. In our particular example, this would be equivalent to showing that the system SSys($\tilde{\phi},\tilde{\top},\tilde{\bot}$) is $\simeq_k$-determinate. However a simple example illustrates that it is not. Suppose we have the following situation.

$$\begin{tikzcd}
 & S_1 \arrow[dl, no head] \arrow[dr, no head] & \\
S_2 & & S_3
\end{tikzcd}$$

where we assume that we have already issued the messages

$\overline{offer_2}(\{1\})$ and $\overline{offer_3}(\{1\})$.

If we now output an $offer_1(\{2,3\})$ message then the system may arrange a communication between either s_1 and s_2 or between s_1 and s_3. In the first case, the system will then be willing to output a $select_1(2)$ message and in the second case, a $select_1(3)$ message will be offered. These two possible states are therefore not related by $\simeq_k$ and so the system is not $\simeq_k$-determinate.

$\sqsubseteq_k$ and $\sqsubseteq_w$ for the most part treat processes identically. Therefore it is reasonable to hope that there may be other constraints that we may impose on processes **p** and **q** that let us deduce $\mathbf{p} \sqsubseteq_w \mathbf{q}$ from $\mathbf{p} \sqsubseteq_k \mathbf{q}$, but our searches in that direction have so far proved to be unproductive. An alternative approach might be to restrict the class of observers that $\sqsubseteq_k$ is defined over in an attempt to bring the two preorders even closer. However, given that $\sqsubseteq_w$ is preserved by | whereas $\sqsubseteq_k$ is not, it seems that restricting the observer class would only provide part of the answer if we wanted to preserve the properties of $\sqsubseteq_w$.

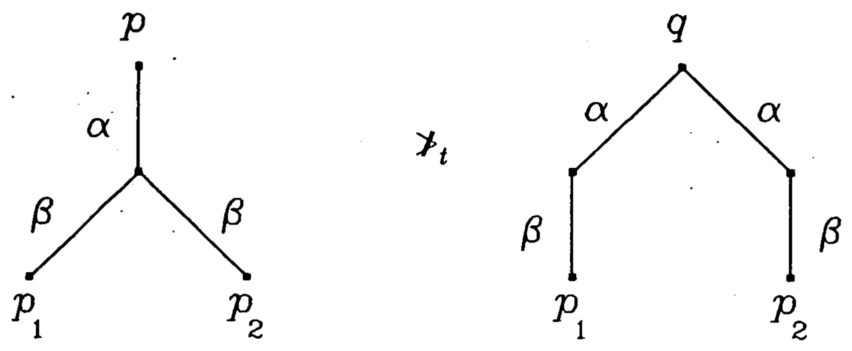
§5.6 A proof that SSys $>_t$ PSys'

Chapter 4 introduced one other way of proving $\mathbf{p} \sqsubseteq_w \mathbf{q}$ indirectly. This approach used the preorder $>_t$. If we could show that

$$\mathrm{SSys}(\tilde{\phi},\tilde{\top},\tilde{\bot}) >_t \mathrm{PSys}(\tilde{\top},\tilde{\phi},\tilde{\bot},\tilde{\top},\tilde{\bot})$$

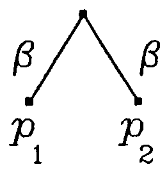
then we could directly infer the required result. However, there is a problem that occurs when trying to use $>_t$. This is due to the fact that

(†)

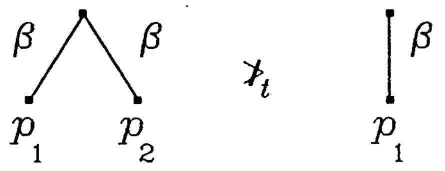


if $Traces(p_1)=Traces(p_2)$

To see why, consider what happens when **q** performs an α move. Let us assume that it takes the left-hand branch. Then process **p** after an α move reaches a state



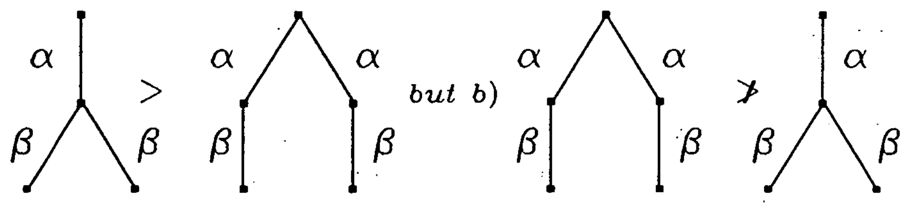
and



as the possible traces are different.

Note that with the relation > the following is true

(‡) *a*)



The proof for SSys > PSys exploited this fact in the following way. When a process is in a state satisfying $P_i(n,k,\perp,2,PC_i[n])$, then at that point it may be committed to a certain action even though we have equated it to $S_i(k,2,\perp)$. The communication we are committed to depends on the states of the communicating partners and their relative orders. The reason we were successful in proving that SSys > PSys was because, although the $s_i(k,2,\perp)$ term was uncommitted, it had no control over what communication would eventually occur and so the situation was similar to (‡). However (†) shows that this technique will not work for $>_t$.

In order to use the $>_t$ preorder, two alternatives are open to us. The first approach would be to equate a term satisfying $P_i(n,k,\perp,2,PC_i[n])$ to a term satisfying $S_i(k,3,j)$ for some j, if a global analysis of the state indicated that $Poller_i$ was committed to establishing a communication with $Poller_j$ due to the order in which $Poller_i$ polled its partners. If we did this, the commitment point in the simple synchronisers and the pollers would then be identical as far as the relation $\mathcal{R}$ was concerned. Unfortunately, this approach would greatly complicate what is already a lengthy proof.

The first approach can be viewed as moving the point of commitment of the simple synchronisers so as to coincide with that of the pollers. The other approach is in some sense the opposite. We can modify the algorithm so that the commitment point of the pollers coincides with the commitment point of the simple synchronisers. If we allow the component p_i, when in the state $(n,k,\perp,2,PC_i[n])$, to spontaneously "give up" and revert back to a state satisfying $P_i(n+1,k,\perp,1,\perp)$, then it allows the p_i component complete freedom of choice until it reaches the state $(n,k,j,1,\perp)$ or $(n,k,\perp,3,\perp)$, where in either case the equivalent state of s_i is also committed.

Obviously, by taking such an approach, we accept the criticism that we are changing the problem to suit the proof. However, in this case we believe the approach is justified for the following reasons. In the first place, there appears to be no other way of completing the proof using the $>_t$ preorder without a great deal of additional complexity. We feel that this cannot be justified while there is a possibility that some relation between $\sqsubseteq_k$ and $\sqsubseteq_w$ can be found that allows us to complete the original and simpler proof. Secondly, the modification gives a very similar effect to a variant of the original algorithm where, instead of incrementing the index into PC_i each time, we choose the next index at random. As long as we have a random number generator that guarantees that each process will eventually be polled, then this random version of the scheme effectively performs identically to the original version. Furthermore, the random version and our proposed modification have very similar behaviours. For these reasons we choose the second approach.

