Proof Methods and Pragmatics for Parallel Programming

Chris Tofts

Doctor of Philosophy
University of Edinburgh
1990

Abstract

We define a parallel extension of a standard imperative programming language, which we call CIMP. This language has a semantics in the process calculus CCS. In order to demonstrate that a sub-language of CIMP is deterministic the notion of a *semi-confluent* process is defined. This process structure is shown to be preserved by the appropriate equivalences of CCS, and to have a set of operators which conserve it. We show that a simple *syntactic* condition on the parallel components of a program is sufficient to ensure its determinacy. Furthermore we demonstrate that the deterministic sub-language of CIMP has a simple functional semantics. We also show that there is a set of simple transformation rules that can manipulate parallelism whilst maintaining determinism.

Two proof systems for the language CIMP are studied, they are based respectively on the approaches of Owicki-Gries and Jones. These systems are shown to be sound with respect to an interpretation of the proof rules in Hennessey–Milner logic. We demonstrate that a knowledge of determinism can greatly ease the proof burden in both of these methods.

We define a calculus, based on CCS, which permits reasoning about the temporal properties of concurrent systems. This calculus is shown to have natural notions of equality and order. Some example systems are included, in particular a temporal analysis of an *alternating bit protocol.* We use this calculus to provide a *timed semantics* for the language CIMP, and examine a timed proof system based on the system for sequential programs presented by Nielson.

Acknowledgements.

Firstly I would like to thank Robin Milner for his tolerance and effort, without which this thesis would both not have been completed and also been a much poorer document. Many useful remarks and suggestions were made on this work by Colin Stirling, David Walker and the other members of the Concurrency Club.

I should like to thank my friends for providing support and a willingness to discuss even the most outrageous ideas; especially my family, Carolyn, Douglas, Nikki, James, David, Neil, Shaun, Nicci and Simon.

Lastly I would like to thank Faroun 'mad' Mullah for his joyous attitude and misspelling my name.

This work was funded by a SERC PhD research grant.

Declaration.

I hereby declare that the work of this thesis has been composed by myself, and that the work reported has not been presented for any university degree before; with the exception of chapters one and two, which are reproductions of some material by other authors, and chapters six and seven, which are joint work with Robin Milner, except where the work is stated to be by Robin Milner.

Chris Tofts.

Table of Contents

1. Introduction. **8**

1.1 Overview. 8

1.1.1 The language CIMP 9

1.1.2 This Thesis 9

1.2 CCS 11

1.2.1 The Language 11

1.2.2 Operational Semantics. 12

1.2.3 Strong Bisimulation. 12

1.2.4 Observation. 14

1.2.5 Equational Theory. 16

1.2.6 Value Passing. 18

1.3 Semantics of CIMP. 20

1.3.1 Expression Translation. 20

1.3.2 Command Translation. 20

1.3.3 Process Logic. 22

2. Process Structure **25**

2.1 Introduction. 25

2.2 Determinacy. 26

2.3 Confluence. 28

2.4 Adding Values. 31

3. Semi-Confluence. 34

3.1 Introduction. 34

3.2 Partial Confluence. 35

3.3 Semi-Confluence. 36

3.3.1 Recursion. 51

3.4 Value Passing. 54

3.5 Conclusion 58

4. Determinism in CIMP. 59

4.1 Introduction. 59

4.1.1 Read and Write sets. 61

4.2 DCIMP is Deterministic. 62

4.3 Conclusions. 67

5. Denotational Semantics of DCIMP. 69

5.1 Introduction. 69

5.2 Denotational Semantics. 70

5.2.1 Expression Evaluation. 70

5.2.2 Command Evaluation. 70

5.3 Closure. 72

5.4 Intermediate Process Semantics. 73

5.4.1 Expression Evaluation. 73

5.4.2 Command Evaluation. 74

5.5 $< C > \approx < C >'$. 78

5.6 Final Equivalence. 81

5.6.1 Evaluation Equivalence. 81

5.6.2 Command Equivalence. 81

5.7 Determinism Preserving Transformations. 83

5.8 Conclusions. 87

6. Axiomatic Methods I. 88

6.1 Introduction. 88

6.1.1 Hoare's Logic. 89

6.2 Owicki-Gries 90

6.2.1 Adding Parallel Commands. 90

6.2.2 Example Proof. 92

6.3 DCIMP. 93

6.3.1 Translating Hoare Triples. 95

6.3.2 Soundness Proof 97

6.3.3 Non-Interference Implies Interference Freedom. 101

6.3.4 Adding Logical Variables. 103

6.4 Conclusions. 104

7. Axiomatic Methods II. 105

7.1 Introduction. 105

7.1.1 Stirling's Inference System. 105

7.2 Interpreting Quintuples. 108

7.3 Soundness. 110

7.4 Relation to Non-Interfering Methods. 118

7.5 Rewriting the Definitions in PL. 118

7.6 Conclusions. 119

8. Timing Concurrent Processes. 120

8.1 Introduction. 120

8.2 Language Definition. 121

8.2.1 Derivation Laws. 122

8.2.2 An Example. 122

8.2.3 Temporal Deadlock. 125

8.2.4 Deadlock. 126

8.3 Strong Time Sensitive Pre-order. 127

8.4 An Equality for sTCCS. 131

8.5 Temporal Structure independence 134

8.6 Equational Characterisation of sTCCS. 137

8.7 Observational Evolutions. 140

8.8 Process Logic For Timed CCS. 140

8.9 Examples. 142

8.9.1 A Simple Timer. 142

8.9.2 Action Available For a Period. 143

8.10 Value Passing. 144

8.11 The Alternating Bit Protocol. 144

8.12 Conclusions. 150

9. Timing Concurrent While Programs. 151

9.1 Introduction. 151

9.2 Timed Semantics for CIMP. 151

9.2.1 Expression Translation. 152

9.2.2 Command Translation. 153

9.3 Timed Hoare's Logic for TIMP. 158

9.4 Conclusions. 165

10. Conclusions and Further Work. 166

10.1 Conclusions. 166

10.2 Further Work. 168

10.2.1 Expressive Confluent Operators. 168

10.2.2 Extending the Language CIMP. 169

10.2.3 An Order for sTCCS. 169

10.2.4 Adding Probabilities to sTCCS. 170

10.2.5 Automatic Transformation Tools. 170

10.2.6 General Transformation to Parallelism. 171

10.2.7 Expression Evaluation. 172

10.2.8 Functional Programming. 172

11. Bibliography. 173

Chapter 1

Introduction.

1.1 Overview.

Programming languages which contain some form of parallelism have proliferated recently. Examples include Occam [Occ], Ada [Ada], concurrent Pascal [Hart], parallel Fortran[For], concurrent C [Son] and many more. All of these languages are based upon one or more of the following data handling methodologies;

- vector processing,
- channels,
- shared variables.

Vector processing is not an interesting form of parallelism, and tends to be used in large numerical applications only. There is a great similarity between concurrent programs written in languages with shared variables or channels. Shared variables are, however, more transparent to the programmer. Working with shared variables requires no explicit control of the communication between the components of a program. Although this extra transparency can leave the programmer with difficulty in achieving some complex control tasks, most of the time it reduces the programming burden. The languages Occam and Ada are channel based, whilst versions of concurrent Pascal have been based both on vector processing and on shared variables.

Throughout this thesis we shall be concentrating upon a simple concurrent imperative programming language with shared variables.

1.1.1 The language CIMP

The simple imperative language consisting of assignment, sequential composition, alternation and iteration, sometimes referred to as IMP, has been much studied. In [Mil6] a version of this language extended to parallelism is given. The language we shall be studying, which we call CIMP, is a reduced version of the language of [Mil6]. We assume that there are a *fixed* number of program variables. In other words, that there is some fixed index set I and that we have a finite collection of variables X_i for all $i \in I$; we refer to the set $\{X_i \mid i \in I\}$ as $\mathcal{X}$. The syntax of the language CIMP is as follows;

Expressions: $E ::= X_i \mid F(E_1, \ldots E_n)$,

Commands: $C ::= X_i := E \mid C;C \mid \mathit{IF}\ \ E\ \ \mathit{THEN}\ \ C_1\ \ \mathit{ELSE}\ \ C_2 \mid$
$C_1\ \mathit{PAR}\ C_2 \mid \mathit{SKIP} \mid \mathit{WHILE}\ \ E\ \ \mathit{DO}\ \ C \mid$
$\mathit{LET}\ \ X := E\ \ \mathit{IN}\ \ C.$

Expressions include any arbitrary function that can be written in the required form. Any program in the original imperative concurrent language can be written in CIMP, simply by taking a sufficiently large set of variables. Variables will be denoted by X, Y, ... when the index is irrelevant. We assume that the domain of values over which variables range has the name $\mathcal{V}$.

1.1.2 This Thesis

Most of the results obtained on CIMP have been formulated using the calculus CCS [Mil1, Mil6]. The remainder of this chapter is an introduction to CCS and contains those concepts from [Mil6] which we shall be using throughout this thesis.

Chapter 2 is an introduction to the structure of processes containing work on ideas that lead to the definition of semi-confluence. In chapter 3 we introduce and study the concept of semi-confluence, which will be used to demonstrate determinism properties of CIMP.

In chapter 4 we identify a sublanguage of CIMP which is deterministic; this language is characterised by a simple syntactic condition on programs. The sufficiency of this condition is demonstrated using the notion of semi-confluence defined in chapter 3.

The semantical consequences of restricting to deterministic programs are considered in chapter 5. We shall demonstrate that the denotational semantics of a deterministic concurrent programming language are little more complicated than that of a sequential programming language, and moreover that it is effectively equivalent to the process based semantics. We also present a simple transformation system on programs that preserves determinism whilst manipulating the amount of concurrency in that program, and demonstrate that the transformations are correct using the denotational semantics defined in this chapter.

In chapter 6 we present a proof method for concurrent programs based on the approach of Owicki-Gries[Owi]. We show that the proof rules are sound with respect to an interpretation given in Hennessey-Milner logic [Mil5], and that non-interference proofs are unnecessary if the program is deterministic.

The concept of parallel programs that co-operate, in the sense that they are mutually compatible, is expressed in terms of *rely* and *guarantee* conditions, and forms the basis of a proof method due to Jones [Jon]. In chapter 7 we study this approach to program proof in the presentation given by Stirling [Stil]. We shall show that this is sound with respect to a Hennessey-Milner logic interpretation, and demonstrate that determinism is a strong restriction on the proof technique.

An extended version of CCS that admits time is defined in chapter 8. We define timed bisimulations and a natural order which expresses the property of one processes being faster than another, whilst computationally equivalent. A study of the properties of an implementation of the *alternating bit protocol* is presented as an extended example. A modal logic is defined which characterises the notion of timed bisimulation in the same way that Hennessey-Milner logic characterises strong bisimulation.

In chapter 9, the calculus defined in chapter 8 is used to give a timed semantics for CIMP. A proof system which can express both correctness and timing properties of concurrent programs is defined in the style of Nielson [Nie], and is shown to be sound with respect to the timed semantics.

1.2 CCS

The calculus CCS [Mil1, Mil6] is one of a number of formal approaches to the study of concurrency which include CSP [Hoa1], ACP [Ber], Unity [Cha] and Petri nets [Rei]. The calculus consists of the notion of *agents* (or *processes*), which have the ability to evolve by performing *actions*, and to communicate with other agents using these actions. Synchronisation between agents is achieved by the simultaneous performance of a complementary action pair, resulting in the combination performing a silent (τ) action. This hiding of communication (we can only see the τ action) permits a notion of observation which is used extensively in the study of systems defined using CCS. The calculus CCS consists of operators for defining processes, and derivation laws in the style of [Plo1], for inferring the evolution of processes.

1.2.1 The Language

The syntax of CCS is defined as follows. We assume a set of action names Λ with ϵ and τ not in Λ, and let $Act = \Lambda \cup \{\tau\}$. There is a bijection, complementation, written $\bar{\ }$ from Λ to Λ which is its own inverse. The letter λ ranges over Λ and μ ranges over Act. We let S range over the set of renaming functions, i.e. those functions $S : Act \rightarrow Act$ such that $\overline{S(\lambda)} = S(\overline{\lambda})$ and $S(\lambda) \neq \tau$ for $\lambda \in \Lambda$. The language CCS consists of: an infinite set Var of variables ranged over by X, Y; a constant symbol *nil*; the unary *action prefix* functions $\mu.$ for $\mu \in Act$; *restriction* $\backslash L$ for $L \subseteq \Lambda$; *relabelling* $[S]$ for a renaming function S; *recursion* Fix; the binary *non-deterministic sum* function $+$; and the binary *parallel composition* function $|$. The set E of CCS expressions is the following:

$$E ::= X \mid nil \mid \mu.E \mid E + E' \mid E \mid E' \mid E\backslash L \mid E[S] \mid Fix(\tilde{X} = \tilde{E}).$$

The set of CCS process, written $\mathcal{P}$, is the set of closed CCS expressions; i.e. those expressions where all the variables are bound. When we write processes, we will often exclude trailing *nil*'s, thus writing for instance $a.nil$ as a.

Definition 1.2.1 *The* sort *of a CCS expression E, written $\mathcal{L}(E)$, is the set of action names in a process, and is defined recursively over the syntax of CCS expressions as follows;*

- $\mathcal{L}(nil) = \emptyset$,
- $\mathcal{L}(X) = \emptyset$,
- $\mathcal{L}(\mu.E) = \{\mu\} \cup \mathcal{L}(E) - \{\tau\}$,
- $\mathcal{L}(E + F) = \mathcal{L}(E) \cup \mathcal{L}(F)$,
- $\mathcal{L}(E \mid F) = \mathcal{L}(E) \cup \mathcal{L}(F)$,
- $\mathcal{L}(E\backslash L) = \mathcal{L}(E) - (L \cup \overline{L})$,
- $\mathcal{L}(E[S]) = S(\mathcal{L}(E))$,
- $\mathcal{L}(Fix(X = E)) = \mathcal{L}(E)$.

1.2.2 Operational Semantics.

An operational semantics is given for CCS by means of a family of transition relations $\{ \xrightarrow{\mu} \mid \mu \in Act\}$. The relation we require is the least one satisfying the derivation laws presented in figure 1-1.

1.2.3 Strong Bisimulation.

The notion of equivalence between processes is that of bisimulation.

Definition 1.2.2 *A binary relation $B \subseteq \mathcal{P} \times \mathcal{P}$ is a* strong bisimulation *if $(P, Q) \in B$ implies that for all $\mu \in Act$,*

$$\text{ACT:}\ \frac{}{\mu.E \xrightarrow{\mu} E}$$

$$\text{SUM0:}\ \frac{E \xrightarrow{\mu} E'}{E + F \xrightarrow{\mu} E'} \qquad \text{SUM1:}\ \frac{F \xrightarrow{\mu} F'}{E + F \xrightarrow{\mu} F'}$$

$$\text{COM0:}\ \frac{E \xrightarrow{\mu} E'}{E \mid F \xrightarrow{\mu} E' \mid F} \qquad \text{COM1:}\ \frac{F \xrightarrow{\mu} F'}{E \mid F \xrightarrow{\mu} E \mid F'}$$

$$\text{COM2:}\ \frac{E \xrightarrow{\lambda} E' \quad F \xrightarrow{\overline{\lambda}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\text{REL:}\ \frac{E \xrightarrow{\mu} E'}{E[S] \xrightarrow{S(\mu)} E'[S]}$$

$$\text{RES:}\ \frac{E \xrightarrow{\mu} E'}{E\backslash L \xrightarrow{\mu} E'\backslash L} \quad (\mu, \overline{\mu} \notin L)$$

$$\text{REC:}\ \frac{E_j\{\tilde{E}/\tilde{X}\} \xrightarrow{\mu} E_j'}{Fix_j\{X_i = E_i;\ i \in I\} \xrightarrow{\mu} E_j'}$$

Figure 1–1: Operational rules for CCS

1. *whenever* $P \xrightarrow{\mu} P'$ *then, for some* Q', $Q \xrightarrow{\mu} Q'$ *and* $(P', Q') \in B$,

2. *whenever* $Q \xrightarrow{\mu} Q'$ *then, for some* P', $P \xrightarrow{\mu} P'$ *and* $(P', Q') \in B$.

Proposition 1.2.3 *If B_i $(i = 1, 2, \ldots)$ are strong bisimulations, then so are the following:*

1. $Id_{\mathcal{P}}$,
2. B_i^{-1},
3. $B_1 B_2$,
4. $\bigcup_{i \in I} B_i$.

Definition 1.2.4 *P and Q are strongly congruent, written $P \sim Q$, whenever there is some strong bisimulation B with $(P, Q) \in B$. In other words,*

$$\sim\ = \bigcup\{B \mid\ B \text{ is a strong bisimulation }\}.$$

Proposition 1.2.5

1. *$\sim$ is the largest strong bisimulation,*
2. *$\sim$ is an equivalence relation.*

Proposition 1.2.6 *Strong Congruence is substitutive for the finite CCS operators. In other words, whenever $P_i \sim Q_i$ $(i = 1, 2)$,*

1. *$\mu.P_1 \sim \mu.Q_1$,*
2. *$P_1 + P_2 \sim Q_1 + Q_2$,*
3. *$P_1 \mid P_2 \sim Q_1 \mid Q_2$,*
4. *$P_1[S] \sim Q_1[S]$,*
5. *$P_1 \backslash L \sim Q_1 \backslash L$.*

Definition 1.2.7 *Let $\tilde{E}$ and $\tilde{F}$ contain variables $\tilde{X}$ at most. Then $\tilde{E} \sim \tilde{F}$ if, for all sets of processes $\tilde{P}$, $\tilde{E}[\tilde{P}/\tilde{X}] \sim \tilde{F}[\tilde{P}/\tilde{X}]$.*

Proposition 1.2.8 *Let $\tilde{E}$ and $\tilde{F}$ contain variables $\tilde{X}$ at most. Let $\tilde{A} \overset{def}{\equiv} \tilde{E}[\tilde{A}/\tilde{X}]$ and $\tilde{B} \overset{def}{\equiv} \tilde{F}[\tilde{B}/\tilde{X}]$ and $\tilde{E} \sim \tilde{F}$. Then $\tilde{A} \sim \tilde{B}$. In other words, strong congruence is substitutive in recursive expressions.*

1.2.4 Observation.

One of the fundamental notions of CCS is that of observational equivalence. We assume that we cannot observe silent actions τ and derive an equivalence. We shall be working with sequences of actions names; sequences will be denoted by letters s, t and we will denote concatenation by juxtaposition

Definition 1.2.9 *If $t \in Act^*$, then $\hat{t}$ is the sequence obtained by deleting all the occurrences of τ from t.*

Definition 1.2.10 *If $t = \mu_1 \ldots \mu_n \in Act^*$, then $E \xrightarrow{t} E'$ iff $E \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_n} E'$.*

Definition 1.2.11 *If* $t = \mu_1 \ldots \mu_n \in Act^*$ *then* $E \overset{t}{\Longrightarrow} E'$ *iff*

$$E(\xrightarrow{\tau})^* \overset{\mu_1}{\rightarrow} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \overset{\mu_n}{\rightarrow} (\xrightarrow{\tau})^* E'.$$

Definition 1.2.12 *A binary relation,* $B \subseteq \mathcal{P} \times \mathcal{P}$, *is a* (weak) bisimulation, *if* $(P, Q) \in B$ *implies that for all* $\mu \in Act$,

1. *whenever* $P \xrightarrow{\mu} P'$ *then, for some* Q', $Q \overset{\hat{\mu}}{\Longrightarrow} Q'$ *and* $(P', Q') \in B$,

2. *whenever* $Q \xrightarrow{\mu} Q'$ *then, for some* P', $P \overset{\hat{\mu}}{\Longrightarrow} P'$ *and* $(P', Q') \in B$.

Definition 1.2.13 *P and Q are* observation equivalent *or* bisimular, *written* $P \approx Q$, *if* $(P, Q) \in B$ *for some (weak) bisimulation B. In other words,*

$$\approx \, = \bigcup \{ B \mid \; B \textit{ is a bisimulation } \}.$$

Proposition 1.2.14

1. $\approx$ *is the largest bisimulation,*

2. $\approx$ *is an equivalence relation.*

Definition 1.2.15 *Let expression* $\tilde{E}$ *and* $\tilde{F}$ *have at most variables* $\tilde{X}$. *Then* $\tilde{E} \approx \tilde{F}$ *if for all sets of process* $\tilde{P}$, $\tilde{E}[\tilde{P}/\tilde{X}] \approx \tilde{F}[\tilde{P}/\tilde{X}]$.

Proposition 1.2.16 *Bisimularity is substitutive with respect to the following operators. Let* $P_i \approx Q_i$ $(i = 1, 2)$, $\tilde{E} \approx \tilde{F}$, $\tilde{A} \overset{def}{\equiv} \tilde{E}[\tilde{A}/\tilde{X}]$ *and* $\tilde{B} \overset{def}{\equiv} \tilde{F}[\tilde{B}/\tilde{X}]$. *Then,*

1. $\mu.P_1 \approx \mu.Q_1$

2. $P_1 \mid P_2 \approx Q_1 \mid Q_2$

3. $P_1[S] \approx Q_1[S]$,

4. $P_1 \backslash L \approx Q_1 \backslash L$,

5. $\tilde{A} \approx \tilde{B}$.

Bisimilarity is not substitutive for non-deterministic sum, since

$$a \approx \tau.a,$$

but it is not the case that,

$$a + b \approx \tau.a + b.$$

Definition 1.2.17 *P and Q are* equal *or* (observation-)congruent, *written $P = Q$, if for all $\mu \in Act$:*

1. *whenever* $P \xrightarrow{\mu} P'$ *then, for some* Q', $Q \overset{\mu}{\Longrightarrow} Q'$ *and* $P' \approx Q'$,

2. *whenever* $Q \xrightarrow{\mu} Q'$ *then, for some* P', $P \overset{\mu}{\Longrightarrow} P'$ *and* $P' \approx Q'$.

Proposition 1.2.18 *Equality is substitutive for* all *the CCS operators.*

Definition 1.2.19 *X is* guarded *in E if every free occurence of X lies within a subexpression $\lambda.F$ of E*

1.2.5 Equational Theory.

The following laws are all valid for observational congruence.

1. Monoid laws;

   (a) $P + Q = Q + P$,

   (b) $P + (Q + R) = (P + Q) + R$,

   (c) $P + P = P$,

   (d) $P + nil = P$.

2. τ laws;

   (a) $\mu.\tau.P = \mu.P$,

   (b) $P + \tau.P = \tau.P$,

   (c) $\mu.(P + \tau.Q) + \mu.Q = \mu.(P + \tau.Q)$.

3. Parallel composition laws;

   (a) $P \mid Q = Q \mid P$,

   (b) $(P \mid Q) \mid R = P \mid (Q \mid R)$,

   (c) $P \mid Nil = P$.

4. Restriction laws;

   (a) $P\backslash L = P$ if $\mathcal{L}(P) \cap (L \cup \overline{L}) = \emptyset$,

   (b) $P\backslash K\backslash L = P\backslash K \cup L$,

   (c) $P[S]\backslash L = P\backslash S^{-1}(L)[S]$,

   (d) $(P \mid Q)\backslash L = (P\backslash L) \mid (Q\backslash L)$ if $\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \cap (L \cup \overline{L}) = \emptyset$.

5. Relabelling laws,

   (a) $P[Id] = P$,

   (b) $P[S] = P[S']$ if $S\lceil\mathcal{L}(P) = S'\lceil\mathcal{L}(P)$,

   (c) $P[S][S'] = P[S' \circ S]$,

   (d) $(P \mid Q)[S] = P[S] \mid Q[S]$ if $S\lceil(L \cup \overline{L})$ is injective, where $L = \mathcal{L}(P \mid Q)$.

6. Recursion laws,

   (a) $Fix\,(X = E) = E[(Fix\,(X = E))/X]$,

   (b) If $F = E[F/X]$ then $F = Fix\,(X = E)$, provided that X is guarded in E,

   (c) $Fix\,(X = X + E) = Fix\,(X = E)$,

   (d) $Fix\,(X = \tau.X + E) = Fix\,(X = \tau.E)$,

   (e) $Fix\,(X = \tau.(X + E) + F) = Fix\,(X = \tau.X + E + F)$.

The final law is the Expansion theorem; let $P \equiv (P_1[S_1] \mid \ldots \mid P_n[S_n])\backslash L$, with $n \geq 1$. Then

$$\begin{aligned} P = {} & \Sigma\{S_i(\lambda).\ (P_1[S_1] \mid \ldots \mid P_i'[S_i] \mid \ldots \mid P_n[S_n])\backslash L; \\ & \quad P_i \xrightarrow{\lambda} P_i', S_i(\lambda) \notin L\} \\ & +\Sigma\{\tau.(P_1[S_1] \mid \ldots \mid P_i'[S_i] \mid \ldots \mid P_j'[S_j] \mid \ldots \mid P_n[S_n])\backslash L; \\ & \quad P_i \xrightarrow{\lambda_1} P_i',\ P_j \xrightarrow{\lambda_2} P_j',\ S_i(\lambda_1) = \overline{S_j(\lambda_2)},\ i < j\} \end{aligned}$$

The above laws form a complete equational reasoning system over finite state CCS processes.

1.2.6 Value Passing.

The CCS calculus with value passing is derived from the basic calculus. We use the variables x, y, ... to range over the values V. Also we assume value expressions e and boolean expressions b built from variables. We define the set $\mathcal{E}$ of agents with values, as the smallest set containing all the variables; and given E and E_i are in $\mathcal{E}$ then so are;

1. $\lambda(x).E$, $\overline{\lambda}(e).E$, $\tau.E$, action prefixes ($\lambda \in \Lambda$),
2. $\sum_{i \in I} E_i$, a summation, (I an indexing set),
3. $E_1 \mid E_2$ a composition,
4. $E\backslash L$ a restriction, ($L \subseteq \Lambda$),
5. $E[S]$ a renaming, (S a renaming function),
6. *if* b *then* E_1 *else* E_2, a conditional.

Furthermore we have constants, for each constant A of arity n we have defining equation,

$$A(x_1, \ldots, x_n) \stackrel{def}{\equiv} E,$$

E	$\hat{E}$
$\mu(x).E$	$\sum_{x\in V}\mu_x.\hat{E}$
$\overline{\mu}e.E$	$\overline{\mu}_e.\hat{E}$
$\tau.E$	$\tau.\hat{E}$
$\sum_{i\in I}E_i$	$\sum_{i\in I}\widehat{E_i}$
$E_1 \mid E_2$	$\widehat{E_1} \mid \widehat{E_2}$
$E\backslash L$	$\hat{E}\backslash\{\mu_x \mid \mu \in L, x \in V\}$
$E[S]$	$\hat{E}[\hat{S}]$ where $\hat{S}(\mu_x) = S(\mu)_x$
if b *then* E_1 *else* E_2	$\sum_{x:b}\widehat{E_1} + \sum_{x:\neg b}\widehat{E_2}$ where x is not in E
$A(e_1,\ldots,e_n)$	$A_{e_1,\ldots,e_n}$

Figure 1–2: Translation of Value Passing Calculus into the Basic Calculus.

where the right hand side E may contain no process variables, and no free variables except $x_1,\ldots,x_n$.

Given an expression E in $\mathcal{E}$ there is an expression $\hat{E}$ which is the translation of that expression into the basic calculus. The translation works by taking $\mu(x)$ and matching it to the set of actions $\{\mu_x \mid x \in V\}$.

The translation, defined recursively, is presented in figure 1-2.

Furthermore, the single defining equation $A(x_1,\ldots,x_n) \overset{def}{\equiv} E$ is translated into the indexed set,

$$\{A_{x_1,\ldots,x_n} \overset{def}{\equiv} \hat{E};\ x_1,\ldots,x_n \in V\}$$

of defining equations.

$$[\![X]\!] = a_X x.\overline{r}x.nil$$

$$[\![F(E_1, \ldots E_n)]\!] = ([\![E_1]\!][r_1/r] \mid \ldots \mid [\![E_n]\!][r_n/r] \mid r_1x_1.\ldots.r_nx_n.\overline{r}_{F(x_1,\ldots,x_n)}.nil) \backslash\{r_1, \ldots, r_n\}$$

Figure 1–3: Expression evaluation in CIMP.

1.3 Semantics of CIMP.

We translate variables as instances of the following process:

$$Loc_X = g_X x.V_X(x)$$
$$V_X(x) = \overline{a}_X x.V_X(x) + g_X y.V_X(y)$$

it should be noted that attempting to read variables before they are intialised leads to deadlock in this description. Moreover the underlying grain of our description is that of memory access not that of expression evaluation.

Definition 1.3.1 *The set* $L_{X_i} = \{a_{X_i}, \overline{g}_{X_i}\}$.

1.3.1 Expression Translation.

The translation of expressions is given in figure 1-3.

1.3.2 Command Translation.

Definition 1.3.2 *We need the following derived operators of CCS.*

- $done = \overline{d}.nil$,
- $B_1 \; result \; B_2 = (B_1 \mid B_2)\backslash\{r\}$,
- $B_1 \; before \; B_2 = (B_1[b/d] \mid b.B_2)\backslash\{b\}$ *with* b *new,*

$[\![X := E]\!] = [\![E]\!]\ \mathit{result}\ (rx.\bar{g}_X x.\mathit{done})$

$[\![C_1; C_2]\!] = [\![C_1]\!]\ \mathit{before}\ [\![C_2]\!]$

$[\![\mathit{IF}\ E\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2]\!] = [\![E]\!]\ \mathit{result}\ rx.(\mathit{if}\ x\ \mathit{then}[\![C_1]\!]\ \mathit{else}\ [\![C_2]\!])$

$[\![\mathit{WHILE}\ E\ \mathit{DO}\ C]\!] =$
$W \overset{def}{=} [\![E]\!]\ \mathit{result}\ rx.(\mathit{if}\ x\ \mathit{then}[\![C]\!]\ \mathit{before}\ W\ \mathit{else}\ \mathit{done})$ with W new.

$[\![C_1\ \mathit{PAR}\ C_2]\!] = [\![C_1]\!]\ \mathit{par}\ [\![C_2]\!]$

$[\![\mathit{SKIP}]\!] = \mathit{done}$

$[\![\mathit{LET}\ X := E\ \mathit{IN}\ C]\!] = [\![E]\!]\ \mathit{result}\ ((rx.\bar{g}_X x.\mathit{done} \mid \mathit{Loc}_X)\ \mathit{before}\ [\![C]\!])\backslash L_X$

Figure 1–4: Command translation for CIMP.

- $B_1\ \mathit{par}\ B_2 = (B_1[d_1/d] \mid B_2[d_2/d] \mid d_1.d_2.\bar{d}.\mathit{done})\backslash\{d_1, d_2\}$.

The processes which represent commands are given in figure 1-4. Since we have no input or output commands in this semantics it seems it could be empty. However as we have global variables whose state we can examine this does not occur. In CCS the process representing a program is not observationally equivalent to *nil*, as it would be if the semantics were empty.

Definition 1.3.3 *The translation of a program P over a set of variables $\mathcal{X}$ is the following process,*

$$([\![P]\!] \mid \mathit{Loc}_{X_1} \mid \ldots \mid \mathit{Loc}_{X_n})\backslash\ L_{X_1} \cup \ldots \cup L_{X_n}$$

with the set $\{X_i \mid 1 \leq i \leq n\} = \mathcal{X}$.

Proposition 1.3.4 *If E is an expression, then*

$$[\![E]\!] \overset{s}{\Longrightarrow} \mathit{nil}$$

for some sequence s of τ,a_X and $\bar{r}$ actions.

Corollary 1.3.5 *The evaluation of an expression does not affect the state of the variable process.*

Proposition 1.3.6 *If C is a terminating CIMP command then,*

$$[\![C]\!] \overset{s\bar{d}}{\Longrightarrow} nil,$$

for some sequence s of τ, a_X and $\overline{g}_X$ actions.

Proposition 1.3.7 *Let C_1, C_2 and C_3 be CIMP commands. Then the following are true;*

1. $[\![(C_1;C_2);C_3]\!] = [\![C_1;(C_2;C_3)]\!]$,
2. $[\![C_1 \;\; PAR \;\; C_2]\!] = [\![C_2 \;\; PAR \;\; C_1]\!]$,
3. $[\![(C_1 \;\; PAR \;\; C_2) \;\; PAR \;\; C_3]\!] = [\![C_1 \;\; PAR \;\; (C_2 \;\; PAR \;\; C_3)]\!]$,
4. $[\![WHILE \;\; E \;\; DO \;\; C]\!] = [\![IF \;\; E \;\; THEN \;\; C; WHILE \;\; E \;\; DO \;\; C \;\; ELSE \;\; SKIP]\!]$.

1.3.3 Process Logic.

We define a simple process logic $\mathcal{PL}$ [Mil6, Sti2] over CCS, whose formulae express conditions on the behaviour of these processes.

Definition 1.3.8 *$\mathcal{PL}$ is the smallest class of formulae containing the following, where F, F_i are already in $\mathcal{PL}$:*

1. $< \alpha > F$, *a progression* ($\alpha \in Act$),
2. $\neg F$, *a negation,*
3. $\bigwedge_{i \in I} F_i$, *a conjunction* (*I some indexing set).*

Note that the first formula in $\mathcal{PL}$ is the empty conjunction $\bigwedge_{i\in\emptyset} F_i$, which we write as $TRUE$. Thereafter we obtain formulae such as $< \alpha > TRUE$, $< \alpha > (< \beta > TRUE \wedge \neg < \gamma > TRUE)$ etc. We use infixed "$\wedge$" for binary conjunction.

Informally $< \alpha > G$, pronounced "*diamond G*" asserts of a process P that:

> "it is possible for P to do α and thereby reach a state Q for which G holds."

Definition 1.3.9 *We define a satisfaction relation* $\models$ *between* processes *and* formulae *as follows; by induction on the structure of formulae,*

1. $P \models < \alpha > F$ *if, for some* P', $P \xrightarrow{\alpha} P'$ *and* $P' \models F$,
2. $P \models \neg F$ *if it is not the case that* $P \models F$,
3. $P \models \bigwedge_{i\in I} F_i$ *if for all* $i \in I$, $P \models F_i$.

We pronounce $P \models F$ as "P satisfies F".

The language $\mathcal{PL}$ is extremely basic, and we would find it hard to express useful specifications or descriptions in it. Let us introduce some derived forms for $\mathcal{PL}$. Firstly some propositional operators:

$$FALSE \equiv \neg TRUE$$

$$F_0 \wedge F_1 \equiv \bigwedge_{i\in\{0,1\}} F_i$$

$$\bigvee_{i\in I} F_i \equiv \neg \bigwedge_{i\in I} \neg F_i$$

$$F_0 \vee F_1 \equiv \bigvee_{i\in\{0,1\}} F_i$$

$$F \supset F' \equiv \neg F \vee F'$$

Second, a variety of modal operators:

$$< \alpha_1 \ldots \alpha_n > F \equiv < \alpha_1 > \ldots < \alpha_n > F \qquad (n \geq 0)$$

$$[t]F \equiv \neg < t > \neg F \qquad (t \in Act^*)$$

Thus $[t]F$ pronounced "*box t*" asserts of a process P that:

"it is necessary for P, if it can do the action sequence t, thereby to reach a state Q where F holds,"

and in particular $[t]FALSE$ asserts that P cannot do t.

Third, some uses of quantifiers:

$$\forall i(\Phi(i) \supset F_i) \equiv \bigwedge_{i\in\{i;\Phi(i)\}} F_i$$

$$\exists i(\Phi(i) \wedge F_i) \equiv \bigvee_{i\in\{i;\Phi(i)\}} F_i$$

$$\forall i(F_i \supset \Phi(i)) \equiv \forall i(\neg\Phi(i) \supset \neg F_i)$$

Notice we are exploiting our ability to choose whatever index set I we wish for a conjunction; moreover, we are free to express that set however we wish, so we imagine that $\Phi(i)$ here is some arbitrary logical formula containing the variable i free. This allows us to mix logical conditions freely with the use of the $<>$ and $[]$ modalities.

As a final pair of derived forms, we may find it convenient to have weak versions of $<>$ and $[]$ which ignore τ actions:

$$<< s >> F \equiv \exists t \in Act^*.(s = \hat{t} \wedge < t > F)$$

$$[[s]]F \equiv \forall t \in Act^*.(s = \hat{t} \supset [t]F)$$

Note, in particular, the case $s = \epsilon$; $< s > F$ and $[s]F$ are both equivalent to F itself, but $<<>> F$ means

"it is possible to satisfy F after a sequence of τ actions".

Chapter 2

Process Structure

2.1 Introduction.

In his books [Mil1,Mil6] Milner discusses the properties required of a process for it to be either confluent or deterministic. Our notion of confluence is motivated by similar requirements on the reductions of the lambda calculus [Chu] and term rewriting systems [Hue]. Determinacy is based upon observation; we require that whenever we observe a particular sequence of actions from a process, all states reachable by that sequence of actions should be identical, i.e. we cannot reach a different place by the same route. This does not amount to saying that there is no non-determinism in the original process, it simply demands that we can know the current state simply by observing the computation the process has gone through.

In his thesis [San], Sanderson demonstrates that a knowledge of the structure of a process can greatly simplify proofs of the properties of that process. Moreover determinism in programs makes computation predictable. Despite its usefullness in specification, most non-determinism in programs is added accidentally by the programmer and can subsequently take a great time to remove. This particular problem is greatly increased when the programming language being used has the capability of computing concurrently. We would like to obtain some simple condition on programs which would allow us to find a deterministic sub-language of CIMP. Semantically we expect that such a condition would force the processes which represent CIMP programs to be deterministic in the sense of [Mil6].

In this chapter the definitions given are in the style of [Mil6] which differs slightly from that of [Mil1]

2.2 Determinacy.

Definition 2.2.1 *A subclass $\mathcal{Q}$ of processes $\mathcal{P}$ is* observation closed *iff whenever $P \in \mathcal{Q}$ and $P \xrightarrow{\mu} P'$ then $P' \in \mathcal{Q}$.*

Definition 2.2.2 *A process P is* strongly determinate *if for every derivative Q of P and for all $\mu \in Act$ whenever $Q \xrightarrow{\mu} Q'$ and $Q \xrightarrow{\mu} Q''$ then $Q' \sim Q''$.*

Proposition 2.2.3 *Strong determinacy is observation closed.*

Proposition 2.2.4 *Whenever P is strongly determinate and $P \sim Q$, then Q is strongly determinate.*

This notion of determinacy unfortunately does not prevent preemption by τ, since this is treated just as any other action. However the notion above has a natural extension to a weak (or observational) form.