One way of implementing our proposed change is by replacing $Buffer_i$ by the definition in Figure 5-3.

$$\begin{aligned}
NBuffer_i &= \sum_{j\in PC_i}^{\circ} \big(\ \overline{Q_{ij}}(NO).\ NBuffer_i \\
&\qquad + set_i(\mathrm{j}).\ NBuffer'_i(\mathrm{j})\ \big)
\end{aligned}$$

$$\begin{aligned}
NBuffer'_i(\mathrm{k}) &= \Big(\ \sum_{j\in PC_i-\{k\}} \overline{Q_{ij}}(NO).\ NBuffer'_i(\mathrm{k})\Big) \\
&\qquad + \overline{Q_{ik}}(YES).\ NBuffer_i \\
&\qquad + \overline{A_{ki}}(NO).\ NBuffer_i
\end{aligned}$$

Figure 5-3: A modified version of the Buffer process

It is easy to check that the new modification has not created any additional synchronisation problems. Furthermore, the additional transition we have introduced does not effect Theorem 5.1, or perhaps more accurately, the extra check for this transition can be added to the proof of Theorem 5.1 without changing the rest of the proof.

In order to prove that

$$\mathrm{SSys}(\tilde{\emptyset},\tilde{\top},\tilde{\bot}) >_t \mathrm{PSys'}(\tilde{\top},\tilde{\emptyset},\tilde{\bot},\tilde{\top},\tilde{\bot}),$$

where PSys' represents the modified poller network, we use the same simulation relation $\mathcal{R}$ as before. Because Theorem 5.1 still holds, and the first part of the definition of $>_t$ is identical to that for $>$, we may use the first part of the previous proof without change, except to verify that the additional transition causes no problems. We are then left with the problem of checking that for each pair $<SSys,PSys> \in \mathcal{R}$ that $Traces(SSys) \subseteq Traces(PSys)$.

We first prove the following theorem.

Theorem 5.4

Let PSys represent a derivative of $PSys(\tilde{T},\tilde{\phi},\tilde{\bot},\tilde{T},\tilde{\bot})$.

Then if PSys $\models$

(i) $P_i(n,k,\bot,1,\bot)$

(ii) $\vee\ P_i(n,k,\bot,2,t) \wedge ((t \neq \bot) \vee P_m(n',k',s',5,\bot))$
where $m = PC_i[n]$

(iii) $\vee\ P_i(n,k,\bot,4,\bot)$

(iv) $\vee\ P_i(n,k,\bot,5,\bot)$

holds then $\forall n'$ s.t. $j = PC_i[n'] \wedge j \in k \wedge PSys \models P_j(n'',k',s,l,t)$
$\exists PSys'$ such that $PSys \overset{\varepsilon}{\Longrightarrow} PSys'$ and

$$PSys' \models P_i(n',k,\bot,1,\bot) \wedge P_j(n'',k',s,l,t)$$

Proof:

We assume that for any process, m, in PC_i where $m \neq j$, if it reaches a state $p_m(n',k',\bot,2,PC_m[n'])$ then it immediately performs a transition to a state $p_m(n'+1,k',\bot,1,\bot)$.

Then if iii) or iv) is true, we may perform a sequence of τ moves to a state satisfying $P_i(n+1,k,\bot,1,\bot)$. If ii) is true we may again perform a transition to $P_i(n+1,k,\bot,1,\bot)$. If i) is true then we can always progress to the next index, either by timing out or by relying on the fact that any partners other than $Poller_j$ will have timed out and so will return *NO* to a Q_{ji} query. Therefore we will eventually get to a state $P_i(n',k,\bot,1,\bot)$ and this sequence of moves does not require any participation by $Poller_j$ and so its state will have remained unchanged. □

We can now show that $SSys(\tilde{\phi},\tilde{\top},\tilde{\bot}) >_t PSys'(\tilde{\top},\tilde{\phi},\tilde{\bot},\tilde{\top},\tilde{\bot})$.

Theorem 5.5 $SSys(\tilde{\phi},\tilde{\top},\tilde{\bot}) >_t PSys'(\tilde{\top},\tilde{\phi},\tilde{\bot},\tilde{\top},\tilde{\bot})$

Proof:

The first part of the proof has already been discussed. All that remains is to check that for all pairs <SSys,PSys> in $\mathcal{R}$, Traces(SSys)$\subseteq$Traces(PSys). We do this by showing that for all sequences s in Traces(SSys), s also exists in Traces(PSys). We use induction on the length of s.

<u>Inductive base</u>, $s=\varepsilon$

Then trivially $\varepsilon \in$Traces(PSys)

<u>Inductive step</u>, $s=\alpha s'$

What possible values can α have? It must either be an *offer* or a *select* message and we treat these two cases separately.

1. $\alpha = offer_i(k)$

   In order for this action to form part of a trace of SSys, the action must be possible which implies that $S_i(\phi,1,\bot)$ holds. This is possible when $P_i(n,\phi,\bot,1,\bot)$, $P_i(n,\phi,\bot,4,\bot) \vee P_i(n,\phi,\bot,5,\bot)$ holds. We may use Theorem 5.2 to deduce that eventually we may progress to a state $P_i(n',\phi,\bot,1,\bot)$ where the transition

   $$p_i(n',\phi,\bot,1,\bot) \xrightarrow{offer_i(k)} p_i(n',k,\bot,1,\bot) \text{ is possible.}$$

   If $k=\phi$ then

   $$s_i(\phi,1,\bot) \xrightarrow{offer_i(k)} s_i(\phi,1,\bot)$$

   and otherwise

   $$s_i(\phi,1,\bot) \xrightarrow{offer_i(k)} s_i(k,2,\bot).$$

   In either case the resulting pairs are in $\mathcal{R}$ so we may use the inductive hypothesis to show that s'$\in$Traces(PSys') where PSys' represents the state after the transition.

2. $\alpha = \overline{select_i}(j)$

   In order for this action to be possible we know that either $S_i(k,3,j)$ or $S_i(k,2,\bot)$ is true. In the first case, we may perform the same analysis as for > to deduce that

$$PSys \xRightarrow{\overline{select}_i(j)} PSys' \text{ and } SSys \xRightarrow{\overline{select}_i(j)} SSys'$$

where $<SSys',PSys'>\in\mathcal{R}$ and so again we may use the inductive hypothesis on s'.

If $S_i(k,2,\perp)$ is true then there are two possibilities.

a. i>j

In this case, by Theorem 5.4, we may eventually reach a state $P_i(n',k,\perp,1,\perp)$ where $j\in k$ and $j=PC_i[n']$. Furthermore, this sequence of transitions doesn't effect $Poller_j$. $Poller_i$ then outputs a $set_i(j)$ action and waits for a reply from $Poller_j$. We can apply Theorem 5.4 to deduce that $P_j(n'',k',\perp,1,\perp)$ will eventually hold, where $i\in k'$ and $i=PC_j[n']$. At this point it can receive a positive Q_{ij} response which means that the system will eventually be able to output a $select_i(j)$ message. Furthermore, the resulting pairs are in $\mathcal{R}$ and so we may apply the inductive hypothesis to s'.

b. i<j

The analysis is similar to the previous case except that $Poller_j$ waits for $Poller_i$.

This completes our proof that Traces(SSys)$\subseteq$Traces(PSys) and hence we may deduce that $SSys(\mathfrak{F},\mathfrak{T},\mathfrak{I}) >_t PSys'(\mathfrak{T},\mathfrak{F},\mathfrak{I},\mathfrak{T},\mathfrak{I})$ □

§5.7 A proof that Tr_{Sync} is an implementation transformation

Before proving that $\mathbf{Tr}_{\mathbf{Poll}}$ is an implementation transformation we first show this property for $\mathbf{Tr}_{\mathbf{Sync}}$. We can then extend the result to $\mathbf{Tr}_{\mathbf{Poll}}$ in a simple way.

To show that $\mathbf{Tr}_{\mathbf{Sync}}$ is an implementation we must show that

$$\prod_{i\in N} \mathbf{p}_i \ \underline{\textit{w-must succeed}} \supset \mathbf{Tr}_{\mathbf{Sync}}\left[\!\left[\prod_{i\in N} \mathbf{p}_i \right]\!\right] \underline{\textit{w-must succeed}}$$

We have previously represented the state of a simple synchroniser by $s_i(k,l,j)$, where k is the set of processes we are trying to synchronise with, j is the identity of a process we have successfully established a communication with (or $\perp$), and l represents the current position in the algorithm. When we place the simple synchroniser in parallel with its host, we need to extend the state to keep track of the process from which this transformation was derived. This extension will be described by means of the following example.

Suppose we have a process $\mathbf{p}_i$ and we translate it to obtain the component $s_i(\phi,1,\perp,\mathbf{p}_i)$, where we have added an extra field, $\mathbf{p}_i$, to indicate the source of the transformed term. Then a communication performed by the original process with process $\mathbf{p}_k$ is translated into the following sequence of actions.

$$\mathbf{p}_i = \sum_{j\in n_i} a_j \cdot \mathbf{p}_{ij} \quad \text{and} \quad \mathbf{p} \xrightarrow{a_m} \mathbf{p}_{im}$$

$$s_i(\phi,1,\perp,\mathbf{p}_i) \xrightarrow{\tau}$$

$$s_i\Big(\bigcup_{j\in n_i} C(a_j)\cup\{i\},\ 2,\perp,\mathbf{p}_i\Big) \xrightarrow{m(i,k)}$$

$$s_i\Big(\bigcup_{j\in n_i} C(a_j)\cup\{i\},\ 3,k,\mathbf{p}_i\Big) \xrightarrow{\tau}$$

$$s_i(\phi,4,\perp,a_m\cdot\mathbf{p}_{im}) \xrightarrow{a_m}$$

$$s_i(\phi,1,\perp,\mathbf{p}_{im})$$

where $C(a_m)=k$

and $m(i,k) = m_{ki}$ if $k<i$ and $\overline{m_{ik}}$ if $k>i$.

This sequence is matched by a similar one in the translation of $\mathbf{p}_k$. We extend the possible locations from 3 to 4 where 4 indicates that the master has not yet performed the synchronised action.

It would be useful if we could extend each process into a known state. Furthermore, we would like this known state to be equivalent to some transformation of a process. A transformation currently starts in a state where all the processes are at location {1}. However, we cannot always extend a computation so that all processes are in this state because a process may already be at location {2} with no possibility of communicating with another process. We will show that it is always possible to extend a computation so that all processes are at location {2}. When a process is at location {1} then it can progress silently to location {2}, so that case presents no problems. If it is at location {3} then this implies that it has just performed an m(k,i) transition which in turn means that s_k has just performed an m(k,i) transition. They may both independently proceed to location {4}, and due to the definition of *C*, they are guaranteed to be able to communicate with each other again in order to reach location {1} which silently brings the component back to location {2}.

The discussion in the previous paragraph leads us to assume that the transformation function results in a state where the initial communication between the host and the simple synchroniser has already taken place. This is equivalent to performing the initial move at compile time instead of at run time, and doesn't alter any of our previous results concerning the transformation.

Altering the starting point of the transformation allows us to prove the following proposition.

Proposition 5.6

$$\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} \mathrm{p}_i]\!] \overset{\varepsilon}{\Longrightarrow} \mathrm{tp} \supset \exists\ \tilde{\mathrm{p}}' \text{ s.t. } \mathrm{tp} \overset{\varepsilon}{\Longrightarrow} \mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} \mathrm{p}_i']\!]$$

Proof:

Immediate as the transformation function starts with each process s_i in a state satisfying $S_i(k,2,\bot,p_i)$ and we have shown that every state can be extended to this form in a computation, so in particular, all the processes in tp may be extended so that they are equivalent to the transformation of some $\prod_{i\in N} p_i'$. □

However, we will need a stronger result than this, namely that the $\prod_{i\in N} p_i'$ processes can be obtained from $\prod_{i\in N} p_i$ by a sequence of silent moves. In other words, we wish to prove the following proposition.

Proposition 5.7

$$\mathrm{Tr}_{\mathrm{Sync}}[\![\prod_{i\in N} p_i]\!] \overset{\varepsilon}{\Longrightarrow} tp \supset \exists\ \tilde{p}' \text{ s.t. } tp \overset{\varepsilon}{\Longrightarrow} \mathrm{Tr}_{\mathrm{Sync}}[\![\prod_{i\in N} p_i']\!]$$

and furthermore, $\prod_{i\in N} p_i \overset{\varepsilon}{\Longrightarrow} \prod_{i\in N} p_i'$

Proof:

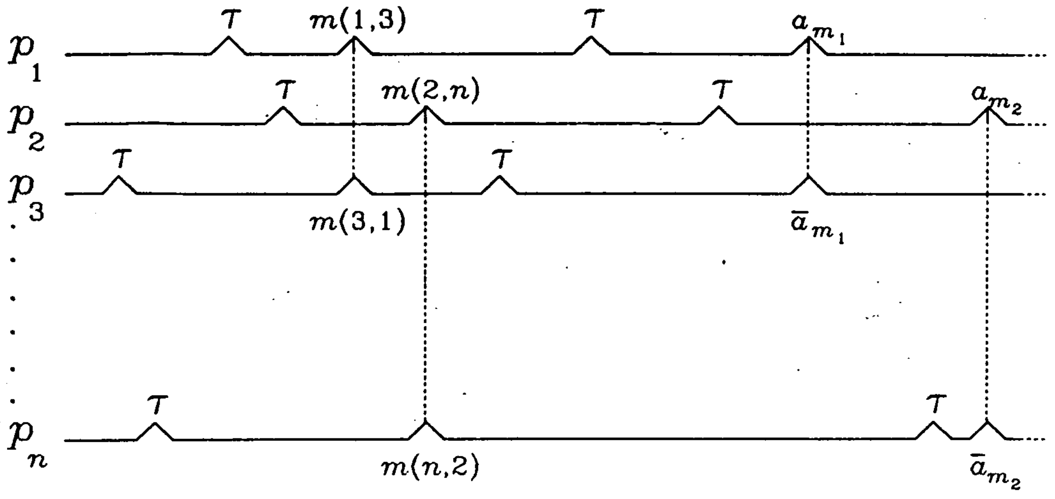
Suppose

$$\mathrm{Tr}_{\mathrm{Sync}}[\![\prod_{i\in N} p_i]\!] \overset{\varepsilon}{\Longrightarrow} \mathrm{Tr}_{\mathrm{Sync}}[\![\prod_{i\in N} p_i']\!].$$