Definition 2.2.5 *P is* weakly determinate *iff for every derivative Q of P and for all $s \in \Lambda^*$, whenever $Q \overset{s}{\Longrightarrow} Q'$ and $Q \overset{s}{\Longrightarrow} Q''$ then $Q' \approx Q''$.*

Proposition 2.2.6 *Weak determinacy is observation closed.*

Proposition 2.2.7 *Whenever P is weakly determinate and $P \approx Q$, then Q is weakly determinate.*

Note that unlike the relationship between the equivalences, a process being strongly determinate does not imply that it need be weakly determinate, nor conversely. For example the process

$$a + \tau$$

is strongly determinate but not weakly determinate, whereas the process

$$a + a.\tau$$

is weakly determinate but not strongly determinate.

Proposition 2.2.8 *The following constructions of CCS preserve strong determinacy. If the processes P and P_i are determinate then so are the following:*

1. *nil, $a.P$, $P\backslash L$,*

2. *$\sum_{i \in I} \mu_i P_i$ where the μ_i's are distinct,*

3. *$P_1 \mid P_2$ where $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$ and $\mathcal{L}(P_1) \cap \overline{\mathcal{L}(P_2)} = \emptyset$,*

4. *$P[f]$ where $f \lceil \mathcal{L}(P)$ is injective.*

The above side conditions on the construction of parallel composition are important and we shall find that similar restrictions will be needed when we introduce constructions which preserve other notions of process structure. The above restriction severely limits the processes we can show determinate by force of construction alone. For instance

$$a \mid a$$

is determinate but cannot be shown to be by construction. Thus unless any class of processes can be constructed using the above, we can only prove it determinate by showing *directly* that every instance is determinate.

2.3 Confluence.

Definition 2.3.1 *P is* strongly confluent *iff it is strongly determinate and for all derivatives Q of P whenever $Q \xrightarrow{\mu_1} Q_1$ and $Q \xrightarrow{\mu_2} Q_2$, with $\mu_1 \neq \mu_2$, then we can find Q_1' and Q_2' such that the following diagram can be completed.*

$$\begin{tikzcd} Q \arrow[r, "\mu_1"] \arrow[d, "\mu_2"'] & Q_1 \arrow[d, "\mu_2"'] \\ Q_2 \arrow[r, "\mu_1"] & Q_2' \sim Q_1' \end{tikzcd}$$

Proposition 2.3.2 *Strong confluence is observation closed and preserved by strong bisimulation.*

Definition 2.3.3 *P is* weakly confluent *if for every derivative Q of P the following diagrams can be completed, in the sense that if the top and left-hand derivations exist, then the bottom and right-hand derivations can be inferred.*

$$\begin{tikzcd} Q \arrow[r, "\tau"] \arrow[d, Rightarrow] & Q_1 \arrow[d, Rightarrow] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd} \quad \begin{tikzcd} Q \arrow[r, "\tau"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow, "\lambda"'] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd} \quad \begin{tikzcd} Q \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2"'] & Q_1 \arrow[d, Rightarrow, "\lambda_2"'] \\ Q_2 \arrow[r, Rightarrow, "\lambda_1"] & \approx \end{tikzcd} \quad \begin{tikzcd} Q \arrow[r, "\lambda"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd}$$

$$\lambda_1 \neq \lambda_2$$

In the books [Mil1, Mil6] there are more elegant versions of the above definition, which are expressed via the notion of excess. Unfortunately, for the later styles of confluence which we will introduce this method cannot be used so it is not included here.

Proposition 2.3.4 *Weak confluence is observation closed.*

Proposition 2.3.5 *If $P \approx Q$ and P is weakly confluent then so is Q.*

Proposition 2.3.6 *If $P \Longrightarrow Q$ and P is weakly confluent then $P \approx Q$.*

This last proposition shows that τ actions do not affect the state of a weakly confluent agent.

Proposition 2.3.7 *If P is weakly confluent then it is weakly determinate.*

Definition 2.3.8 *For $\mu_i \in Act$, $1 \leq i \leq n$ ($n \geq 0$), we define the* confluent action prefix *of a process P, written $(\mu_1 \mid \ldots \mid \mu_n)P$, as follows;*

$$()P \equiv P,$$
$$(\mu_1 \mid \ldots \mid \mu_n)P \equiv \sum_{1 \leq i \leq n} \mu_i . (\mu_1 \mid \ldots \mid \mu_{i-1} \mid \mu_{i+1} \mid \ldots \mid \mu_n)P.$$

Note that given $(\mu_1 \mid \ldots \mid \mu_n)P$ and any permutation function σ of $\{1, \ldots, n\}$ then,

$$(\mu_{\sigma(1)} \mid \cdots \mid \mu_{\sigma(n)})P \sim (\mu_1 \mid \cdots \mid \mu_n)P.$$

Definition 2.3.9 *Given processes P_1 and P_2 we write $P_1 \parallel P_2$ to mean $(P_1 \mid P_2)\backslash L$ where $L = \overline{\mathcal{L}(P_1)} \cap \mathcal{L}(P_2)$.*

Proposition 2.3.10 *The following constructions of CCS preserve weak confluence. Whenever P, P_1 and P_2 are weakly confluent then so are;*

1. *nil and $P\backslash L$,*
2. *$P[f]$ where $f\lceil\mathcal{L}(P)$ is injective,*
3. *$(\mu_1 \mid \cdots \mid \mu_n)P$,*
4. *$P_1 \parallel P_2$ where $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$,*
5. *where E is built from the above constructions then $P \overset{def}{\equiv} Fix(X = E)$ is weakly confluent.*

To see why we cannot compose arbitrary processes in parallel consider the process

$$P = a.b \mid a.$$

Both the processes $a.b$ and a are weakly confluent, but unfortunately P is *not* since

$$P \xrightarrow{a} a.b \text{ and } P \xrightarrow{a} a \mid b$$

and the above derivatives are not weakly equivalent. Hence the above P is not weakly determinate and therefore cannot be weakly confluent. Whilst the restriction in the parallel composition is sufficient to ensure that the composed process will be weakly confluent, it once again leaves us unable to prove that certain processes which are weakly confluent are so, simply as a result of the manner of their construction. For example,

$$a \mid a$$

is weakly confluent but violates the sort independence required, for it to be deduced as weakly confluent from its construction. The parallel composition used in proposition 2.3.10 is not the usual one; we have 'forced' communication on all complementary actions. If we did not then the construction would not preserve weak confluence, for consider

$$Q = a \mid \overline{a}.$$

This can perform

$$Q \xrightarrow{a} \overline{a} \text{ and } Q \xrightarrow{\tau} nil$$

but as the process $\overline{a}$ cannot evolve trivially to *nil* these evolutions violate the determinism requirement upon silent actions. This seems less restrictive than the earlier requirement, as there appear to be no natural examples where permitting complementary actions to be observed does not lead immediately to a violation of the requirements for a process to be weakly confluent.

2.4 Adding Values.

As Sanderson shows in his thesis [San] these notions of determinacy and confluence do not apply in the value passing calculus, owing to an asymmetry between the input and output actions in that calculus which is not present in the underlying calculus. The following example should clarify the point:

$$A \overset{def}{\equiv} ax.by.\overline{g}(x+y).$$

The process A can perform the following actions (where the domain of values is the integers):

$$A \xrightarrow{a5} by.\overline{g}(5+y),$$
$$A \xrightarrow{a7} by.\overline{g}(7+y).$$

Under the definitions presented earlier we would be forced to deduce that this process is not confluent since it can perform the actions $a5$ or $a7$ and it is subsequently incapable of performing the requisite matching $a7$ or $a5$ actions. However we would expect that such a process should be both confluent and determinate. Thus the notions of confluence and determinism are extended to take account of substitution on input actions.

Definition 2.4.1 *(Sanderson) A process P is* strongly value confluent *if for all derivatives Q of P whenever $Q \xrightarrow{a} Q_1$ and $Q \xrightarrow{b} Q_2$ then one of the following hold:*

1. *$a = b$ and $Q_1 \sim Q_2$ or,*

2. *$a \neq b$ and either*

   (a) *$Q_1 \xrightarrow{b} Q_1'$ and $Q_2 \xrightarrow{a} Q_2'$ with $Q_1' \sim Q_2'$ or*

   (b) *$a = \lambda n$ and $b = \lambda m$ for some* positive *λ and values m,n, and there exists some U with variable x such that $Q_1 \sim U[n/x]$ and $Q_2 \sim U[m/x]$.*

Note that this definition is asymmetric in its treatment of positive and negative actions.

There is a similar extended definition of determinacy but that is not included as we shall use a slightly different technique.

This process structure has the usual properties of being derivative closed and preserved by strong bisimulation, and it has a definable sub-calculus of the value passing calculus. In [Mil4] Milner suggested a different approach to handling values which is more elegant.

Definition 2.4.2 *The set* $\{\Lambda_i \mid i \in I\}$ *is a* partition *of Act iff* $\Lambda_i \cap \Lambda_j = \emptyset$ *whenever* $i \neq j$ *and* $\bigcup_{i\in I}\Lambda_i = Act$.

Definition 2.4.3 *Let* $S = \{\Lambda_i \mid i \in I\}$ *be a partition of Act. We write* $\lambda_1 \sharp_S \lambda_2$ *to mean that the actions* λ_1 *and* λ_2 *are not in the same partition set of* S*, i.e. given* $\lambda_1 \in \Lambda_i$ *then* $\lambda_2 \in \Lambda_j$ *with* $i \neq j$.

When it is clear from the context by which partition the actions are separated, we will drop the subscript in the inequality statement.

Definition 2.4.4 *Given a partition S then P is* S-partition confluent *if for any derivative Q of P the following diagrams can be completed.*

$$\begin{tikzcd}
Q \arrow[r, "\tau"] \arrow[d, Rightarrow] & Q_1 \arrow[d, Rightarrow] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}
\quad
\begin{tikzcd}
Q \arrow[r, "\tau"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow, "\lambda"'] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}
\quad
\begin{tikzcd}
Q \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2"'] & Q_1 \arrow[d, Rightarrow, "\lambda_2"'] \\
Q_2 \arrow[r, Rightarrow, "\lambda_1"] & \approx
\end{tikzcd}
\quad
\begin{tikzcd}
Q \arrow[r, "\lambda"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}$$

$\lambda_1 \sharp \lambda_2$

Proposition 2.4.5 *S-partition confluence is observation closed.*

Proposition 2.4.6 *Given* $P \approx Q$ *and P is S-partition confluent then Q is S partition confluent.*

Proposition 2.4.7 *If P is S-partition confluent and $P \Longrightarrow P'$ then $P' \approx P$.*

The above definition has no constraints on actions from the same partition other than determinacy. Now let S be the partition which contains the following sets (for values taken from a domain $\mathcal{D}$):

for each positive $\lambda \in \Lambda$, the set $\{\lambda n \mid n \in \mathcal{D}\}$,

for each negative $\lambda \in \Lambda$, and each $n \in \mathcal{D}$, the singleton set $\{\lambda n\}$.

Then the resulting S-confluence is equivalent to the value passing confluence as defined by Sanderson.

Definition 2.4.8 *An action map f* preserves *a partition S iff for each $\Lambda_i \in S$ there is a fixed $\Lambda_j \in S$ such that for all $a \in \Lambda_i$, $f(a) \in \Lambda_j$.*

Proposition 2.4.9 *Given P, P_i are S-partition confluent then the following are also S-partition confluent:*

1. *$(\mu_1 \mid \ldots \mid \mu_n)P$, nil , $P \backslash L$,*
2. *$\sum_{i \in I} \lambda_i . P_i$ where for some $\Lambda_j \in S$, $\lambda_i \in \Lambda_j$ for all λ_i,*
3. *$P[f]$ where f preserves S,*
4. *$P_1 \parallel P_2$ where $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$,*
5. *where E is formed using the above constructions then $P \overset{def}{\equiv} Fix(X = E)$ is S-partition confluent.*

Definition 2.4.10 *Given two partitions S and S' we shall say S refines S', written $S' \sqsubseteq S$ iff for all $\Lambda' \in S'$ there exists $\Lambda \in S$ such that $\Lambda' \subseteq \Lambda$.*

Proposition 2.4.11 *Given P is S-partition confluent and $S \sqsubseteq S'$ then P is S' partition confluent.*

Chapter 3

Semi-Confluence.

3.1 Introduction.

In the last chapter we introduced some notions of determinacy and confluence and some constructions that preserved those properties of processes. Unfortunately there are many processes which have those properties which can not be constructed using the derived operators provided. In this chapter we will provide an extended confluence structure that will permit us to demonstrate constructively the confluence of a larger class of processes. Throughout this chapter we will only be interested in weak forms of confluence and shall therefore be using the phrase 'confluent' rather than 'weakly confluent'. For instance it has already been stated that the process

$$a \mid a$$

cannot be shown confluent by its construction, since it is not constructed using the operators provided in the previous chapter. There are two methods, in CCS, for removing unwanted actions; the first is restriction, and the second is internalisation (providing a communicating complementary action and then restricting). If we use the first method to make a process confluent then we lose all the behaviour guarded by the restricted actions. The second approach allows us to hide actions and given that the silent actions produced obey the requirements the later behaviour is still available. We should like to include these effects in our account of confluence and thus provide a more expressive set of constructors for confluent processes. Consider the following processes:

$$P \overset{def}{\equiv} a.b \mid a,$$
$$Q \overset{def}{\equiv} \overline{a}.\overline{a}.$$

The process P is not confluent, yet the process $P \parallel Q$ is confluent, since it is observationally equivalent to the process b. We should like the above activity to be expressible in a system of constructions for confluent processes.

3.2 Partial Confluence.

In order to express the ability of a process to become non-confluent, Milner in [Mil4] suggested the notion of partial confluence. We will divide actions into two sets, actions in the first set are 'well-behaved' and are fully confluent, while actions in the second have no requirement on them. Informally this means that we have a set of actions on which our process has ceased to behave as we should wish, but there are still areas where it behaves in a consistent manner. The following is a formal definition of the above.

Definition 3.2.1 *A process P is K-partially confluent (for some set $K \subseteq \Lambda$) iff for all derivatives Q of P the following diagrams can be completed: (letting κ range over K and λ range over Λ).*

$$\begin{tikzcd}
Q \arrow[r, "\tau"] \arrow[d, Rightarrow] & Q_1 \arrow[d, Rightarrow] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}
\quad
\begin{tikzcd}
Q \arrow[r, "\tau"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow, "\lambda"'] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}
\quad
\begin{tikzcd}
Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow, "\lambda"'] \\
Q_2 \arrow[r, Rightarrow, "\kappa"] & \approx
\end{tikzcd}
\quad
\begin{tikzcd}
Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "\kappa"'] & Q_1 \arrow[d, Rightarrow] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}$$

$$\kappa \neq \lambda$$

In the above definition we have demanded that for all actions $\kappa \in K$ we have precisely the requirements for a process to be confluent, but have specified no conditions on actions not in K other than τ. The actions not in the set K are 'bad', in that they are the actions preventing the process from being fully confluent. We cannot guarantee the confluence of the actions not in the set K. An example of a process which is partially confluent but not confluent is the following;

$$P \overset{def}{\equiv} a + a.b,$$

which is $\{b\}$-partially confluent. Whilst this concept of confluence is useful (since it can be shown to have the usual properties of a confluence and a set of constructions that preserve it) once an action has 'gone bad' it can only be removed by *restriction.* This has the effect of removing all future behaviours guarded by a 'bad' action and thus we are no nearer being able to manipulate the amount of 'non-confluence' in our process. There is no way of restoring a process to full confluence which keeps its internal behaviour intact, i.e. removing unwanted actions by internalising them. In the above example if we tried to make the process confluent by internalising the a action then we would obtain a process that is non-deterministic on the τ action and therefore could not be confluent. More precisely we derive the following process

$$Q \equiv (P \mid \overline{a})\backslash\{a\}.$$

Unfortunately this is the process;

$$\tau + \tau.b,$$

and is not even deterministic. We should like a notion that allows actions to become non-confluent but does not permit so much freedom that subsequently we can only remove them by direct restriction.

3.3 Semi-Confluence.

From the above it is clear that we need to find a requirement on processes that lies between full confluence and partial confluence. It cannot be so strong as to forbid all non-confluent behaviour, but it must contain some restriction on the kind of non-confluent behaviour allowed, which will permit us to restore confluence without losing potentialities guarded by non-confluent actions. In the following definition the set of actions identified is that set upon which confluence *cannot* be guaranteed. This is a change in style from that of partial confluence. We will be trying to show that processes are confluent, so we shall be demonstrating that eventually there are *no* actions upon which they are not fully confluent. In this case it is easier to keep track of those actions which are not yet well behaved, than the actions upon which the process can be guaranteed to be behaving as we require. The set of non-confluent actions contains those 'bad' actions which we wish to see removed from a process for it to be confluent.

Definition 3.3.1 *A process P is semi-confluent with respect to a set of actions $F \subseteq \Lambda$ (F-confluent) if for all derivatives Q of P the following diagrams can be completed, given transistions from Q to Q_1 and Q_2;*

$$\begin{tikzcd}
Q \arrow[r, "\tau"] \arrow[d, Rightarrow] & Q_1 \arrow[d, Rightarrow] & Q \arrow[r, "\tau"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow, "\lambda"'] & Q \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2"'] & Q_1 \arrow[d, Rightarrow, "\lambda_2"'] & Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "\kappa"'] & Q_1 \arrow[d, Rightarrow] \\
Q_2 \arrow[r, Rightarrow] & \approx & Q_2 \arrow[r, Rightarrow] & \approx & Q_2 \arrow[r, Rightarrow, "\lambda_1"] & \approx & Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}$$

$$\lambda_1 \neq \lambda_2 \qquad \kappa \in \Lambda - F$$

$$\text{if} \quad \begin{tikzcd}
Q \arrow[r, "\phi"] \arrow[d, Rightarrow, "\phi"'] & Q_1 \\
Q_2 &
\end{tikzcd} \quad \phi \in F \quad \left\{ \begin{array}{l} \textit{then either } Q_1 \approx Q_2 \textit{ or there exists } Q'_1 \textit{ and} \\ Q'_2 \textit{ such that } Q_1 \overset{\phi}{\Longrightarrow} Q'_1 \textit{ and } Q_2 \overset{\phi}{\Longrightarrow} Q'_2 \\ \textit{with } Q'_1 \approx Q'_2. \end{array} \right.$$

In the above definition we have only required that 'good' actions are deterministic. The 'bad' actions (those in F) are permitted to 'delay' determinism in the sense that it may require a further 'bad' action to reach the identified state. The motivation for this definition comes from the process we exhibited earlier;

$$a \mid a.b.$$

This process is not confluent, but when we internalise the a actions by composing with a suitable process it becomes confluent. This process is $\{a\}$-confluent. The earlier example,

$$a + a.b$$

is not semi-confluent to any set of actions, so partial confluence does not imply semi-confluence.

Proposition 3.3.2 *Semi-confluence to any set is observation closed.*

Proof: Immediate from the definition.

Proposition 3.3.3 *If P is F-confluent and $P \xrightarrow{\tau} P'$ then $P \approx P'$.*

Proof: Let $B = \{(P, P')$ such that $P \Longrightarrow P'$, and P is F-confluent $\}$. We shall show that this is an observational bisimulation up to $\approx$. For all $\lambda \in Act$ we must show that if $P \overset{\lambda}{\Longrightarrow} P_1$ then there exists P_2 such that $P' \overset{\lambda}{\Longrightarrow} P_2$ with $(P_1, P_2) \in \approx B \approx$ and conversely.

There are two cases:

1. $\lambda = \tau$. In this case either $P_1 \equiv P'$ or from

$$\begin{tikzcd} P \arrow[r, "\tau"] \arrow[d, Rightarrow] & P' \arrow[d, Rightarrow] \\ P_1 \arrow[r, Rightarrow] & P_3 \approx P_4 \end{tikzcd}$$

   we can deduce that $(P_1, P_4) \in B$ and $P_3 \approx P_4$ hence $(P_1, P_3) \in \approx B \approx$.

2. $\lambda = \mu$. Then from

$$\begin{tikzcd} P \arrow[r, "\tau"] \arrow[d, Rightarrow, "\mu"'] & P' \arrow[d, Rightarrow, "\mu"'] \\ P_1 \arrow[r, Rightarrow] & P_3 \approx P_4 \end{tikzcd}$$

   we can deduce $P' \overset{\mu}{\Longrightarrow} P_4 \approx P_3$ and therefore $(P_4, P_1) \in \approx B \approx$ as required.

The converse follows by an identical argument. □

Proposition 3.3.4 *If P is F-confluent and $F \subseteq F'$ then P is F'-confluent.*

Proof: Immediate from the definition.

Proposition 3.3.5 *Let P be F-confluent and consider $\phi \in F$ such that for any derivative Q of P the following can be completed;*

$$\begin{tikzcd} Q \arrow[r, "\phi"] \arrow[d, Rightarrow, "\phi"'] & Q_1 \arrow[d, Rightarrow] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd}$$

then P is $(F - \{\phi\})$-confluent.

Proof: Immediate from the definition of semi-confluence.

w

Proposition 3.3.6 *If P is F-confluent and $P \approx Q$ then Q is F-confluent.*

Proof: We have to show, that for any derivative R of Q, that the diagrams can be completed. Consider the second diagram in particular. Since $P \approx Q$ there exists a derivative S of P such that $S \approx R$ but as S is a derivative of P the following can be completed

$$\begin{tikzcd} S \arrow[r, "\tau"] \arrow[d, Rightarrow, "\lambda"'] & S_1 \arrow[d, Rightarrow, "\lambda"'] \\ S_2 \arrow[r, Rightarrow] & S_4 \approx S_3 \end{tikzcd} \quad \text{for any } S_1 \text{ and } S_2.$$

Now consider the second requirement upon the process R;

$$\text{if} \quad \begin{tikzcd} R \arrow[r, "\tau"] \arrow[d, Rightarrow, "\lambda"'] & R_1 \\ R_2 & \end{tikzcd}$$

then we choose $S_1 \approx R_1$ and $S_2 \approx R_2$. Thus there exist derivatives $R_3 \approx S_3$ and $R_4 \approx S_4$ with $R_1 \overset{\lambda}{\Longrightarrow} R_3$ and $R_2 \Longrightarrow R_4$ with $R_3 \approx R_4$ so the diagram can be completed as required. The proof proceeds by a similar argument in the other cases. □

Proposition 3.3.7 *If P is $\emptyset$-confluent then it is confluent.*

Proof: Immediate, observe that if P is $\emptyset$-confluent then, as $\Lambda - \emptyset = \Lambda$ the definition of semi-confluence is precisely that of confluence, when the set is empty.

It is the above proposition which permits us to demonstrate that a semi-confluent process has been converted to a confluent process. In our usual example,

$$a \mid a.b$$

which is $\{a\}$-confluent we can construct the following process.

$$(a \mid a.b \mid \overline{a}.\overline{a})\backslash\{a\}$$

which is the process

$$\tau.\tau.b + \tau.b.\tau$$

which is $\emptyset$-confluent or confluent. In this example we have removed the 'bad' action by providing a process which communicates on its complement sufficiently often and then restricting the action away. We shall find that this technique can be used in general to remove 'bad' actions from semi-confluent processes and restore our process to being fully confluent. The provision of sufficient complementary actions is very important. The following process

$$(a \mid a.b \mid \overline{a})\backslash\{a\}$$

is *not* confluent. If we use the expansion theorem we discover that the above is in fact

$$\tau + \tau.b,$$

which once again is not deterministic. Thus in attempting to remove 'bad' actions we must be careful to supply sufficient complementary actions to remove all the instances of the failing actions, or else we may produce a process which is not deterministic. It is this requirement which leads to the side conditions on the semi-confluence preserving parallel composition which will be defined later.

Definition 3.3.8 *A process P is a-*stable *if for every derivative Q of P if $Q \overset{a}{\Longrightarrow} Q'$ then $Q \approx Q'$.*

This definition states that a process can perform those actions upon which it is stable infinitely often without any observable change. To remove an action from a process which is in the set of 'bad' actions we shall require that its complement be stable in the process with which it is composed in parallel. This requirement is very strong. Thus returning a process from being semi-confluent to being fully confluent requires the presence of a highly constrained process which can in some sense 'hide' the undesired properties of a semi-confluent process.

Definition 3.3.9 *A process P is S-stable (for any action set S) iff $\forall a \in S$, P is a-stable.*

Note: if P is S-stable then for any $a \notin \mathcal{L}(P)$, P is $S \cup \{a\}$-stable.

Proposition 3.3.10 *If P is F_1-confluent and Q is F_2-confluent, then the following hold:*

1. *nil is $\emptyset$-confluent,*
2. *$(\mu_1 \mid \ldots \mid \mu_n)P$ is F_1-confluent,*
3. *$P \backslash L$ is $(F_1 - L)$-confluent,*
4. *$P[f]$ is $f(F_1)$-confluent given $f \lceil \mathcal{L}(P)$ is injective.*
5. *$P \mid\mid Q$ is F-confluent given:*
   - *(a) Q is $\overline{F_1}$-stable,*
   - *(b) P is $\overline{F_2}$-stable,*
   - *(c) $F \supseteq (F_1 \cup F_2 - (L \cup \overline{L})) \cup (\mathcal{L}(P) \cap \mathcal{L}(Q))$ where $L = \mathcal{L}(P) \cap \overline{\mathcal{L}(Q)}$.*

The last clause requires that an F-confluent process can only communicate with processes which are $\overline{F}$-stable in order that it can even remain semi-confluent. This condition ensures that there will be sufficient complementary actions to hide all the instances of semi-confluent actions and moreover each of these communicating actions do not change the state of the communicating process.

In order to prove the above proposition we need the following property of semi-confluent processes.

Lemma 3.3.11 *Let P be an F-confluent process, and let Q be a derivative of P. Then for $s \in \Lambda^*$, $\kappa \in \Lambda - F$, and $\phi \in F$, one of the following holds:*

1. *if*
$$\begin{tikzcd} Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "s"'] & Q_1 \\ Q_2 & \end{tikzcd}$$
then either

   (a)
$$\begin{tikzcd} & Q_1 \arrow[d, Rightarrow, "s"] \\ Q_2 \arrow[r, Rightarrow, "\kappa"] & \approx \end{tikzcd}$$
when $\kappa \notin s$ or

   (b) *there exists $s_1, s_2 \in \Lambda^*$ with $s = s_1 \kappa s_2$ and*
$$\begin{tikzcd} & Q_1 \arrow[d, Rightarrow, "s_1 s_2"] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd}$$
;

2. *if*
$$\begin{tikzcd} Q \arrow[r, "\phi"] \arrow[d, Rightarrow, "s"'] & Q_1 \\ Q_2 & \end{tikzcd}$$
then either

   (a) *$\phi \notin s$ and*
$$\begin{tikzcd} & Q_1 \arrow[d, Rightarrow, "s"] \\ Q_2 \arrow[r, Rightarrow, "\phi"] & \approx \end{tikzcd}$$
or,

   (b) *$\phi \in s$ and either*

      i. *there exists $s_1, s_2 \in \Lambda^*$ such that $s = s_1 \phi s_2$ and*
$$\begin{tikzcd} & Q_1 \arrow[d, Rightarrow, "s_1 s_2"] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd}$$
,or

ii.

$$\begin{tikzcd} & Q_1 \arrow[d, Rightarrow, "s"'] \\ Q_2 \arrow[r, Rightarrow, "\phi"] & \approx \end{tikzcd}$$

Proof: We proceed by induction on the length of s.

Consider case 1.a, then we have

$$\begin{tikzcd} Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "s"'] & Q_1 \\ Q_2 & \end{tikzcd}$$

and for $\kappa \notin s$ we wish to show that

$$\begin{tikzcd} Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "s"'] & Q_1 \arrow[d, Rightarrow, "s"'] \\ Q_2 \arrow[r, Rightarrow, "\kappa"] & \approx \end{tikzcd}$$

When $s = \epsilon$, this follows immediately from the definition of semi-confluence (second diagram). Otherwise, let $s = \mu s'$; then using the third diagram and assuming the completion for the shorter action sequence s', (if $\kappa \notin s$ then $\kappa \notin s'$) we can construct

$$\begin{tikzcd} Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "\mu"'] & Q_1 \arrow[d, Rightarrow, "\mu"'] \\ Q_3 \arrow[r, Rightarrow, "\kappa"] \arrow[d, Rightarrow, "s'"'] & Q_3' \arrow[d, Rightarrow, "s'"'] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd}$$

But if $Q_3 \overset{k}{\Longrightarrow} Q_3'$, then there exists R, R' such that $Q_3 \overset{\epsilon}{\Longrightarrow} R \xrightarrow{k} R' \overset{\epsilon}{\Longrightarrow} Q_3'$ with $Q_3 \approx R$ and $Q_3' \approx R'$ (by proposition 3.3.3). Hence as required the following can be constructed:

$$\begin{tikzcd}
Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "\mu"'] & Q_1 \arrow[d, Rightarrow, "\mu"'] \\
R \arrow[r, "\kappa"] \arrow[d, Rightarrow, "s'"'] & R' \arrow[d, Rightarrow, "s'"'] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}$$

The proofs of the other parts follows by a similar construction using the other properties of a semi-confluent process. □

Proof: (of cases 1 to 4 of proposition 3.3.10)

1. *nil* is $\emptyset$-confluent follows directly.
2. Given P is F_1-confluent then $(\mu_1 \mid \ldots \mid \mu_n)P$ is F_1-confluent, since the confluent actions prefix is well behaved; we simply retain those actions in P which were not confluent.
3. That $P\backslash L$ is $(F_1 - L)$-confluent whenever P is F_1-confluent follows immediately from the fact that we are only removing actions; since previously the diagrams could be completed they certainly can be now.
4. Given P is F_1-confluent and $f\lceil\mathcal{L}(P)$ is injective, then it follows directly that $P[f]$ is $f(F_1)$-confluent.

Proof: (of case 5 of proposition 3.3.10)

If P is F_1-confluent and Q is F_2-confluent, then to prove that there is an F such that $P \mid\mid Q$ is F-confluent we shall proceed in two stages: we shall show

1. that if $P \mid\mid Q \Longrightarrow R$ then $R \approx P \mid\mid Q$; and
2. that the visible actions behave as required.

1. We shall prove that $B = \{(P \parallel Q, R) \mid P \parallel Q \Longrightarrow P' \parallel Q' \approx R\}$ with P and Q, F_1 and F_2 confluent respectively, is a bisimulation.

   Let $P \overset{s}{\Longrightarrow} P'$ and $Q \overset{\bar{s}}{\Longrightarrow} Q'$ with $s \in Act^*$ and all visible actions $a \in s$ restricted in the composition $P \parallel Q$, so we have one of the following cases.

   Case 1: $P \parallel Q \xrightarrow{\tau} P_1 \parallel Q_1$; we require $R \Longrightarrow R'$ with $(P_1 \parallel Q_1, R') \in B$. We could have inferred $P \parallel Q \xrightarrow{\tau} P_1 \parallel Q_1$ in a number of ways:

   Subcase 1.1: $P \xrightarrow{\tau} P_1$ with $Q \equiv Q_1$; then

$$\begin{tikzcd} P \arrow[r, "\tau"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s"'] \\ P_2 \arrow[r, Rightarrow] & P_4 \approx P_3 \end{tikzcd} \qquad \text{(lemma 3.3.11)}$$

   we require $Q \overset{\bar{s}}{\Longrightarrow} Q_2$ and since from proposition 3.3.10 $P_2 \approx P_3$, then $P_1 \parallel Q_1 \Longrightarrow P_2 \parallel Q_2 \approx P_1 \parallel Q_2 \approx R'$. Thus $(P_1 \parallel Q_1, R')$ is in the relation, and we can prove likewise for $Q \xrightarrow{\tau} Q_1$.

   Subcase 1.2: $P \xrightarrow{\lambda} P_1$ and $Q \xrightarrow{\bar{\lambda}} Q_1$, (Note: if $\lambda \in F_1$ then $Q_1 \approx Q$ from the stability requirement on Q, similarly if $\bar{\lambda} \in F_2$.) We have two cases dependent on whether λ is fully confluent or not:

   Subcase 1.2.1: if $\lambda \notin F_1 \cup F_2$ then given

$$\begin{tikzcd} P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \\ P_2 & \end{tikzcd}$$

   this divides into two subcases:

   Subcase 1.2.1.1: if $\lambda \in s$; then

$$\begin{tikzcd} P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s_1 s_2"'] \\ P_2 \arrow[r, Rightarrow] & P_3 \approx P_4 \end{tikzcd}$$

   with $s = s_1 \lambda s_2$, and we can find

$$\begin{tikzcd} Q \arrow[r, "\bar{\lambda}"] \arrow[d, Rightarrow, "\bar{s}"'] & Q_1 \arrow[d, Rightarrow, "\overline{s_1 s_2}"'] \\ Q_2 \arrow[r, Rightarrow] & Q_3 \approx Q_4 \end{tikzcd}$$

so $P_1 \parallel Q_1 \Longrightarrow P_4 \parallel Q_4 \approx P_3 \parallel Q_3$ and $P_2 \parallel Q_2 \Longrightarrow P_3 \parallel Q_3$ hence $R \Longrightarrow R'' \approx P_3 \parallel Q_3$ and $R'' \approx R_4 \parallel Q_4$ thus $R \approx P_2 \parallel Q_2$ as required,

Subcase 1.2.1.2: if $\lambda \notin s$ then from lemma 3.3.11we can find

$$\begin{tikzcd}
P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_2 \arrow[d, Rightarrow, "s"'] & Q \arrow[r, "\bar{\lambda}"] \arrow[d, Rightarrow, "\bar{s}"'] & Q_2 \arrow[d, Rightarrow, "\bar{s}"'] \\
P_1 \arrow[r, Rightarrow, "\lambda"] & P_3 \approx P_4 & Q_1 \arrow[r, Rightarrow, "\bar{\lambda}"] & Q_3 \approx Q_4
\end{tikzcd}$$

so $P_1 \parallel Q_1 \Longrightarrow P_3 \parallel Q_3 \approx P_4 \parallel Q_4$ and $P_2 \parallel Q_2 \Longrightarrow P_4 \parallel Q_4$ hence $R \Longrightarrow R'' \approx P_3 \parallel Q_3$ and $R'' \approx P_4 \parallel Q_4$ thus $R \approx P_2 \parallel Q_2$ as required;

Subcase 1.2.2: if $\lambda \in F_1 \cup F_2$ (assume wlog $\lambda \in F_1$) once again $P \xrightarrow{\lambda} P'$ and $Q \xrightarrow{\bar{\lambda}} Q'$ but we require Q to be $\overline{F_1}$-stable, so from the definition of stability $Q' \approx Q$; (Letting s/λ denote the string $s \in \Lambda^*$ with a single instance of λ deleted from it.) Once again we have two sub-cases:

Subcase 1.2.2.1: if $\lambda \notin s$

$$\begin{tikzcd}
P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s"'] & Q \arrow[r, "\bar{\lambda}"] \arrow[d, Rightarrow, "\bar{s}"'] & Q_1 \arrow[d, Rightarrow, "\bar{s}"'] \\
P_2 \arrow[r, Rightarrow, "\lambda"] & P_4 \approx P_3 & Q_2 \arrow[r, Rightarrow, "\bar{\lambda}"] & Q_4 \approx Q_3
\end{tikzcd}$$

now $Q \approx Q_1$ from stability, hence $Q_3 \approx Q_2$ and $Q_4 \approx Q_2$ (transitivity and stability),

so $P_1 \parallel Q_1 \Longrightarrow P_3 \parallel Q_3 \approx P_4 \parallel Q_4 \approx P_4 \parallel Q_2$ also $R \Longrightarrow R'' \approx P_3 \parallel Q_3$ thus $(P_1 \parallel Q_1, R) \in B$,

Subcase 1.2.2.2: if $\lambda \in s$

$$\begin{tikzcd}
P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s/\lambda"'] & Q \arrow[r, "\bar{\lambda}"] \arrow[d, Rightarrow, "\bar{s}"'] & Q_1 \arrow[d, Rightarrow, "\bar{s}/\bar{\lambda}"'] \\
P_2 \arrow[r, Rightarrow] & P_4 \approx P_3 & Q_2 \arrow[r, Rightarrow] & Q_4 \approx Q_3
\end{tikzcd}$$

(The completion of both squares is given by lemma 3.3.11.) so $P_2 \parallel Q_2 \Longrightarrow P_4 \parallel Q_4 \approx P_3 \parallel Q_3$ and $P_1 \parallel Q_1 \Longrightarrow P_3 \parallel Q_3$ thus $R \Longrightarrow R'' \approx P_4 \parallel Q_4$ and hence $(P_1 \parallel Q_1, R) \in B$.

Note: if $\lambda \in F_1 \cap F_2$ then its complement must be stable in one of P or Q, for it to take part in a communication; and having been restricted the other action can take no further part. So we need only consider actions in each of F_1 and F_2 separately. A symmetric argument will prove the case for $\lambda \in F_2$.

Case 2: Consider $P \parallel Q \xrightarrow{\lambda} P_1 \parallel Q_1$ for instance $P \xrightarrow{\lambda} P_1$ with $Q_1 \equiv Q$ we require $R \overset{\lambda}{\Longrightarrow} R_1$ with $(P_1 \parallel Q_1, R_1) \in B$. From lemma 3.3.11 we have for $\lambda \notin s$:

$$\begin{tikzcd} P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s"'] \\ P_2 \arrow[r, Rightarrow, "\lambda"] & P_4 \approx P_3 \end{tikzcd} \quad \text{with } Q \overset{\bar{s}}{\Longrightarrow} Q_1$$

so $P_2 \parallel Q_2 \overset{\lambda}{\Longrightarrow} P_4 \parallel Q_2 \approx P_3 \parallel Q_2 \approx P_1 \parallel Q$ thus $R \overset{\lambda}{\Longrightarrow} R_1 \approx P_4 \parallel Q_4$ and hence $R_1 \approx P_1 \parallel Q_1$ as required. The other case follows by an almost identical argument, as does the requirement for $Q \xrightarrow{\lambda} Q'$.