Let σ_i be the sequence of actions contributed by component i. In other words, the set $\{\sigma_i | i\in N\}$ can be merged in some way to form an ε sequence. Each sequence σ_i consists of a number of subsequences each of the form

$$\xrightarrow{\tau} \xrightarrow{m(i.k)} \xrightarrow{\tau} \xrightarrow{{}^{B}m}$$

The situation may be viewed as follows



We define the function f on sequences such as these as follows

$$f(\tau.m(i,k).\tau.a_m.s) = a_m.f(s)$$
$$f(NIL) = NIL$$

Let us denote by $\mathbf{Tr}_{\mathbf{Sync}_i}$ the transformation applied to the ith component. If $\mathbf{Tr}_{\mathbf{Sync}_i}[\![\mathbf{p}_i]\!] \xrightarrow{\sigma_i} \mathbf{Tr}_{\mathbf{Sync}_i}[\![\mathbf{p}_i']\!]$ then $\mathbf{p}_i \xrightarrow{f(\sigma_i)} \mathbf{p}_i'$.

Furthermore, by examining the effect of f on our previous diagram, we may conclude that the set of sequences $\{f(\sigma_i)|i\in N\}$ can also be merged to form an ε sequence. Therefore we may deduce that

$$\prod_{i\in N} \mathbf{p}_i \overset{\varepsilon}{\Longrightarrow} \prod_{i\in N} \mathbf{p}_i' \qquad \Box$$

By examination of the transformation function, we may deduce that

$$\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} \mathbf{p}_i]\!] \xrightarrow{\surd} \Leftrightarrow \prod_{i\in N} \mathbf{p}_i \xrightarrow{\surd}$$

We may also extend this to our last proposition. If the transformation passes through a state where a $\surd$ move is possible on its way from

$$\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} \mathbf{p}_i]\!] \text{ to } \mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} \mathbf{p}_i']\!]$$

then the path from

$$\prod_{i\in N} p_i \text{ to } \prod_{i\in N} p_i'$$

also has this possibility, and vice versa.

We now have to show that every communication that was possible in the original system may be mirrored in the transformed system. This is the purpose of the next proposition.

Proposition 5.8

$$\prod_{i\in N} p_i \xrightarrow{\tau} \prod_{i\in N} p_i' \supset \mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} p_i]\!] \overset{\tau}{\Longrightarrow} \mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} p_i']\!]$$

Proof:

If $\prod_{i\in N} p_i \xrightarrow{\tau} \prod_{i\in N} p_i'$ then there exists a pair p_i, p_j such that

$$p_i \xrightarrow{a} p_i',\ p_j \xrightarrow{\bar{a}} p_j' \text{ and}$$

$$\prod_{i\in N} p_i = p_1 \mid \ldots \mid p_i \mid \ldots \mid p_j \mid \ldots \mid p_n \text{ and}$$

$$\prod_{i\in N} p_i' = p_1 \mid \ldots \mid p_i' \mid \ldots \mid p_j' \mid \ldots \mid p_n$$

Consider $\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} p_i]\!]$. Then $S_i(k_i,2,\perp,p_i)$ and $S_j(k_j,2,\perp,p_j)$ hold where $i\in k_j$ and $j\in k_i$. Therefore these two processes may evolve to $s_i(k_i,3,j,p_i)$ and $s_j(k_j,3,i,p_j)$. These may then separately evolve to $s_i(\phi,4,\perp,a.p_i')$ and $s_j(\phi,4,\perp,\bar{a}.p_j')$ because, due to our assumptions about C, there is a unique port between s_i and s_j. These two processes may then communicate to produce $s_i(\phi,1,\perp,p_i')$ and $s_j(\phi,1,\perp,p_j')$. Finally, both of these processes may move independently to $s_i(k_i',2,\perp,p_i')$ and $s_j(k_j',2,\perp,p_j')$ where k_i' and k_j' contain the set of processes that p_i' and p_j' wish to communicate with. The rest of the components have remained unchanged and so the resulting system is equivalent to $\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i\in N} p_i']\!]$. □

We are now able to show that $\mathbf{Tr}_{\mathbf{Sync}}$ is an implementation transformation.

Theorem 5.9

$$\prod_{i \in N} \mathbf{p}_i \ \underline{\textit{w-must succeed}} \supset \mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i \in N} \mathbf{p}_i]\!] \ \underline{\textit{w-must succeed}}$$

Proof:

Suppose that this is not the case. Then there is a prefix of a computation of $\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i \in N} \mathbf{p}_i]\!]$ which cannot be extended to a successful state, i.e.

$$\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i \in N} \mathbf{p}_i]\!] \overset{\varepsilon}{\Longrightarrow} \text{tp s.t. } \nexists \text{tp}'.\ \text{tp} \overset{\varepsilon}{\Longrightarrow} \text{tp}' \xrightarrow{\checkmark}$$

By Proposition 5.7 we know that there exists a $\prod_{i \in N} \mathbf{p}_i'$ such that

$$\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i \in N} \mathbf{p}_i]\!] \overset{\varepsilon}{\Longrightarrow} \text{tp} \overset{\varepsilon}{\Longrightarrow} \mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i \in N} \mathbf{p}_i']\!].$$

where $\prod_{i \in N} \mathbf{p}_i \overset{\varepsilon}{\Longrightarrow} \prod_{i \in N} \mathbf{p}_i'$

But then there are two possibilities. Either $\prod_{i \in N} \mathbf{p}_i$ passed through a successful state on the way to $\prod_{i \in N} \mathbf{p}_i'$, and therefore $\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i \in N} \mathbf{p}_i]\!]$ must have also passed through a successful state, or

$$\prod_{i \in N} \mathbf{p}_i' \overset{\varepsilon}{\Longrightarrow} \prod_{i \in N} \mathbf{p}_i'' \xrightarrow{\checkmark}$$

But then by the last proposition,

$$\mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i \in N} \mathbf{p}_i']\!] \overset{\varepsilon}{\Longrightarrow} \mathbf{Tr}_{\mathbf{Sync}}[\![\prod_{i \in N} \mathbf{p}_i'']\!] \xrightarrow{\checkmark}$$

In other words, we have achieved a contradiction. □

We have therefore proved that $\mathbf{Tr}_{\mathbf{Sync}}$ is an implementation transformation.