- The symmetric case $R \xrightarrow{\mu} R'$, can be proved as follows; we require $P \parallel Q \overset{\hat{\mu}}{\Longrightarrow} P_1 \parallel Q_1$ with $(P_1 \parallel Q_1, R') \in B$ but $P \parallel Q \Longrightarrow P_2 \parallel Q_2 \approx R$ by definition so $P \parallel Q \overset{\hat{\mu}}{\Longrightarrow} P_1 \parallel Q_1 \approx R_1$ directly. So our construction does not introduce any non-determinism on the silent actions.

2. Visible actions.

   (a) Determinism requirement, there are two cases:

   Case 1: $\lambda \notin F$; given the following:

$$\begin{tikzcd} P \parallel Q \arrow[r, "\lambda"] \arrow[d, Rightarrow] & S_1 \\ S_2 \arrow[r, Rightarrow, "\lambda"] & S_4 \end{tikzcd}$$

   We must prove that it can be completed. Assume the λ action comes from P then we have:

$$\begin{tikzcd} P \parallel Q \arrow[r, "\lambda"] \arrow[d, Rightarrow] & P_1 \parallel Q \\ P_2 \parallel Q_1 \arrow[r, Rightarrow, "\lambda"] & P_3 \parallel Q_2 \end{tikzcd}$$

(Since if $\lambda \notin F$ then if $\lambda \in \mathcal{L}(P)$ then $\lambda \notin \mathcal{L}(Q)$.) so for some s (We only consider the case $\lambda \notin s$.):

$$\begin{tikzcd} P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s"'] \\ P_2 \arrow[r, Rightarrow, "\lambda"] \arrow[d, Rightarrow, "\lambda"'] & P_4 \approx P_3 \arrow[d, Rightarrow] \\ P_5 \arrow[r, Rightarrow] & P_6 \approx P_7 \end{tikzcd} \qquad \text{with } Q \overset{\bar{s}}{\Longrightarrow} Q_2,$$

now $P_5 \approx P_6$ and $P_7 \approx P_4$ (proposition 3.3.10) hence $P_1 \parallel Q \Longrightarrow P_3 \parallel Q_2$ so $P_1 \parallel Q \approx P_5 \parallel Q_2$ and therefore we can form:

$$\begin{tikzcd} P \parallel Q \arrow[r, "\lambda"] \arrow[d, Rightarrow] & S_1 \arrow[d, Rightarrow] \\ S_2 \arrow[r, Rightarrow, "\lambda"] & S_3 \approx S_4 \end{tikzcd} .$$

The other case is identical up to erasure of one occurrence of λ and therefore omitted.

Case 2 $\lambda \in F$ there are two subdivisions of this case:

Subcase 2.1: $\lambda \notin \mathcal{L}(Q)$ (In other words, $\lambda \in F_1$ and in a sense is not a new 'bad' action, it has not been made non-confluent by the parallel composition, it already was non-confluent.) So we need to be able to complete the following:

$$\begin{tikzcd} P \parallel Q \arrow[r, "\lambda"] \arrow[d, Rightarrow] & S_1 \\ S_2 \arrow[r, Rightarrow, "\lambda"] & S_4 \end{tikzcd}$$

and since we are working with the fifth requirement we need either that $S_1 \approx S_4$ or the existence of S_3 and S_5 such that $S_2 \overset{\lambda}{\Longrightarrow} S_3$ and $S_4 \overset{\lambda}{\Longrightarrow} S_5$ with $S_3 \approx S_5$; so once again we have two cases:

Subcase 2.1.1: in the former situation:

$$\begin{tikzcd} P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s"'] \\ P_2 \arrow[r, Rightarrow, "\lambda"] & P_4 \approx P_3 \end{tikzcd}$$

(From lemma 3.3.11) and we can complete with $Q \overset{\bar{s}}{\Longrightarrow} Q_2$ as usual,

Subcase 2.1.2: and in the latter where we have:

$$\begin{tikzcd} P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s/\lambda"'] \\ P_2 \arrow[r, Rightarrow] & P_4 \approx P_3 \end{tikzcd}$$

and we can complete using $Q \overset{\overline{s/\lambda}}{\Longrightarrow} Q_2$.

The case for $\lambda \notin \mathcal{L}(P)$ follows by symmetry.

Subcase 2.2: $\lambda \in \mathcal{L}(P) \cap \mathcal{L}(Q)$ so we require that when

$$\begin{tikzcd} P \parallel Q \arrow[r, "\lambda"] \arrow[d, Rightarrow] & S_1 \\ S_2 \arrow[r, Rightarrow, "\lambda"] & S_4 \end{tikzcd}$$

then either (as before) $S_1 \approx S_4$ or there exists S_5 and S_3 such that $S_1 \overset{\lambda}{\Longrightarrow} S_3$ and $S_4 \overset{\lambda}{\Longrightarrow} S_5$ with $S_3 \approx S_5$; once again we have two sub-cases:

Subcase 2.2.1: $\lambda \notin F_1 \cup F_2$ so if $P \xrightarrow{\lambda} P'$ and $P \overset{\lambda}{\Longrightarrow} P''$ then $P'' \approx P'$ and similarly if $Q \xrightarrow{\lambda} Q'$ and $Q \overset{\lambda}{\Longrightarrow} Q''$ then $Q'' \approx Q'$. So the following diagrams can be completed (lemma 3.3.11)

$$\begin{tikzcd} P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s"'] & Q \arrow[r, "\lambda"] \arrow[d, Rightarrow, "\bar{s}"'] & Q_1 \arrow[d, Rightarrow, "\bar{s}"'] \\ P_2 \arrow[r, Rightarrow, "\lambda"] & P_4 \approx P_3 & Q_2 \arrow[r, Rightarrow, "\lambda"] & Q_3 \approx Q_4 \end{tikzcd}$$

and combining them we have:

$$\begin{tikzcd} P \parallel Q \arrow[r, "\lambda"] \arrow[d, Rightarrow, "\lambda"'] & P_1 \parallel Q_1 \arrow[d, Rightarrow, "\lambda"'] \\ P_2 \parallel Q_2 \arrow[r, Rightarrow, "\lambda"] & P_4 \parallel Q_3 \approx P_3 \parallel Q_4 \end{tikzcd}$$

Subcase 2.2.2: $\lambda \in F_1 \cup F_2$ then from earlier we can always complete the diagram as appropriate just using the visible evolution of one of the processes and since our processes are semi-confluent up to the appropriate sets that is always possible.

(b) Confluence Requirement. Given $\lambda_1 \neq \lambda_2$ we must show that the following diagram can be completed:

$$\begin{tikzcd}
P \parallel Q \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2"'] & S_1 \arrow[d, Rightarrow, "\lambda_2"'] \\
S_2 \arrow[r, Rightarrow, "\lambda_1"] & S_4 \approx S_3
\end{tikzcd}$$

The proof that follows is precisely that of [Mill]. There are three cases (but owing to symmetry we only need consider two of them).

Case 1: $\lambda_1, \lambda_2 \in \mathcal{L}(P)$ (or equally both in $\mathcal{L}(Q)$). then we have the following completion:

$$\begin{tikzcd}
P \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2 s"'] & P_1 \arrow[d, Rightarrow, "\lambda_2 s"'] \\
P_2 \arrow[r, Rightarrow, "\lambda_1"] & P_4 \approx P_3
\end{tikzcd}$$

using lemma 3.3.11 and $Q \overset{\bar{s}}{\Longrightarrow} Q_2$ we can form:

$$\begin{tikzcd}
P \| Q \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2"'] & P_1 \| Q \arrow[d, Rightarrow, "\lambda_2"'] \\
P_2 \| Q_2 \arrow[r, Rightarrow, "\lambda_1"] & P_4 \| Q_2 \approx P_3 \| Q_2
\end{tikzcd}$$

Case 2: $\lambda_1 \in \mathcal{L}(P)$ and $\lambda_2 \in \mathcal{L}(Q)$, consider

$$\begin{tikzcd}
P \parallel Q \arrow[r, "\lambda_1"] \arrow[d, Rightarrow] & P_1 \parallel Q \\
P_2 \parallel Q_1 \arrow[r, Rightarrow, "\lambda_2"] & P_2 \parallel Q_2
\end{tikzcd}$$

now applying lemma 3.3.11 twice we have:

$$\begin{tikzcd}
P \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "s"'] & P_1 \arrow[d, Rightarrow, "s"'] & Q \arrow[r, "\lambda_2"] \arrow[d, Rightarrow, "\bar{s}"'] & Q_1 \arrow[d, Rightarrow, "\bar{s}"'] \\
P_2 \arrow[r, Rightarrow, "\lambda_1"] & P_4 \approx P_3 & Q_2 \arrow[r, Rightarrow, "\lambda_2"] & Q_4 \approx Q_3
\end{tikzcd}$$

so combining we have:

$$\begin{tikzcd}
P \| Q \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2"'] & P_1 \| Q \arrow[d, Rightarrow, "\lambda_2"'] \\
P_2 \| Q_4 \arrow[r, Rightarrow, "\lambda_1"] & P_4 \| Q_4 \approx P_3 \| Q_3
\end{tikzcd}$$

as required.

We claimed that $P \parallel Q$ is semi-confluent to F where $F \supseteq (F_1 \cup F_2 - (L \cap \overline{L})) \cup (\mathcal{L}(P) \cap \mathcal{L}(Q))$ and $L = \mathcal{L}(P) \cap \overline{\mathcal{L}(Q)}$; this is obtained from the cases in the proof above. This is true due to the following reasoning. Firstly since any communicating pair of actions is restricted all such pairs can be removed from the semi-confluent action set. Secondly we must add all those actions which appear both in P and in Q. Thirdly all 'bad' actions which were not removed by communication must still be 'bad' actions. □

3.3.1 Recursion.

So far our semi-confluent constructors have only permitted us to construct finite processes. Unfortunately in order to build stable processes and other interesting processes, we have to use recursive definitions.

Proposition 3.3.12 *Given a closed term, where the only constructions used in its defintion are the semi-confluent operators (given in Proposition 3.3.10), then the process given by that term is semi-confluent, and moreover it is semi-confluent up to the set defined by the constructions used and given in Proposition 3.3.10.*

Proof: Consider the set of recursive equations:

$$P_i \overset{def}{\equiv} E_i$$

We will show that if the E_i are built only from semi-confluent operators, then for some F the processes P_i are F-confluent. We proceed by induction over the depth of inference required to infer the derivation of $P_i \xrightarrow{\mu} P'$ and the number of confluent action prefixes used. Thus we shall assume that there is some F' such that P' is F'-confluent whenever the depth of inference on P' is less than that for P, or there are fewer action prefixes in the defintion of P'. Now we simply proceed by case analysis over the structure of P. For simplicity we only consider one recursive equation.

1. $P \equiv A$; then since $A \equiv E$ we have by a shorter inference the action $P \xrightarrow{\mu} P'$ thus P is semi-confluent up to some set F.

2. $P \equiv (\mu_1 \mid \ldots \mid \mu_n)Q$; the process Q has fewer confluent action prefixes than the process P. So there is a set F up to which Q is semi-confluent. We wish to show the above process is semi-confluent up to the same set F (i.e. the requisite diagrams can be completed.)

   (a) If we can infer:

$$\begin{tikzcd} P \arrow[r, "\tau"] \arrow[d, Rightarrow] & P_1 \\ P_2 & \end{tikzcd}$$

   then there is a $\mu_i = \tau$ (wlog let it be μ_1). Then $P_1 \equiv (\mu_2 \mid \ldots \mid \mu_n)Q$ and either directly $P_1 \approx P_2$, otherwise there is a $\mu_i = \tau$ for $i > 1$ and a (possibly empty) set of actions $\mu'_1, \ldots \mu'_k$ which are not τ, then:

$$P_1 \Longrightarrow (\mu'_1 \mid \ldots \mid \mu'_k)Q,$$
$$P_2 \Longrightarrow (\mu'_1 \mid \ldots \mid \mu'_k)Q,$$

   as required.

   (b) If we can infer:

$$\begin{tikzcd} P \arrow[r, "\tau"] \arrow[d, Rightarrow, "\lambda"'] & P_1 \\ P_2 & \end{tikzcd}$$

   then there exists $\mu_i = \tau$ (say μ_1) and we can assume wlog (since the above held) that no other $\mu_i = \tau$. (Moreover if $P \equiv (\tau)Q$ then closure follows immediately from semi-confluence of Q.) So let $\mu_2 = \lambda$. Then

$$P_1 \overset{\lambda}{\Longrightarrow} (\mu_3 \mid \ldots \mid \mu_n)Q,$$
$$P_2 \Longrightarrow (\mu_3 \mid \ldots \mid \mu_n)Q,$$

   completing the diagram as required.

   (c) Given that we can infer:

$$\begin{tikzcd} P \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2"'] & P_1 \\ P_2 & \end{tikzcd} \qquad \text{with } \lambda_1 \neq \lambda_2$$

(We can assume wlog there are no τ actions in $\mu_1 \dots \mu_n$ since we can remove them using the two sub-proofs above.) and choose $\lambda_1 = \mu_1$ and $\lambda_2 = \mu_2$ then the following can be inferred:

$$P_1 \overset{\lambda_2}{\Longrightarrow} (a_3 \mid \dots \mid a_n)Q,$$
$$P_2 \overset{\lambda_1}{\Longrightarrow} (a_3 \mid \dots \mid a_n)Q,$$

as required to close the diagram.

(d) Given the following:

$$\begin{tikzcd} P \arrow[r, "\lambda"] \arrow[d, Rightarrow, "\lambda"'] & P_1 \\ P_2 & \end{tikzcd}$$

once again we can assume that none of the μ_i is τ. Thus there is either a unique $\mu_i = \lambda$ in which case $P_1 \approx P_2$ follows immediately, or if more than one of the μ_i are λ; then whichever we choose the remaining confluent action prefixes are simply permutations of each other and as was noted earlier they are therefore equivalent. So $P_1 \approx P_2$ as required.

The derivation closure properties follow from the semi-confluence of Q and any derivative of P which is not a derivative of Q can be shown confluent by the above argument as it will be a shorter confluent action prefix of Q.

3. $P \equiv nil$; is $\emptyset$-confluent immediately.

4. $P \equiv Q \backslash L$; is $(F - L)$ confluent if Q is F-confluent.

5. $P \equiv Q \parallel R$; since we infer $Q \xrightarrow{\mu} Q'$ and $R \xrightarrow{\mu'} R'$ by a shorter inference we can assume that Q and R are F_1 and F_2-confluent respectively, and that the appropriate stability requirements are met, then P is F-confluent where F is (as before)
$F \supseteq (F_1 \cup F_2 - (L \cup \overline{L})) \cup (\mathcal{L}(Q) \cap \mathcal{L}(R))$ with $L = \mathcal{L}(P) \cap \overline{\mathcal{L}(Q)}$. The proof is identical to the one given earlier excepting that the induction is over the depth of inference, and is therefore omitted. □

3.4 Value Passing.

In order to be able to express semi-confluence in the value passing calculus, we need a partition extension in the same form as the extension to confluence given in the last chapter.

Definition 3.4.1 *The pair $[S_1, S_2]$ is a* double partition[1] *of the action set Λ, where $S_1 = \{\Phi_i \mid i \in I\}$ and $S_2 = \{\Psi_j \mid j \in J\}$ iff the following hold:*

1. *for all $i \in I$, $\Phi_i \subseteq \Lambda$ and for all $j \in J$, $\Psi_j \subseteq \Lambda$,*

2. *$\Phi_i \cap \Phi_k = \emptyset$ for $i \neq k$,*

3. *$\Psi_j \cap \Psi_l = \emptyset$ for $j \neq l$,*

4. *$\Phi_i \cap \Psi_j = \emptyset$ for all i, j.*

Definition 3.4.2 *Let $[S_1, S_2]$ be a double partition; then we write $\lambda_1 \sharp_{[S_1,S_2]} \lambda_2$ if $\lambda_1 \sharp_{S_1} \lambda_2$ and $\lambda_1 \sharp_{S_2} \lambda_2$.*

As before we will drop the suffix when the double partition used is clear from the context.

Definition 3.4.3 *A process P is* partition semi-confluent to $[S_1, S_2]$ *(written $[S_1, S_2]$-confluent) iff for any derivative Q of P the following diagrams can be completed:*

$$\begin{tikzcd}
Q \arrow[r, "\tau"] \arrow[d, Rightarrow] & Q_1 \arrow[d, Rightarrow] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}
\qquad
\begin{tikzcd}
Q \arrow[r, "\tau"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow, "\lambda"'] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}
\qquad
\begin{tikzcd}
Q \arrow[r, "\lambda_1"] \arrow[d, Rightarrow, "\lambda_2"'] & Q_1 \arrow[d, Rightarrow, "\lambda_2"'] \\
Q_2 \arrow[r, Rightarrow, "\lambda_1"] & \approx
\end{tikzcd}
\qquad
\begin{tikzcd}
Q \arrow[r, "\kappa"] \arrow[d, Rightarrow, "\kappa"'] & Q_1 \arrow[d, Rightarrow] \\
Q_2 \arrow[r, Rightarrow] & \approx
\end{tikzcd}$$

$$\lambda_1 \sharp \lambda_2 \qquad\qquad \kappa \in \Phi_i \in S_1$$

[1]This is a non-standard use of the word parition, as we do not require coverage of the whole base set, strictly it is a double sub-partition

$$\text{if}\quad \begin{tikzcd} Q \arrow[r, "\phi"] \arrow[d, Rightarrow, "\phi"'] & Q_1 \\ Q_2 & \end{tikzcd} \qquad \phi \in \Psi_j \in S_2 \qquad \left\{ \begin{array}{l} \textit{then either } Q_1 \approx Q_2 \textit{ or there exists } Q_3 \textit{ and} \\ Q_4 \textit{ such that } Q_1 \stackrel{\phi}{\Longrightarrow} Q_3 \textit{ and } Q_2 \stackrel{\phi}{\Longrightarrow} Q_4 \\ \textit{with } Q_3 \approx Q_4. \end{array} \right.$$

The interpretation of the above is that in the double-partition $[S_1, S_2]$ the actions in the sets of S_1 are well-behaved, being both deterministic and confluent, and the actions in the sets of S_2 have the ability to be non-deterministic for one step. If we use the double partition:

$$[\{\{\lambda_i\} \mid \lambda_i \in \Lambda - F\}, \{\{\lambda_j\} \mid \lambda_j \in F\}],$$

then we have F-confluence, and using the following double partition:

$$[\{\{\lambda_i\} \mid \lambda_i \in \Lambda\}, \emptyset]$$

we recover the original definition of weak confluence.

Proposition 3.4.4 *Partition semi-confluence is observation closed.*

Proposition 3.4.5 *If $P \xrightarrow{\tau} P'$ and P partition semi-confluent then $P' \approx P$.*

Proposition 3.4.6 *If P is $[S_1, S_2]$-confluent and $Q \approx P$ then Q is $[S_1, S_2]$-confluent.*

The proofs of the above proposition are identical to those given for semi-confluence (propositions 2.4.5, 2.4.7 and 2.4.6) and are therefore omitted.

Definition 3.4.7 *Two partitions S and S' are* compatible *iff given $\Phi_i \in S$ and $\lambda_i \in \Phi_i$, if $\lambda_i \in \Psi_j$ for some $\Psi_j \in S'$ then $\Psi_j = \Phi_j$.*

We will use the abbreviation "$S_1, \ldots S_n$ *are compatible*" to mean that the partitions are pairwise compatible.

Definition 3.4.8 *Let L be a set of action names, and let Φ_i be a set in some partition. Then Φ_i is L-*closed *iff given $\lambda \in \Phi_i$, if $\lambda \in L$ then for all $\lambda' \in \Phi_i$, $\lambda' \in L$. In other words, $\Phi_i \cap L \neq \emptyset$ implies $\Phi_i \subseteq L$.*

Proposition 3.4.9 *(Weakening) If P is $[S_1, S_2]$-confluent and $S_1 \sqsubseteq S$ and $S_2 \sqsubseteq S'$ with $S \cap S' = \emptyset$ and S, S', S_1, S_2 compatible, then P is $[S, S']$-confluent.*

Proof: Immediate from the definition of partition semi-confluence.

Proposition 3.4.10 *(Strengthening) If P is $[S_1, S_2]$-confluent and for all derivatives Q of P and for all $\lambda \in \Psi_j \in S_2$ the following holds:*

$$\begin{tikzcd} Q \arrow[r, "\lambda"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd}$$

then P is $[S_1 \cup \Psi_j, S_2 - \Psi_j]$-confluent.

Proof: Immediate from the definition of partition semi-confluence.

Proposition 3.4.11 *The following constructions preserve partition semi-confluence. Given P_i $[S_1, S_2]$-confluent and Q $[S_3, S_4]$-confluent, then:*

1. *nil is $[S, \emptyset]$-confluent (for any S);*

2. *$(a_1 \mid \ldots \mid a_n)P_1$ is $[S_1, S_2]$-confluent;*

3. *$\Sigma a_i.P_i$ is $[S_1, S_2]$-confluent whenever for all $a_i \in \Lambda_m \in S_1 \cup S_2$, all the sets in $S_1 \cup S_2$;*

4. *$P_1 \backslash L$ is $[S_1', S_2']$-confluent, where $S_1' = \{\Phi - L \mid \Phi \in S_1\}$ and $S_2' = \{\Psi - L \mid \Psi \in S_2\}$;*

5. *$P_1[f]$ is $[S_1', S_2']$-confluent, where $f\lceil\mathcal{L}(P_1)$ is injective with $S_1' = \{f(\Phi) \mid \Phi \in S_1\}$ and $S_2' = \{f(\Psi) \mid \Psi \in S_2\}$;*

6. *$P_1 \parallel Q$ is $[S, S']$-confluent, where:*

   (a) *P_1 is $\overline{\Psi_j}$-stable for all $\Psi_j \in S_4$,*

   (b) *Q is $\overline{\Psi_j}$-stable for all $\Psi_j \in S_2$,*

   (c) *S_1, S_2, S_3, S_4 are compatible,*

   (d) *for all $\Phi_i \in S_1 \cup S_3$, Φ_i is $\mathcal{L}(P) \cup \mathcal{L}(Q)$ closed,*

   (e) *for all $\Psi_j \in S_2$, Ψ_j is $\overline{\mathcal{L}(Q)}$ closed,*

   (f) *for all $\Psi_j \in S_4$, Ψ_j is $\overline{\mathcal{L}(P)}$ closed;*

   let $L = \mathcal{L}(P) \cup \overline{\mathcal{L}(Q)}$ with

   $$S = S_1 \cup S_3 - \{\Phi_i \in (S_1 \cup S_3) \mid \lambda \in \Phi_i \text{ and either } (\lambda \in \mathcal{L}(P) \cup \mathcal{L}(Q) \text{ or } \lambda \in (L \cup \overline{L}))\}$$

   and

   $$S' = S_2 \cup S_4 - \{\Psi_j \in (S_2 \cup S_4) \mid \lambda \in \Psi_i \text{ and } \lambda \in (L \cup \overline{L})\} \cup \{\Phi_i \in (S_1 \cup S_3) \mid \lambda \in \Phi_i \text{ and } \lambda \in (\mathcal{L}(P) \cap \mathcal{L}(Q))\};$$

7. *any closed term formed using the above is $[S, S']$-confluent with S, S' given by the above constructions.*

The extensive conditions on the parallel composition are in fact just those we had before with some additions to ensure that partitions are maintained throughout the construction. We omit a proof of the above as it replicates that for proposition 3.3.10 and is identical in every detail up to the conditions required for the partition to be maintained, which is a technical nicety and not at all informative.

3.5 Conclusion

We have presented in the last two chapters a family of confluence definitions related in the following fashion:

$$\begin{array}{ccccc} \text{confluence} & \subset & \text{semi-confluence} & \subset & \text{partial confluence} \\ \cup & & \cup & & \\ \text{partition-confluence} & \subset & \text{partition semi-confluence.} & & \end{array}$$

Semi-confluence is (up to partitioning) the largest confluence we have discovered where it is possible to allow processes to become temporarily non-confluent and to restore the confluence without just removing all future behaviours guarded by the actions that lead to non-confluence. There may be many notions of confluence lying between semi-confluence and partial confluence which have this desirable property, but it is not clear how we should weaken our definition to find them.

Unfortunately we are still not able to provide a complete set of confluence-preserving operators for confluent processes. We cannot yet show that the process

$$a \mid a$$

is confluent by force of construction, other than by demonstrating that we can construct an equivalent process:

$$(a \mid a)nil$$

which is confluent.

Chapter 4

Determinism in CIMP.

4.1 Introduction.

Within the language CIMP there is the capability to introduce non-deterministic behaviour. There are valid programs whose output is not uniquely determined by the initial values of the program variables and the program text. In order to ascertain variable values during execution, we have to refer to the particular computation path chosen by any CIMP interpreter/compiler. For example the following program fragment:

$$X := 1 \ \mathit{PAR} \ X := 2$$

can result in the variable X having value 1 or 2. The potential presence of such non-determinism makes programming in CIMP a difficult task. The lack of determinism removes the predictability from the programming language, which is essential to straight forward programming. We wish to identify a sub-language of CIMP within which the results of computation are independent of the computation path chosen. Such a sub-language can be considered effectively deterministic. Demanding this property returns us to conventional sequential programming, where a variable's value is uniquely determined by the initial values and the program text. This statement has to be slightly modified; in the case of CIMP at the termination of the execution of any sequential fragment the variable values will be determined by the initial state and the commands executed up to this point. For example we can predict the variable values at the points $1, 2, 3$ in the following, if it is deterministic

$$(C_a \ \mathit{PAR} \ C_b);_1 C_c;_2 (C_d \ \mathit{PAR} \ C_e);_3 .$$

The only construct in our language which can introduce non-determinism is *PAR* . In the absence of that construct we certainly have a language which is deterministic in the manner required, but unfortunately we can no longer compute concurrently. Thus in order to have a concurrent deterministic language we must identify those parallel statements which introduce non-determinism. It is certainly *not* the case that all parallel statements introduce non-determinism. Consider for example;

$$X := 1 \ \mathit{PAR} \ Y := 4$$

which is deterministic in the above sense.

In the first example the non-determinism resulted from attempting to make two simultaneous changes to a variables value. Do we obtain determinism by forbidding simultaneous *changing* of variable's value? Consider the program fragment:

$$\begin{gathered} X := 4; \\ X := 6 \ \mathit{PAR} \ X := X \end{gathered}$$

this obeys the above restriction but unfortunately is *not* deterministic. At termination the variable X will have a value of either 4 or 6. The non-determinism results not from simultaneous changes, but from simultaneous writing to a particular variable. Can determinism be enforced by forbidding simultaneous writing to a variable?

$$\begin{gathered} X := 4; \\ X := 6 \ \mathit{PAR} \ Y := X \end{gathered}$$

In the above fragment the variable X's behaviour is deterministic, and we do not attempt to perform any simultaneous writing to any variable; but the variable Y can have a value of either 4 or 6 at the execution's completion. Thus we see that accessing a variable written to in another fragment of a *PAR* command can lead to the introduction of non-determinism. The condition this leads us to impose is that in a *PAR* command no fragment may access a variable *written* by another either for reading or for writing.

4.1.1 Read and Write sets.

In order to formalise the above restriction we will define a read set of a program fragment, written $\mathcal{R}(C)$, and the write set, written $\mathcal{W}(C)$.

Definition 4.1.1 *The sets $\mathcal{R}(C)$ and $\mathcal{W}(C)$ are defined inductively over the structure of program fragments as follows:*

Phrase	$\mathcal{W}(Phrase)$	$\mathcal{R}(Phrase)$
Xi	$\emptyset$	$\{Xi\}$
$F(E_1, ...E_n)$	$\emptyset$	$\mathcal{R}(E_1) \cup ... \cup \mathcal{R}(E_n)$
$Xi := E$	$\{Xi\}$	$\mathcal{R}(E)$
$C_1; C_2$	$\mathcal{W}(C_1) \cup \mathcal{W}(C_2)$	$\mathcal{R}(C_1) \cup \mathcal{R}(C_2)$
C_1 *PAR* C_2	$\mathcal{W}(C_1) \cup \mathcal{W}(C_2)$	$\mathcal{R}(C_1) \cup \mathcal{R}(C_2)$
IF E *THEN* C_1 *ELSE* C_2	$\mathcal{W}(C_1) \cup \mathcal{W}(C_2)$	$\mathcal{R}(E) \cup \mathcal{R}(C_1) \cup \mathcal{R}(C_2)$
LET $Xi = E$ *IN* C	$\mathcal{W}(C) - \{Xi\}$	$\mathcal{R}(E) \cup \mathcal{R}(C) - \{Xi\}$
WHILE E *DO* C	$\mathcal{W}(C)$	$\mathcal{R}(E) \cup \mathcal{R}(C)$

Definition 4.1.2 *Two commands C_1 and C_2 are* non-interfering *iff*

- $(\mathcal{R}(C_1) \cup \mathcal{W}(C_1)) \cap \mathcal{W}(C_2) = \emptyset$,
- $(\mathcal{R}(C_2) \cup \mathcal{W}(C_2)) \cap \mathcal{W}(C_1) = \emptyset$.

Definition 4.1.3 *(The derived deterministic parallel construct PAR_d)*

We can write C_1 PAR_d C_2 precisely when C_1 and C_2 are non-interfering.

We call the sub-language of CIMP produced when only this restricted form of parallel composition is allowed DCIMP. In order to prove that this restriction of language constructs enforces determinism we shall show that a corresponding effect has taken place within our semantics. In other words, we show that the processes representing programs in DCIMP are confluent.

4.2 DCIMP is Deterministic.

It is not possible to prove that for any program fragment C, the process $[\![C]\!]$ representing it is deterministic, since for instance

$$[\![Y := X \;\; PAR_d \;\; Z := X]\!]$$

is the following process (as defined in chapter 1)

$$P \equiv ((a_X x.\overline{r}x \mid rx.\overline{g}_Y x.\overline{d}_1)\backslash\{r\} \mid (a_X x.\overline{r}x \mid rx.\overline{g}_Z x.\overline{d}_2)\backslash\{r\} \mid d_1.d_2.\overline{d})\backslash\{d_1, d_2\}$$

and we have composed two agents which are *not* sort disjoint in parallel. This does not directly imply that the process is not confluent but in this case it is sufficient to ensure non-confluence, since (ignoring values) we can infer

$$\begin{tikzcd} P \arrow[r, "a_X x"] \arrow[d, Rightarrow, "a_X x"'] & Q \\ R & \end{tikzcd}$$

with $Q \not\approx R$, violating the determinism requirement. However when we combine this process with the processes representing the variables we then have a process which is deterministic. All the 'local' non-determinism has been hidden as silent actions. We shall proceed by demonstrating that for any command C the process $[\![C]\!]$ is $[S_1, S_2]$-confluent for suitable S_1 and S_2, and that any *complete* program P which is produced by forming

$$([\![P]\!] \mid Loc_{X_1} \mid \ldots \mid Loc_{X_n})\backslash\{L_{X_1} \cup \ldots \cup L_{X_n}\}$$

is deterministic. In other words it is $[S, \emptyset]$-confluent for some S and therefore S partition confluent.

As a simple outline of the proof which follows, consider the language DCIMP without values. For any program fragment C we can show that $[\![C]\!]$ is semi-confluent up to $\{a_{X_1}, \ldots, a_{X_n}\}$ and that the process Loc_{X_i} is stable on the action $\overline{a}_{X_i}$ and confluent; thus the representation of any complete program will be $\emptyset$-confluent, i.e. confluent. In order to prove determinism in the presence of values we have to use double-partitions, which adds greatly to the bookkeeping involved but does not change the underlying approach of the proof.

Proposition 4.2.1 *Given any program fragment C,$[\![C]\!]$ is $[S_1, S_2]$-confluent where*

- $S_1 = \{\{\overline{g}_X j\} \mid X \in \mathcal{X},\ j \in \mathcal{V}\} \cup \{\overline{d}\}$
- $S_2 = \{\{a_X j \mid j \in \mathcal{V}\} \mid X \in \mathcal{X}\}$

Lemma 4.2.2 *For any expression E, the process $[\![E]\!]$ is $[S_1 \cup R, S_2]$-confluent for S_1, S_2 as above and R as defined as follows:*

$$R = \{\{\overline{r}i\} \mid i \in \mathcal{V}\}.$$

Proof: We first define the following auxilary sets:

- $R' = \{\{ri \mid i \in \mathcal{V}\}\}$,
- $R_k = \{\{\overline{r}_k i\} \mid i \in \mathcal{V}\}$,
- $R'_k = \{\{r_k i \mid i \in \mathcal{V}\}\}$.

We proceed by induction over the structure of expressions.

- $[\![X]\!] = a_X x.rx$ which is $[S_1 \cup R, S_2]$-confluent immediately.
- for $[\![F(E_1, \ldots, E_n)]\!]$ we prove by induction on k, that $[\![E_1]\!][r_1/r] \mid \ldots \mid [\![E_k]\!][r_k/r]$ is $[S_1 \cup R_1 \cup \ldots \cup R_k, S_2]$-confluent. This is clearly true for $k = 1$. So assuming it to be true for k we shall show that it is true for $k + 1$.

- $r_1x_1.\ldots.r_nx_n.\overline{r}_{f(x_1,\ldots,x_n)}$ is clearly $[S_1 \cup R \cup R'_1 \cup \ldots \cup R'_n, S_2]$-confluent,
- if $[\![E_1]\!][r_1/r] \mid \ldots \mid [\![E_k]\!][r_k/r]$ is $[S_1 \cup R_1 \cup \ldots \cup R_k, S_2]$-confluent, and since $[\![E_{k+1}]\!][r_{k+1}/r]$ is $[S_1 \cup R_{k+1}, S_2]$-confluent then $[\![E_1]\!][r_1/r] \mid \ldots \mid [\![E_{k+1}]\!][r_{k+1}/r]$ is $[S_1 \cup R_1 \cup \ldots \cup R_{k+1}, S_2]$-confluent. (The composition is valid since the partition sets are compatible, there are no complementary action pairs, and if the sorts intersect they must intersect with actions already in S_2 and those sets are closed.)

Hence we can deduce that

$$([\![E_1]\!][r_1/r] \mid \ldots \mid [\![E_n]\!][r_n/r] \mid r_1x_1.\ldots.r_nx_n)\backslash\{r_1,\ldots,r_n\}$$

is $[S_1 \cup R, S_2]$-confluent. The composition is valid as we have restricted all complementary labels and the closedness requirements were met. There is no stability requirement as we only permitted communication with actions in the first partition.

Proof:(proposition 4.2.1) We proceed by induction over the structure of program fragments.

- Assignment, $X := E$;

$$[\![X := E]\!] \equiv ([\![E]\!] \mid rx.\overline{g}_Xx.\overline{d})\backslash\{r\}$$

now $\mathcal{L}([\![E]\!]) \cap \overline{\mathcal{L}(rx.\overline{g}_Xx.\overline{d})} = r$ so the above is a parallel composition of the required form. $rx.\overline{g}_Xx.\overline{d}$ is $[S_1 \cup R', S_2]$-confluent and $[\![E]\!]$ is $[S_1 \cup R, S_2]$-confluent, and hence $[\![X := E]\!]$ is $[S_1, S_2]$-confluent.

- Sequence, given $[\![C_1]\!]$ and $[\![C_2]\!]$ are $[S_1, S_2]$-confluent we wish to show that $[\![C_1; C_2]\!]$ is $[S_1, S_2]$-confluent. Now

$$[\![C_1; C_2]\!] \equiv ([\![C_1]\!][b/d] \mid b.[\![C_2]\!])\backslash\{b\} \qquad (b \text{ new})$$

so if we proceeded by a direct construction we could only deduce $[\![C_1; C_2]\!]$ is $[S_1, S_2]$-confluent whenever C_1 and C_2 do not *write* to the same variable. (A slightly strong restriction.) As in that case for any $\lambda \in \mathcal{L}([\![C_1]\!]) \cap \mathcal{L}([\![C_2]\!])$, $\lambda \in \Psi \in S_2$ as by the above restriction it can only be a read action. Let $Z = \{\{\overline{g}_Y x\} x \in \mathcal{V}\}$. We will consider (for simplicity) that there is only *one* write variable in common between C_1 and C_2 and that it is called Y. So we can show by construction that $[\![C_1; C_2]\!]$ is $[S_1 - Z, S_2 \cup Z]$-confluent. Consider a derivative Q of $[\![C_1; C_2]\!]$ now try to find a Q such that we can derive, for $\lambda \in \Psi_i \in Z$;

$$\begin{tikzcd} Q \arrow[r, "\lambda"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \\ Q_2 & \end{tikzcd} \quad \text{with } Q_1 \not\approx Q_2.$$

This cannot happen whilst we are within $[\![C_1]\!]$ or in the process $[\![C_2]\!]$ alone (In other words the other process has been exhausted in the parallel CCS composition) since these processes are $[S_1, S_2]$-confluent. So the only possible candidate is when $[\![C_1]\!] \xrightarrow{\lambda} T$ and $[\![C_2]\!] \xrightarrow{\lambda} T'$ but the construction forces $[\![C_1]\!]$ to become *nil* before any action can be observed form $[\![C_2]\!]$, i.e. there is no Q, derivative of $[\![C_1; C_2]\!]$, for which the diagram;

$$\begin{tikzcd} Q \arrow[r, "\lambda"] \arrow[d, Rightarrow, "\lambda"'] & Q_1 \arrow[d, Rightarrow] \\ Q_2 \arrow[r, Rightarrow] & \approx \end{tikzcd} \quad \text{for } \lambda \in \Psi_i \in Z$$

cannot be completed. Thus by strengthening $[\![C_1; C_2]\!]$ is $[(S_1 - Z) \cup Z, (S_2 \cup Z) - Z]$-confluent, i.e. it is $[S_1, S_2]$-confluent as required. We can extend this to an arbitary number of coincident write variables by varying the choice of Z as appropriate.