§5.8 A proof that $\mathrm{Tr}_{\mathrm{Poll}}$ is an implementation transformation

The end is in sight. We show that $\mathrm{Tr}_{\mathrm{Poll}}$ is an implementation transformation by showing that it is an implementation of $\mathrm{Tr}_{\mathrm{Sync}}$ and then appealing to transitivity.

The transformation function $\mathrm{Tr}_{\mathrm{Sync}}$ consists of two parts. The first translates the processes into their *offer/select* form, and the second part consists of the simple synchronisers. $\mathrm{Tr}_{\mathrm{Poll}}$ may be broken down in a similar way and, loosely speaking, we may view the two translations as follows.

$$\mathrm{Tr}_{\mathrm{Sync}}[\![\prod_{i\in N} p_i]\!] \equiv \left(\mathrm{Tr}_i[\![\prod_{i\in N} p_i]\!] \mid \mathrm{Tr}_{sy}[\![\prod_{i\in N} p_i]\!]\right)$$

$$\mathrm{Tr}_{\mathrm{Poll}}[\![\prod_{i\in N} p_i]\!] \equiv \left(\mathrm{Tr}_i[\![\prod_{i\in N} p_i]\!] \mid \mathrm{Tr}_{po}[\![\prod_{i\in N} p_i]\!]\right)$$

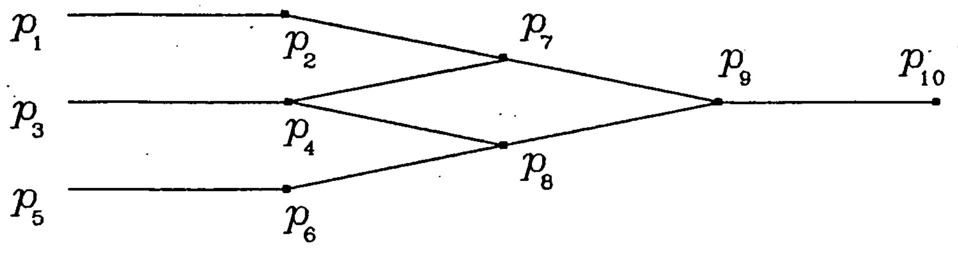
Suppose that $\mathrm{Tr}_{\mathrm{Sync}}[\![\prod_{i\in N} p_i]\!]$ <u>*w-must succeed*</u>.
Then $\left(\mathrm{Tr}_i[\![\prod_{i\in N} p_i]\!] \mid \mathrm{Tr}_{sy}[\![\prod_{i\in N} p_i]\!]\right)$ <u>*w-must succeed*</u>
and so $\left(\mathrm{Tr}_i[\![\prod_{i\in N} p_i]\!] \mid \mathrm{Tr}_{po}[\![\prod_{i\in N} p_i]\!]\right)$ <u>*w-must succeed*</u>
which implies that $\mathrm{Tr}_{\mathrm{Poll}}[\![\prod_{i\in N} p_i]\!]$ <u>*w-must succeed*</u>.

which is sufficient to show that $\mathrm{Tr}_{\mathrm{Poll}}$ is an implementation of $\mathrm{Tr}_{\mathrm{Sync}}$ and hence is an implementation transformation.

§5.9 Partial application of synchronisation transformations

One of the advantages of treating synchronisation schemes as program transformations lies in their ability to be partially applied to a system. We now investigate this possibility in more detail.

Let us consider again the example presented in Chapter 3, page 63.



PC	1	2	3	4	5	6	7	8	9	10
	$\{\{2\}\}$	$\left\{\begin{matrix}\{7\}\\\{1\}\end{matrix}\right\}$	$\{\{4\}\}$	$\left\{\begin{matrix}\{3\}\\\{7,8\}\end{matrix}\right\}$	$\{\{6\}\}$	$\left\{\begin{matrix}\{5\}\\\{8\}\end{matrix}\right\}$	$\left\{\begin{matrix}\{2,4\}\\\{9\}\end{matrix}\right\}$	$\left\{\begin{matrix}\{4,6\}\\\{9\}\end{matrix}\right\}$	$\left\{\begin{matrix}\{7,8\}\\\{10\}\end{matrix}\right\}$	$\{\{9\}\}$

The algorithm for determining synchronising annotations produced the following dominance relation for this system.

$$\begin{array}{lll} p_1 < p_2 & p_2 \# p_7 & p_9 < p_7 \\ p_3 < p_4 & p_4 \# p_7 & p_9 < p_8 \\ p_5 < p_6 & p_4 \# p_8 & p_9 < p_{10} \\ & p_6 \# p_8 & \end{array}$$

A fundamental property of the algorithm involved its treatment of the propagation of incomparable processes. If two processes **p** and **q** are attempting to communicate and **p#q** then these processes cannot be affected by any other communications between pairs of comparable processes. For example, suppose **q** could also simultaneously attempt to communicate with **r** and **r** with **s** where **r<s**. This might potentially influence the communication between **p** and **q** because one of the summands of **q** (with **r**) may be withdrawn due to **r** communicating with **s**. However, this situation is not possible because **p#q** and **q** can simultaneously attempt to communicate with **r** which implies **q#r** and hence **r#s**. As a result, all communications that take place between comparable processes are in some sense disjoint from those between the incomparable processes. This allows us to partially apply a transformation only to those processes that are incomparable to some other process. In the case of Schwarz' scheme, for example, even the transformed processes only need to communicate via a poller when

synchronising with a set of incomparable processes. For the comparable cases they may communicate directly with the other processes. Thus in the example above, p_1 could always communicate directly with its partners (in this case p_2) whereas p_4 must communicate with p_7 and p_8 via a poller although it can communicate directly with p_3.

The correctness proof for Schwarz' scheme still holds in the partial application case because the direct communications between comparable terms manifest themselves as silent moves among the hosts. If communications between comparable terms could influence the other communications then this would not be the case as a successful direct communication would potentially require the ability to send retraction messages to the pollers which we have not considered.

Partial application of the transformation has obvious advantages. Implementing pollers, even when done in hardware, is an expensive process. It is therefore desirable to omit unnecessary uses of them. Furthermore, this approach has more general applicability. Other synchronisation schemes may also be expressed as transformations and partially applied. The success of this approach relies on natural limits to the propagation of incomparable processes in the dominance relation. One reason for expecting this to be the case is that many programs satisfy the restrictions imposed by the asymmetric version of CSP, for example. For these programs no incomparable processes will be necessary, although some may be generated due to the simplifications assumed by the current algorithm. We expect in the more general case that programs will contain a sizable subset that will have no incomparable processes. These subsets will be connected together by more elaborate synchronisation nets containing incomparable processes at the interfaces.