- Alternation,

$$[\![\mathit{IF}\ E\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2]\!] \equiv$$
$$[\![E]\!]\ \mathit{result}\ rx.(\mathit{if}\ x\ \mathit{then}\ [\![C_1]\!]\ \mathit{else}\ [\![C_2]\!])$$

$[\![E]\!]$ is $[S_1 \cup R, S_2]$ confluent, and $[\![C_1]\!]$ and $[\![C_2]\!]$ are both $[S_1, S_2]$-confluent. We need firstly to show that $rx.(\mathit{if}\ x\ \mathit{then}\ [\![C_1]\!]\ \mathit{else}\ [\![C_2]\!])$ is $[S_1 \cup R', S_2]$-confluent, but this process is precisely,

$$\sum_{x\neq 0} rx.[\![C_1]\!] + \sum_{x=0} rx.[\![C_2]\!];$$

and given that $[\![C_1]\!]$ and $[\![C_2]\!]$ are $[S_1 \cup R', S_2]$-confluent by weakening, thus the above process is $[S_1 \cup R', S_2]$-confluent by construction. The complete process $[\![\mathit{IF}\ \ E\ \ \mathit{THEN}\ \ C_1\ \ \mathit{ELSE}\ \ C_2]\!]$ is

$$([\![E]\!] \mid rx.(if\ x\ then\ [\![C_1]\!]\ else\ [\![C_2]\!]))\backslash\{r\}$$

but this is the required form for our construction and hence is $[S_1, S_2]$-confluent.

- Iteration,

$$\begin{aligned}&[\![\mathit{WHILE}\ \ E\ \ \mathit{DO}\ \ C]\!] \equiv\\ W = {}&[\![E]\!]\ result\ rx.(if\ x\ then([\![C]\!]\ before\ W)\ else\ \overline{d}) \qquad (W \text{ new})\end{aligned}$$

  $\overline{d}$ is $[S_1, S_2]$ confluent, from the earlier parts of this proof all the sub-expressions are $[S_1, S_2]$-confluent thus as it is a recursion with all the sub-parts $[S_1, S_2]$-confluent, hence the process is itself $[S_1, S_2]$-confluent.

- Deterministic Parallel composition,

$$[\![C_1\ \ PAR_d\ \ C_2]\!] \equiv ([\![C_1]\!][d_1/d] \mid [\![C_2]\!][d_2/d] \mid d_1.d_2.\overline{d})\backslash\{d_1, d_2\}$$

  in the above construction $d_1.d_2.\overline{d}$ is $[S_1 \cup \{\{d_1\}\} \cup \{\{d_2\}\}, S_2]$-confluent, $[\![C_1]\!][d_1/d]$ is $[S_1 \cup \{\{\overline{d_1}\}\} - \{\{\overline{d}\}\}, S_2]$-confluent, $[\![C_2]\!][d_2/d]$ is $[S_1 \cup \{\{\overline{d_2}\}\} - \{\{\overline{d}\}\}, S_2]$-confluent, thus by construction $[\![C_1\ \ PAR_d\ \ C_2]\!]$ is $[S_1, S_2]$-confluent. Since the only common actions are read actions and those are already in the partition S_2.

- Null, $[\![\mathit{SKIP}]\!] \equiv \overline{d}$ is is therefore $[S_1, S_2]$-confluent.

- Local variable,

$$\begin{aligned}&[\![\mathit{LET}\ \ X := E\ \ \mathit{IN}\ \ C]\!]\\ &\equiv ([\![E]\!]\ result\ ((rx.\overline{a}_X x.done \mid Loc_X)\ before\ \mid [\![C]\!])\backslash L_X)\end{aligned}$$

  we must show that Loc_X is stable on $\overline{a}_X y$ actions. From its definition:

$$Loc_X = g_X y.Var_X(y)$$
$$Var_X(y) = \overline{a}_X y.Var_X(y) + g_X z.Var_X(z)$$

this is stable on the required actions by inspection. This process is $[\{\{g_X y \mid y \in \mathcal{V}\} \cup \{\overline{a}_X y \mid y \in \mathcal{V}\}\}, \emptyset]$-confluent. Thus by construction $[\![LET\ \ X := E\ \ IN\ \ C]\!]$ is $[S_1, S_2]$-confluent. (We have to weaken to re-introduce the read and write actions for the variable X.) □

Proposition 4.2.3 *If P is a program in DCIMP then the process representing that program is confluent.*

Proof: The translation of P is

$$\mathcal{P} \equiv ([\![P]\!] \mid Loc_{X_1} \mid \ldots \mid Loc_{X_n}) \backslash L_{X_1} \cup \ldots \cup L_{X_n}$$

from proposition 4.2.1 $[\![P]\!]$ is $[S_1, S_2]$-confluent and each of the Loc_{X_i} are $[\{\{g_{X_i} y \mid y \in \mathcal{V}\} \cup \{\overline{a}_{X_i} y \mid y \in \mathcal{V}\}\}, \emptyset]$-confluent. and stable on $\{\overline{a}_{X_i} y \mid y \in \mathcal{V}\}$. Thus the construction of $\mathcal{P}$ is $[\{\{\overline{d}\}\}, \emptyset]$-confluent, and since $\overline{d}$ is the only observable action of $\mathcal{P}$ it is confluent. □

4.3 Conclusions.

We have exhibited a deterministic sub-language, DCIMP, of CIMP. The condition required to obtain DCIMP is natural and simple to check. Therefore it is not unreasonable to enforce as a programming discipline. There does not appear to be a weaker *syntactic* condition which is sufficient to ensure determinism, but a proof that we have the weakest possible syntactic condition does not seem at all evident. In fact, it is even hard to formulate this proposition precisely.

This sub-fragment of CIMP is possibly that part of the language which is 'useful', in the sense that the introduction of non-determinism is usually something that a programmer copes with post-hoc rather than seeks as a useful language feature. In fact the only direct use of such constructions in shared variable languages seems to be as a short cut to a random number generator.

In subsequent chapters we shall show that if we write concurrent programs subject to the determinism constraint then many traditionally hard problems of concurrent programs are greatly simplified. Moreover we shall show that such programs can be manipulated with ease.

Chapter 5

Denotational Semantics of DCIMP.

5.1 Introduction.

The denotational approach to semantics [Sto, Sch] is very successful at explaining properties of programs. However in order to denote concurrent programs, techniques such as powerdomains [Plo2] or metric spaces [Bak] have been necessary. The reason such methods were required is the non-determinism possible in general concurrent programming languages. DCIMP is deterministic and we therefore expect that it should be possible to provide a reasonably simple denotational semantics for it (for example of a simplicity similar to that for IMP).

We should like a functional semantics for DCIMP as our current semantics is non-modular. Until we have translated the complete program we can only see actions requesting and sending values, *not* the behaviour of a program fragment. We are thus unable to reason about properties of program fragments since their definition is incomplete in the current semantics. The production of a *natural* functional semantics which can be shown equivalent to our process based semantics, in some sense, will also lead to a greater credence of that more general semantics.

$$\mathcal{E}[\![Xi]\!]\tilde{m} = \tilde{m}_i$$
$$\mathcal{E}[\![F(E_1, ..., E_n)]\!]\tilde{m} = F(\mathcal{E}[\![E_1]\!]\tilde{m}, \ldots, \mathcal{E}[\![E_n]\!]\tilde{m})$$

Figure 5–1: Expression evaluation in DCIMP.

5.2 Denotational Semantics.

The state space of the language is defined by the values in the shared variables. Since there are a fixed finite number of these variables the state space is simply a vector of fixed length. We will call this vector a memory and denote it as $\tilde{m}$, the elements of which we refer to by indices, i.e. $\tilde{m}_i$ will be the value of the i th element of $\tilde{m}$. Thus we define the following partial semantical functions.

$$\mathcal{C} : Commands \rightarrow val^N \rightarrow val^N.$$
$$\mathcal{E} : Expressions \rightarrow val^N \rightarrow val.$$

Here, *val* is the domain of values that the variables can take.

5.2.1 Expression Evaluation.

Expression evaluation is defined as figure 5-1 in the usual style.

5.2.2 Command Evaluation.

The semantics of commands is as given in figure 5-2 and is for deterministic programs only.

Note, if $\mathcal{W}(C_1) \cap \mathcal{W}(C_2)$ is non-empty for any C_1 PAR C_2, then the function defining its denotation would not be well defined.

$$(\mathcal{C}[\![Xi := E]\!]\tilde{m})_j = \begin{cases} \tilde{m}_j \text{ for } j \neq i \\ \mathcal{E}[\![E]\!]\tilde{m} \text{ otherwise} \end{cases}$$

$$\mathcal{C}[\![SKIP]\!]\tilde{m} = \tilde{m}$$

$$\mathcal{C}[\![C_1; C_2]\!]\tilde{m} = (\mathcal{C}[\![C_1]\!]; \mathcal{C}[\![C_2]\!])\tilde{m}$$

$$(\mathcal{C}[\![C_1 \ \ PAR_d \ \ C_2]\!]\tilde{m})_i = \begin{cases} (\mathcal{C}[\![C_1]\!]\tilde{m})_i \text{ for } i \in \mathcal{W}(C_1) \\ (\mathcal{C}[\![C_2]\!]\tilde{m})_i \text{ for } i \in \mathcal{W}(C_2) \\ \tilde{m}_i \text{ otherwise} \end{cases}$$

$$\mathcal{C}[\![IF \ \ E \ \ THEN \ \ C_1 \ \ ELSE \ \ C_2]\!]\tilde{m} = \mathit{if} \ \mathcal{E}[\![E]\!]\tilde{m} \ \mathit{then} \ \mathcal{C}[\![C_1]\!]\tilde{m} \ \mathit{else} \ \mathcal{C}[\![C_2]\!]\tilde{m}$$

$$(\mathcal{C}[\![LET \ \ Xi := E \ \ IN \ \ C]\!]\tilde{m})_j = \begin{cases} \tilde{m}_i \ \ i = j \\ (\mathcal{C}[\![C]\!]\tilde{m}')_j \text{ for } j \neq i \end{cases}$$

$$\text{with } \tilde{m}'_j = \begin{cases} \tilde{m}_j \text{ for } j \neq i \\ (\mathcal{E}[\![E]\!]\tilde{m})_j \text{ for } j = i \end{cases}$$

$$\mathcal{C}[\![WHILE \ \ E \ \ DO \ \ C]\!]\tilde{m} =$$

$$\mathit{if} \ \mathcal{E}[\![E]\!]\tilde{m} \ \mathit{then} \ \mathcal{C}[\![C; WHILE \ \ E \ \ DO \ \ C]\!]\tilde{m} \ \mathit{else} \ \mathcal{C}[\![SKIP]\!]\tilde{m}$$

Figure 5–2: Semantics of DCIMP commands.

5.3 Closure.

Since the process semantics bears little resemblance to a functional form we will need to add to its structure. We will form the closure of a command process $[\![C]\!]$, by adding variable processes for each variable which the command C could access. This extended process will then read in all the values for the complete variable set, and after its execution has been completed it will then write out all the computed values of the variables.

Definition 5.3.1 *The* closure *of a command C is the following process.*

$$\begin{aligned} < C >= &((get(\tilde{x})).\overline{g_{x1}}x_1, \ldots, \overline{g_{xN}}x_N.\ \bar{b}.nil \mid \\ &b.[\![C]\!] \mid Loc_{x1} \mid \ldots \mid Loc_{xN})\backslash b \mid d.a_{x1}y_1 \ldots a_{xN}y_n.\overline{put}(\tilde{y}).nil) \\ &\backslash d, L_{x1}, \ldots, L_{xN} \end{aligned}$$

and $\mathcal{R}(C) \cup \mathcal{W}(C) \subseteq \{x1, \ldots, xN\}$.

Lemma 5.3.2 *For any deterministic terminating phrase C,*

$$< C > \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil$$

where $y_i = f_{C,i}(\tilde{x})$ for some $f_{C,i}$ dependent on the structure of C.

Proof: directly from the structure of the closure, the appropriate $f_{C,i}$ coming from the denotation of C.

Note: if C were to be non-terminating then $< C > \approx get(\tilde{x})$, which is not a particularly interesting command and hence we do not take account of such processes. This equality results from observational equivalence not preserving divergence [Wal].

We now need to be able to construct programs in the closure form so we will define an intermediate process semantics, which will be of closure processes.

5.4 Intermediate Process Semantics.

We will denote translations in the intermediate form by $< C >'$. We will also need two new processes defined as follows:

$$Split \equiv get(\tilde{x}).\overline{put}(\tilde{x}).\overline{put}(\tilde{x}).nil$$

$$Unify_S \equiv c_1(\tilde{x}).c_2(\tilde{y}).\overline{put}(\tilde{z}).nil$$

$$\text{where } z_i = \begin{cases} x_i \text{ for } i \in S, \\ y_i \text{ otherwise.} \end{cases}$$

5.4.1 Expression Evaluation.

We will use the same denotation of expression evaluation as in the original process based semantics.

Definition 5.4.1 Expression closure; *this is only needed for expressions occurring within IF THEN ELSE and WHILE DO .*

$< E >' = ((get(\tilde{x}).\overline{g_{x1}}x_1 \ldots \overline{g_{xN}}x_N.\bar{b}.nil \mid$

$b.[\![E]\!] \mid Loc_{x1} \mid ... \mid Loc_{xN})\backslash b \mid ry.a_{x1}y_1...a_{xN}y_n.\bar{t}y.\overline{put}(\tilde{y}).nil) \backslash\{r, d, L_{X_1}, ..., L_{X_N}\}$

and $\{x1, \ldots, xN\} \subseteq \tilde{x}$.

Note: all memory values are preserved by this process i.e. $\tilde{y} = \tilde{x}$.

Lemma 5.4.2 *If E is an expression then*

$$< E >' \approx get(\tilde{x}).\bar{t}z.\overline{put}(\tilde{x}).nil$$

where z is the value of E given the variables have the value $\tilde{x}$.

Proof: a simple induction over the structure of expression evaluating processes.

$$< Xi := E >' = < Xi := E >$$

$$< SKIP >' = < SKIP >$$

$$< C_1; C_2 >' = (< C_1 >' [c/put] \mid < C_2 >' [c/get]) \backslash c$$

$$< C_1 \;\; PAR_d \;\; C_2 >' =$$
$$(Split[c/put] \mid < C_1 >' [c/get, c_1/put] \mid < C_2 >' [c/get, c_2/put] \mid Unify_{W(C_1)}) \backslash c, c_1, c_2$$

$$< IF \;\; E \;\; THEN \;\; C_1 \;\; ELSE \;\; C_2 >' =$$
$$(< E >' [c/put] \mid tx.(if\ x\ then\ < C_1 >' [c/get]\ else < C_2 >' [c/get]) \backslash c, t$$

$$\begin{aligned} < LET \;\; Xi := E \;\; IN \;\; C >' = {} & (< E >' [c/put] \mid tx.c(\tilde{x}).\overline{c_1}(\tilde{y}).\ \overline{\zeta}\tilde{x}_i.nil \\ & \mid < C >' [c_1/get][c_2/put] \mid \zeta y.c_2(\tilde{z}).\overline{put}(\tilde{w}).nil) \\ & \backslash c, c_1, c_2, t, \zeta \end{aligned}$$

$$\text{With } \tilde{y}_j = \begin{cases} \tilde{x}_j \text{ for } j \neq i \\ x \text{ otherwise,} \end{cases} \qquad \text{and } \tilde{w}_j = \begin{cases} \tilde{z}_j \text{ for } j \neq i \\ y \text{ otherwise.} \end{cases}$$

$$< WHILE \;\; E \;\; DO \;\; C >' = W \Leftarrow$$

$$\begin{gathered} (< E >' [c/put] \mid (tx.if\ x\ then\ ((< C >' [c/put] \mid W[c/get]) \backslash c) \\ else < SKIP >')[c/get]) \backslash c, t \quad (W \text{ new}) \end{gathered}$$

Figure 5–3: Intermediate Semantics for Commands in DCIMP.

5.4.2 Command Evaluation.

The intermediate process semantics is given in figure 5-3 and is only for programs in DCIMP.

In figure 5-3 the $Unify_S$ process will be well defined, as we only have instances of $C_1 \;\; PAR_d \;\; C_2$.

Lemma 5.4.3 *If C is a deterministic terminating command then,*

$$< C >' \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil.$$

Where $y_i = g_{C,i}(x_1, ..., x_N)$ *for some* $g_{C,i}$ *dependent on the structure of* C.

Proof: We will proceed by structural induction over commands.

1. $C = SKIP$ or $C = X := E$;

   since these are both translated directly to *closures*, by lemma 5.3.2 they both have the required form.

2. $C = C_1; C_2$;

   from the inductive assumption:

   $< C_1 >' \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil$ and $< C_2 >' \approx get(\tilde{y}).\overline{put}(\tilde{z}).nil$.

   With $y_i = g_{C_1,i}(\tilde{x})$ and $z_i = g_{C_2,i}(\tilde{y})$.

   $$\begin{aligned} < C_1; C_2 >' &= (< C_1 >' [c/put] \mid < C_2 >' [c/get])\backslash c \\ &\approx (get(x_1, ...x_N).\overline{c}(\tilde{y}).nil \mid c(\tilde{w}).\overline{put}(\tilde{z}).nil)\backslash c \\ &\approx get(\tilde{x}).\tau.\overline{put}(\tilde{z}).nil \\ &\approx get(\tilde{x}).\overline{put}(\tilde{z}).nil \end{aligned}$$

   The w_i are bound in the communication to the y_i, and the overall evaluation is that of the composition of $g_{C_1,i}$ and $g_{C_2,i}$ for each i.

3. $C = C_1 \;\; PAR_d \;\; C_2$;

   from the inductive assumption,

   $$< C_1 >' \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil,$$

   $$< C_2 >' \approx get(\tilde{x}).\overline{put}(\tilde{z}).nil.$$

   With $y_i = g_{C_1,i}(\tilde{x})$ and $z_i = g_{C_2,i}(\tilde{x})$.

   We will split the process into two parts:

$$\begin{aligned} P1 &= (Split[c/put] \mid < C_1 >' [c/get, c_1/put] \mid < C_2 >' [c/get, c_2/put])\backslash c \\ &\approx (get(\tilde{x}).\overline{c}(\tilde{x}).\overline{c}(\tilde{x}).nil \mid c(\tilde{x}).\overline{c_1}(\tilde{y}).nil \mid c(\tilde{x}).\overline{c_2}(\tilde{z}).nil) \\ &\approx get(\tilde{x}).(\overline{c_1}(\tilde{y}).\overline{c_2}(\tilde{z}).nil + \overline{c_2}(\tilde{z}).\overline{c_1}(\tilde{y}).nil) \end{aligned}$$

$$\begin{aligned} C &= (P1 \mid Unify_{\mathcal{W}(C_1)})\backslash c_1, c_2 \\ &\approx (P1 \mid c_1(\tilde{u}).c_2(\tilde{v}).\overline{put}(\tilde{w}).nil)\backslash c_1, c_2 \\ &\approx get(\tilde{x}).\tau.\tau.\overline{put}(\tilde{w}).nil \\ &\approx get(\tilde{x}).\overline{put}(\tilde{w}).nil \end{aligned}$$

with $w_i = \begin{cases} y_i \text{ for } Xi \in \mathcal{W}(C_1), \\ z_i \text{ otherwise.} \end{cases}$

We can see that $g_{C_1 \ PAR_d \ C_2,i}(\tilde{x}) = \begin{cases} g_{C_1,i}(\tilde{x}) & i \in \mathcal{W}(C_1), \\ g_{C_2,i}(\tilde{x}) & i \in \mathcal{W}(C_2), \\ \text{identity otherwise.} \end{cases}$

4. $C = IF \ \ E \ \ THEN \ \ C_1 \ \ ELSE \ \ C_2;$

Just consider the case E true,

$$< C_1 >' \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil,$$

$$< E >' \approx get(\tilde{x}).\overline{t}y.\overline{put}(\tilde{x}).nil.$$

From assumption the value bound in $\overline{t}y$ will be *true.*

$$\begin{aligned} < C >' &= (< E >' [c/put] \mid (tx.if \ x \ then(< C_1 >' [c/get]) \ else \ (< C_2 >' \\ &\quad [c/get])))\backslash c, t \\ &\approx get(\tilde{x}).\tau.\tau.\overline{put}(\tilde{y}).nil \\ &\approx get(\tilde{x}).\overline{put}(\tilde{y}).nil \end{aligned}$$

With $y_i = g_{C_1,i}(\tilde{x})$, since the condition evaluation leaves values unchanged. Also $y_i = g_{C_2,i}(\tilde{x})$ if the condition was *false.*

5. $C = LET \ \ Xi := E \ \ IN \ \ C';$

$$< C >' = (< E >' [c/put] \mid tx.c(\tilde{x}).\overline{c_1}(\tilde{y}).\overline{\zeta}\tilde{x}_i.nil \mid < C' >' [c_1/get][c_2/put] \mid \zeta y.c_2(\tilde{z}).\overline{put}(\tilde{w}).nil)\backslash c, c_1, c_2, t, \zeta$$

with $\tilde{y}_j = \begin{cases} \tilde{x}_j \text{ for } j \neq i \\ x \text{ otherwise,} \end{cases}$ and $\tilde{w}_j = \begin{cases} \tilde{z}_j \text{ for } j \neq i \\ y \text{ otherwise.} \end{cases}$

Since $< E >' \approx get(\tilde{x}).\bar{t}.\overline{put}(\tilde{y}).nil$ and writing in the variable bindings explicitly where possible.

$$\begin{aligned} < C >' \approx\ & (get(\tilde{x}).\bar{t}x.c(\tilde{y}) \mid tx.c(\tilde{y}).\overline{c_1}(\tilde{z}).\bar{\zeta}\tilde{y}_i.nil \mid c_1(\tilde{y}).\overline{c_2}(\tilde{z}).nil \mid \\ & \zeta y.c_2(\tilde{z}).\overline{put}(\tilde{w}).nil)\backslash c, c_1, c_2, t, \zeta \\ & \approx get(\tilde{x}).\tau.\tau.\tau.\tau.\tau.\overline{put}(\tilde{w}).nil \\ & \approx get(\tilde{x}).\overline{put}(\tilde{w}).nil \end{aligned}$$

where $w_j = \begin{cases} \tilde{x}_j \text{ for } j = i \\ g_{C',j}(\tilde{x}) \text{ otherwise.} \end{cases}$

6. $C = WHILE\ \ E\ \ DO\ \ C'$;

we define: $W_n = IF\ \ E\ \ THEN\ \ C'; W_{n-1}\ \ ELSE\ \ SKIP$
$W_0 = SKIP$

From the inductive assumption:

$< C' >' = get(\tilde{x}).\overline{put}(\tilde{y}).nil$

We will now proceed by induction on n to show that W_n has the required form.

For $n = 0$, $W_0 = SKIP$ thus $< W_0 >' \approx get(\tilde{x}).\overline{put}(\tilde{x}).nil$ as required.

For n=k assuming $< W_{k-1} >'$ has the required form,

$< W_k >' \approx < IF\ \ E\ \ THEN\ \ C'; W_{k-1}\ \ ELSE\ \ SKIP >'$.

From part 2 of this proof $< C'; W_{k-1} >' \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil$.

With $\tilde{y} = (\widetilde{g_{C'}}; \widetilde{g_{W_{k-1}}})\tilde{x}$.

Thus from part 4 above; if E is true then $< W_k >' \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil$.

With $\tilde{y} = (\widetilde{g_{C'}}; \widetilde{g_{W_{k-1}}})\tilde{x}$.

Otherwise, if E is false then $< W_k >' \approx get(\tilde{x}).\overline{put}(\tilde{x}).nil$.

For any terminating *WHILE DO* there exists a finite m such W_m will evaluate to the same value. Thus *WHILE* E *DO* C' is of the required form given that it terminates.

5.5 $< C > \approx < C >'$.

Lemma 5.5.1 *For any deterministic command* C, $< C > \approx < C >'$.

Proof: Since for all terminating commands both $< C >$ and $< C >'$ have the same observational structure, i.e. $get(\tilde{x}).\overline{put}(\tilde{y}).nil$, we simply have to prove that for the same input $\tilde{x}$ we get the same output $\tilde{y}$ in every case. Note that if the command in non-terminating then both processes are $get(\tilde{x}).nil$ and are therefore observationally equivalent. This problem of observational bisimulation ignoring divergence is discussed in [Wal]. We will proceed by induction over the structure of commands.

1. $C = Xi := E$ or *SKIP*;

   since by definition of atomic commands $< C > = < C >'$ the result follows directly.

2. $C = C_1; C_2$;

   from the inductive assumption,

   if $< C_1 > \approx get(\tilde{x}).\overline{put}(\tilde{z}).nil$ then $< C_1 >' \approx get(\tilde{x}).\overline{put}(\tilde{z}).nil$,

   and if $< C_2 > \approx get(\tilde{z}).\overline{put}(\tilde{y}).nil$ then $< C_2 >' \approx get(\tilde{z}).\overline{put}(\tilde{y}).nil$.

   Thus $< C > \approx get(\tilde{x}).\overline{put}(\tilde{w}).nil$, and from the definition of $< C >$, $w_i = y_i$. Since the state of the variables will be the result of any writing done by C_1 followed by that of C_2; lemma 5.4.2 implies that $< C >' \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil$ as required.

3. $C = C_1 \;\; PAR_d \;\; C_2$;

   from the inductive assumption,

   if $< C_1 > \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil$ then $< C_1 >' \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil$,

   and if $< C_1 > \approx get(\tilde{x}).\overline{put}(\tilde{z}).nil$ then $< C_2 >' \approx get(\tilde{x}).\overline{put}(\tilde{z}).nil$.

   Since C is deterministic any variables modified by C_1 are not written or read by C_2, and vice-versa. Thus the value of the variables after the execution of $C_1 \;\; PAR_d \;\; C_2$ is either that due to the variable being written by C_1 or C_2 or the same as its initial value. Since the variables read by C_1 are not written by C_2, the values written are the same as those written in isolation of C_2 and vice-versa. So for $\tilde{w}$ defined as:

$$w_i = \begin{cases} y_i \text{ if } Xi \in \mathcal{W}(C_1) \\ z_i \text{ if } Xi \in \mathcal{W}(C_2) \\ x_i \text{ otherwise.} \end{cases}$$

   $< C > \approx get(\tilde{x}).\overline{put}(\tilde{w}).nil$

   From lemma 5.4.2, we have $< C >' \approx get(\tilde{x}).\overline{put}(\tilde{v}).nil$, with

$$v_i = \begin{cases} y_i \text{ if } Xi \in \mathcal{W}(C_1) \\ z_i \text{ otherwise.} \end{cases}$$

   But for $Xi \notin \mathcal{W}(C_2)$, $z_i = x_i$, thus the definitions of w_i and v_i are equivalent, as required.

4. $C = IF \;\; E \;\; THEN \;\; C_1 \;\; ELSE \;\; C_2$;

   By the same argument as given above for sequence (part 2), but with two cases for E true or false.

5. $C = LET\ \ Xi := E\ \ IN\ \ C'$;

   $< C' > \approx get(\tilde{x}).\overline{put}(\tilde{y}).nil$ as does $< C' >'$.

   $< C > \approx get(\tilde{x}).\overline{put}(\tilde{z}).nil$

   $< C >' \approx get(\tilde{x}).\overline{put}(\tilde{w}).nil$

   $$\text{where } \tilde{z}_j = \begin{cases} x_j \text{ for } j = i \\ y_j \text{ otherwise.} \end{cases}$$

   (The value in the process reprsenting Xi will remain constant over the execution of $< C >$, since all writes to it are restricted. Similarly in $< C >'$ we simply mantian the value of Xi accross the execution of the command.)

   $$\text{and } \tilde{w}_j = \begin{cases} x_j \text{ for } j = i \\ y_j \text{ otherwise.} \end{cases}$$

   (From lemma 5.4.2.) Thus the two processes are bisimular.

6. $C = WHILE\ \ E\ \ DO\ \ C_1$;

   as in the proof of lemma 5.4.2, (using sections 4 and 2 of this proof,) since the fragment terminates we can replace it by a sequence of finite approximations, and prove the required result by induction on the depth of the approximation.

$\square$

5.6 Final Equivalence.

5.6.1 Evaluation Equivalence.

We show that the representation of an expression in both semantics evaluates to the same value when the variables have the same values.

Definition 5.6.1 *The value of an expression E in a memory $\tilde{m}$ is written $\mathcal{V}(E)\tilde{m}$, and takes the value x when $(\overline{get}(m_1, ..., m_N).nil \mid< E >')\backslash get, put \approx \overline{t}x$.*

Lemma 5.6.2 *For any expression E,*

$\mathcal{E}[\![E]\!]\tilde{m} = \mathcal{V}(E)\tilde{m}$.

Proof:

We proceed by induction over the structure of expressions.

1. $E = Xi$;

   then $\mathcal{E}[\![E]\!]\tilde{m} = m_i$ and $\mathcal{V}(E)\tilde{m} = m_i$ since the value read will be placed directly into ty.

2. $E = F(E_1, ..., E_n)$;

   since (by inductive assumption) $\mathcal{E}[\![E_i]\!]\tilde{m} = \mathcal{V}(E_i)\tilde{m}$ and hence the final value in both cases is the same.

□

5.6.2 Command Equivalence.

Lemma 5.6.3 *If C is a deterministic terminating command then $\mathcal{C}[\![C]\!]\tilde{m}_i = g_{C,i}(m_1, ..., m_N)$ for $g_{C,i}$ defined as in lemma 5.4.2. We will call this vector function $\widetilde{g_C}$.*

Proof:

We will proceed by induction over the structure of commands.

1. $C = SKIP$;

   then $\mathcal{C}[\![C]\!]$ is the identity map, as is $\tilde{g_C}$.

2. $C = (Xi := E)$;

   in this case $\mathcal{C}[\![C]\!]$ is the identity on the variable vector excepting the i th. element, which becomes the value $\mathcal{E}[\![E]\!]\tilde{m}$, for an initial value vector $\tilde{m}$. The closure $< C >'$ is the identity on all the variable values except for the i th. element, which gets the value $\mathcal{V}[\![E]\!]\tilde{m}$. Since these evaluate to the same value from the same state (lemma 5.6.1) the functions are identical.

3. $C = C_1; C_2$;

   Given (from the inductive assumption)

   $\mathcal{C}[\![C_j]\!] = \widetilde{g_{C_j}}$, for $j = 1, 2$.

   Now $g_{\widetilde{C_1;C_2}} = \widetilde{g_{C_1}}; \widetilde{g_{C_2}}$, and $\mathcal{C}[\![C_1; C_2]\!] = \mathcal{C}[\![C_1]\!]; \mathcal{C}[\![C_2]\!]$;

   thus $\mathcal{C}[\![C_1; C_2]\!] = g_{\widetilde{C_1;C_2}}$.

4. $C = C_1 \;\; PAR_d \;\; C_2$;

   from lemma 5.4.2, we have

   $$(g_{C_1 \; \widetilde{PAR_d} \; C_2})_i = \begin{cases} (\widetilde{g_{C_1}})_i \text{ for } i \in \mathcal{W}(C_1), \\ (\widetilde{g_{C_2}})_i \text{ otherwise.} \end{cases}$$

   But except on $\mathcal{W}(C_2)$, $\widetilde{g_{C_2}}$ is the identity map. So we can rewrite this as

   $$(g_{C_1 \; \widetilde{PAR_d} \; C_2})_i = \begin{cases} (\widetilde{g_{C_1}})_i \text{ for } i \in \mathcal{W}(C_1) \\ (\widetilde{g_{C_2}})_i \text{ for } i \in \mathcal{W}(C_2) \\ id_i \text{ otherwise.} \end{cases}$$

Since $\mathcal{W}(C_1) \cap \mathcal{W}(C_2) = \emptyset$.

Now from the inductive assumption, $\mathcal{C}[\![C_1]\!] = \widetilde{g_{C_1}}$ and $\mathcal{C}[\![C_2]\!] = \widetilde{g_{C_2}}$.

Thus $\mathcal{C}[\![C]\!] = \tilde{g}_c$ as required.

5. $C = \mathit{IF}\ \ E\ \ \mathit{THEN}\ \ C_1\ \ \mathit{ELSE}\ \ C_2;$

   Proved directly from part 3 of this proof , by considering the cases when E is true, or false.

6. $C = \mathit{LET}\ \ Xi := E\ \ \mathit{IN}\ \ C'.$

   From the inductive assumption $C[\![C]\!] = \widetilde{g_{C'}}$.

$$\widetilde{g_C} = \begin{cases} \text{identity on element } i \\ \widetilde{g_{C'}} \text{ otherwise.} \end{cases}$$

   (From lemma 5.6.1.) However this is precisely the definition of $\mathcal{C}[\![C]\!]$, as required.

7. $C = \mathit{WHILE}\ \ E\ \ \mathit{DO}\ \ C'.$

   Proved by taking a sequence of finite approximations and inducing upon the depth. (Using sections 5 and 3).

□

5.7 Determinism Preserving Transformations.

Program transformation [Dar] is a well established technique for improving the efficiency of existing programs without performing extensive rewriting. In concurrent applications it is frequently one of the simplest methods for generating correct implementations [Ros2]. In particular we would like to be able to start with sequential programs and produce concurrent versions without affecting the program execution and without additional programming effort.

Definition 5.7.1 *we will say that two program fragments C_1 and C_2 are* interchangeable, *written $C_1 <=> C_2$ iff $\mathcal{C}[\![C_1]\!] = \mathcal{C}[\![C_2]\!]$.*

Proposition 5.7.2 *(Transformation system $\mathcal{T}$)*
the following are valid interchanges

T1: $C_1;(C_2;C_3) <=> (C_1;C_2);C_3$,

T2: $C_1 \ PAR_d \ C_2 <=> C_2 \ PAR_d \ C_1$,

T3: $C_1 \ PAR_d \ (C_2 \ PAR_d \ C_3) <=> (C_1 \ PAR_d \ C_2) \ PAR_d \ C_3$,

T4: $C_1;C_2 <=> C_1 \ PAR_d \ C_2$ *given C_1 and C_2 are non-interfering.*

Proof:

1. T1 follows directly from the associativity of function composition.

2. To prove the validity of T2 consider; $(\mathcal{C}[\![C_1 \ PAR_d \ C_2]\!]\tilde{m})_i$

$$= \begin{cases} (\mathcal{C}[\![C_1]\!]\tilde{m})_i & i \in \mathcal{W}(C_1), \\ (\mathcal{C}[\![C_2]\!]\tilde{m})_i & i \in \mathcal{W}(C_2), \\ \tilde{m}_i \text{ otherwise,} \end{cases}$$

   which is precisely $(\mathcal{C}[\![C_2 \ PAR_d \ C_1]\!]\tilde{m})_i$ as required.

3. To prove T3 valid consider; $(\mathcal{C}[\![(C_1 \ PAR_d \ C_2) \ PAR_d \ C_3]\!]\tilde{m})_i$

$$= \begin{cases} (\mathcal{C}[\![C_1 \ PAR_d \ C_2]\!]\tilde{m})_i & i \in \mathcal{W}(C_1 \ PAR_d \ C_2), \\ (\mathcal{C}[\![C_3]\!]\tilde{m})_i & i \in \mathcal{W}(C_3), \\ \tilde{m}_i \text{ otherwise,} \end{cases}$$

   but $\mathcal{W}(C_1 \ PAR_d \ C_2) = \mathcal{W}(C_1) \cup \mathcal{W}(C_2)$. So

$$= \begin{cases} (\mathcal{C}[\![C_1]\!]\tilde{m})_i & i \in \mathcal{W}(C_1), \\ (\mathcal{C}[\![C_2]\!]\tilde{m})_i & i \in \mathcal{W}(C_2), \\ (\mathcal{C}[\![C_3]\!]\tilde{m})_i & i \in \mathcal{W}(C_3), \\ \tilde{m}_i \text{ otherwise} \end{cases}$$

from the definition of $\mathcal{C}[\![C_1 \;\; PAR_d \;\; C_2]\!]$ which is

$$= \begin{cases} (\mathcal{C}[\![C_1]\!]\tilde{m})_i \;\; i \in \mathcal{W}(C_1), \\ (\mathcal{C}[\![C_2 \;\; PAR_d \;\; C_3]\!]\tilde{m})_i \;\; i \in \mathcal{W}(C_2 \;\; PAR_d \;\; C_3), \\ \tilde{m}_i \text{ otherwise}, \end{cases}$$

$$= (\mathcal{C}[\![C_1 \;\; PAR_d \;\; (C_2 \;\; PAR_d \;\; C_3)\tilde{m}]\!])_i$$

as required.

4. To prove T4 valid consider; $\mathcal{C}[\![C_1; C_2]\!] = \mathcal{C}[\![C_1]\!]; \mathcal{C}[\![C_2]\!]$

   given that C_1 and C_2 are non-interfering; and since $\mathcal{C}[\![C]\!]$ only affects those variables in $\mathcal{W}(C)$ for any C, we can deduce that

$$(\mathcal{C}[\![C_1; C_2]\!]\tilde{m})_i = \begin{cases} (\mathcal{C}[\![C_1]\!]\tilde{m})_i \;\; i \in \mathcal{W}(C_1), \\ (\mathcal{C}[\![C_2]\!]\tilde{m})_i \;\; i \in \mathcal{W}(C_2), \\ \tilde{m}_i \text{ otherwise}, \end{cases}$$

$$= (\mathcal{C}[\![C_1 \;\; PAR_d \;\; C_2]\!]\tilde{m})_i.$$

□

The translation rules we have given are very simple and yet they can extract all purely syntactically detectable parallelism in a program, without affecting its determinacy.

Definition 5.7.3 *A DCIMP program P is in* shortest sequential form *if whenever a sequence $C'; C; C''$ occurs in P, where*

$$C \equiv C_1 \;\; PAR_d \;\; C_2 \ldots \;\; PAR_d \;\; C_n$$

then there is no *j such that $1 \leq j < n$, and such that both the pairs*

$$C' \; and \, C_1 \;\; PAR_d \;\; \ldots \;\; PAR_d \;\; C_j,$$
$$C_{j+1} \;\; PAR_d \;\; \ldots \;\; PAR_d \;\; C_n \; and \, C''$$

are non-interefering.

The above definition captures the notion that there is no application of T4 which will leave us with a shorter program (in terms of sequential components) once it is in shortest sequential form. In some senses this is the 'fastest' deterministic rewriting of the original program, when we use the measure that all sequential components cost the same amount of time. The only speed improvements which can be made upon this program involve the possible introduction of non-determinism.

Proposition 5.7.4 *The transformation system $\mathcal{T}$ can reduce any DCIMP program to shortest sequential form.*

Proof: We provide an algorithm that, given any program, computes its shortest sequential form. Assume the program is of the form;

$$C_1; C_2 \ldots; C_n.$$

1. For each C_i, $1 < i \leq n$ in turn check to see if a split (using T2 and T3) can be found which allows it to be absorbed into the two surrounding components, or preceding component if considering C_n;
2. If such a split exists then apply T4 to produce the two separate components; and then apply T4 twice to combine them to the appropriate commands.
3. Continue until C_n is reached.

□

Note: interference is preserved by addition of parallel constructs. Once a pair of commands C_1 and C_2 are interfering there is no addition of parallel program fragments to either command which will make them non-interfering. Hence we need no back tracking in the above algorithm, and the algorithm above will terminate. When it terminates the program will be in shortest sequential form.

Unfortunately a program can have many shortest sequential forms and there will be many transformation paths leading to programs with different arrangements of the original program components. However starting from any given program the length (in terms of sequential components) of the shortest sequential form is unique and reflects the causal dependencies on variables. This means that in order to get more parallelism we must know more about the causality on variable values; so that any introduced non-determinism does not affect the correctness of the computation.

5.8 Conclusions.

The deterministic language DCIMP has a simple denotational semantics from which it is possible to deduce properties of programs with reasonable ease. The set of transformations we exhibited was not extensive; but starting from a sequential program can produce a concurrent program containing all the available parallelism without affecting determinacy. Moreover the transformation system relies solely on the imposition of a simple *syntactic* condition and so should therefore be readily implementable.

In the algorithm presented for converting programs to parallel forms no account was taken of 'balancing' the parallelism. We made no attempt to distribute the task evenly over the sequential execution path. In practice we would wish to minimise the maximum number of processing sites required whilst maintaining the execution length of our program.

It is not clear what definition to use to specify the condition of a program being balanced. However it is clear that we can use the system $\mathcal{T}$ to derive programs where each sequential component has at most a fixed arbitrary number of parallel subcomponents, and that given a fixed number of processing sites we can convert any DCIMP program to one which uses at most that number of sites.

Chapter 6

Axiomatic Methods I.

6.1 Introduction.

The axiomatic methods of program proof for sequential while languages introduced by Hoare [Hoa2] in 1969 have proved extremely successful. There have been many extensions [Owi, Jon, Stil, Zwi] of this approach to enable the production of proofs of properties of concurrent programs. In his original paper Hoare states that the assignment axiom holds only;

'in the absence of side-effects'.

Whilst it is possible to guarantee the absence of side-effects in a well designed sequential programming language, this guarantee *cannot* in general be given for a concurrent programming language. There are some evident methods that may enable us to perform proofs on concurrent systems:

- restrict proof forms,
- restrict language constructs,
- extend the information in the proof system.

The first and last methods are demonstrated in [Owi] and [Jon, Sti] respectively. The second appears as a semantical restriction in [Zwi].

In this chapter we will examine methods based on the first two solution approaches and show that they are closely related. In the following chapter we will examine techniques based upon the third method.

6.1.1 Hoare's Logic.

Hoare's method of proof is based on statements of partial correctness. We use the following notation; given assertions P and Q about program variables and a statement C in our language then

$$\{P\}C\{Q\}$$

informally means that; if P is true of the program variables before the execution of C then, whenever C terminates Q will hold of the program variables after the execution of C. Note, no account is taken of the actual termination of C; if C fails to terminate then we can say nothing about the variable state from the above triple.

A logic of such triples is usually presented in a natural deduction style. For brevity we only present a single example rule as we will be presenting a full inference system in the next section.

$$\text{Sequential Composition: } \frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

This rule captures our natural understanding of the sequential composition of commands. After a terminating execution of C_1, if P held initially then Q will hold, and if Q holds at the start of C_2 then R will hold when it terminates. Thus if P holds initially then a terminating computation of $C_1; C_2$ will leave R holding. It is the naturality of such rules that is the appeal of this treatment of program proof. There is a full presentation of Hoare's logics for sequential programming languages in the survey paper [Apt].

Null: $\dfrac{}{\{P\}SKIP\{P\}}$

Assignment: $\dfrac{}{\{P[E/X]\}X := E\{P\}}$

Alternation: $\dfrac{\{P \wedge b\}C_1\{Q\} \quad \{P \wedge \neg b\}C_2\{Q\}}{\{P\}IF \; b \; THEN \; C_1 \; ELSE \; C_2\{Q\}}$

Iteration: $\dfrac{\{P \wedge b\}C\{P\}}{\{P\}WHILE \; b \; DO \; C\{P \wedge \neg b\}}$

Composition: $\dfrac{\{P_1\}C_1\{P_2\} \ldots \{P_n\}C_n\{P_{n+1}\}}{\{P_1\}BEGIN \; C_1; \ldots ; C_n \; END\{P_{n+1}\}}$

Consequence: $\dfrac{P \vdash P_1 \quad \{P_1\}C\{Q_1\} \quad Q_1 \vdash Q}{\{P\}C\{Q\}}$

Figure 6–1: The Owicki-Gries inference system.

6.2 Owicki-Gries

The proof method of Owicki and Gries [Owi] is given over the following language:

$$\begin{aligned} C ::= {} & SKIP \mid X := E \mid IF \; b \; THEN \; C_1 \; ELSE \; C_2 \mid \\ & WHILE \; b \; DO \; C \mid BEGIN \; C_1; \ldots ; C_n \; END \mid \\ & AWAIT \; b \; THEN \; C \mid COBEGIN \; C_1 \| \ldots \| C_n \; COEND. \end{aligned}$$

In figure 6-1 we present the inference rules for the sequential fragment of the language above.

The notation $P \vdash Q$ means it is possible to prove Q with P as an assumption in the language of propositions. This system is sound and complete in the sense of Cook [Coo] for the sequential fragment.

6.2.1 Adding Parallel Commands.

There is a requirement in the Owicki-Gries system that each assignment statement and each expression evaluation is atomic, i.e. the presence of other processes cannot disturb the value produced. They provide a syntactic restriction on parallel

Await: $\dfrac{\{P \wedge b\}C\{Q\}}{\{P\}\mathit{AWAIT}\ b\ \mathit{THEN}\ C\{Q\}}$

Cobegin: $\dfrac{\{P_1\}C_1\{Q_1\} \ldots \{P_n\}C_n\{Q_n\} \text{ are interference free}}{\{P_1 \wedge \ldots \wedge P_n\}\mathit{COBEGIN}\ C_1 \| \ldots \| C_n\ \mathit{COEND}\{Q_1 \wedge \ldots \wedge Q_n\}}$

Figure 6–2: Parallel Inference for the Owicki-Gries proof system.

composition which they state merely requires that memory access be indivisible for the machine state to be well defined. That condition is as follows:

> each expression E may refer to at most one variable y which can be changed by another process while E is being evaluated and E may refer to y at most once. A similar restriction holds for assignment statements.

The rules in figure 6-2 complete the deduction system.

The inference rule Cobegin requires that the proofs for program fragments do not interfere with each other. This demand means that certain assertions, used in the proof of $\{P_i\}C_i\{Q_i\}$ for each process, are left invariantly true under parallel execution of the other processes. If these assertions are not falsified then the proof of $\{P_i\}C_i\{Q_i\}$ will still hold, in the context of a parallel execution, and therefore Q_i will hold for each i on termination. For example the assertion $X \geq Y$ remains true under the execution of $X := X + 1$, while the assertion $X = Y$ does not.

Definition 6.2.1 *We will write pre(C) and post(C) to mean the predicates P and Q respectively, in the triple $\{P\}C\{Q\}$.*

Definition 6.2.2 *(Owicki-Gries) given a proof $\{P\}C\{Q\}$ and a statement C' with precondition pre(C') and postcondition post(C') say that C'* does not interfere *with C if the following conditions hold:*

- *$\{Q \wedge pre(C')\}C'\{Q\}$;*
- *if D is any statement within C but* not *within an AWAIT THEN , then $\{pre(D) \wedge pre(C')\}C'\{pre(D)\}$.*

Definition 6.2.3 $\{P_1\}C_1\{Q_1\}\ldots\{P_n\}C_n\{Q_n\}$ *are interference-free if the following holds. Let C' be an AWAIT THEN or assignment statement (which does not appear within an AWAIT THEN) of process C_i. Then for all j with $i \neq j$ C' does not interfere with $\{P_j\}C_j\{Q_j\}$.*

Showing non-interference is quite mechanical; make a list of the C_i's preconditions, a second list of C_j's assignments and *AWAIT THEN* statements and show that each element of the second list does not disturb the truth of the first. In the paper [Owi] this system is shown to be both sound and complete for the class of programs described.

6.2.2 Example Proof.

To prove the following.

$$\{X = 0\}\mathit{COBEGIN} \begin{array}{c} \mathit{AWAIT}\ \mathit{true}\ \mathit{THEN}\ X := X + 1 \\ \| \\ \mathit{AWAIT}\ \mathit{true}\ \mathit{THEN}\ X := X + 2 \end{array} \mathit{COEND}\{X = 3\}$$

We must firstly verify the following formulae:

Let $C_1 \equiv \mathit{AWAIT}\ \mathit{true}\ \mathit{THEN}\ X := X + 1$ and
$C_2 \equiv \mathit{AWAIT}\ \mathit{true}\ \mathit{THEN}\ X := X + 2$ then

1. $\{pre(C_1) \wedge pre(C_2)\}C_2\{X = 0 \vee X = 2\}$

$$\begin{array}{ll}
\{(X = 0 \vee X = 2) \wedge (X = 0 \vee X = 1)\} & \\
\{X = 0\} & \\
\mathit{AWAIT}\ \mathit{true}\ \mathit{THEN} & \{X = 0\} \\
 & X := X + 2 \\
 & \{X = 2\} \\
\{X = 2\} & \\
\{X = 0 \vee X = 2\} &
\end{array}$$

2. $\{(X = 1 \vee X = 3) \wedge pre(C_2)\}C_2\{(X = 1 \vee X = 3)\}$

3. $\{(pre(C_2) \wedge pre(C_1))\}C_1\{pre(C_2)\}$

4. $\{(X = 2 \vee X = 3) \wedge pre(C_1)\}C_1\{(X = 1 \vee X = 3)\}$

As we have established the non-interference of C_1 and C_2. We can now obtain the following proof:

```
{X = 0}
COBEGIN {X = 0}
        {X = 0 ∨ X = 2}
        AWAIT true THEN X := X + 1
        {X = 1 ∨ X = 3}
        ||
        {X = 0}
        {X = 0 ∨ X = 1}
        AWAIT true THEN X := X + 2
        {X = 2 ∨ X = 3}
COEND
{(X = 1 ∨ X = 3) ∧ (X = 2 ∨ X = 3)}
{X = 3}.
```

Note, in the above proof most of the detail is required to establish the non-interference of the commands.

6.3 DCIMP.

In his paper [Mil5] Milner produced a Hoare like proof system for a language which is essentially DCIMP. We will reproduce that system and extend it (adding rules for *LET*, *IF* and *WHILE*) to cover the whole of the language DCIMP. Firstly Hoare triples are extended to the following form,

$$\{P\}C\{Q\}_{\tilde{Y}}^{\tilde{X}}$$

where $\tilde{X}$ and $\tilde{Y}$ are disjoint sets of variable names; such that $\tilde{X}$ contains all the variables in $\mathcal{R}(C) - \mathcal{W}(C)$ and $\tilde{Y}$ contains all the variables in $\mathcal{W}(C)$. Furthermore

Assignment: $$\frac{}{\{P[E/X]\}X := E\{P\}^{\tilde{X}}_{\tilde{Y}}}$$

Null: $$\frac{}{\{P\}SKIP\{P\}^{\tilde{X}}_{\tilde{y}}}$$

Sequential: $$\frac{\{P\}C_1\{Q\}^{\tilde{X}_1}_{\tilde{Y}_1} \quad \{Q\}C_2\{R\}^{\tilde{X}_1}_{\tilde{Y}_2}}{\{P\}C_1; C_2\{R\}^{(\tilde{X}_1-\tilde{Y}_2)\cup(\tilde{X}_2-\tilde{Y}_1)}_{\tilde{Y}_1\cup\tilde{Y}_2}}$$

Alternation: $$\frac{\{P\wedge b\}C_1\{Q\}^{\tilde{X}_1}_{\tilde{Y}_1} \quad \{P\wedge\neg b\}C_2\{Q\}^{\tilde{X}_2}_{\tilde{Y}_2}}{\{P\}IF\ b\ THEN\ C_1\ \ ELSE\ \ C_2\{Q\}^{(\tilde{X}_1-\tilde{Y}_2)\cup(\tilde{X}_2-\tilde{Y}_1)}_{\tilde{Y}_1\cup\tilde{Y}_2}}$$

Iteration: $$\frac{\{P\wedge b\}C\{P\}^{\tilde{X}}_{\tilde{Y}}}{\{P\}WHILE\ \ b\ \ DO\ \ C\{P\wedge\neg b\}^{\tilde{X}}_{\tilde{Y}}}$$

Weakening: $$\frac{P\vdash P_1 \quad \tilde{X}_1\subseteq\tilde{X} \quad \tilde{Y}_1\subseteq\tilde{Y} \quad \tilde{Y}\cap\tilde{X}=\emptyset \quad \{P_1\}C\{Q_1\}^{\tilde{X}_1}_{\tilde{Y}_1} \quad Q_1\vdash Q}{\{P\}C\{Q\}^{\tilde{X}}_{\tilde{Y}}}$$

Let: $$\frac{\{P\wedge Z=E\}C\{Q\}^{(\tilde{X}\cup\{Z\})-(\{Z\}\cap\tilde{Y})}_{\tilde{Y}}}{\{P\wedge(Z=F)\}LET\ \ Z:=E\ \ IN\ \ C\{Q\wedge(Z=F)\}^{\tilde{X}}_{\tilde{Y}}} \quad \begin{cases} Z \text{ not in } P \text{ or } Q \text{ and} \\ \text{some } \textit{value } F. \end{cases}$$

Parallel:(on condition that $\tilde{X}_1\cap\tilde{Y}_2=\tilde{X}_2\cap\tilde{Y}_1=\tilde{Y}_1\cap\tilde{Y}_2=\emptyset$)

$$\frac{\{P_1\}C_1\{Q_1\}^{\tilde{X}_1}_{\tilde{X}_2} \quad \{P_2\}C_2\{Q_2\}^{\tilde{X}_2}_{\tilde{Y}_2}}{\{P_1\wedge P_2\}C_1\ \ PAR_d\ \ C_2\{Q_1\wedge Q_2\}^{\tilde{X}_1\cup\tilde{X}_2}_{\tilde{Y}_1\cup\tilde{Y}_2}}$$

Figure 6–3: Inference Rules for DCIMP.

all program variables occurring in P or Q must be in $\tilde{X}\cup\tilde{Y}$. This extension allows us to check that parallel compositions obey our non-interference condition. A Hoare logic for the language DCIMP would contain the rules given in figure 6-3.

Once again the side condition on the parallel is purely syntactic and therefore very easy to check. It ensures that the commands C_1 and C_2 are non-interfering since the only non-empty intersection of the variable sets permitted is in the read sets. In his book [Mil6] Milner provides the following example proof of

$$\{X=1\wedge Y=3\}Y:=Y+X\ \ PAR_d\ \ Z:=X+1\{Y=4\wedge Z=2\}^{\{X\}}_{\{Y,Z\}}$$

1. $\{Y + X = 4\} Y := Y + X \{Y = 4\}^{\{X\}}_{\{Y\}}$ (assignment),

2. $X = 1 \wedge Y = 3 \vdash Y + X = 4$ (arithmetic),

3. $\{X = 1 \wedge Y = 3\} Y := Y + X \{Y = 4\}^{\{X\}}_{\{Y\}}$ (weakening 1,2),

4. $\{X + 1 = 2\} Z := X + 1 \{Z = 2\}^{\{X\}}_{\{Z\}}$ (assignment),

5. $X = 1 \vdash X + 1 = 2$ (arithmetic),

6. $\{X = 1\} Z := X + 1 \{Z = 2\}^{\{X\}}_{\{Z\}}$ (weakening 4,5),

7. $\{X = 1 \wedge Y = 3 \wedge X = 1\} Y := Y + X \;\; \mathit{PAR}_d \;\; Z := X + 1 \{Y = 4 \wedge Z = 2\}^{\{X\}}_{\{Y,Z\}}$ (parallel 3,6),

8. $\{X = 1 \wedge Y = 3\} Y := Y + X \;\; \mathit{PAR}_d \;\; Z := X + 1$ $\{Y = 4 \wedge Z = 2\}^{\{X\}}_{\{Y,Z\}}$ (weakening).

In order to show that this inference system is sound with respect to our process based semantics for our programming language Milner translates extended Hoare triples into the logic $\mathcal{PL}$.

6.3.1 Translating Hoare Triples.

For any program fragment C we take as its representation the process$[\![C]\!]$. The process $[\![C]\!]$ is incomplete in the sense that it does not include the processes which represent program variables. However, the traces of such a process inform us of precisely the effect it would have on the program variables, e.g. given the command

$$C \equiv X := X + X$$

the process $[\![C]\!]$ can perform the following; $[\![C]\!] \overset{s\bar{d}}{\Longrightarrow} nil$, where $s = a_X(5)a_X(6)\overline{g}_X(11)$. In this sequence we have read X twice and received the values 5 and 6, (clearly the variable X has been interfered with whilst we were reading it) and then the value 11 is written back to the variable X. To represent this information about the internal history of processes Milner introduced the notion of memory maps: a function from some set of program variable names to values. The letters m, m' will denote memory maps and the functional $Dom(m)$ will represent the set of variable names upon which m is defined. We define the effect of an action sequence $s \in \Lambda^*$ in relation to memory maps. This is written $m\{\!|\, s \,|\!\}m'$, and means informally that in the absence of interference, starting from memory m and executing the sequence of actions s we will arrive in the memory m'.

Definition 6.3.1 *The triples* $m\{\!|\, s \,|\!\}m'$ *are defined inductively on the length of* s;

1. $m\{\!|\, \epsilon \,|\!\}m'$ *iff* $m' = m$,

2. $m\{\!|\, s\tau \,|\!\}m'$ *iff* $m\{\!|\, s \,|\!\}m'$,

3. $m\{\!|\, s\bar{g}_X z \,|\!\}m'$ *iff* $m\{\!|\, s \,|\!\}m''$ *for some* m'' *such that* $m'(X) = z$ *and for all* $Y \neq X$ $m'(Y) = m''(Y)$,

4. $m\{\!|\, sa_X y \,|\!\}m'$ *iff* $m\{\!|\, s \,|\!\}m'$ *and* $m'(X) = y$.

It is now possible to write a formula $F \in \mathcal{PL}$ which is dependent solely upon $P, Q, \tilde{X}, \tilde{Y}$, such that $[\![C]\!] \models F$ precisely when $\{P\}C\{Q\}^{\tilde{X}}_{\tilde{Y}}$ holds. The formula F asserts that when P holds of a memory m then if $[\![C]\!]$ terminates producing a memory m' after a sequence of actions s then given that $m\{\!|\, s \,|\!\}m'$ then Q must hold of m'. We can restrict our attention to memory maps such that $Dom(m) = Dom(m') = \tilde{X} \cup \tilde{Y}$ since no other variables can occur in P, Q or C. For convenience we let $\tilde{Z} = \tilde{X} \cup \tilde{Y}$ and the expression $P[m(\tilde{Z})/\tilde{Z}]$ denote the substitution of the memory values for each of the program variables in P.

Definition 6.3.2 *(Milner) a Hoare triple* $\{P\}C\{Q\}^{\tilde{X}}_{\tilde{Y}}$ *is valid iff* $[\![C]\!] \models F(P, Q, \tilde{X} \cup \tilde{Y})$ *where* $F(P, Q, \tilde{X} \cup \tilde{Y})$ *is the following* $\mathcal{PL}$ *formula:*

$$\forall m, m', s.((Dom(m) = Dom(m') = \tilde{Z}) \wedge m\{\!|\, s \,|\!\}m' \wedge P[m(\tilde{Z})/\tilde{Z}] \supset [[s\bar{d}]]Q[m'(\tilde{Z})/\tilde{Z}]).$$

In his book [Mil6] and the paper [Mil5] Milner presents an outline proof that a reduced version (the parallel, sequence, weakening and assignment inference rules) of the inference system is sound with respect to the above notion of validity. We will present a proof of the full inference system for DCIMP firstly for a pure proposition language and then in the presence of logical variables.

6.3.2 Soundness Proof

Proposition 6.3.3 *the proof system is figure 6-3 is sound with respect to the validity defined in 6.3.2 over the process interpretation of DCIMP.*

Proof:

1. Null:

$$\frac{}{\{P\}\mathit{SKIP}\{P\}^{\tilde{X}}_{\tilde{Y}}}$$

The process $[\![\mathit{SKIP}]\!]$ can only perform the action $\bar{d}$. Thus given any initial memory m the memory after the execution of *SKIP* will be identical to it and hence:

$$[\![\mathit{SKIP}]\!] \models F(P, P, \tilde{X} \cup \tilde{Y})$$

follows immediately.

2. Assignment:

$$\frac{}{\{P[E/X]\}X := E\{P\}^{\tilde{X}}_{\tilde{Y}}}$$

For the process $[\![X := E]\!]$ all the evolution sequences are of the form $s = a_{X_j}y_1 \ldots a_{X_k}y_k\bar{g}_X E\bar{d}$ (see proposition 1.3.4). Hence for any initial m and $m\{\!|\, s \,|\!\}m'$ then m' is identical to m excepting for the variable X, where it has value E (from the definition of $m\{\!|\, s \,|\!\}m'$). So for any m where $P[E/X][m(\tilde{Z}/\tilde{Z})]$ holds, then for any s evolution of $[\![X := E]\!]$ and $m\{\!|\, s \,|\!\}m'$, $P[m'(\tilde{Z})/\tilde{Z}]$ holds. Thus we have

$$[\![X := E]\!] \models F(P[E/X], P, \tilde{X} \cup \tilde{Y}).$$

3. Sequential:

$$\frac{\{P\}C_1\{Q\}^{\tilde{X}_1}_{\tilde{Y}_1} \quad \{Q\}C_2\{R\}^{\tilde{X}_1}_{\tilde{Y}_2}}{\{P\}C_1;C_2\{R\}^{(\tilde{X}_1-\tilde{Y}_2)\cup(\tilde{X}_2-\tilde{Y}_1)}_{\tilde{Y}_1\cup\tilde{Y}_2}}$$

The observed sequences of $[\![C_1;C_2]\!]$ are just those formed by arbitrary concatenation of action sequences from $[\![C_1]\!]$ to those from $[\![C_2]\!]$. Any *terminating* sequence of $[\![C_1;C_2]\!]$ is of the form $s = s_1\tau s_2$, where $s_1\overline{d}$ is a terminating sequence of $[\![C_1]\!]$ and $s_2\overline{d}$ is a terminating sequence of $[\![C_2]\!]$. Let $\tilde{Z} = \tilde{X}_1 \cup \tilde{Y}_1 \cup \tilde{X}_2 \cup \tilde{Y}_2$. For any memory maps m, m' such that $\tilde{Z} = Dom(m) = Dom(m')$ and $m\{\!|\, s \,|\!\}m'$ there exists m'', such that $m\{\!|\, s_1 \,|\!\}m''$ and $m''\{\!|\, s_2 \,|\!\}m'$. From the left hand premise; if P holds of m then Q holds of m'' (possibly with some extension of the variable range), and from the right hand premise if Q holds of m'' then R holds of m' (once again the variable range may need extending) since the sequences s_1 and s_2 where terminating. In other words if

$$[\![C_1]\!] \models F(P,Q,\tilde{Z}) \text{ and}$$
$$[\![C_2]\!] \models F(Q,R,\tilde{Z}) \text{ then}$$
$$[\![C_1;C_2]\!] \models F(P,R,\tilde{Z}).$$

4. Alternation:

$$\frac{\{P \wedge b\}C_1\{Q\}^{\tilde{X}_1}_{\tilde{Y}_1} \quad \{P \wedge \neg b\}C_2\{Q\}^{\tilde{X}_2}_{\tilde{Y}_2}}{\{P\}\mathit{IF}\ b\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\{Q\}^{(\tilde{X}_1-\tilde{Y}_2)\cup(\tilde{X}_2-\tilde{Y}_1)}_{\tilde{Y}_1\cup\tilde{Y}_2}}$$

We can assume that;

$$[\![C_1]\!] \models F(P \wedge b, Q, \tilde{X}_1 \cup \tilde{Y}_1), \text{ and } [\![C_2]\!] \models F(P \wedge \neg b, Q, \tilde{X}_2 \cup \tilde{Y}_2).$$

Let $\tilde{Z}_1 = \tilde{X}_1 \cup \tilde{Y}_1$, $\tilde{Z}_2 = \tilde{X}_2 \cup \tilde{Y}_2$ and $\tilde{Z} = \tilde{Z}_1 \cup \tilde{Z}_2$. Since $\tilde{Z}_1 \subseteq \tilde{Z}$ and $\tilde{Z}_2 \subseteq Z$ then we have:

$$[\![C_1]\!] \models F(P \wedge b, Q, \tilde{Z}) \text{ and } [\![C_2]\!] \models F(P \wedge \neg b, Q, \tilde{Z}).$$

We require to prove that:

$$[\![\mathit{IF}\ b\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2]\!] \models F(P,Q,\tilde{Z}).$$

The computation sequences of the process $[\![\mathit{IF}\ b\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2]\!]$ are simply those produced by performing the evaluation of the condition b followed by the appropriate choice of a terminating evaluation sequence from $[\![C_1]\!]$ or $[\![C_2]\!]$. In evaluating b we leave any memory unchanged, since there are no write actions in a pure evaluation. Thus if P holds before the evaluation of b it will hold after it. So P will be true of the memory when we start executing either $[\![C_1]\!]$ or $[\![C_2]\!]$. There are two cases.

(a) b is true; we will now execute $[\![C_1]\!]$ with an initial memory where $P \wedge b$ holds, thus from the left hand premise if $[\![C_1]\!]$ terminates it will terminate in a memory where Q holds.

(b) b is false; we execute $[\![C_2]\!]$ from an initial memory where $P \wedge \neg b$ holds and can similarly deduce that on any termination Q will hold of the final memory state.

Thus for all sequences of $[\![\mathit{IF}\ \ b\ \ \mathit{THEN}\ \ C_1\ \ \mathit{ELSE}\ \ C_2]\!]$ we can see that

$$[\![\mathit{IF}\ b\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2]\!] \models F(P, Q, \tilde{Z}).$$

5. Iteration:

$$\frac{\{P \wedge b\}C\{P\}_{Y}^{X}}{\{P\}\mathit{WHILE}\ \ b\ \ \mathit{DO}\ \ C\{P \wedge \neg b\}_{Y}^{X}}$$

If the process $[\![\mathit{WHILE}\ \ b\ \ \mathit{DO}\ \ C]\!]$ terminates then the terminating action sequence will be an arbitrary number of copies of evaluation sequences of b followed by a terminating action sequences of $[\![C]\!]$. Proceeding inductively along the evaluation sequence; if P holds of the initial memory then it will hold after the evaluation of b. If b is true we then perform a *terminating* sequence of $[\![C]\!]$ with $P \wedge b$ holding of the initial memory, and hence P holds of the final memory. So proceeding we can see that P holds throughout the execution of any terminating action sequence of $[\![\mathit{WHILE}\ \ b\ \ \mathit{DO}\ \ C]\!]$. Finally for termination to occur we must reach a state where $P \wedge \neg b$ holds of a memory after the execution of an instance of $[\![C]\!]$; and thus it will hold of any final memory, i.e. given;

$$[\![C]\!] \models (P \wedge b, P, \tilde{X} \cup \tilde{Y}) \text{ then}$$
$$[\![\mathit{WHILE}\ \ b\ \ \mathit{DO}\ \ C]\!] \models (P, P \wedge \neg b, \tilde{X} \cup \tilde{Y}).$$

6. Weakening:

$$\frac{P \vdash P_1 \quad X_1 \subseteq X \quad X_1 \cap W(C) = \emptyset \quad Y_1 \subseteq Y \quad Y \cap X = \emptyset \quad \{P_1\}C\{Q_1\}^{\tilde{X}_1}_{\tilde{Y}_1} \quad Q_1 \vdash Q}{\{P\}C\{Q\}^{\tilde{X}}_{\tilde{Y}}}$$

This follows immediately from logical considerations; and that the variable set extensions preserve the appropriate restrictions with respect to the read and write sets.

7. Let:

$$\frac{\{P \wedge Z = E\}C\{Q\}^{(\tilde{X} \cup \{Z\}) - (\{Z\} \cap \tilde{Y})}_{\tilde{Y}}}{\{P \wedge (Z = F)\} \mathit{LET} \;\; Z := E \;\; \mathit{IN} \;\; C\{Q \wedge (Z = F)\}^{\tilde{X}}_{\tilde{Y}}} \quad \begin{cases} Z \text{ not in } P \text{ or } Q \text{ and} \\ \text{some } \mathit{value}\ F. \end{cases}$$

If $[\![\mathit{LET} \;\; Z := E \;\; \mathit{IN} \;\; C]\!]$ has a terminating action sequence then that sequence is formed from an assignment to the variable Z followed by a terminating sequence for $[\![C]\!]$. The evaluation of $Z := E$ does not affect the memory since the variable Z is local. There is no $a_Z y$ action visible in the action sequence of $[\![\mathit{LET} \;\; Z := E \;\; \mathit{IN} \;\; C]\!]$ as a result of its construction. From the view of the process C it is executing using the initial memory excepting that the variable Z now has the value of E evaluated in that memory. But then $P \wedge (Z = E)$ will hold of that memory and hence we can deduce that Q will hold of any final memory. Since Z is not in P or Q we will not have a contradiction. Thus $Q \wedge (Z = F)$ will hold of the final memory of $[\![\mathit{LET} \;\; Z := E \;\; \mathit{IN} \;\; C]\!]$ given $P \wedge (Z = F)$ held of its initial memory. In other words,

$$[\![\mathit{LET} \;\; Z := E \;\; \mathit{IN} \;\; C]\!] \models F(P \wedge (Z = F), Q \wedge (Z = F), \tilde{X} \cup \tilde{Y}).$$

8. Parallel:(on condition that $\tilde{X}_1 \cup \tilde{Y}_2 = \tilde{X}_2 \cup \tilde{Y}_1 = \tilde{Y}_1 \cup \tilde{Y}_2 = \emptyset$)

$$\frac{\{P_1\}C_1\{Q_1\}^{\tilde{X}_1}_{\tilde{X}_2} \quad \{P_2\}C_2\{Q_2\}^{\tilde{X}_2}_{\tilde{Y}_2}}{\{P_1 \wedge P_2\}C_1 \;\; \mathit{PAR}_d \;\; C_2\{Q_1 \wedge Q_2\}^{\tilde{X}_1 \cup \tilde{X}_2}_{\tilde{Y}_1 \cup \tilde{Y}_2}}$$

Any terminating sequence $[\![C_1 \;\; \mathit{PAR}_d \;\; C_2]\!]$ is an arbitrary interleaving of terminating sequences of $[\![C_1]\!]$ and $[\![C_2]\!]$. Since $P_1 \wedge P_2$ holds of the initial memory; both P_1 and P_2 hold separately of that memory. So we have the required initial conditions to apply the premises. However we must be careful that the presence of the actions from the other process do not disturb the resulting final memory of either computation.

Now Q_1 can only refer to variables written to or read by C_1 and since C_2 is *not premitted* to write to any of these variables its presence will leave them unaffected. Thus the truth or falsehood of Q_1 will be unaffected by the presence of $[\![C_2]\!]$ whilst $[\![C_1]\!]$ is executing. A symmetric argument shows the condition Q_2 is unaffected by the execution of C_1.

Wlog we assume that the final action of $[\![C_1]\!]$ occurs in our terminating sequence before that of $[\![C_2]\!]$. Any variable used by $[\![C_1]\!]$ is unaffected by the presence of $[\![C_2]\!]$, and since P_1 held initially Q_1 will hold now; and it will continue to hold until $[\![C_2]\!]$ terminates. At which point Q_2 will also hold of the memory and hence $Q_1 \wedge Q_2$ will hold of the final memory. We can see that

$$[\![C_1 \;\; PAR_d \;\; C_2]\!] \models F(P_1 \wedge P_2, Q_1 \wedge Q_2, \tilde{X}_1 \cup \tilde{Y}_1 \cup \tilde{X}_2 \cup \tilde{Y}_2).$$

□

6.3.3 Non-Interference Implies Interference Freedom.

Definition 6.3.4 *given a program in DCIMP without the LET IN construct, there is an equivalent program in the language of Owicki-Gries defined by direct translation as follows.*

DCIMP	*Owicki-Gries*
C	$\tilde{C}$
$X := E$	$X := E$
$C_1; C_2$	*BEGIN* $\tilde{C}_1; \tilde{C}_2$ *END*
C_1 PAR_d C_2	*COBEGIN* $\tilde{C}_1 \| \tilde{C}_2$ *COEND*
IF E *THEN* C_1 *ELSE* C_2	*IF* E *THEN* $\tilde{C}_1$ *ELSE* $\tilde{C}_2$
WHILE E *DO* C	*WHILE* E *DO* $\tilde{C}$

Proposition 6.3.5 *If we can prove* $\{P\}C\{Q\}_{\tilde{Y}}^{\tilde{X}}$ *then we can prove* $\{P\}\tilde{C}\{Q\}$ *in the inference system of Owicki-Gries.*

Proof: We proceed by induction over the depth of proof. We distinguish proof systems by their different triples and drop the ˜ signifying translation between the two languages. The only point of interest is the parallel rules. We must demonstrate the interference freedom of the proofs of C_1 and C_2; given that we can prove $\{P_1 \wedge P_2\}C_1\ \ PAR_d\ \ C_2\{Q_1 \wedge Q_2\}_{\tilde{Y}}^{\tilde{X}}$. From the inductive assumption, if the above was deduced from

$$\frac{\{P_1\}C_1\{Q_1\}_{\tilde{X}_2}^{\tilde{X}_1} \quad \{P_2\}C_2\{Q_2\}_{\tilde{Y}_2}^{\tilde{X}_2}}{\{P_1 \wedge P_2\}C_1\ \ PAR_d\ \ C_2\{Q_1 \wedge Q_2\}_{\tilde{Y}_1 \cup \tilde{Y}_2}^{\tilde{X}_1 \cup \tilde{X}_2}}$$

then we can prove

$$\{P_1\}C_1\{Q_1\} \text{ and } \{P_2\}C_2\{Q_2\}.$$

So if we can establish non-interference of the above then we are finished. For non-interference the following conditions must hold hold for both C_1 and C_2:

1. $\{Q_1 \wedge P_2\}C_2\{Q_1\}$;
2. if D is any statement within C_1 then $\{pre(D) \wedge P_2\}C_2\{pre(D)\}$.

The first clause is an immediate consequence of the fact that the presence of C_2 cannot affect the truth of Q_1. (See proof of soundness of parallel inference rule.) The second requirement follows from the fact that in part of the sub-proof of $\{P_1\}C_1\{Q_1\}_{\tilde{Y}_1}^{\tilde{X}_1}$ no reference can be mades to variables outside of $\tilde{X}_1$ and $\tilde{Y}_1$; and therefore its truth is unaffected by the presence of $[\![C_2]\!]$. Moreover we can find a matching proof in Owicki-Gries; which uses no more variables. Any extra variables would be redundant, they cannot appear in C_1; and hence can be removed from our predicates by weakening. Thus the condition of non-interference guarantees that proofs are interference free.

□

6.3.4 Adding Logical Variables.

Definition 6.3.6 *let the propositions P and Q contain the* logical variables $v_1 \ldots v_n$ *(which we will call $\underline{v}$.) we interpret the extended triple:*

$$\{P\}C\{Q\}_{\tilde{Y}}^{\tilde{X}}$$

as

$$\forall \underline{v}([\![C]\!] \models F(P, Q, \tilde{X} \cup \tilde{Y})),$$

in other words that the interpretation holds for all possible variable values. In that if we can find a substitution that makes P true, then the same substitution will make Q true if the command C terminates.

Proposition 6.3.7 *the rules of inference in figure 6-3 are sound with the interpretation of logical variables given above.*

Proof: We present the outline proof for the parallel rule, the others follow from similar arguments. On condition that $\tilde{X}_1 \cap \tilde{Y}_2 = \tilde{X}_2 \cap \tilde{Y}_1 = \tilde{Y}_1 \cap \tilde{Y}_2 = \emptyset$;

$$\frac{\{P_1\}C_1\{Q_1\}_{\tilde{X}_2}^{\tilde{X}_1} \quad \{P_2\}C_2\{Q_2\}_{\tilde{Y}_2}^{\tilde{X}_2}}{\{P_1 \wedge P_2\}C_1 \;\; PAR_d \;\; C_2\{Q_1 \wedge Q_2\}_{\tilde{Y}_1 \cup \tilde{Y}_2}^{\tilde{X}_1 \cup \tilde{X}_2}}$$

We wish to show that (given $\tilde{Z} = \tilde{X}_1 \cup \tilde{Y}_1 \cup \tilde{X}_2 \cup \tilde{Y}_2$):

$$\forall \underline{v}([\![C_1 \;\; PAR_d \;\; C_2]\!] \models F(P_1 \wedge P_2, Q_1 \wedge Q_2, \tilde{Z}))$$

whenever

$$\forall \underline{v}([\![C_1]\!] \models F(P_1, Q_1, \tilde{X}_1 \cup \tilde{Y}_1))$$

and

$$\forall \underline{v}([\![C_2]\!] \models F(P_2, Q_2, \tilde{X}_2 \cup \tilde{Y}_2)).$$

We know that if for an initial memory $\forall \underline{v} P_1 \wedge P_2$ holds then we can apply the premises; as $\forall \underline{v} P_1$ and $\forall \underline{v} P_2$ hold of that memory. We demonstrated earlier the the processes do not disturb the evolution of each others memories. Thus if the composition terminates, then it terminates in a memory where $\forall \underline{v} Q_1$ and $\forall \underline{v} Q_2$ hold. So $\forall \underline{v} Q_1 \wedge Q_2$ holds of that memory as required.