CHAPTER 6
Conclusion and further work

The aim of this thesis has been to study the problems associated with the implementation of concurrent languages based on the synchronous handshaking view of process communication. These problems arise not only in the design of the implementation algorithms themselves, but also in the associated proofs of these algorithms. The language chosen as the basis of these investigations was Milner's Calculus of Communicating Systems. CCS was an appropriate vehicle for this research due to the well developed body of proof techniques and equivalences that already existed for the language. Because of the unusual nature of the proofs conducted in this thesis, however, the previous work on CCS has had to be extended to accommodate fairness and transformational correctness in an intuitive fashion.

The theoretical problems tackled in this thesis have been problem-driven to a great extent. The development of the weak-must preorder, and the definitions of implementation and transformation, were prompted by the desire to perform a correctness proof for a CCS implementation. Such a problem-driven approach has the advantage that the theoretical investigations are well motivated. Furthermore, the particular problems under consideration may drive the investigations in directions that may not otherwise be contemplated. The problem-driven approach also has its drawbacks. There is a danger that the theoretical work is incomplete and the results may also be of relevance to only a small class of problems. The quest for intuitively appealing definitions may also lead to equivalences that are mathematically intractable. While we do not believe that these potential drawbacks of the problem-driven approach are applicable in this case, the implications of the work presented in the thesis need to be examined in more detail from a theoretical standpoint. Even if the definitions themselves do not gain wide acceptance, it is hoped that the motivation behind the choice of the definitions will influence future work in this area.

We stated in the introduction that an important use of concurrent programming constructs was as a structuring tool in program design. This prompted the work in Chapter 2 where the Edinburgh version of the language PFL was described. It was argued that this was a natural way of extending CCS to a complete programming language, and preliminary feedback from the teaching of this language, both in Edinburgh and Goteborg, reinforces this view. The chapter also pointed out the scope for future enhancements, including the need for a more sophisticated user interface, and the possibility of extending the language with features from other languages such as Synchronous CCS, MEIJE and CIRCAL. Care must be taken in PFL to create an acceptable impression of non-determinism to the user. The additional manipulations required to achieve this complicate the implementation of the concurrent primitives. To what extent these measures are necessary in a large PFL program needs to be investigated. The preliminary work on PFL has been encouraging and we believe that further work on single processor implementations of CCS, not necessarily based on PFL, should be encouraged.

Chapter 3 demonstrated the problems involved in providing a distributed implementation of Static CCS. A subset of Static CCS based on synchronising annotations was identified, and an efficient implementation strategy based on this subset outlined. A method was proposed for computing these annotations automatically under certain, simplifying, assumptions. The results are limited to Static CCS (and CSP). The more general case of CCS has not been considered for simplicity. The effect of these results in the presence of dynamic process creation therefore needs to be investigated.

A new approach to the implementation of process synchronisation, based on program transformations, was also proposed in Chapter 3.

Some of the existing synchronisation schemes may be reformulated as program transformations, and by applying these transformations selectively to parts of the source program, an efficient implementation strategy may be achieved. The problems associated with proving these transformations correct led to the theoretical investigations of Chapter 4. The *weak-must* testing equivalence was proposed as a way of incorporating fairness constraints into the *must* testing equivalence, while retaining the validity of the expansion theorem. The most obvious weakness of the new equivalence is the lack of an alternative characterisation that admits some form of bisimulation style proof technique. This omission is important as there currently does not exist a proof technique that is directly applicable to the *weak-must* testing equivalence. We hope that the motivation behind the introduction of this equivalence will be sufficiently appealing that others will also attempt to find such a characterisation, or propose alternative equivalences of a similar nature to $\simeq_w$. A connection was established between $\sqsubseteq_w$ and the $\sqsubseteq_k$ and $<$ preorders, under certain conditions, and perhaps these may form the basis of future searches in this direction. An important area for future research involves the development of transformations that produce, as output, programs that have no output guards in summations. This would allow the resulting programs to be run on existing implementations of languages such as CSP and OCCAM. Partial application of such transformations would allow the asymmetric nature of these languages to be hidden, although a performance penalty would obviously have to be paid. Program transformations could be developed that produced programs with bounds on their interconnection patterns. Such transformations would be useful for processors, such as the Transputer [INMOS 84b], where there are physical constraints on the interconnectivity of the processors which would otherwise create difficulties when mapping processes onto processors. The techniques presented in Chapter 4 may help in proving these transformations correct. The work presented in Chapters 3 and 4, especially the proposal to use program transformations as an aid to process synchronisation, and the development of the *weak-must* testing preorder, forms the major achievement of this thesis.

A new definition of implementation was presented, based on the *weak-must* testing preorder. Although this definition, along with the definition of transformation correctness, was motivated by the need to prove the Schwarz transformation scheme correct, these definitions are of far wider applicability. The areas to which these techniques may be applied will increase as larger proofs become more feasible, and the problems tackled become more complex.

There have been other approaches to the treatment of weak and strong fairness that have been introduced since the *weak-must* testing equivalence was defined. While these approaches do not currently respect the expansion theorem, they may form the basis of techniques that do, and these should then be compared with our approach. While we strongly believe that the expansion theorem should still hold when fairness constraints are taken into consideration, this is open to debate and further, more convincing, arguments should perhaps be developed to resolve this matter one way or the other.

Chapter 5 presented a comparatively large proof of the Schwarz transformation scheme. The problem raised a number of interesting issues concerning the proof methods that had to be employed. We believe these problems, and the techniques developed to treat them, are of a general significance. In particular, the need to conduct the proof without appealing to inductive arguments, and the notation developed to keep the proof manageable, may aid in the analysis of similar problems. Another beneficial aim of exhibiting such proofs is perhaps less obvious. In order to develop theorem provers for languages such as CCS, it is necessary to be able to identify the types of operations that need to be performed in a verification, and the detailed presentation of proofs is one way of helping this process.

In conclusion, the problems associated with implementing a language such as CCS have been analysed within a formal framework. Furthermore, new techniques have been developed to aid the implementation process and also to analyse the resulting synchronisation algorithms. Implementations of languages such as CCS are still in their infancy, as is their associated theoretical support. It is hoped that the work presented in this thesis will contribute in some small way to the growing process.

References

[Austry 84]	Austry, D., and Boudol, G. Algèbre de Processus et Synchronisation. *Theoretical Computer Science* (30):91-131, 1984.
[Backhouse 83]	Backhouse, R. *Specification and Proof of a Regular Language Recogniser in Synchronous CCS.* Technical Report CSR-130-83, Computer Science Department, Edinburgh University, 1983.
[Banino 79]	Banino, J.S., Kaiser, C., and Zimmermann, H. Synchronization for Distributed Systems using a Single Broadcast Channel. In *Proceedings of First International Conference on Distributed Computing Systems*, pages 330-338. Oct, 1979.
[Bernstein 80]	Bernstein, A.J. Output Guards and Nondeterminism in "Communicating Sequential Processes". *ACM Transactions on Programming Languages and Systems* 2(2), 1980.
[Brinch 73]	Brinch hansen, P. *Operating System Principles.* Prentice-Hall, 1973.
[Brinch 81]	Brinch Hansen, P. Edison - a multiprocessor language. *Software-Practice and Experience* 11, 1981.
[Buckley 83]	Buckley, G.N., and Silberschatz, A. An Effective Implementation for the Generalized Input-Output Construct of CSP. *ACM Transactions on Programming Languages and Systems* 5(2), 1983.
[Burstall 80]	Burstall, R.M., MacQueen, D.B., and Sannella, D.T. *HOPE: An Experimental Applicative Language.* Technical Report CSR-62-80, Computer Science Department, Edinburgh University, 1980.
[Cardelli 82]	Cardelli, L. *ML under Unix.* Technical Report, Bell Laboratories, Murray Hill, New Jersey, 1982.
[Costa 84]	Costa, G., and Stirling, C. A Fair Calculus of Communicating Systems. *Acta Informatica* 21:417-441, 1984.