6.4 Conclusions.

We have provided a simple syntactic restriction which permits formal Hoare style proofs over parallel while programs. Unfortunately this condition is strictly stronger than the non-interference condition of Owicki-Gries. This is not surprising as the comparison did not take into account the atomicity assumptions used in the Owicki-Gries system. The languages are comparable but the underlying semantics are not. Whilst there is a condition given by Owicki and Gries for non-atomic assignments it is not clear how to interpret it.

So far we have been unable to prove that the proof system of DCIMP is complete, in the usual sense. The absence of an *AWAIT THEN* command makes the proof considerably harder and prevents us from easily detecting weakest preconditions of parallel constructs.

Chapter 7

Axiomatic Methods II.

7.1 Introduction.

In the paper [Jon] Jones presents a methodology for the development of programs that have the potential for interference. In order to achieve this, the specification of program fragments is extended via conditions; on how much a program fragment is willing to *guarantee* to the environment and how much it *relies* on that environment. These are natural notions in the development of a program, as a programmer will know how much any program interferes with its environment, and what it requires from that environment to function correctly. Only compatible program fragments, i.e. those that guarantee at least as much as the other fragments rely on, can be executed in parallel. In this chapter we will be using the unrestricted version of the parallel language CIMP.

7.1.1 Stirling's Inference System.

This idea of rely and guarantee conditions is used by Stirling [Stil] to present an extended Hoare like logic over a shared variable while language.

Definition 7.1.1 *Let P and Q be predicates on variables (from some language of predicates $\mathcal{L}$), and let Γ and Δ be sets of such predicates. Then the extended Hoare triple:*

$$\{\Gamma, P\}C\{Q, \Delta\}$$

is interpreted as follows. If P holds before *the execution of C and no predicate within Γ is falsified* during *the execution of C, then*

1. *any predicate within Δ will not be falsified during the execution of C and;*

2. *if C terminates, it will terminate with Q holding of the variables.*

In the above, Γ is the predicate set that the execution of C relies on being invariant, and Δ the predicate set it guarantees invariant. It should be noted that the preservation of Δ is independent of the termination of C. A deduction system (figure 7-1) is presented over the following language:

$$\begin{aligned} C ::= &\ SKIP \mid X := E \parallel C_1;C_2 \mid C\backslash X \mid. \\ &IF\ b\ THEN\ C_1\ ELSE\ C_2 \mid WHILE\ b\ DO\ C \\ &\mid C_1\ PAR\ C_2 \mid AWAIT\ b\ THEN\ C \end{aligned}$$

Definition 7.1.2 *In figure 7-1 the interpretation of the statement $\Gamma \Rightarrow P$ or $\Gamma \Rightarrow \Gamma'$ is that all execution paths that leave the antecedent invariant, leave the consequent invariant.*

Definition 7.1.3 *The ordinary triple $\{P\}C\{Q\}$ is an abbreviation for $\{\mathcal{L}, P\}C\{Q, \emptyset\}$ where $\mathcal{L}$ is the language from which the propositions are taken.*

Definition 7.1.4 *the notation $P \models Q$ means that Q is deducible from P in the language of propositions $\mathcal{L}$.*

Definition 7.1.5 *$C\backslash X$ is the program obtained by replacing all occurrences of assignment to X in C with SKIP and transforming any AWAIT true THEN C to C when C is SKIP or $Y := E$.*

In [Stil] the inference system of figure 7-1 is shown to be both sound and complete. These proofs rely on a semantics of potential computation sequences, an extension of the semantical presentation of [Hen1].

We will show that for the fragment of the inference system that refers to the following language:

Null: $\dfrac{}{\{\Gamma, P\} SKIP \{P, \Delta\}}$

Assignment: $\dfrac{\Gamma \Rightarrow P \quad P \models R[t/X] \quad \forall Q \in \Delta, P \wedge Q \models Q[t/X]}{\{\Gamma, P\} X := t \{R, \Delta\}}$

Sequence: $\dfrac{\{\Gamma, P\} C_1 \{Q, \Delta\} \quad \{\Gamma, Q\} C_2 \{R, \Delta\}}{\{\Gamma, P\} C_1 ; C_2 \{R, \Delta\}}$

Alternation: $\dfrac{\Gamma \Rightarrow P \quad \{\Gamma, P \wedge b\} C_1 \{Q, \Delta\} \quad \{\Gamma, P \wedge \neg b\} C_2 \{Q, \Delta\}}{\{\Gamma, P\} IF \ b \ THEN \ C_1 \ ELSE \ C_2 \{Q, \Delta\}}$

Iteration: $\dfrac{\Gamma \Rightarrow P \quad \{\Gamma, P \wedge b\} C \{P, \Delta\}}{\{\Gamma, P\} WHILE \ b \ DO \ C \{P \wedge \neg b, \Delta\}}$

Await: $\dfrac{\Gamma \Rightarrow P \quad \{P \wedge b\} C \{Q\} \quad \forall R \in \Delta \{P \wedge b \wedge R\} C \{R\}}{\{\Gamma, P\} AWAIT \ b \ THEN \ C \{Q, \Delta\}}$

Parallel: $\dfrac{\Gamma \Rightarrow R \quad \{\Gamma, P\} C_1 \{R, \Sigma \cup \Delta\} \quad \{\Sigma, Q\} C_2 \{S, \Gamma \cup \Delta\} \quad \Sigma \Rightarrow R}{\{\Gamma \cup \Sigma, P \wedge Q\} C_1 \ PAR \ C_2 \{R \wedge S, \Delta\}}$

Auxiliary Variable: $\dfrac{\{P\} C \{Q\}}{\{P\} C \backslash X \{Q\}}$ (P and Q do not contain free occurrences of X.)

Weakening: $\dfrac{\Gamma \Rightarrow \Gamma' \quad P \models P_1 \quad \{\Gamma', P_1\} C \{Q_1, \Delta'\} \quad Q_1 \models Q \quad \Delta' \Rightarrow \Delta}{\{\Gamma, P\} C \{Q, \Delta\}}$

Figure 7–1: Inference System for Rely-Guarantee

$$C ::= SKIP \mid X := E \mid IF\ \ b\ \ THEN\ \ C_1\ \ ELSE\ \ C_2 \mid C_1; C_2.$$
$$WHILE\ \ b\ \ DO\ \ C \mid C_1\ \ PAR\ \ C_2$$

we can give an interpretation of the quintuples in the same manner as Milner gave for the extended Hoare triples, of the last chapter. We shall prove that in this interpretation a reduced set of deduction rules is sound with respect to the process based semantics.

7.2 Interpreting Quintuples.

The notion of memory that we used in the last chapter we will use again but with the following extension:

$$m\{\!|\, s\overline{d}\,|\!\}m' \text{ iff } m\{\!|\, s\,|\!\}m'$$

this simply states that actions signaling termination do not affect the memory.

Definition 7.2.1 *Given a proposition P and a memory m the notation $m \models P$ means that $P[m(\tilde{X})/\tilde{X}]$ holds. Where $\tilde{X}$ is the set of all the program variables.*

Definition 7.2.2 *We will say that*

$$\{\Gamma, P\}C\{Q, \Delta\}$$

is valid *precisely when for all $a_1, \ldots a_n$ such that $[\![C]\!] \xrightarrow{a_1} \ldots \xrightarrow{a_n} D$ then*

for all memory maps $m_0 m'_0 \ldots m_n m'_n$

if $m_0 \models P$

and for all $0 \leq i \leq n$ for all $R \in \Gamma$, $m_i \models R \supset m'_i \models R$

and for all $1 \leq i \leq n$ $m'_{i-1}\{\!|\, a_i\,|\!\}m_i$

then for all $1 \leq i \leq n$ for all $S \in \Delta$, $m'_{i-1} \models S \supset m_i \models S$

and moreover if $a_n = \overline{d}$ then $m'_n \models Q$ and $D \sim nil$.

Informally we think of the transitions between m_i and m'_i as external actions, and we require that external actions do not falsify anything in the rely condition. The transitions between m'_{i-1} and m_i are internal transitions; so we require that they are consistent with internal actions. So given an initial memory m_0 at which P holds, then over all internal actions the guarantee conditions will not be falsified provided that the rely conditions are maintained. Finally if the command terminates then Q will hold of the final memory. We could have chosen the following slightly different definition.

Definition 7.2.3 *The quintuple*

$$\{\Gamma, P\}C\{Q, \Delta\}$$

is valid *precisely when for all $a_1, \ldots a_n$ such that $[\![C]\!] \xrightarrow{a_1} \ldots \xrightarrow{a_n} D$ then*

for all memory maps $m_0^0 \ldots m_0^{k_0}, \ldots, m_n^0 \ldots m_n^{k_n}$
if $m_0^0 \models P$
and for all $0 \leq i \leq n$ for all $0 \leq j \leq k_i - 1$ for all $R \in \Gamma$,
$m_i^j \models R \supset m_i^{j+1} \models R$
and for all $1 \leq i \leq n$ $m_{i-1}^{k_{i-1}} \{\!|a_i|\!\} m_i^0$
then for all $1 \leq i \leq n$ for all $S \in \Delta$,
$m_{i-1}^{k_{i-1}} \models S \supset m_i^0 \models S$
and moreover if $a_n = \overline{d}$ then $m_n^{k_n} \models Q$ and $D \sim nil$.

This definition differs from the earlier one in that we have allowed multiple external actions, instead of assuming that they can all be represented by one step in the memory chain.

Proposition 7.2.4 *The above two definitions of validity are equivalent.*

Proof:

1. Definition 7.2.3 implies 7.2.2, choose $k_i = 1$ for all i and then let $m_i \equiv m_i^0$ and $m'_i \equiv m_i^1$, the definitions are now identical.

2. In the other direction, let

$$[\![C]\!] \xrightarrow{a_1} \ldots \xrightarrow{a_n} D$$

and let $m_0^0 \ldots m_0^{k_0}, \ldots m_n^0, \ldots m_n^{k_n}$ be such that they obey the antecedent of definition 7.2.3. So we pick $m_i \equiv m_i^0$ and $m_i' \equiv m_i^{k_i}$ for all i. Then we can deduce the antecedent of definition 7.2.2, namely

$m_0 \models P$

for all $0 \leq i \leq n$ for all $R \in \Gamma$, $m_i \models R \supset m_i' \models R$

for all $1 \leq i \leq n$ $m_{i-1}' \{\!| a_i |\!\} m_i$

thus from definition 7.2.2 we can deduce that

for all $1 \leq i \leq n$ for all $S \in \Delta$ $m_{i-1}' \models S \supset m_i \models S$

and if $a_n = \bar{d}$ then $m_n' \models Q$ and $D \sim nil$

but this is precisely the consequent of Definition 7.2.3 (up to relabelling). □

Before we can attempt a soundness proof of the reduced inference system we need an interpretation of invariant containment, for predicate sets, with respect to CCS processes. We also need a definition of satisfaction for predicates with respect to the process model.

Definition 7.2.5 *We interpret $\Gamma \Rightarrow A$ to mean that, for all m and m', whenever $m \models R \supset m' \models R$ for all $R \in \Gamma$, then also $m \models A \supset m' \models A$.*

Definition 7.2.6 *$A \models B$ if for all memories m, $m \models A$ implies $m \models B$.*

7.3 Soundness.

Proposition 7.3.1 *The rules* null, assignment, sequence, iteration, alternation, weakening *and* parallel *of figure 7-1 are sound with respect to the process based interpretation.*

Proof:

1. Null:

$$\frac{}{\{\Gamma, P\} SKIP \{P, \Delta\}}$$

this holds vacuously; since the only action $[\![SKIP]\!]$ can perform is $\bar{d}$ and this leaves memories unchanged by definition.

2. Assignment:

$$\frac{\Gamma \Rightarrow P \quad P \models R[t/X] \quad \forall Q \in \Delta, P \wedge Q \models Q[t/X]}{\{\Gamma, P\} X := t \{R, \Delta\}}$$

Given $[\![X := t]\!] \xrightarrow{a_1} \ldots \xrightarrow{a_n} D$ and we have memories $m_0 m'_0 \ldots m_n m'_n$ such that

$m_0 \models P$

and for all $0 \leq i \leq n$ for all $S \in \Gamma$, $m_i \models S \supset m'_i \models S$

and for all $1 \leq i \leq n$ $m'_{i-1}\{\!| a_i |\!\}m_i$.

For all $R \in \Delta$, if $m_{i-1} \models R$ then either

- $m'_{i-1} \models A$ in which case from $P \wedge R \models R[t/X]$, $m_i \models R$ or,
- $m_i \not\models P$ in which case the action which performs the assignment has been made and since $[\![X := t]\!]$ has only one write action subsequent actions do not affect the memory, hence $m_i \models R$.

hence for all $1 \leq i \leq n$ for all $R \in \Delta$, $m'_{i-1} \models R \supset m_i \models R$.

If $[\![X := t]\!]$ terminates then as there is only one write action and all subsequent actions do not affect the memory. From $\Gamma \Rightarrow P$ and $P \models Q[t/X]$ we can deduce $\Gamma \Rightarrow Q[t/X]$, thus if $m_i \models Q$ then $m'_i \models Q$ and since after the a_k which is the write action there exists m'_k such $m'_k \models Q$. All subsequent actions (either environmental or local) do not affect the validity of Q and hence $m'_n \models Q$ as required.

3. Alternation:

$$\frac{\Gamma \Rightarrow P \quad \{\Gamma, P \wedge b\} C_1 \{Q, \Delta\} \quad \{\Gamma, P \wedge \neg b\} C_2 \{Q, \Delta\}}{\{\Gamma, P\} IF\ b\ THEN\ C_1\ ELSE\ C_2 \{Q, \Delta\}}$$

Let $[\![IF\ b\ THEN\ C_1\ ELSE\ C_2]\!] \xrightarrow{a_1} \ldots \xrightarrow{a_n} D$ then there are two direct sub-cases;

(a) a_n occurs before the execution of either C_1 or C_2 starts. Then this process so far leaves memories unchanged since it is performing an evaluation. Hence trivially the guarantee conditions are met.

(b) Otherwise there is some a_k which is the first action from whence the execution of C_1 or C_2 then continues. Since $\Gamma \Rightarrow P$ and conditional evaluation leaves memory unchanged; we know that $m'_{k-1} \models P$ and $m_k \models P$. For the memory sequence $m_k m'_k \dots m_n m'_n$ assume wlog that $m'_{k-1} \models P \wedge b$. From the left hand premise we have that given $[\![C_1]\!] \xrightarrow{b_1} \dots \xrightarrow{b_{n-k}} D'$, with $D' = D$;
for all memory maps $l_0 l'_0 \dots l_{n-k} l'_{n-k}$
if $l_0 \models P$
and for all $0 \leq i \leq n-k$ for all $R \in \Gamma$, $l_i \models R \supset l'_i \models R$
and for all $1 \leq i \leq n-k$ $l'_{i-1} \{\!| a_i |\!\} l_i$
then for all $1 \leq i \leq n-k$ for all $S \in \Delta$, $l'_{i-1} \models S \supset l_i \models S$
and moreover if $a_n = \overline{d}$ then $m_n \models Q$ and $D \sim nil$.
By identifying m_{j+k} with l_j and m'_{j+k} with l'_{j+k} for all $1 \leq j \leq n-k$, we have for all $1 \leq j \leq n-k$ and for all $S \in \Delta$, that $m'_{j+k-1} \models S \supset m_{j+k} \models S$. Combining this with the earlier proof for the first $k-1$ internal actions we have for all $1 \leq i \leq n$ for all $S \in \Delta$, $m'_{i-1} \models S \supset m_i \models S$.

If $[\![C_1]\!]$ (for instance) terminates then from the left hand premise (given b is true) we have that $l'_{n-k} \models Q$ and $D' \sim nil$ thus $m'_n \models Q$ and $D \sim nil$ as required. A symmetric argument proves the case when b is false.

4. Sequence:

$$\frac{\{\Gamma, P\} C_1 \{Q, \Delta\} \quad \{\Gamma, Q\} C_2 \{R, \Delta\}}{\{\Gamma, P\} C_1 ; C_2 \{R, \Delta\}}$$

There are three cases.

(a) $[\![C_1]\!]$ does not terminate. In which case we map memories $m_0 m'_0 \dots m_n m'_n$ to the memories in the definition of the left hand premise and obtain the required guarantee property directly.

(b) $[\![C_1]\!]$ terminates but $[\![C_2]\!]$ does not. Then in $[\![C_1; C_2]\!] \xrightarrow{a_1} \ldots \xrightarrow{a_n} D$ either we have not started executing C_2; in which case the immediately preceding proof would still apply; or there is an a_k which is the τ corresponding to the termination signal of $[\![C_1]\!]$. So for a memory sequence $m_0 m_0' \ldots m_k m_k'$ we have from the left hand premise that for all $1 \leq i \leq k$ and for all $S \in \Delta$, $m_{i-1}' \models S \supset m_i \models S$ and $m_k \models Q$. Thus choosing the memories in the right hand premise as $m_{k+1} m_{k+1}' \ldots m_n m_n'$ then these memories have all the requisite properties (silent actions do not change memory values) and we can deduce that for all $k+2 \leq i \leq n$ for all $S \in \Delta$, $m_{i-1}' \models S \supset m_i \models S$ (from the right hand premise). Hence for all $1 \leq i \leq n$ for all $S \in \Delta$ then $m_{i-1} \models S \supset m_i \models S$.

(c) Both $[\![C_1]\!]$ and $[\![C_2]\!]$ terminate, then from (b) we have that the guarantee requirement is already met and since if $[\![C_2]\!]$ terminates then matching memories as for part (b), we have that $m_n' \models R$ and $D \sim nil$ as required.

5. Iteration:

$$\frac{\Gamma \Rightarrow P \quad \{\Gamma, P \wedge b\} C \{P, \Delta\}}{\{\Gamma, P\} \mathit{WHILE}\ b\ \mathit{DO}\ C \{P \wedge \neg b, \Delta\}}$$

given $\mathit{WHILE}\ b\ \mathit{DO}\ C \xrightarrow{a_1} \ldots \xrightarrow{a_n} D$

for all memory maps $m_0 m_0' \ldots m_n m_n'$

if $m_0 \models P$

and for all $0 \leq i \leq n$ for all $Q \in \Gamma$, $m_i \models Q \supset m_i' \models Q$

and for all $1 \leq i \leq n$ $m_{i-1}' \{\!| a_i |\!\} m_i$

Now as in the proof for *IF THEN ELSE* we know that there are (say) k actions which are required to evaluate the condition b and that these do no affect the memory. Thus for all $1 \leq i \leq k$ for all $R \in \Delta$, $m_{i-1}' \models R \supset m_i \models R$. Since $\Gamma \Rightarrow P$ we can deduce that if $m_0 \models P$ then $m_{k+1} \models P$ (*). So we can use the memories $m_{k+1} m_{k+1}' \ldots m_l m_l'$ up to some l in the premise $\{\Gamma, P \wedge b\} C \{P, \Delta\}$. (Assuming the loop does not terminate trivially.)

Thus for all $1 \leq i \leq l$ for all $R \in \Delta$, $m'_{i-1} \models R \supset m_i \models R$; and moreover if $[\![WHILE \;\; b \;\; DO \;\; C]\!]$ terminates; after one particular execution of C then there will be a $m'_{l+k} \models P \wedge \neg b$, as memory is invariant under condition evaluation.

Proceeding inductively along the execution path of $[\![WHILE \;\; b \;\; DO \;\; C]\!]$; we can see that for all $1 \leq i \leq n$ for all $R \in \Delta$, $m'_{i-1} \models R \supset m_i \models R$. Moreover if the process $[\![WHILE \;\; b \;\; DO \;\; C]\!]$ terminates then it terminates after an evaluation which gives b false; and at which point as in (*) we know that $m'_n \models P$ and $m' \models \neg b$ hence $m'_n \models P \wedge \neg b$.

6. Parallel:

$$\frac{\Gamma \Rightarrow Q_1 \;\; \{\Gamma, P_1\}C_1\{Q_1, \Sigma \cup \Delta\} \;\; \{\Sigma, P_2\}C_2\{Q_2, \Gamma \cup \Delta\} \;\; \Sigma \Rightarrow Q_2}{\{\Gamma \cup \Sigma, P_1 \wedge P_2\}C_1 \;\; PAR \;\; C_2\{Q_1 \wedge Q_2, \Delta\}}$$

We must show that if $[\![C_1 \;\; PAR \;\; C_2]\!] \xrightarrow{a_1} \ldots \xrightarrow{a_n} D$,
then for all memory maps $m_0 m'_0 \ldots m_n m'_n$
if $m_0 \models P_1 \wedge P_2$
and for all $0 \leq i \leq n$ for all $R \in \Gamma \cup \Sigma$,
$m_i \models R \supset m'_i \models R$
and for all $1 \leq i \leq n$ $m'_{i-1}\{\!| a_i |\!\}m_i$
then for all $1 \leq i \leq n$ for all $S \in \Delta$, $m'_{i-1} \models S \supset m_i \models S$
and moreover if $a_n = \overline{d}$ then $m'_n \models Q$ and $D \sim nil$.
We can assume that

$$[\![C_1]\!] \xrightarrow{b_1} \ldots \xrightarrow{b_{n'}} E,$$
$$[\![C_2]\!] \xrightarrow{c_1} \ldots \xrightarrow{c_{n''}} F;$$

with $E \mid F = D$ and that the actions a_i are some arbitrary interleaving of the actions $b_1 \ldots b_{n'}$ and the actions $c_1 \ldots c_{n''}$. Let the associated memory sequences of $[\![C_1]\!]$ and $[\![C_2]\!]$ evolve through be $p_0 p'_0 \ldots p_{n'} p'_{n'}$ and $q_0 q'_0 \ldots q_{n''} q'_{n''}$ respectively. These sequences of memories are subsequences of the sequence $m_0 m'_0 \ldots m_n m'_n$ which can be defined as follows.

There is a partition of the indexes $\{1, \ldots, n\}$ into $I = \{i_1, \ldots, i_{n'}\}$ and $J = \{j_1, \ldots, j_{n''}\}$, such that $a_{i_1} \ldots a_{i_{n'}} = b_1 \ldots b_{n'}$ and

$a_{j_1} \dots a_{j_{n''}} = c_1 \dots c_{n''}$. Also $b_{n''} = \tau$ if $[\![C_1]\!]$ terminates, and $c_{n''} = \tau$ if $[\![C_2]\!]$ terminates.

Then we set $p_0 = q_0 = m_0$, and $p'_{n'} = q'_{n''} = m'_n$ and otherwise, for $k = 1, \dots, n'$;

$$p'_{k-1} = m'_{i_k-1} \text{ and } p_k = m_{i_k};$$

and for $k = 1, \dots, n''$;

$$q'_{k-1} = m'_{j_k-1} \text{ and } q_k = m_{j_k}.$$

A diagram will make the construction clear. We have taken $n = 6$, $n' = 4$, $n'' = 2$, with $a_1a_2a_3a_4a_5a_6 = b_1c_1b_2b_3c_2b_4$.

$$\boxed{\begin{array}{cccccccccccccc}
 & & a_1 & & a_2 & & a_3 & & a_4 & & a_5 & & a_6 & \\
m_0 & m'_0 & m_1 & m'_1 & m_2 & m'_2 & m_3 & m'_3 & m_4 & m'_4 & m_5 & m'_5 & m_6 & m'_6 \\
 & & b_1 & & & & b_2 & & b_3 & & & & b_4 & \\
p_0 & p'_0 & p_1 & & & p'_1 & p_2 & p'_2 & p_3 & & & p'_3 & p_4 & p'_4 \\
 & & & & c_1 & & & & & & c_2 & & & \\
q_0 & & & q'_0 & q_1 & & & & & q'_1 & q_2 & & & q'_2
\end{array}}$$

The memory sequences $p_0p'_0 \dots p_{n'}p'_{n'}$ and $q_0q'_0 \dots q_{n''}q'_{n''}$ will now be shown to obey the antecedents for the quintuples,

$$\{\Gamma, P_1\}C_1\{Q_1, \Sigma \cup \Delta\} \quad ,$$
$$\{\Sigma, P_2\}C_2\{Q_2, \Gamma \cup \Delta\} \quad ,$$

to hold. Since $m_0 \models P_1 \wedge P_2$, we have that $p_0 \models P_1$ and $q_0 \models P_2$. Consider the first action of $[\![C_2]\!]$ either

- this is preceded by no evolutions of $[\![C_1]\!]$ in which case $q'_0 = m'_0$ and by assumption for all $R \in \Sigma \cup \Gamma$, $m_0 \models R \supset m'_0 \models R$ thus for all $S \in \Sigma$, $q_0 \models S \supset q'_0 \models S$.
- Or there are one or more evolutions of $[\![C_1]\!]$ (say k) before the first action of $[\![C_2]\!]$. In this case we have $p_0p'_0 \dots p_k$ identical to $m_0m'_0 \dots m_k$ and, as above, for all $0 \le i \le k-1$ for all $R \in \Gamma \cup \Sigma$, $m_i \models R \supset m'_i \models R$.

In other words for all $0 \leq i \leq k-1$ for all $S \in \Gamma$,
$p_i \models S \supset p'_i \models S$
for all $1 \leq i \leq k$, $p'_{k-1}\{\| b_i \|\} p_i$
and $p_0 \models P_1$,
thus for all $1 \leq i \leq k$ for all $R \in \Sigma \cup \Delta$,
$p'_{i-1} \models R \supset p_i \models R$
or for all $1 \leq i \leq k$ for all $S \in \Sigma$, $p'_{i-1} \models S \supset p_i \models S$ and
$p_i \models S \supset p'_i \models S$.
moreover from for all $R \in \Gamma \cup \Sigma$, $m_k \models R \supset m'_k \models R$, by definition $q'_0 = m'_k$ and from the above it is clear that for all $S \in \Sigma$,
$q_0 \models S \supset q'_0 \models S$.

By an identical argument for all $R \in \Gamma$, $p_0 \models R \supset p'_0 \models R$.

We now proceed inductively over the following statements:

(a) for all $0 \leq i \leq k_p$ for all $R \in \Gamma$, $p_i \models R \supset p'_i \models R$,

(b) for all $0 \leq j \leq k_q$ for all $S \in \Sigma$, $q_j \models S \supset q'_j \models S$;

with $k_p \leq n'$ and $k_q \leq n''$. To prove the second clause we assume that for all $0 \leq j \leq k_q$ for all $S \in \Sigma$, $q_j \models S \supset q'_j \models S$ given this we can use $q_0 q'_0 \cdots q_{k_q} q'_{k_q}$ in the definition for $\{\Sigma, P_2\} C_2 \{Q_2, \Gamma \cup \Delta\}$ and deduce that for all $1 \leq i \leq k_q$ for all $R \in \Gamma$, $q'_{i-1} \models R \supset q_i \models R$. There will be a maximal (possibly zero) b_k such that b_k occurs before c_{k_q+1}. For all memory pairs $p_i p'_i$, $0 \leq i \leq k$ the transitions come either from the environment which guarantees $\Gamma \cup \Sigma$ or from an interleaving of environment or C_2 moves. Environment moves guarantee $\Gamma \cup \Sigma$, C_2 moves as far as action k_q guarantee Γ; so when they are interleaved Γ is maintained. But when Γ is maintained over environment moves of C_1 it will maintain $\Sigma \cup \Delta$, so at least up to action b_k, $\Sigma \cup \Delta$ will be maintained.

For the memory transition $q_{k_q+1} q'_{k_q+1}$ the move is either purely environmental in which case $\Gamma \cup \Sigma$ is ensured, or it is an interleaving of environment moves and moves from C_1 which ensure $\Sigma \cup \Delta$ (at least up to this point). Therefore for all $S \in \Sigma$ $q_{k_{q+1}} \models S \supset q'_{k_q+1} \models S$.

The proof of the other case is symmetrical.

Thus the memory sequences $p_0 p'_0 \cdots p_n p'_{n'}$ and $q_0 q'_0 \cdots q_{n''} q'_{n''}$ are of the form admitted by

$$\{\Gamma, P_1\} C_1 \{Q_1, \Sigma \cup \Delta\} \ ,$$
$$\{\Sigma, P_2\} C_2 \{Q_2, \Gamma \cup \Delta\} \ ;$$

so for all $1 \leq i \leq n$ and $m'_{i-1} m_i$ there is a matching $p'_{k-1} p_k$ or $q'_{j-1} q_j$ for some k or j and from the above for all $R \in \Gamma \cup \Delta$, $p'_{k-1} \models R \supset p_k \models R$ or for all $S \in \Gamma \cup \Delta$, $q'_{j-1} \models S \supset q_j \models S$. Thus for all $1 \leq i \leq n$ for all $T \in \Delta$, $m'_{i-1} \models T \supset m_i \models T$

Finally if both $[\![C_1]\!]$ and $[\![C_2]\!]$ terminate then $p_{n'} \models Q_1$ and $q_{n''} \models Q_2$. We can assume wlog that $[\![C_1]\!]$ terminates first; in which case the only subsequent actions of $[\![C_1 \ \ PAR \ \ C_2]\!]$ are those of $[\![C_2]\!]$, τ and $\bar{d}$, of which the latter two do not affect memory values. From $\Gamma \Rightarrow Q_1$ we know that all the actions of $[\![C_2]\!]$ will leave Q_1 invariant; and since the environment guarantees $\Gamma \cup \Sigma$, that will also leave the validity of Q_1 invariant.

So once Q_1 holds of a memory in the evolution path of $[\![C_1 \ \ PAR \ \ C_2]\!]$ it will continue to hold until the process terminates. When $[\![C_2]\!]$ terminates Q_2 will also hold. From the definition of $[\![C_1 \ \ PAR \ \ C_2]\!]$, a_n will be $\bar{d}$ and $D \sim nil$ upon termination. So we have $m_n \models Q_1 \wedge Q_2$, by assumption for all $R \in \Gamma \cup \Sigma$, $m_n \models R \supset m'_n \models R$ and $\Gamma \Rightarrow Q_1$ with $\Sigma \Rightarrow Q_2$, thus $\Gamma \cup \Sigma \Rightarrow Q_1 \wedge Q_2$ hence $m'_n \models Q_1 \wedge Q_2$.

7. Weakening:

$$\frac{\Gamma \Rightarrow \Gamma' \quad P \models P_1 \quad \{\Gamma', P_1\} C \{Q_1, \Delta'\} \quad Q_1 \models Q \quad \Delta' \Rightarrow \Delta}{\{\Gamma, P\} C \{Q, \Delta\}}$$

This follows immediately from the definitions of $\Rightarrow$ and the logic of propositions. □

7.4 Relation to Non-Interfering Methods.

We can obtain a system of proof identical to that for our non-interfering language, if the following rely and guarantee conditions are used;

$$\text{Rely: } \{X = v \mid X \in \mathcal{W}(C) \cup \mathcal{R}(C) \text{ and } v \in \mathcal{V}\},$$
$$\text{Guarantee: } \{X = v \mid X \notin \mathcal{W}(C) \text{ and } v \in \mathcal{V}\}.$$

Informally we rely on the environment not changing the value of any variable we read or write. Similarly we can guarantee that the value of variables we do not write to will remain unchanged. The above rely and guarantee conditions are exceedingly strong which reflects just how restrictive the determinism condition is.

7.5 Rewriting the Definitions in $\mathcal{PL}$.

We originally stated that we were going to express validity in terms of the satisfaction of a $\mathcal{PL}$ formula but have so far only provided and used a long hand definition. For completeness, the following is a re-expression of definition 7.2.2.

Definition 7.5.1 *let $s \in \Lambda^*$ be an action sequence and s_i its ith member; then the quintuple*

$$\{\Gamma, P\}C\{Q, \Delta\}$$

is valid *precisely when*

$$[\![C]\!] \models \forall m_0 m'_0 \ldots m_n m'_n.\ (P[m(\tilde{X}/\tilde{X})] \wedge \forall i.\ m'_{i-1}\{\!|\, s_i \,|\!\}m_i$$
$$\wedge \forall i \forall R \in \Gamma, R[m_i(\tilde{X})/\tilde{X}]$$
$$\supset R[m'_i(\tilde{X})/\tilde{X}] \wedge [[s]]\ (\forall i \forall S \in \Delta, S[m'_{i-1}(\tilde{X})/\tilde{X}] \supset S[m_i(\tilde{X})/\tilde{X}] \wedge$$
$$((s_n = \overline{d}) \supset Q[m'_n(\tilde{X})/\tilde{X}])))$$

7.6 Conclusions.

We have presented two different proof systems over concurrent while languages. Their validity has been shown to be expressible as formulae of $\mathcal{PL}$ and hence we have been able to prove the soundness of such systems relative to a process based semantics. The correspondence between our condition ensuring determinacy and the restrictions/extensions to proof systems needed to manage concurrency has been demonstrated. Restricting ourselves to programs that are deterministic greatly eases the proof burden when trying to establish the properties of such programs. Thus as we would expect the sub-class of CIMP programs which are deterministic is of some value, when we need to show program correctness.

Chapter 8

Timing Concurrent Processes.

8.1 Introduction.

The temporal properties of concurrent processes give an insight into some interesting aspects of concurrent programming. There have been some attempts to provide a formalism within which these concepts can be expressed [Ros1, Koy1, Jef]. Most of these assume synchrony which results in some of the more interesting temporal properties of processes being inexpressible. In a previous paper [Tof] we provided an extension to CCS which admitted a notion of timing. That approach was unsatisfactory in that the account it gave of time was somewhat eccentric. Processes could only evolve simultaneously by communication, time and action were interleaved otherwise. In order to give a fuller account of time for asynchronous processes we need to just let time pass, and observe what the processes produce and when they produce it.

In order to give this fuller account the state transitions have been split into two orthogonal parts; one part is our normal notion of action (which can be regarded as computation) and the other part is the passage of time. There are sound reasons for making this separation. Computation involves energy change and there is a result of quantum mechanics which states that energy changes and time cannot be measured simultaneously [Dir, Sch]. Thus it seems reasonable when producing models of time and computation not to permit the simultaneous observance of the two activities.

We therefore assume that actions have no duration; although if we wish to construct actions with duration we can in the manner of [Cas].

8.2 Language Definition.

We define a timed extension of the language CCS [Mil1] as follows.

Let Λ be a set of (atomic action) symbols not containing τ or ϵ, and let $Act = \Lambda \cup \{\tau\}$. We also have times t taken from one of the following: positive integers, positive rationals or positive reals, representing the divisions of time. We assume a complementation bijection $\bar{\cdot} : Act \longrightarrow Act$ which is its own inverse. The letter λ ranges over Λ, the letter μ over Act, and S over *relabelling functions*, i.e. those $S : Act \longrightarrow Act$ such that $S(\overline{\mu}) = \overline{S(\mu)}$ and $S(\mu) \neq \tau$, unless $\mu = \tau$. The languages wTCCS (weakly timed CCS) and sTCCS (strongly timed CCS) consist of an infinite set Var of variables ranged over by X and Y, a constant symbol Nil, unary function symbols $\mu.$, $\backslash\lambda$, $[S]$, and FIX_X ($X \in Var$), and the binary function symbols $+$ and $|$. With for each action μ a unary function $\mu.$ taking a process and prefixing the process by that given action. In wTCCS the function symbol [], denotes a function which takes a process and a time and yields a process prefixed by that amount of time. The function () takes a process and a time (not zero), and returns the process prefixed by that time, along with the unary function δ which returns a delayed process in the calculus sTCCS.

The set P of wTCCS-expressions ranged over by P is the set given by the following definition:

$$P \ ::= \ Nil \mid \mu.P \mid [t]P \mid P \mid P \mid P + P \mid FIX_X P \mid X \mid P\backslash L \mid P[S].$$

The set P of sTCCS-expressions ranged over by P is the set given by the following definition, with $t \neq 0$:

$$P \ ::= \ Nil \mid \mu.P \mid (t)P \mid \delta P \mid P \mid P \mid P + P \mid FIX_X P \mid X \mid P\backslash L \mid P[S].$$

(It will be clear from the context which version of the timing system is being used.) The intention of the time action prefix is as follows. In wTCCS, $[t]P$ means that after a period of time t' where $t' \geq t$ the process P is reached; this is very similar to the *wait* introduced into CSP by Roscoe and Reed [Ros1]. On the other hand, the system sTCCS represents separation of delay from the initial timing so $(t)P$ represents a process that will becomes P in precisely a period of time t. The δ operator providing a way of introducing delays to allow for synchronisation.

8.2.1 Derivation Laws.

The action-evolution of a process can be derived from the operational rules presented in Figure 8-1. The temporal evolutions of wTCCS are derived using the operational rules presented in Figure 8-2. The alternative set of rules presented in Figure 8-3, give the temporal evolutions for sTCCS. The transition relations between processes is the least set of transitions satisfying the set of action laws plus the appropriate set of temporal laws.

Definition 8.2.1 *We define the operator $\dot{-}$ on times as follows;*

$$t \dot{-} t' = \begin{cases} t - t' \textit{ when } t' \leq t \\ 0 \textit{ otherwise,} \end{cases}$$

where $-$ is the usual subtraction operator.