[DeNicola 82] De Nicola, R. and Hennessy, M.C.B.
Testing Equivalences for Processes.
Technical Report CSR-123-82, Computer Science Department, Edinburgh University, 1982.

[Dijkstra 65] Dijkstra, E.W.
Cooperating Sequential Processes.
Technical Report, Technological University, Eindhoven, The Netherlands, 1965.

[DoD 80] *Reference Manual for the Ada programming Language*
United States Department of Defence, 1980.

[Engelfriet 84] Engelfriet, J.
Determinacy —> (Observation Equivalence = Trace Equivalence).
Technical Report, Twente University of Technology, January, 1984.

[Francez 80] Francez, N., and Rodeh, M.
A distributed data type implemented by a probabilistic communication scheme.
In *Proceedings of 21st Annual Symposium on Foundations of Computer Science*, pages 373-379. Syracuse, N.Y., Oct, 1980.

[Goldberg 83] Goldberg, A. and Robson, D.
Smalltalk-80: *The Language and its Implementation.*
Addison-Wesley, 1983.

[Gordon 79] Gordon, M.J., Milner: A.J. and Wadsworth, C.P.
Lecture Notes in Computer Science. Volume 78.*Edinburgh LCF.*
Springer-Verlag, 1979, Chapter 2.

[Gries 77] Gries, D.
An exercise in Proving Parallel Programs Correct.
CACM 20(12), 1977.

[Habermann 72] Habermann, A.N.
Synchronization of Communicating Processes.
CACM 15(3), 1972.

[Hennessy 84a] Hennessy, M.
Proving Systolic Systems Correct.
Technical Report CSR-162-84, Computer Science Department, Edinburgh University, 1984.

[Hennessy 84b] Hennessy, M.
An Algebraic Theory of Fair Asynchronous Communicating Processes.
Technical Report CSR-171-84, Computer Science Department, Edinburgh University, 1984.

[Hennessy 84c] Hennessy, M.
A Proof Technique for MUST-Testing Preorder.
1984.
Unpublished notes.

[Hewitt 77] Hewitt, C.
Viewing Controls Structures as Patterns of Passing Messages.
Artificial Intelligence 8(3), 1977.

[Hewitt 80] Hewitt, C.
The Apiary Network Architecture for Knowledgeable Systems.
In *Conference Record of the* 1980 *LISP Conference*. Stanford University, 1980.

[Hoare 78] Hoare, C.A.R.
Communicating Sequential Processes.
CACM 21(8), 1978.

[Holmstrom 83] Holmstrom, S.
PFL: A Functional Language for Parallel Programming.
In *Declarative Programming Workshop*. Programming Methodology Group, Chalmers University of Technology, University of Goteborg, Sweden, 1983.

[INMOS 84a] INMOS.
Computer Science: OCCAM Programming Manual.
Prentice-Hall, 1984.

[INMOS 84b] INMOS.
IMS T424 Transputer Reference Manual.
Technical Report, INMOS Limited, 1984.

[Kennaway 81] Kennaway, J.R.
Formal Semantics of Nondeterminism and Parallelism.
PhD thesis, St. John's College, Oxford University, 1981.

[Lamport 78] Lamport, L.
Time, Clocks, and the Ordering of Events in a Distributed System.
CACM 21(7), 1978.

[Larsen 85] Larsen, K.G.
A Context Dependent Equivalence between Processes.
In *To appear in the proceedings of ICALP*'85. 1985.

[Li 83] Li, W.
An Operational Approach to Semantics and Translation for Concurrent Programming Languages.
PhD thesis, Computer Science Department, Edinburgh University, 1983.

[Lynch 80] Lynch, N.A.
Fast Allocation of Nearby Resources in a Distributed System.
In *Proceedings of* 12*th Annual Symposium on Theory of Computing*. ACM, Los Angeles, Calif, 1980.

[Marsan 84] Marsan, M.A., Conte, G., and Balbo, G.
A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems.
ACM Transactions on Computer Systems 2(2), 1984.

[Martin 80] Martin, A.J.
A Distributed Implementation Method for Parallel Programming.
In S.H. Lavington (editor), *Information Processing* 80. 1980.

[Millington 82] Millington, M., and Hennessy, M.C.B.
Towards a Theory of Translation.
1982.
Unpublished Paper, Computer Science Department, Edinburgh University.

[Milne 85] Milne, G.J.
CIRCAL and the Representation of Communication, Concurrency, and Time.
ACM Transactions on Programming Languages and Systems 7(2), 1985.

[Milner 80] Milner, R.
Lecture Notes in Computer Science. Volume 92: *A Calculus of Communicating Systems.*
Springer-Verlag, 1980.

[Milner 83] Milner, R.
Calculi for Synchrony and Asynchrony.
Theoretical Computer Science (25):267-310, 1983.

[Milner 84] Milner, R.
The Standard ML Core Language.
Technical Report CSR-168-84, Computer Science Department, Edinburgh University, 1984.

[Mitchell 84] Mitchell, K.
A User's Guide to PFL.
1984.
Edinburgh Computer Science Department Lecture Notes.

[Mitchell 85] Mitchell, K. and Mycroft, A.
The Edinburgh Standard ML Compiler.
1985.
In preparation.

[Olderog 84] Olderog, E. and Apt, K.R.
Fairness in Parallel Programs: The Transformational Approach.
Technical Report 8402, Christian-Albrechts-Universitat, Kiel, July, 1984.

[Owicki 75] Owicki, S.
Axiomatic proof techniques for parallel programs.
PhD thesis, Department of Computer Science, Cornell University, 1975.

[Park 81] Park, D.J.
Concurrency and Automata on Infinite Sequences.
In *Lecture Notes in Computer Science* 104. Springer-Verlag, 1981.

[Parrow 84] Parrow, J., and Gustavsson, R.
Modelling Distributed Systems in an Extension of CCS with Infinite Experiments and Temporal Logic.
In *Fourth International Workshop on Protocol Specification, Testing, and Verification.* IFIP W66.1, June, 1984.

[Peterson 77] Peterson, J.L.
Petri Nets.
ACM Computing Surveys 3(9), 1977.