8.2.2 An Example.

In wTCCS the following process deadlocks,

$$(a.b.Nil \mid \overline{a}.[5]\overline{b}.Nil)\backslash a, b.$$

The process has the following derivation,

$$(a.b.nil \mid \overline{a}.[5]\overline{b}.nil)\backslash a, b \xrightarrow{\tau} (b.nil \mid [5]\overline{b}.nil)\backslash a, b.$$

$$\text{ACT:}\ \frac{}{\mu.P \xrightarrow{\mu} P}$$

$$\text{SUM0:}\ \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \qquad \text{SUM1:}\ \frac{Q \xrightarrow{\mu} Q'}{P + Q \xrightarrow{\mu} Q'}$$

$$\text{COM0:}\ \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \qquad \text{COM1:}\ \frac{Q \xrightarrow{\mu} Q'}{P \mid Q \xrightarrow{\mu} P \mid Q'}$$

$$\text{COM2:}\ \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{REL:}\ \frac{P \xrightarrow{\mu} P'}{P[S] \xrightarrow{S(\mu)} P'[S]}$$

$$\text{RES:}\ \frac{P \xrightarrow{\mu} P'}{P\backslash L \xrightarrow{\mu} P'\backslash L} \qquad \mu, \overline{\mu} \notin L$$

$$\text{REC:}\ \frac{P_j\{\tilde{P}/\tilde{X}\} \xrightarrow{\mu} P_j'}{FIX_j\{X_i = P_i;\ i \in I\} \xrightarrow{\mu} P_j'}$$

Figure 8–1: Operational rules for wTCCS and sTCCS

$$\text{TIME:}\ \frac{}{[t]P \overset{t'}{\rightsquigarrow} [t \dot{-} t']P} \qquad \text{TRANS:}\ \frac{P \overset{t}{\rightsquigarrow} P' \quad P' \overset{s}{\rightsquigarrow} P''}{P \overset{s+t}{\rightsquigarrow} P''}$$

$$\text{COM-TIME:}\ \frac{P \overset{t}{\rightsquigarrow} P' \quad Q \overset{t}{\rightsquigarrow} Q'}{P \mid Q \overset{t}{\rightsquigarrow} P' \mid Q'} \qquad \text{INV:}\ \frac{}{P \overset{t}{\rightsquigarrow} P}$$

$$\text{SUM-TIME:}\ \frac{P \overset{t}{\rightsquigarrow} P' \quad Q \overset{t}{\rightsquigarrow} Q'}{P + Q \overset{t}{\rightsquigarrow} P' + Q'}$$

$$\text{ACT-TIME:}\ \frac{P \xrightarrow{\mu} P'}{[0]P \xrightarrow{\mu} P'}$$

$$\text{REL-TIME:}\ \frac{P \overset{t}{\rightsquigarrow} P'}{P[S] \overset{t}{\rightsquigarrow} P'[S]}$$

$$\text{RES-TIME:}\ \frac{P \overset{t}{\rightsquigarrow} P'}{P\backslash L \overset{t}{\rightsquigarrow} P'\backslash L}$$

$$\text{REC-TIME:}\ \frac{P_j\{\tilde{P}/\tilde{X}\} \overset{t}{\rightsquigarrow} P_j'}{FIX_j\{X_i = P_i;\ i \in I\} \overset{t}{\rightsquigarrow} P_j'}$$

Figure 8–2: Temporal rules for wTCCS

$$\text{TIME:}\ \frac{}{(t)P \overset{t'}{\rightsquigarrow} (t-t')P} \qquad \text{for } t > t'.$$

$$\text{TIME1:}\ \frac{}{(t)P \overset{t}{\rightsquigarrow} P} \qquad\qquad \text{TRANS:}\ \frac{P \overset{t}{\rightsquigarrow} P' \quad P' \overset{s}{\rightsquigarrow} P''}{P \overset{s+t}{\rightsquigarrow} P''}$$

$$\text{DELAY:}\ \frac{}{\delta P \overset{t}{\rightsquigarrow} \delta P}$$

$$\text{UN-DELAY:}\ \frac{P \xrightarrow{\mu} P'}{\delta P \xrightarrow{\mu} P'}$$

$$\text{COM-TIME:}\ \frac{P \overset{t}{\rightsquigarrow} P' \quad Q \overset{t}{\rightsquigarrow} Q'}{P \mid Q \overset{t}{\rightsquigarrow} P' \mid Q'}$$

$$\text{SUM-TIME:}\ \frac{P \overset{t}{\rightsquigarrow} P' \quad Q \overset{t}{\rightsquigarrow} Q'}{P + Q \overset{t}{\rightsquigarrow} P' + Q'}$$

$$\text{REL-TIME:}\ \frac{P \overset{t}{\rightsquigarrow} P'}{P[S] \overset{t}{\rightsquigarrow} P'[S]}$$

$$\text{RES-TIME:}\ \frac{P \overset{t}{\rightsquigarrow} P'}{P\backslash L \overset{t}{\rightsquigarrow} P'\backslash L}$$

$$\text{REC-TIME:}\ \frac{P_j\{\tilde{P}/\tilde{X}\} \overset{t}{\rightsquigarrow} P_j'}{FIX_j\{X_i = P_i;\ i \in I\} \overset{t}{\rightsquigarrow} P_j'}$$

Figure 8–3: Temporal rules for sTCCS

EXT-DELAY-1: $\dfrac{P \overset{t}{\leadsto} P'}{P \mid Q \overset{t}{\leadsto} P' \mid Q}$	EXT-DELAY-2: $\dfrac{Q \overset{t}{\leadsto} Q'}{P \mid Q \overset{t}{\leadsto} P \mid Q'}$

Figure 8–4: Delay Rules (DR).

In the parallel composition above, the right hand process now requires a period of time of length at least 5 to pass before it will undergo any further activity; the left hand process requires a matching $\bar{b}$ action to proceed but will not delay. No further action is therefore possible.

So actions not separated by delays must be performed sequentially and immediately in time; failure to do so leads to a deadlock.

This would seem to imply that it would be useful to add *temporal* evolution laws as in figure 8-4.

Proposition 8.2.2 *Let P be a process in wTCCS in which all actions are guarded by time actions, and all Nil processes are guarded by* [0]. *Then P has exactly the same evolutions in wTCCS + DR.*

Proof: Since all actions are guarded by time prefixes, we never reach a state where we must infer (by ACT alone) that $P \xrightarrow{\lambda} P'$. In other words for all processes P we can always infer $P \overset{t'}{\leadsto} P'$ for t' larger than some arbitrary t, thus we can never reach a state where in a parallel composition $P \mid Q$, either P or Q can only evolve through an action, therefore we can infer any evolution that we could infer with the addition of the rules DR.

Notation: we will use the abbreviation P^t to represent the t evolution of the process P i.e $P \overset{t}{\leadsto} P'$, whenever it is inconvenient to introduce a new process name for all the timed intermediates of a process.

8.2.3 Temporal Deadlock.

The *Nil* or *deadlocked* process is one which is capable of no action. There is an equivalent temporal process which can always permit the passage of time, but never produces an action. The following are the simplest definitions for both wTCCS and sTCCS respectively:

$$[0]Nil,$$
$$\delta Nil.$$

There are, however, an infinite variety of processes which are equivalent to this temporal nil. For any P in wTCCS and Q in sTCCS, the following processes are equivalent to the temporal nil:

$$[0][1]P,$$
$$\delta(1)Q,$$

since in neither case can we infer a direct action, and so we cannot remove the leading [0] or the δ operators.

8.2.4 Deadlock.

In sTCCS the Nil process acts like a *deadlock* [Ber] with respect to the continuing temporal evolution of the system. If we examine the derivation laws $COM - TIME$ and $SUM - TIME$, we observe that the following are true for any non-zero t.

$$Nil + (t)P \text{ has the same derivatives as } Nil$$
$$Nil \mid (t)P \text{ has the same derivatives as } Nil$$

Thus once we have an unguarded Nil process in any leading binary term *all* further temporal evolution is blocked. Any composite process is stopped immediately, the process cannot evolve further in time. We can use this property to compare the initial time behaviour of any processes.

Definition 8.2.3 *Given any equivalence [order] we can construct the time pre-fix equivalence [order] simply by composing in parallel both processes we wish to show equivalent[related] with the process $(t)Nil$.*

This is motivated from the proceeding observation. After a period of time t has passed the processes $(t)Nil \mid P$ and $(t)Nil \mid Q$ become equivalent to Nil; until t has passed they exhibit all the possible behaviour of the respective processes within that period.

8.3 Strong Time Sensitive Pre-order.

It is possible, much as for CCS, to produce two different basic notions of equivalence (strong and weak), which are based on orders with respect to time. The first requires that the time taken after any action be matched directly by the time taken after a similar action in the other process, not that the total time taken to go from one state to another via an action is greater.

Definition 8.3.1 *(Strong pre-order) We will say that P is* faster *than Q iff there exists a relation R with $(P,Q) \in R$ if for all $\mu \in Act$ and for all times t;*

1. *if $P \xrightarrow{\mu} P'$ then there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $(P',Q') \in R$,*
2. *if $P \overset{t}{\leadsto} P'$ then there exists Q',t' such that $Q \overset{t'}{\leadsto} Q'$ and $(P',Q') \in R$ and $t' \geq t$,*
3. *if $Q \xrightarrow{\mu} Q'$ then there exists P' such that $P \xrightarrow{\mu} P'$ and $(P',Q') \in R$,*
4. *if $Q \overset{t}{\leadsto} Q'$ then there exists P',t' such that $P \overset{t'}{\leadsto} P'$ and $(P',Q') \in R$ and $t' \leq t$.*

The relation R is called s strong pre-order.

Proposition 8.3.2 *If R, R' and R_i for $i \in I$ are all* strong pre-orders *then so are;*

1. *Id_P,*
2. *RR',*
3. *$\bigcup_{i \in I} R_i$.*

Proof:

Parts 1 and 3 are obvious so we will prove part 2.

Given $P\ RR'\ Q$ then there exists S such that $P\ R\ S$ and $S\ R'\ Q$. Since R and R' are both *strong pre-orders* from the definition.

If $P \xrightarrow{\mu} P'$ then there exists S' such that $S \xrightarrow{\mu} S'$ and $P'\ R\ S'$. Similarly from the second equivalence there is a Q' such that $Q \xrightarrow{\mu} Q'$ and $S'\ R'\ Q'$.

Thus for $P \xrightarrow{\mu} P'$ then there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P'\ RR'Q'$.

If $P \overset{t}{\leadsto} P'$ then there exists S' and t' such that $S \overset{t'}{\leadsto} S'$ with $P'\ R\ S'$ and $t' \geq t$. Similarly from the second equivalence there is a Q' and t'' such that $Q \overset{t''}{\leadsto} Q'$ with $S'\ R'\ Q'$ and $t'' \geq t'$.

Thus for $P \overset{t}{\leadsto} P'$ then there exists Q' and t'' such that $Q \xrightarrow{t''} Q'$ with $P'\ RR'\ Q'$ and $t'' \geq t$.

The symmetric cases can be proved similarly. □

Definition 8.3.3 *A functional $\mathcal{F}$ on binary relations $R \subseteq P \times P$ such that $(P,Q) \in \mathcal{F}(R)$ iff for all $\mu \in Act$ and for all time t:*

1. *if $P \xrightarrow{\mu} P'$ then there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $(P',Q') \in R$,*
2. *if $P \overset{t}{\leadsto} P'$ then there exists Q', t' such that $Q \overset{t'}{\leadsto} Q'$ and $(P',Q') \in R$ and $t' \geq t$,*
3. *if $Q \xrightarrow{\mu} Q'$ then there exists P' such that $P \xrightarrow{\mu} P'$ and $(P',Q') \in R$,*
4. *if $Q \overset{t}{\leadsto} Q'$ then there exists P', t' such that $P \overset{t'}{\leadsto} P'$ and $(P',Q') \in R$ and $t' \leq t$.*

Proposition 8.3.4

- *$\mathcal{F}$ is monotonic,*
- *R is a strong pre-order iff $R \subseteq \mathcal{F}(R)$.*

Proof: both parts directly from the definition of $\mathcal{F}$.

Definition 8.3.5 *Call* $\geq \equiv \{ \bigcup R \mid R$ *is a* strong pre-order $\}$.

Proposition 8.3.6 $\geq$ *is the largest strong pre-order.*

Proposition 8.3.7 $\geq$ *is the maximal fixed point of* $\mathcal{F}$*.*

Proof: $\geq$ is a strong pre-order hence $\geq \subseteq \mathcal{F}(\geq)$,
$\mathcal{F}$ is monotonic thus $\mathcal{F}(\geq) \subseteq \mathcal{F}(\mathcal{F}(\geq))$,
(In other words $\mathcal{F}(\geq)$ is a pre-fixed point of $\mathcal{F}$.)
but $\geq$ contains all pre-fixed points of $\mathcal{F}$ hence $\mathcal{F}(\geq) \subseteq \geq$ □

Proposition 8.3.8 Strong Pre-Order *is substitutive with respect to the finite operators of wTCCS. Thus for* $P \geq Q$ *the following hold;*

1. $[t]P \geq [t]Q$,
2. $\mu.P \geq \mu.Q$,
3. $P + E \geq Q + E$,
4. $P \mid E \geq Q \mid E$,
5. $P \backslash L \geq Q \backslash L$,
6. $P[S] \geq Q[S]$.

Proof:(following [Mil6], Proposition 5.17)

Most of the cases are self-evident so we shall provide a proof of one as an example. We shall show that given $P \geq Q$ then $P \mid E \geq Q \mid E$. We will show that the relation $T = \{(P \mid E \ , \ Q \mid E) \mid P \geq Q\}$ is a strong pre-order. Consider firstly the action evolutions of $P \mid E$ so if $P \mid E \xrightarrow{\mu} R$ then there are three cases;

1. $R \equiv P' \mid E$ then $P \xrightarrow{\mu} P'$ and from the definition $Q \xrightarrow{\mu} Q'$ with $P' \geq Q'$ so $Q \mid E \xrightarrow{\mu} Q' \mid E$ with $(P' \mid E, Q' \mid E) \in T$.
2. $R \equiv P \mid E'$ then $Q \mid E \xrightarrow{\mu} Q \mid E'$ with $(P \mid E', Q \mid E') \in T$.
3. $R \equiv P' \mid E'$ and $\mu = \tau$; then there is a μ such that $P \xrightarrow{\mu} P'$ and $E \xrightarrow{\bar{\mu}} E'$ so $Q \xrightarrow{\mu} Q'$ with $P' \geq Q'$ therefore we can infer $Q \mid E \xrightarrow{\tau} Q' \mid E'$ and $(P' \mid E', Q' \mid E') \in T$.

The time evolutions are much simpler since we know that if $P \mid E \overset{t}{\rightsquigarrow} R$ then $R \equiv P' \mid E'$ where $P \overset{t}{\rightsquigarrow} P'$ and $E \overset{t}{\rightsquigarrow} E'$ thus there exists Q' and $t' \geq t$ such that $Q \mid E \overset{t'}{\rightsquigarrow} Q' \mid E''$ with $P' \mid E' \geq Q' \mid E''$, since from $E \overset{t}{\rightsquigarrow} E'$ we can infer $E \overset{t'}{\rightsquigarrow} E''$ with $E' \equiv E''$ from the time evolution rule plus the rule INV.

Unfortunately this pre-order is *not* substitutive in sTCCS, for consider the following processes;

$$P = (5)a \quad Q = (7)a \quad E = (6)b,$$
$$R = P \mid E \text{ and } S = Q \mid E.$$

Clearly $P \geq Q$ but there does not exists a $t \geq 5$ such that $S \overset{t'}{\rightsquigarrow} S'$ with $a \mid (1)b \geq S'$, since we can *never* reach an S' which can perform an a action without performing a b action first. A similar lack of congruence can be observed with respect to the non-determinism operator.

Strong pre-order is not substitutive with respect to sTCCS, owing to the introduction of causality via timing information. In the system sTCCS the following two processes are equivalent,

$$(5)a \mid (7)b \text{ and } (5)a.(2)b.$$

The process on the left has the causality introduced implicitly by timing, while the other process has the causality explicitly introduced by action-prefix. Any interleaving 'faster than' relation will not preserve timing causality, but will preserve structural causality. Thus to obtain a notion of 'faster than' that will be an order for sTCCS, we shall need to distinguish between the *explicit* introduction of causality through structure and the *implicit* causality introduced by timing.

We will not proceed with further study of the strong pre-order. Since we regard the system sTCCS as the more fundamental we will instead attempt to find an equivalence relation which is substitutive for that system.

8.4 An Equality for sTCCS.

Definition 8.4.1 *Processes P and Q are* time-equivalent *iff there exists a relationship R between P and Q such that for all $\mu \in Act$ and for all times t;*

1. *if $P \xrightarrow{\mu} P'$ then there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $(P', Q') \in R$,*
2. *if $P \overset{t}{\leadsto} P'$ then there exists Q' such that $Q \overset{t}{\leadsto} Q'$ and $(P', Q') \in R$,*
3. *if $Q \xrightarrow{\mu} Q'$ then there exists P' such that $P \xrightarrow{\mu} P'$ and $(P', Q') \in R$,*
4. *if $Q \overset{t}{\leadsto} Q'$ then there exists P' such that $P \overset{t}{\leadsto} P'$ and $(P', Q') \in R$,*

the relation R is called a time-equivalence.

Proposition 8.4.2 *If S, S' and S_i for all $i \in I$, are time-equivalences; then the following are also time-equivalences;*

1. *Id_P,*
2. *SS',*
3. *S^{-1},*
4. *$\bigcup_{i \in I} S_i$.*

Proof: A trivial extension to the proof of proposition 8.3.2

Definition 8.4.3 $\sim_T \equiv \bigcup\{S \mid S \textit{ is a time-equivalence}\}$

Proposition 8.4.4

1. $\sim_T$ *is a time-equivalence,*

2. $\sim_T$ *is the largest time-equivalence.*

Proof:

1. directly from the above proposition.
2. immediately from the definition of $\sim_T$.

Definition 8.4.5 *T is a time-equivalence up to $\sim_T$ whenever $\sim_T T \sim_T$ is a time-equivalence.*

Proposition 8.4.6 *If T is a* time-equivalence up to $\sim_T$ *then* $T \subseteq \sim_T$.

Proof: for $(P, Q) \in T$ we have $P \sim_T PTQ \sim_T Q$ and thus $P \sim_T T \sim_T Q$.

The usual functional defintion in the style of [Par] can be used to demonstrate time equivalence correct; as the details are precisely those of the earlier definition and proof we omit them.

Proposition 8.4.7 *Time-equivalence is substitutive for the finite operators of sTCCS. In other words, given $P \sim_T Q$ then;*

1. $\mu.P \sim_T \mu.Q$,

2. $(t)P \sim_T (t)Q$,

3. $\delta P \sim_T \delta Q$,

4. $P + E \sim_T Q + E$,

5. $P \mid E \sim_T Q \mid E$,

6. $P\backslash L \sim_T Q\backslash L$,

7. $P[S] \sim_T Q[S]$.

In order to prove the above we need to demonstrate that appropriate bisimulations can be found. The detail has been presented earlier and is therefore omitted.

Definition 8.4.8 *Let E and F be two expressions, with free variables $\tilde{X}$. Then we will say that $E \sim_T F$ iff for all vectors of processes $\tilde{P}$*

$$E[\tilde{P}/\tilde{X}] \sim_T F[\tilde{P}/\tilde{X}]$$

Proposition 8.4.9 *given $E \sim_T F$ then $Fix_{\tilde{X}}E \sim_T Fix_{\tilde{X}}F$.*

Proof: consider only a single pair of equations:

$$A \equiv E[A/X] \text{ and } B \equiv F[B/X]$$

with $E \sim_T F$. We will show that the relation $\{(G[A/X], G[B/X]) \mid G \text{ contains at most } X \text{ free }\}$ is a time-equivalence up to $\sim_T$. We proceed by induction over the depth of inference by which either $G[A/X] \xrightarrow{\mu} P'$ or $G[A/X] \overset{t}{\rightsquigarrow} P''$ is inferred. Then we continue by case analysis over the structure of G. The resulting proof mirrors that of [Mil1,Mil6]; but we have to perform each case both for action and temporal evolutions. For example consider the case for the parallel composition operator. Suppose $G \equiv G_1 \mid G_2$ then

- if $G_1[A/X] \xrightarrow{\mu} G_1'[A/X]$ then $G[A/X] \xrightarrow{\mu} G_1'[A/X] \mid G_2[A/X]$ but from the inductive assumption $G_1[A/X] \sim_T G_1[B/X]$ as there actions are derived by a shorter inference; as we can then immediately obtain the desired result.

- Similarly for $G_2[A/X] \xrightarrow{\mu} G_2'[A/X]$.
- if $G_1[A/X] \mid G_2[A/X] \xrightarrow{\tau} G_1'[A/X] \mid G_2'[A/X]$ then there is a λ such that

  $$G_1[A/X] \xrightarrow{\lambda} P_1' \text{ and } G_2[A/X] \xrightarrow{\bar{\lambda}} P_2'.$$

  So from the inductive assumption we can find:

  $$G_1[B/X] \xrightarrow{\lambda} Q_1' \text{ and } G_2[B/X] \xrightarrow{\bar{\lambda}} Q_2';$$
  $$\text{with } P_1' \sim_T T \sim_T Q_1' \text{ and } P_2' \sim_T T \sim_T Q_2'.$$

  Take $P' = P_1' \mid P_2'$ and $Q' = Q_1' \mid Q_2'$ then $P_i = H_i[A/X]$ for some H and similarly $Q_i = H_i[B/X]$. But $(P_i, Q_i) \in \sim_T T \sim_T$ so letting $H = H_1 \mid H_2$ we have $(P', Q') \equiv (H[A/X], H[B/X]) \in T$ and hence

  $$G[A/X] \xrightarrow{\tau} P' \text{ and } G[B/X] \xrightarrow{\tau} Q'$$
  $$\text{with } (P', Q') \in T.$$

- $G_1[A/X] \mid G_2[B/X] \overset{t}{\rightsquigarrow} P' \mid Q'$ this is the same as the above case but with time actions replacing the normal actions.

The other structural cases follow the same pattern.

8.5 Temporal Structure independence

We wish to equate processes where performing the same action costs the same amount of time. Whilst the strong equivalence appears to do this, it is only natural if we assume that all of the time cost of an event is located either immediately before it or immediately after it, not if it has a cost distributed either side of it. Consider the following two processes;

$$P \equiv (1)a.(5)R,$$
$$Q \equiv (3)a.(3)R,$$

where R is an arbitary process. If we take the time actions above as the cost of performing the a action in both cases, then these processes should be considered equivalent in that sense, but they are certainly not equivalent in the strong sense.

Essentially we wish to identify processes where we can find time slices between which they are always capable of the same computation, but not necessarily at precisely the same time. We would like to base this notion of equivalence on an order. We start by abstracting actions to remove the precise location with respect to time and replace it with a notion of associated cost in time.

Proposition 8.5.1 *Let P be a process in wTCCS, or sTCCS over a dense time (ie, the reals or the rationals). If $P \overset{t}{\leadsto} P'$ then there exists P'' such that $P \overset{t_1}{\leadsto} P''$ and $P'' \overset{t_2}{\leadsto} P'$, with $t_1 + t_2 = t$.*

> **Proof:** Immediate from the nature of the TIME rule in both the case of sTCCS and wTCCS.

Definition 8.5.2 *For any process P, $P \xrightarrow[t]{\mu} Q$ iff one of the following holds*

1. *there exists P' and P'' such that :*

   (a) $P \overset{t_1}{\leadsto} P'$,

   (b) $P' \xrightarrow{\mu} P''$,

   (c) $P'' \overset{t_2}{\leadsto} Q$,

   (d) $t = t_1 + t_2$.

2. *There exists P' such that:*

   (a) $P \overset{t}{\leadsto} P'$,

   (b) $P' \xrightarrow{\mu} Q$.

3. *There exists P' such that:*

   (a) $P \xrightarrow{\mu} P'$,

   (b) $P' \overset{t}{\leadsto} Q$.

4. *Failing the above.*

(a) $P \xrightarrow{\mu} Q$,

(b) $t = 0$.

Now we can define a temporal simulation. The evolution defined above is an abstraction from the underlying timed system; a similar method is used in [Smo].

Definition 8.5.3 *We will say P is faster than Q iff there exists a relation T, called a* Temporal Simulation, *such that for all $\mu \in Act$ and for all times t;*

1. *if $P \xrightarrow[t]{\mu} P'$ then there exists Q', t' such that $Q \xrightarrow[t']{\mu} Q'$ and $(P', Q') \in T$ with $t' \geq t$,*

2. *if $Q \xrightarrow[t]{\mu} Q'$ then there exists P', t' such that $P \xrightarrow[t']{\mu} P'$ and $(P', Q') \in T$ with $t' \leq t$.*

Unfortunately, owing to the different nature of TCCS[Tof] and both sTCCS and wTCCS this is not the same as the form of temporal simulation as defined in [Tof]. However it does have the property of equating the processes P and Q defined earlier.

Proposition 8.5.4 *If T, T' and T_i for all $i \in I$, are temporal simulations then the following are all temporal simulations;*

1. Id_P,

2. TT',

3. $\bigcup_{i \in I} T_i$.

Definition 8.5.5 $\geq_T = \bigcup\{T \mid T \text{ is a temporal simulation}\}$.

Proposition 8.5.6

- $\geq_T$ *is a temporal simulation,*
- $\geq_T$ *is the largest temporal simulation.*

Once again the above can be demonstrated correct by a functional definition in the style of [Par].

Proposition 8.5.7 *If $P \geq_T Q$ then for finite wTCCS processes P and Q then;*

- $[t]P \geq_T [t']Q$ *with* $t' \geq t$,
- $P + E \geq_T Q + E$,
- $P \mid E \geq_T Q \mid E$,
- $P[S] \geq_T Q[S]$,
- $P \backslash L \geq_T Q \backslash L$.

Unfortunately the problems with implicit causality prevent this order being substitutive for sTCCS. We do not include a proof of the above proposition since it is in essence identical to that given earlier.

8.6 Equational Characterisation of sTCCS.

Consider the following equations.

Given $P \sim_T Q$ and $t_1 \leq t_2$ then the following equations are true of $\sim_T$:

1. Action Prefix;

   (a) $\mu.P = \mu.Q$,

   (b) $(t)P = (t)Q$,

   (c) $(t_1)(t_2)P = (t_1 + t_2)P$,

   (d) $\delta P = \delta Q$,

2. Non-determinism;

   (a) $P + P = P$,

   (b) $P + \delta Nil = P$,

   (c) $(t)P + Nil = Nil$,

   (d) $P + R = R + P$,

   (e) $(t_1)P + (t_2)Q = (t_1)(P + (t_2 - t_1)Q)$,

   (f) $(t)P + (t)Q = (t)(P + Q)$,

   (g) $P + (R + S) = (P + R) + S$,

   (h) $(t_1)\delta P + (t_2)\delta P = (t_1)\delta P$,

   (i) $\delta a.P + a.P = a.P$,

   (j) $\delta P + \delta R = \delta(P + R)$,

   (k) $(t_1)a.P + (t_2)R = (t_1)a.P$,

   (l) $a.P + \delta R + (t)S = a.P + \delta R$,

3. Composition;

   (a) $P \mid \delta Nil = P$,

   (b) $(t)P \mid Nil = Nil$,

   (c) $P \mid R = R \mid P$,

   (d) $P \mid (R \mid S) = (P \mid R) \mid S$,

   (e) $(t_1)P \mid (t_2)Q = (t_1)(P \mid (t_2 - t_1)Q)$,

   (f) $(t)P \mid (t)Q = (t)(P \mid Q)$,

4. Restriction;

   (a) $a.P\backslash L = Nil$ if a, $\overline{a} \in L$,

   (b) $a.(P\backslash L) = (a.P)\backslash L$ if a, $\overline{a} \notin L$,

(c) $(t)(P\backslash L) = ((t)P)\backslash L$,

(d) $\delta(P\backslash L) = (\delta P)\backslash L$,

(e) $P = P\backslash L$ if for all $a \in \mathcal{L}(P)$ $a \notin L$,

(f) $P\backslash L_1\backslash L_2 = P\backslash L_1 \cup L_2$,

(g) $P\backslash L + Q\backslash L = (P + Q)\backslash L$,

(h) $(P\backslash L) \mid Q = (P \mid Q)\backslash L$, if for all $a \in \mathcal{L}(Q)$ a, $\overline{a} \notin L$,

We have not given rules for wTCCS since we cannot obtain the same structural identities. But there is no manipulation possible of the temporal operators owing to the property that processes of the form

$$[2][4]P$$

deadlock. In other words, in

$$R \equiv [2]a.P \mid [3]Q$$

if we tried to replace this process by the obvious,

$$S \equiv [2](a.P \mid [1]Q);$$

then this would temporally deadlock in the context $S\backslash a$ whereas $R\backslash a$ does not. Since further passage of time will permit the process Q to evolve. In the strong system however both the processes;

$$R' = (2)a.P \mid (3)Q \text{ and } S' = (2)(a.P \mid (1)Q),$$

deadlock in the context $\backslash a$. Thus it is possible to manipulate the time action prefixes in the strong system. A similar property ensures that we cannot distribute weak time over a non-deterministic pair of processes.

Most of the above equations come directly from CCS and from the natural properties of time. The non-determinism equations come from the unwillingness of a sTCCS process to delay when it can perform a normal action. As in, for instance, equation 2.k where we are stating that the process will immediately evolve either through the action prefix or by performing the delayed alternative at once.

There is no general equation analogous to the expansion theorem, so this equational system is probably not complete. This results from the problems of introducing unintentional causality. We cannot yet give a simple normal form.

8.7 Observational Evolutions.

We can define the usual observational notions of evolution.

Definition 8.7.1 *For* $s = \mu_1 \ldots \mu_n \in Act^*$ *we say that,* $P \xrightarrow[t]{s} P'$ *iff* $P \xrightarrow[t_1]{\mu_1} \ldots \xrightarrow[t_n]{\mu_n} P'$ *with* $t = t_1 + \ldots + t_n$.

Definition 8.7.2 $P \overset{s}{\underset{t}{\Longrightarrow}} P'$ *iff* $P \xrightarrow[t]{s_0} P'$ *for some* $\hat{s_0} = s$

Time pressure means that we have not looked at the equivalence induced on processes by this evolution, but we suspect that we will obtain results on wTCCS which mirror those for CCS. However since we cannot yet provide a substitutive order for sTCCS, we do not suspect that an order induced by observational evolutions will be substitutive.

8.8 Process Logic For Timed CCS.

We introduce a simple extension of the process logic $\mathcal{PL}$, with a timed modal operator. Since our space of times may be dense, it is not sufficient to add a next operator which is interpreted as at the next instant the proposition holds. A similar logic is presented in [Koy2]. The formulae of our logic are defined as follows:

$$F ::= \bigwedge_{i \in I} F_i \mid \neg F \mid < a > F \mid \{t\}F.$$

Definition 8.8.1 *The satisfaction relation between processes and formulae is defined as follows; $P \models_T F$ iff:*

- *$P \models_T \bigwedge_{i \in I} F_i$ iff for all $i \in I$, $P \models_T F_i$,*
- *$P \models_T \neg F$ iff $P \models_T F$ is* false,
- *$P \models_T < a > F$ iff there exits P' such that $P \xrightarrow{a} P'$ and $P' \models_T F$,*
- *$P \models_T \{t\}F$ iff either*
  - *there exists a, P' such that $P \xrightarrow{a} P'$ and $P \models_T F$,*
  - *or there exists P', t' such that $P \overset{t'}{\rightsquigarrow} P'$ with $t' \leq t$ and $P' \models_T F$.*

The idea behind the time modality is that we can reach a place that can satisfy the remainder of the formula in no more time than that given. So we are either able to satisfy the remiander of the formula now, or can reach some future point *sufficiently rapidly* were the remiander can be satisified. This avoids prolems with 0 time actions.

Proposition 8.8.2 *If $P \models_T \{t\}F$ then for all $t' \geq t$, $P \models_T \{t'\}F$.*

Proof: Immediate from the definition of the operator $\{t\}$.

We believe it is possible to show that if two processes of sTCCS satisfy all the same temporal formula to the same depth, then they are temporally equivalent up to the same depth, by a proof essentially identical to that in [Mill]. Note: since zero times are not permitted in sTCCS the first clause of the temporal modality only applies to delays and these are matched over the equivalence.

To handle observational congruences we define an extended operator, for $s \in \Lambda^*$ and $t \in Times$, $[[s,t]]_T^*$, in the following way;

Definition 8.8.3 *$P \models [[s,t]]^*_T F$ iff there exists $\{t_0\}[s_1]\{t_1\}...[s_n]\{t_n\}$ such that $P \models \{t_0\}[s_1]\{t_1\}...[s_n]\{t_n\}F$ and $t_0 + t_1 + ... + t_n \leq t$.*

Definition 8.8.4 *The observational version of the modal operators, by using the hat operator,*

$$P \models_T [[\hat{s},t]]_T F \text{ iff } P \models_T [[s,t]]^*_T F.$$

Note: we can add quantifiers and implications in the same manner as for $\mathcal{PL}$.

8.9 Examples.

8.9.1 A Simple Timer.

Consider the example of the timed and the timer controlled actions presented in [Tof]. We wish to produce a sequence of actions seperated by fixed times. In one case we use internal timing, in the other we sperate the functions into a clock and a producer.

$$\begin{aligned} E_a &\equiv a.[t]E_a \\ Timer &\equiv \overline{alarm}.[t']Timer \\ E'_a &\equiv alarm.[t_1]a.[0].E'_a \\ R_a &\equiv (Timer \mid E'_a)\backslash alarm \end{aligned}$$

We consider the evolutions of the two systems with $t_1 < t'$ and $t_2 = t' - t_1$:

$$\begin{gathered} E_a \xrightarrow{a}\overset{t}{\rightsquigarrow} E_a, \\ \text{thus } E_a \overset{a}{\underset{t}{\Longrightarrow}} E_a, \\ R_a \xrightarrow{\tau}\overset{t_1}{\rightsquigarrow}\xrightarrow{a}\overset{t_2}{\rightsquigarrow} R_a, \\ \text{thus } R_a \overset{a}{\underset{t'}{\Longrightarrow}} R_a. \end{gathered}$$

Note, these are minimum time paths.

It seems that the time of the process R_a is independent of the value of t_1 provided it remains less than t'. Possibly we could achieve an analogous result to that of weakness replacement [Tof]. The following example shows that in these systems weakness replacement will hold only when processes behaviours are much more constrained than the requirements of the original result.

Consider the following processes;

$$\begin{aligned} P &= \bar{b}.[5]P_1, \\ Q &= \bar{b}.[10]Q_1, \\ E &= b.[20]Nil, \\ R_1 &= E \parallel P, \\ R_2 &= E \parallel Q. \end{aligned}$$

Even if P_1 and Q_1 are identical then R_1 will not be the same as R_2. The former can evolve to a state equivalent to P_1, twice as fast as the latter. Thus the condition for the processes to be made identical by communicating with a slower process must include repeated communication.

8.9.2 Action Available For a Period.

In the following process:

$$(5)\delta a.P + (7)a.P$$

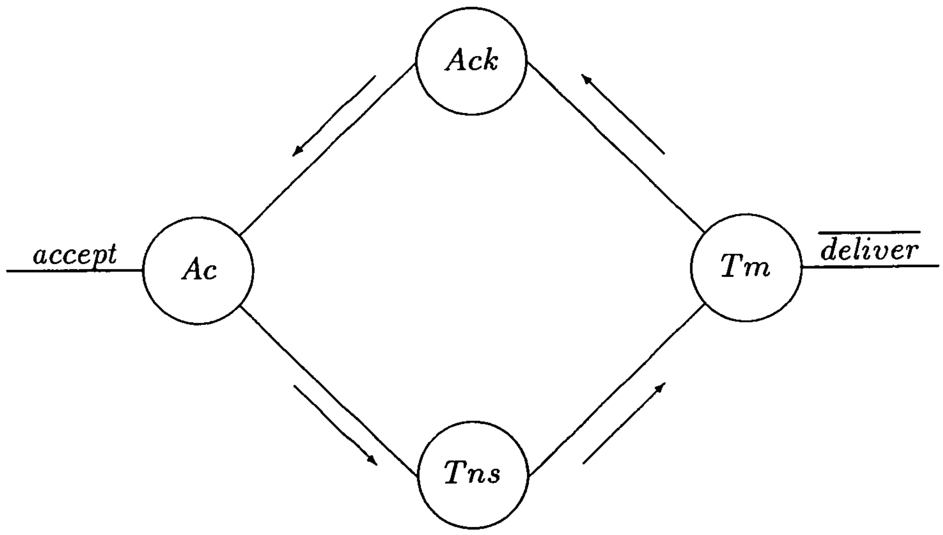
the action a can be inferred at any time between 5 and 7, but it must have been used by at latest 7 or the process will deadlock. This can be used to represent a process that requires a certain time to start and is then only available for a limited period.

8.10 Value Passing.

If we use the operators of CCS with values and either set of temporal operators then we can derive a value passing calculus from our basic calculus in an identical manner to that used for CCS. The translation was presented in chapter 1.

8.11 The Alternating Bit Protocol.

In his book [Mil6] Milner presents an implementation of the alternating bit protocol in CCS, and demonstrates that the protocol is correct. Perforce this implementation ignores the exact temporal properties of the system and its components. We extend the implementation in CCS to one in sTCCS, where we can take advantage of the temporal properties of the system components. In this version some of the system complexity can be reduced by exploiting the temporal information. An alternating bit protocol realisation can be viewed as follows. (We let $\tilde{b}$ stand for the negation of the boolean value b.)



The process Ac will work in the following manner. After accepting a message, it sends it with bit b along the channel Tns and waits, subsequently there are three possibilities:

- it times out, and re-Transmits the message;
- it gets an acknowledgement b from the Ack line (correct transmission), so it can now accept another message;
- it gets an acknowledgement $\tilde{b}$ (superfluous extra acknowledgement of earlier message) which is ignored.

The replier Tm works in a dual manner. After a message is delivered it sends an acknowledgement with bit b along the Ack line. There are then three possibilities:

- it times out, and re-transmits the acknowledgement;
- it gets a new message with bit $\tilde{b}$ from the Tns line, which it delivers and acknowledges with bit $\tilde{b}$;
- it gets a repetition of the old message with bit b which is ignored.

The channels in the implementation are identical and have the ability to duplicate or lose an arbitrary message, arbitrarily many times. For convenience we ignore actual messages and concentrate on the value of the control bits.