[Prasad 84] Prasad, K.V.S.
Specification and Proof of a Simple Fault Tolerant System in CCS.
Technical Report CSR-178-84, Computer Science Department, Edinburgh University, 1984.

[Reif 84] Reif, J.H., and Spirakis, P.G.
Real-time Synchronization of Interprocess Communications.
ACM Transactions on Programming Languages and Systems 6(2), 1984.

[Ron 84] Ron, D., Rosemberg, F. and Pnueli, A.
A Hardware Implementation of the CSP Primitives and its Verification.
In *ICALP* 84. Springer-Verlag, Lecture Notes in Computer Science 172, 1984.

[Sanderson 82] Sanderson, M.T.
Proof Techniques for CCS.
PhD thesis, Computer Science Department, Edinburgh University, 1982.

[Schneider 82] Schneider, F.B.
Synchronization in Distributed Programs.
ACM Transactions on Programming Languages and Systems 4(2), 1982.

[Schwarz 78] Schwarz, J.S.
Distributed Synchronisation of Communicating Sequential Processes.
Technical Report DAI TR 56, Artificial Intelligence Department, Edinburgh University, 1978.

[Shrira 81] Shrira, L., and Francez, N.
An Experimental Implementation of CSP.
In *Proceedings of 2nd International Conference on Distributed Systems*, pages 126-136. Paris, France, April, 1981.

[Silberschatz 79] Silberschatz, A.
Communication and Synchronization in Distributed Systems.
IEEE Transactions on Software Engineering SE-5(6), 1979.

[Silberschatz 81] Silberschatz, A.
Port directed communication.
The Computer Journal 24(1), 1981.

[Snepscheut 81] Van de Snepscheut, J.L.A.
Synchronous Communication between Asynchronous Components.
Information Processing Letters 13(3), 1981.

[Tarski 55] Tarski, A.
A Lattice-theoretical Fixpoint theorem and its Applications.
Pacific Journal of Mathematics 5, 1955.

[Wirth 77] Wirth, N.
Modula: a Language for Modular Multiprogramming.
Software-Practice and Experience 7, 1977.

Appendix A
The Original "Kennaway" Equivalence

This appendix describes the equivalence due to Kennaway as it was presented in [Kennaway 81]. We justify using the version due to DeNicola and Hennessy [DeNicola 82] by showing that the original definition is not an observational equivalence in any meaningful sense.

We start by defining a weak form of testing with a set of actions.

Definition For any finite $L \subseteq \mathcal{A}ct \cup \{\tau\}$.

$$p \ \underline{\underline{\text{must}}}\ L \iff \exists \mu \in L.\ p \overset{\mu}{\Longrightarrow}$$

$$P \ \underline{\underline{\text{must}}}\ L \iff \forall p \in P.\ p \ \underline{\underline{\text{must}}}\ L.$$

Note that τ may be an element of L. We also extend the definition of after to allow τ to be one of the admissible labels.

Definition For any $\mu \in \mathcal{A}ct \cup \{\tau\}$

$$p \ \underline{\underline{\text{after}}}\ \mu = \{p' \mid p \overset{\mu}{\Longrightarrow} p' \text{ for some } p'\}$$

$$P \ \underline{\underline{\text{after}}}\ \mu = \bigcup \{p \ \underline{\underline{\text{after}}}\ \mu \mid p \in P\}$$

Note that as a consequence of the definition of $\overset{\tau}{\Longrightarrow}$ in [Kennaway 81], $p \ \underline{\underline{\text{after}}}\ \tau \neq p \ \underline{\underline{\text{after}}}\ \varepsilon$ as $p \ \underline{\underline{\text{after}}}\ \tau$ is the set of processes that can be reached after one or more τ moves whereas $p \ \underline{\underline{\text{after}}}\ \varepsilon$ is the set of processes that can be reached after zero or more τ moves. Thus, in general, $p \in p \ \underline{\underline{\text{after}}}\ \varepsilon$ whereas $p \notin p \ \underline{\underline{\text{after}}}\ \tau$.

Kennaway then goes on to describe his equivalence $\approx_k$ by means of a recurrence relation.

$$P \approx_k^0 Q \text{ is always true.}$$

$$P \approx_k^{n+1} Q \Leftrightarrow \begin{array}{l} \text{i)}\ \ \forall \text{finite } L \subseteq \mathcal{A}ct \cup \{\tau\}.\ P\ \underline{\text{must}}\ L \supset Q\ \underline{\text{must}}\ L \\ \text{ii)}\ \ \forall \mu \in \mathcal{A}ct \cup \{\tau\}.\ P\ \underline{\text{after}}\ \mu \approx_k^n Q\ \underline{\text{after}}\ \mu \end{array}$$

$$P \approx_k Q \ \Leftrightarrow\ \forall n \geq 0.\ P \approx_k^n Q.$$

We extend this definition to single processes in the obvious way, i.e.

$$p \approx_k q \ \Leftrightarrow\ \{p\} \approx_k \{q\}.$$

Unfortunately, although this equivalence is intended to be weak, or observational, in the sense that $\alpha.\tau.p$ and $\alpha.p$ are indistinguishable, this is not so. To see why the definition differentiates between these terms, consider the following analysis.

$$\begin{array}{lcl} \alpha.\beta.NIL\ \underline{\text{after}}\ \alpha & = & \{\beta.NIL\} \\ \alpha.\tau.\beta.NIL\ \underline{\text{after}}\ \alpha & = & \{\tau.\beta.NIL,\ \beta.NIL\} \end{array}$$

$$\{\beta.NIL\}\ \underline{\text{after}}\ \tau \ = \ \phi$$
$$\{\tau.\beta.NIL,\ \beta.NIL\}\ \underline{\text{after}}\ \tau \ = \ \{\beta.NIL\}$$

We can therefore deduce that

$$((\alpha.\beta.NIL\ \underline{\text{after}}\ \alpha)\ \underline{\text{after}}\ \tau)\ \underline{\text{must}}\ \{\gamma\}$$

whereas

$$((\alpha.\tau.\beta.NIL\ \underline{\text{after}}\ \alpha)\ \underline{\text{after}}\ \tau)\ \not{\underline{\text{must}}}\ \{\gamma\}$$

and so $\alpha.\beta.NIL \not\approx_k^3 \alpha.\tau.\beta.NIL$.

Unfortunately if we restrict the definition of after to the case where μ cannot equal τ we still run into difficulties as the equivalence then equates

$$\alpha.NIL + \beta.NIL + \tau.NIL \quad \text{and} \quad \tau.\alpha.NIL + \beta.NIL$$

This is because the ability to specify "after τ" coupled with the definition of must acted as a substitute for must. Take away the ability to use "after τ" and the system breaks down.

The definition presented by DeNicola and Hennessy [DeNicola 82] was originally intended as an alternative definition of Kennaway's equivalence. However this appendix, along with the work in Chapter 5, has shown that the DeNicola version is not an alternative characterisation of the original definition but rather a corrected version that manages to avoid the deficiencies of the original definition.