We now give the definition of a timed version of the alternating bit protocol in the following fashion. There are 4 fundamental times involved, the transmission times on both channels and the re-try times in both senders.

$$Ac(b) = \delta ack(b).Ac(b) + \delta ack(\tilde{b}).Ac(b) + \delta accept.Sd(\tilde{b})$$
$$Sd(b) = \overline{send}(b).Sd_1(b) + \delta ack(b).Sd(b) + \delta ack(\tilde{b}).Ac(\tilde{b})$$
$$Sd_1(b) = (t_{rt})\overline{send}(b).Sd_1(b) + \delta ack(b).(t_a)Sd_1(b) + \delta ack(\tilde{b}).Ac(\tilde{b})$$

$$Tm(b) = \delta transmit(b).Tm(b) + \delta transmit(\tilde{b}).Tm(b) + \overline{deliver}.Rp(b)$$

$$Rp(b) = \overline{reply}(b).Rp_1(b) + \delta transmit(\tilde{b}).Rp(b) + \delta transmit(b).Tm(\tilde{b})$$
$$Rp_1(b) = (t_{rt}).\overline{reply}(b).Rp_1(b) + \delta transmit(\tilde{b})(t_t).Rp_1(b) + \delta transmit(b).Tm(\tilde{b})$$

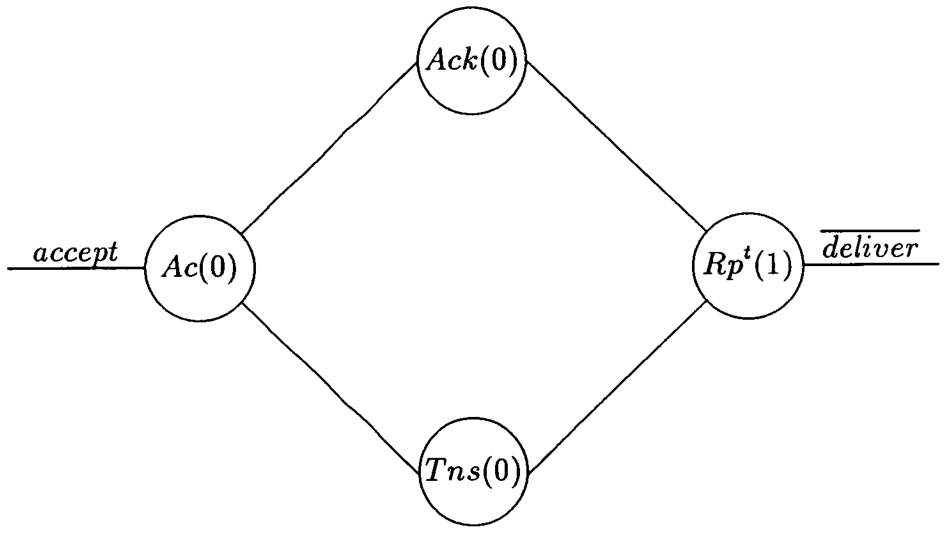
$$\text{Let } s = s_1 b_2 s_2$$
$$Tns(b_1 s_1 b_2 s_2) = \delta send(b).Tns(bb_1 s) + (t_s)\overline{Transmit}(b_1).Tns(s) + \delta\tau.Tns(b_1 s_1 s_2) + \delta\tau.Tns(s)$$

$$Ack(b_1 s_1 b_2 s_2) = \delta reply(b).Ack(bb_1 s) + (t_s)\overline{ack}(b).Ack(s) + \delta\tau.Ack(b_1 s_1 s_2) + \delta\tau.Ack(s)$$

With initial state (for some $t < t_{rt}$),

$$Ab \equiv Ac(0) \parallel Ack(0) \parallel Tns(0) \parallel Rp^t(1)$$

A static flow diagram for the above is



We firstly impose the restriction that t_a and t_t are less than t_s; this is sufficient to ensure that the *ack* and *transmit* ports are always available to the transmission channels. We impose the condition on the re-try rate that $t_{rt} > 2t_s$. This seems reasonable as we cannot be sure that a transmission has failed until this period of time has elapsed. The external environment is forced to take delivery as soon as possible. This enables us to calculate the period of time between the reception of an *accept* action, and the return of the process to a position where the next transmission may be attempted.

From the constraint on re-tries we can observe that there is no path with a time short enough to require that the transmission channel should contain more than one data item at once. We redefine our channels as follows.

$$Ack = \delta reply(b).Ack(b)$$
$$Ack(b) = (t_s)\overline{ack}.Ack + \delta\tau.Ack$$

$$Tns = \delta send(b).Tns(b)$$
$$Tns(b) = (t_s)\overline{transmit}.Tns + \delta\tau.Tns$$

With initial state,

$$Ab = Ac(0) \mid\mid Ack \mid\mid Tns \mid\mid Rp_1^{t_s}(1).$$

It should be noted that we have arranged that at no time is an immediate *ack* or *transmit* action of either type impossible, apart for a period immediately after the reception of such an action, and then the evolution is possible before any more actions of that type can be produced by the channels. Thus we can use actions with no delay guarding in the channels to force the evolution to proceed at certain intervals.

Let us examine the evolution of the process *Ab*. (Note that we will not divide time actions below the minimum to permit another physical evolution. We will include a path only if two immediate evolutions lead eventually to the same state. For clarity we will subscript the τ actions with the initial letter of the original action from which they where inferred. We will mark points at which genuine divergence can take place with a number and the evolution will be reconsidered from there later.)

$$
\begin{aligned}
& Ac(0) \parallel Ack \parallel Tns \parallel Rp^{t_s}(1) \\
& \xrightarrow{accept} Sd(1) \parallel Ack \parallel Tns \parallel Rp_1^{t_s}(1) \\
& \xrightarrow{\tau_s} Sd_1(1) \parallel Ack \parallel Tns(1) \parallel Rp_1^{t_s}(1) \\
& \overset{t_s}{\leadsto} Sd_1^{t_s}(1) \parallel Ack \parallel transmit1.Tns \parallel Rp_1^{2t_s}(1) && (1) \\
& \xrightarrow{\tau_t} Sd_1^{t_s}(1) \parallel Ack \parallel Tns \parallel Tm(0) \\
& \xrightarrow{deliver} Sd_1^{t_s}(1) \parallel Ack \parallel Tns \parallel Rp(0) \\
& \xrightarrow{\tau_a} Sd_1^{t_s}(1) \parallel Ack(0) \parallel Tns \parallel Rp_1(0) \\
& \overset{t_s}{\leadsto} Sd_1^{2t_s}(1) \parallel ack0.Ack \parallel Tns \parallel Rp_1^{t_s}(0) && (2) \\
& \xrightarrow{\tau_a} Ac(1) \parallel Ack \parallel Tns \parallel Rp_1^{t_s}(0)
\end{aligned}
$$

This is the inversion of the initial state so their has been a correct transmission in time $2t_s$, if there are no errors, as we would expect. We now consider the evolution if there is an error on the first transmission (which we labelled (1) above) some period of time t, after the transmission was started. Note that t must be less than or equal to our transmission time t_s.

$$
\begin{aligned}
& Sd_1(1) \parallel Ack \parallel Tns(1) \parallel Rp_1^{t_s}(1) \\
& \overset{t}{\leadsto} Sd_1^{t}(1) \parallel Ack \parallel Tns \parallel Rp_1^{t_s+t}(1) \\
& \overset{t_{rt}-t_s-t}{\leadsto} Sd_1^{t_{rt}-t_s}(1) \parallel Ack \parallel Tns \parallel Rp(1) && (*) \\
& \xrightarrow{\tau_r} Sd^{t_{rt}-t_s}(1) \parallel Ack(1) \parallel Tns \parallel Rp_1(1) \\
& \overset{t_s}{\leadsto} Sd_1^{t_{rt}} \parallel ack1.Ack \parallel Tns \parallel Rp_1^{t_s}(1)
\end{aligned}
$$

There are now two possible evolution paths, one leading to the re-try as desired the other requiring a further complete cycle before the re-try can even be attempted. If the following alternative is chosen,

$$
\begin{aligned}
& \xrightarrow{\tau_a} Sd_1(1) \parallel Ack \parallel Tns \parallel Rp_1^{t_s}(1) \\
& \overset{t_{rt}-t_s}{\leadsto} Sd_1^{t_{rt}-t_s} \parallel Ack \parallel Tns \parallel Rp(1)
\end{aligned}
$$

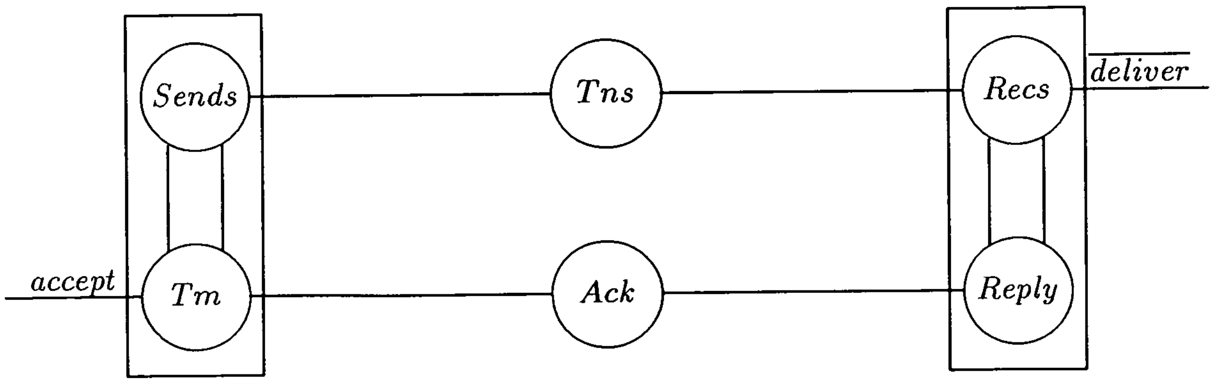
This is the same as state (*) and thus has the same behaviour. Thus we have an implementation of the protocol which has a non-linear response to error: after one error we have a potential computation which will *never* correctly transmit. This can be corrected by separating the activities of re-trying, and responding to enquiries. The next version of the alternating-bit protocol is produced along these lines, and has a linear response to errors.

$$\begin{gathered}
Tmc(b) = \delta ack(b).Tmc(b) + \delta ack(\tilde{b}).Tmc(b) + accept.\overline{start_s}(b).Tms(\tilde{b}) \\
Tms(b) = \delta ack(b).Tms(b) + \delta ack(\tilde{b}).\overline{stop_s}.\ Tmc(\tilde{b}) \\
Sends = \delta start_s(b).\overline{send}(b).Send(b) + \delta stop_s.Sends \\
Send(b) = (t_{rt})\overline{send}(b).Send(b) + \delta stop_s.Sends \\
Recc(b) = \delta transmit(b).Recc(b) + \delta transmit(\tilde{b})Recc(b) \\
+\overline{deliver}.\overline{start_r}(b).Recs(b) \\
Recs(b) = \delta transmit(\tilde{b}).Recs(b) + \delta transmit(b).\ \overline{stop_r}.Recc(\tilde{b}) \\
Reply = \delta start_r(b).\overline{reply}(b).Replys(b) + \delta stop_r.Reply \\
Replys = (t_{rt})\overline{reply}(b) + \delta stop_r.Reply
\end{gathered}$$

With the restriction that $t_{rt} > 2t_s$ we can re-use our earlier channels and the complete protocol's initial state is;

$$Tmc(0) \mid\mid Sends \mid\mid Tns \mid\mid Ack \mid\mid Recs(1) \mid\mid Reply^{t_s}(1).$$

A static flow diagram for the above is.



It is relatively easy to check that the time performance of this process is linearly affected by the number of errors in transmission.

8.12 Conclusions.

We have presented a temporal model which is an extension of CCS. This system has most of the properties we desire, excepting that we have so far not demonstrated an order over processes. We believe that such an order may be obtained using the techniques of [Cas]. The temporal behaviour of the systems sTCCS and wTCCS has little effect on the operational behaviour derived from CCS, and we believe that the methodology of separating action and temporal evolutions can successfully extend any underlying operational reasoning system for computation actions.

Chapter 9

Timing Concurrent While Programs.

9.1 Introduction.

Often the reason for introducing parallelism into programs is that their sequential version does not execute with sufficient speed. Since we are not interested in fast programs that will not work, we should like to obtain a methodology within which we can produce both time information and correctness proofs. In her thesis [Nie] Nielson presented methods for deducing the run time of sequential programs using an extension of Hoare's logic. We will extend these methods to concurrent programs, and demonstrate that the reasoning system we produce is sound with respect to an interpretation over a timed semantics, which we will construct in the timed calculus sTCCS presented in the last chapter. Throughout we will be working with statements of total correctness and therefore some constructions will not work for non-terminating program fragments.

9.2 Timed Semantics for CIMP.

We will put all the time/cost of a computation into the variable access. This is both for simplicity, and from the expectation that most of the cost of a computation is incurred in moving values to and from memory. Assuming that all variable read/writes take the same unit amount of time, we translate variables as instances of the following process:

$$
\begin{aligned}
[\!\lhd X \rhd\!] &= \delta a_X x.\delta\overline{r}x.\delta Nil \\[1em]
[\!\lhd F(E_1,\ldots E_n)\rhd\!] &= (([\!\lhd E_1\rhd\!]\ result\ \delta\rho x.\overline{r_1}x.\overline{b_1}.\delta Nil) \mid \\
&\quad (\delta b_1.[\!\lhd E_2\rhd\!]\ result\ \delta\rho x.\overline{r_2}x.\overline{b_2}.\delta Nil) \mid \ldots \mid \\
&\quad (\delta b_{n-1}.[\!\lhd E_n\rhd\!][r_n/r]) \mid \\
&\quad \delta r_1 x_1 \ldots \delta r_n x_n.\delta Nil\ \overline{r}_{F(x_1,\ldots,x_n)}.\delta Nil) \\
&\quad \backslash\{r_1,\ldots,r_n,b_1,\ldots,b_{n-1}\}
\end{aligned}
$$

Figure 9–1: Timed Expression evaluation for CIMP.

$$
\begin{aligned}
Loc_X &= \delta g_X x.(1)V_X(x) \\
V_X(x) &= \delta\overline{a}_X x.(1)V_X(x) + \delta g_X y.(1)V_X(y)
\end{aligned}
$$

We will work with only a finite number of variables to allow the formation of closures. We will say that a variable, X contains the value y when the process representing it is in the state $V_X(y)$.

Definition 9.2.1 *The set* $L_{X_i} = \{a_{X_i}, \overline{g}_{X_i}\}$.

9.2.1 Expression Translation.

The translation of expressions is given in figure 9-1. Unlike the earlier process based semantics we have not assumed that we have an arbitrary number of processing sites available to perform an expression evaluation. Thus the semantics have been altered to limit ourselves to effectively one processing site for any evaluation. This will keep a more accurate account of how much computing resources we are consuming.

Definition 9.2.2 *The following derived operator of sTCCS is required for the definition of expression evaluation,*

$$B_1\ result\ B_2 = (B_1 \mid B_2)\backslash\{r\}.$$

$[\lhd X := E \rhd] = [\lhd E \rhd]\ \mathit{result}\ (\delta rx.\delta\overline{g}_X x.\mathit{done})$

$[\lhd C_1 ; C_2 \rhd] = [\lhd C_1 \rhd]\ \mathit{before}\ [\lhd C_2 \rhd]$

$[\lhd \mathit{IF}\ \ E\ \ \mathit{THEN}\ \ C_1\ \ \mathit{ELSE}\ \ C_2 \rhd] = [\lhd E \rhd]\ \mathit{result}\ \delta rx.(\mathit{if}\ x\ \mathit{then} [\lhd C_1 \rhd]\ \mathit{else}\ [\lhd C_2 \rhd])$

$[\lhd \mathit{WHILE}\ \ E\ \ \mathit{DO}\ \ C \rhd] =$
$W = [\lhd E \rhd]\ \mathit{result}\ \delta rx.(\mathit{if}\ x\ \mathit{then} [\lhd C \rhd]\ \mathit{before}\ W\ \mathit{else}\ \mathit{done})$ with W new.

$[\lhd C_1\ \ \mathit{PAR}\ \ C_2 \rhd] = [\lhd C_1 \rhd]\ \mathit{par}\ [\lhd C_2 \rhd]$

$[\lhd \mathit{SKIP} \rhd] = \mathit{done}$

$[\lhd \mathit{LET}\ \ X := E\ \ \mathit{IN}\ \ C \rhd] = [\lhd E \rhd]\ \mathit{result}\ ((\delta rx.\delta\overline{g}_X x.\mathit{done} \mid \mathit{Loc}_X)\ \mathit{before}\ [\lhd C \rhd])\backslash L_X$

Figure 9–2: Command translation for CIMP.

9.2.2 Command Translation.

Definition 9.2.3 *We need the following derived operators of sTCCS.*

- $\mathit{done} = \overline{d}.\delta Nil,$
- $B_1\ \mathit{before}\ B_2 = (B_1[b/d] \mid \delta b.B_2)\backslash\{b\}$ *with b new,*
- $B_1\ \mathit{par}\ B_2 = (B_1[d_1/d] \mid B_2[d_2/d] \mid \delta d_1.\delta d_2.\delta\overline{d}.\mathit{done})\backslash\{d_1, d_2\}$

The processes which represent commands are given in figure 9-2.

Proposition 9.2.4 *If C is a terminating command, then $[\lhd C \rhd] \overset{s\overline{d}}{\underset{t}{\Longrightarrow}} \delta Nil$, with s composed only of τ, read and write actions.*

Proof: By structural induction over commands and induction over the length of the action string s.

From the above for any terminating command C then $[\lhd C \rhd] \models_T [[s\overline{d}, t]]_T \mathit{true}$ where t is any time and s is composed solely of read or write actions. This means that we have no time actions in our processes $[\!\lhd C \rhd\!]$; so we shall work with closures of commands.

Definition 9.2.5 *The temporal closure of a command C written $< C >_T$ is the following process;*

$$\begin{gathered}< C >_T = (get(\tilde{x}).\overline{g}_{X_1}x_1 \dots \overline{g}_{X_n}x_n.done \mid [\!\lhd C \rhd\!] \mid Loc_{X_1} \mid \dots \\ \mid Loc_{X_n} \mid \delta d.a_{X_1}y_1.\dots\delta a_{X_n}y_n.\ \overline{put}(\tilde{y}).done)\backslash\ \{d\} \cup L_{X_1} \cup \dots \cup L_{X_n}\end{gathered}$$

Unfortunately we have added an overhead of 1 into the time taken for a command to execute. In the sense that if we examined how long the process $[\!\lhd C \rhd\!]$ took to execute to completion in the presence of variables we would discover that it can complete in 1 unit of time less than $< C >_T$ can.

Definition 9.2.6 *Two commands C_1 and C_2 are* totally independent *iff*

$$(\mathcal{R}(C_1) \cup \mathcal{W}(C_1)) \cap (\mathcal{R}(C_2) \cup \mathcal{W}(C_2))) = \emptyset$$

Definition 9.2.7 *A program is totally independent iff the only PAR commands used in its construction are* totally independent. *We call this sub-language of DCIMP, TIMP.*

Definition 9.2.8 *The time an expression or a terminating command C of TIMP takes to execute, written $dur(C)$, is defined recursively as follows, for any terminating C.*

- $dur(X) = 1$,
- $dur(F(E_1, \dots E_n)) = dur(E_1) + \dots + dur(E_n)$,
- $dur(SKIP) = 0$,
- $dur(X := E) = 1 + dur(E)$,

- $dur(C_1; C_2) = dur(C_1) + dur(C_2)$,
- $dur(C_1 \ \ PAR \ \ C_2) = max(dur(C_1), dur(C_2))$ *given* C_1 *and* C_2 *are totally independent,*
- $dur(IF \ \ E \ \ THEN \ \ C_1 \ \ ELSE \ \ C_2) = dur(E) + if \ E \ then \ dur(C_1) \ else \ dur(C_2)$,
- $dur(WHILE \ \ E \ \ DO \ \ C) =$ *for some integer* n[1] , $n * (dur(E) + dur(C)) + dur(E)$,
- $dur(LET \ \ X := E \ \ IN \ \ C) = dur(E) + dur(C) + 1$.

We need that commands in a parallel composition are totally independent, to guarantee the time taken when they execute in parallel. If they are not totally independent, then they may attempt to simultaneously access the same variable, at which point one of the components will have to wait for a period of time 1 to elapse. Then we shall be unsure of the performance of the composition. For example consider

$$Y := X \ \ PAR \ \ W := X$$

this will take 2 time periods to execute, not the 1 which we would hope. Even more unfortunately,

$$Z := X + Y \ \ PAR \ \ W := X + T$$

will take time 3 to execute, whilst

$$Z := X + Y \ \ PAR \ \ W := T + X$$

will only take time 2 to execute. In order to avoid this indeterminacy in the execution time, we do not allow such programs. Furthermore we may conclude,

[1]since we are only dealing with terminating loops we must execute the body C an integer number of times, and evaluate the expression one more time than that

that if the programs are not totally independent, then they may not truly execute in parallel with respect to time. We are only interested in performance we can guarantee; in these cases we may or may not be able to execute in parallel and hence we can make little accurate prediction of execution time.

Proposition 9.2.9 *If C is a command in TIMP then*

$$< C >_T \models_T [[get(\tilde{x})\overline{put}(\tilde{y}), t]]_T true$$

with $t = dur(C) + 1$.

Proof: We proceed by induction over the structure of commands, relying on the fact that there is a direct correspondence between the time that $[\!\lhd C \rhd\!]$ takes to execute (in the presence of variables), and the time $< C >_T$ takes to execute; as we have not provided a recursive definition of $< C >_T$. The analysis is straightforward and follows almost immediately from the definition of the semantics of commands. We will prove three of the cases.

- Assignment, $X := E$;

  $$\begin{aligned}< X := E >_T = &(get(\tilde{x}).\overline{g}_{X_1} x_1 \ldots\ \overline{g}_{X_n} x_n.done \mid [\!\lhd X := E \rhd\!] \mid \\ &Loc_{X_1} \mid \ldots \mid Loc_{X_n} \mid \delta d.a_{X_1} y_1 \ldots . \delta a_{X_n} y_n . \\ &\overline{put}(\tilde{y}).done) \backslash \{L_{X_1}, \ldots, L_{X_n}\}.\end{aligned}$$

  After the action $get(\tilde{x})$ has been performed, and all the assignments made, there is a period of time of length 1 when no further activity can take place, as each variable process is in a time guarded state. The expression E is then evaluated. This takes a time $dur(E)$, proved by a trivial induction on the structure of E. After the assignment is made there is a period of time 1 when all the variable values can not be read, in order to construct the value vector $\tilde{y}$ as the variable X is in a time guarded state. The action $\overline{put}(\tilde{y})$ can only take place, after at least period of time of length $dur(E) + 1$ has expired. Moreover all the actions involved in the computation of E were silent actions. Thus

$$< X := E >_T \models_T [[get(\tilde{x})\overline{put}(\tilde{y}), dur(X := E) + 1]]_T true.$$

- Sequential composition; $< C_1; C_2 >_T$ is not expressed directly in terms of $< C_1 >_T$ and $< C_2 >_T$ so our result will not follow immediately from the semantical construction. Let us assume that;

  $$< C_1 >_T \models_T [[get(\tilde{x}_1)\overline{put}(\tilde{y}_1), dur(C_1) + 1]]_T true \text{ and}$$
  $$< C_2 >_T \models_T [[get(\tilde{x}_2)\overline{put}(\tilde{y}_2), dur(C_2) + 1]]_T true.$$

  Both of these expressions were formed using $[\triangleleft C_1 \triangleright]$ and $[\triangleleft C_2 \triangleright]$, as is $< C_1; C_2 >$, and from their respective constructions we know that C_1 takes $dur(C_1)$ to execute in the presence of variables and that C_2 takes $dur(C_2)$ to execute in similar circumstances. The construction $[\triangleleft C_1; C_2 \triangleright]$ forces them to execute one after the other; and thus in the presence of variables must take at least a period $dur(C_1) + dur(C_2)$ to execute. So we can deduce that

  $$< C_1; C_2 >_T \models_T [[get(\tilde{x})\overline{put}(\tilde{y}), dur(C_1; C_2) + 1]]_T true.$$

- Totally Independent Parallel composition; the argument is similar to that for sequence. From the independence of the two commands we can see that their mutual presence does not affect the time that each will take to execute (as that is solely determined by variable access). Thus if totally independent C_1 and C_2 take times t_1 and t_2 respectively to execute, with variables present, then (from construction) C_1 *PAR* C_2 will take at least $max(t_1, t_2)$ to execute. Hence

  $$< C_1 \ PAR \ C_2 >_T \models_T [[get(\tilde{x})\overline{put}(\tilde{y}), dur(C_1 \ PAR \ C_2) + 1]]_T true.$$

$\square$

9.3 Timed Hoare's Logic for TIMP.

We extend the triples of Hoare's logic with a predicate which records the time taken to execute a command and the variables present in that command.

Definition 9.3.1 *Given a predicate P then $FV(P)$ is the set of variables occurring free in P.*

Definition 9.3.2 *Given predicates P, Q over program variables and R over program variables and $\underline{time}$, we write*

$$[P]\, C : R\, [Q]_{\tilde{X}}$$

to mean that if the command C starts from a state where P holds then it terminates, and for the state in which it terminates Q will hold. Moreover it will reach that state in at most time given in R. This expression is well formed iff;

- $FV(P) \subseteq \tilde{X}$,
- $FV(Q) \subseteq \tilde{X}$,
- $FV(R) \subseteq \tilde{X} \cup \{\underline{time}\}$,
- $\mathcal{R}(C) \cup \mathcal{W}(C) \subseteq \tilde{X}$.

Note, in the logical system of predicates we require one property of time formulae which is that for $t' \geq t$,

$$R[t/\underline{time}] \models R[t'/\underline{time}].$$

The following abbreviations are useful in presenting the proof system.

Definition 9.3.3 *Given two time formulae R and R', the formula $R \oplus R'$ is an abbreviation for*

$$\exists time_1 . \exists time_2 . \underline{time} = time_1 + time_2 \wedge R[time_1/\underline{time}] \wedge R'[time_1/\underline{time}]$$

Definition 9.3.4 *Given a formula R and a pure formula Q. The formula $Q^{\cdot}R$ is an abbreviation for:*

$$\exists \tilde{y}. Q[\tilde{y}/\tilde{X}] \wedge R[\tilde{y}/\tilde{X}]$$

where $\tilde{X}$ is the set of program variables in R and Q.

Definition 9.3.5 *For any expression E, $dur^+(E)$ is the predicate $\underline{time} = dur(E)$.*

Definition 9.3.6 *The predicate $I_{\tilde{X}}$ is the identity predicate on all the variables in the set $\tilde{X}$.*

We extend Nielson's proof system $\mathcal{R}$ to $\mathcal{R}_{||}$ given in figure 9-3 over the language TIMP.

Definition 9.3.7 *The expression $[P]\ C : R\ [Q]_{\tilde{X}}$ is* valid *iff*

$$< C >_T \models_T P[\tilde{y}/\tilde{X}] \wedge [[get(\tilde{y})\overline{put}(\tilde{z}), t+1]]_T true \supset Q[\tilde{z}/\tilde{X}] \wedge R[\tilde{z}/\tilde{X}, t/\underline{time}].$$

One of the effects of using closures is to greatly simplify the formula we need to express validity over processes.

Proposition 9.3.8 *The proof system $\mathcal{R}_{||}$ is sound with respect to the notion of validity given above.*

Proof:

1. Assignment:

$$\overline{[P[E/X]]\ X := E : \underline{time} = dur(X := E)\ [P]_{\tilde{X}}}$$

Given $P[E/X][\tilde{y}/\tilde{X}]$ holds, and from Proposition 9.2.9

$< X := E >_T \models [[get(\tilde{y}), \overline{put}(\tilde{z}), t+1]]_T$ and

Assignment: $$\frac{}{[P[E/X]]\ X := E : \underline{time} = dur(X := E)\ [P]_{\tilde{X}}}$$

Null: $$\frac{}{[P]\ SKIP : \underline{time} = 0\ [P]_{\tilde{X}}}$$

Sequence: $$\frac{[P]\ C_1 : R_1\ [S]_{\tilde{X}} \quad [S]\ C_2 : R_2\ [Q]_{\tilde{X}}}{[P]\ C_1; C_2 : R_1 \oplus (S \cdot R_2)\ [Q]_{\tilde{X}}}$$

Alternation: $$\frac{[P \wedge E]\ C_1 : R\ [Q]_{\tilde{X}} \quad [P \wedge \neg E]\ C_2 : R\ [Q]_{\tilde{X}}}{[P]\ IF\ \ E\ \ THEN\ \ C_1\ \ ELSE\ \ C_2 : R\ [Q]_{\tilde{X}}}$$

Parallel: $$\frac{[P_1]\ C_1 : R_1 \wedge \underline{time} = t_1\ [Q_1]_{\tilde{X}} \quad [P_2]\ C_2 : R_2 \wedge \underline{time} = t_2\ [Q_2]_{\tilde{Y}} \quad \tilde{X} \cap \tilde{Y} = \emptyset}{[P_1 \wedge P_2]\ C_1\ \ PAR\ \ C_2 : R_1 \wedge R_2 \wedge \underline{time} = max(t_1, t_2)\ [Q_1 \wedge Q_2]_{\tilde{X} \cup \tilde{Y}}}$$

Let: $$\frac{[P \wedge X = E]\ C : R\ [Q]_{\tilde{Y} \cup \{X\}} \quad X \notin FV(P) \cup FV(Q)}{[P \wedge (X = F)]\ LET\ \ X := E\ \ IN\ \ C : R \oplus dur^{+}(E)\ [Q \wedge (X = F)]_{\tilde{Y}}}$$

Iteration: $$\frac{\begin{array}{l}[P(z+1) \wedge E]\ C : R'\ [P(z) \wedge Q']_{\tilde{X} \cup \{z\}} \qquad P(0) \models \neg E \\ P(z) \wedge \neg E \wedge I_{\tilde{X}} \models Q \quad Q' \cdot Q \models Q \\ P(z) \wedge \neg E \wedge dur^{+}(E) \models R \quad dur^{+}(E) \oplus R \oplus (Q' \cdot R') \models R\end{array}}{[\exists z. P(z)]\ WHILE\ \ E\ \ DO\ \ C : R\ [Q]_{\tilde{X}}} \text{ with } z \notin \tilde{X}.$$

Consequence: $$\frac{P \models P' [P']\ C : R'\ [Q']_{\tilde{Y}} \quad R \models R' \quad Q' \models Q \quad \tilde{Y} \subseteq \tilde{X}}{[P]\ C : R\ [Q]_{\tilde{X}}}$$

Invariance: $$\frac{[P]\ C : R\ [Q]_{\tilde{X}}}{[P]\ C : R \wedge P\ [Q]_{\tilde{X}}}$$

Figure 9–3: Timed proof system $\mathcal{R}_{||}$ for TIMP.

$$\tilde{z}_i = \begin{cases} \tilde{y}_i \; i \neq X \\ E \text{ otherwise.} \end{cases}$$

and hence $P[\tilde{z}/\tilde{X}]$ holds. Furthermore $t = dur(X := E)$ and thus $\underline{time} = t = dur(X := E)$ holds.

2. Null:

$$\frac{}{[P] \; SKIP : \underline{time} = 0 \; [P]_{\tilde{X}}}$$

follows immediately.

3. Sequence:

$$\frac{[P] \; C_1 : R_1 \; [S]_{\tilde{X}} \quad [S] \; C_2 : R_2 \; [Q]_{\tilde{X}}}{[P] \; C_1; C_2 : R_1 \oplus (S^{\cdot} R_2) \; [Q]_{\tilde{X}}}$$

we wish to show that this preserves validity. So let us assume:

$$< C_1 >_T \models_T P[\tilde{y_1}/\tilde{X}] \wedge \; [[get(\tilde{y_1}), \overline{put}(\tilde{z}_1), t_1 + 1]]_T true \supset S[\tilde{z}_1/\tilde{X}] \wedge$$
$$R_1[\tilde{y_1}/\tilde{X}, t_1/\underline{time}] \text{ and}$$

$$< C_2 >_T \models_T P[\tilde{y_2}/\tilde{X}] \wedge \; [[get(\tilde{y_2}), \overline{put}(\tilde{z}_2), t_2 + 1]]_T true \supset S[\tilde{z}_2/\tilde{X}] \wedge$$
$$R_1[\tilde{y_2}/\tilde{X}, t_2/\underline{time}].$$

Unfortunately $< C_1; C_2 >_T$ is not expressed in terms of $< C_1 >_T$ and $< C_2 >_T$ but in terms of $[\triangleleft C_1; C_2 \triangleright]$. However, as $< C_1 >_T$ and $< C_2 >_T$ are expressed in terms of $[\triangleleft C_1 \triangleright]$ and $[\triangleleft C_2 \triangleright]$ respectively we can still proceed, by observing that:

$$\begin{aligned}
< C_1, C_2 >_T =& (get(\tilde{x}).\overline{g}_{X_1}x_1 \dots \overline{g}_{X_n}x_n.done \\
& \mid \llbracket C_1; C_2 \rrbracket \mid Loc_{X_1} \mid \dots \mid Loc_{X_n} \mid \\
& \delta d.a_{X_1}y_1. \dots \delta a_{X_n}y_n. \\
& \overline{put}(\tilde{y}).done)\backslash\{L_{X_1}, \dots, L_{X_n}\} \\
=& (get(\tilde{x}).\overline{g}_{X_1}x_1 \dots \overline{g}_{X_n}x_n.done \\
& \mid \llbracket C_1 \rrbracket \; before \; \llbracket C_1 \rrbracket \\
& \mid Loc_{X_1} \mid \dots \mid Loc_{X_n} \mid \\
& \delta d.a_{X_1}y_1. \dots \delta a_{X_n}y_n. \\
& \overline{put}(\tilde{y}).done)\backslash\{L_{X_1}, \dots, L_{X_n}\} \\
=& (get(\tilde{x}).\overline{g}_{X_1}x_1 \dots \overline{g}_{X_n}x_n.done \\
& \mid (\llbracket C_1 \rrbracket[b/d] \mid b.\llbracket C_2 \rrbracket)\backslash\{b\} \\
& \mid Loc_{X_1} \mid \dots \mid Loc_{X_n} \mid \\
& \delta d.a_{X_1}y_1. \dots \delta a_{X_n}y_n. \\
& \overline{put}(\tilde{y}).done)\backslash\{L_{X_1}, \dots, L_{X_n}\}
\end{aligned}$$

Now for a process to perform $\overline{put}(\tilde{z})$, the variables $X_1, \dots, X_N$ must have values $z_1, \dots, z_n$ at the termination of that process. If we start C_1 with P holding of its input value vector, then it will output $\overline{put}(\tilde{z_1})$ with S holding of that vector of values. Thus at the point $\llbracket C_2 \rrbracket$ performs b in the combined process; the processes $Var_1, \dots, Var_n$ must contain the values $z_{1_1}, \dots, z_{1_n}$.

Similarly after a process closure has performed the action $get(\tilde{y})$, the variable processes must have the values $y_1, \dots, y_n$ respectively. In the composite process we start with P holding of the input values. We know that, from the left hand premise, S holds of the values when $\llbracket C_1 \rrbracket$ finishes (i.e performs the hidden action b). From the right hand premise, if S holds of the input values of C_2, then Q will hold for the values output upon termination of $C_1; C_2$.

From Proposition 9.2.9

$$< C_1; C_2 >_T \models_T [[get(\tilde{y}), \overline{put}(\tilde{z}), dur(C_1; C_2) + 1]]_T true.$$

We have already identified a state where $(S^{\cdot} R_2)$ holds; and we know $R_1 \oplus (S^{\cdot} R_2)$ holds, since the time taken by $C_1; C_2$ is precisely the sum of the times for C_1 and C_2 separately; and that in the closure we get a overhead of 1. So we can deduce that

$$< C_1; C_2 >_T \models_T P[\tilde{y}/\tilde{X}] \wedge \ [[get(\tilde{y}), \overline{put}(\tilde{z}), t+1]]_T true \supset S[\tilde{z}/\tilde{X}] \wedge \\ R_1 \oplus (R_2\text{;}S)[\tilde{y}/\tilde{X}, t/\underline{time}]$$

as required.

4. Alternation: by an argument essentially identical to sequence.

5. Parallel:

$$\frac{[P_1]\, C_1 : R_1 \wedge \underline{time} = t_1\ [Q_1]_{\tilde{X}} \quad [P_2]\, C_2 : R_2 \wedge \underline{time} = t_2\ [Q_2]_{\tilde{Y}} \quad \tilde{X} \cup \tilde{Y} = \emptyset}{[P_1 \wedge P_2]\, C_1 \ \ PAR \ \ C_2 : R_1 \wedge R_2 \wedge \underline{time} = max(t_1, t_2)\ [Q_1 \wedge Q_2]_{\tilde{X} \cup \tilde{Y}}}$$

The proof that this inference rule preserves validity starts from similar observations to those made in the proof of sequence (part 3 above).

$$\begin{aligned} < C_1 \ \ PAR \ \ C_2 >_T =& (get(\tilde{x}).\overline{g}_{X_1} x_1 \dots \overline{g}_{X_n} x_n.done \\ & \mid [\!\lhd C_1 \ \ PAR \ \ C_2 \rhd\!] \\ & \mid Loc_{X_1} \mid \dots \mid Loc_{X_n} \mid \\ & \delta d.a_{X_1} y_1 . \dots \delta a_{X_n} y_n . \\ & \overline{put}(\tilde{y}).done) \backslash \{L_{X_1}, \dots, L_{X_n}\} \\ =& (get(\tilde{x}).\overline{g}_{X_1} x_1 \dots \overline{g}_{X_n} x_n.done \\ & \mid [\!\lhd C_1 \rhd\!] \ par \ [\!\lhd C_2 \rhd\!] \\ & \mid Loc_{X_1} \mid \dots \mid Loc_{X_n} \mid \\ & \delta d.a_{X_1} y_1 . \dots \delta a_{X_n} y_n . \\ & \overline{put}(\tilde{y}).done) \backslash \{L_{X_1}, \dots, L_{X_n}\} \end{aligned}$$

Since the commands C_1 and C_2 are totally independent, we can observe that given memory values which satisfy $P_1 \wedge P_2$ the execution of $[\!\lhd C_1 \rhd\!]$ (in the composition) will leave Q_1 holding of the variables it accesses; and the execution of $[\!\lhd C_2 \rhd\!]$ will leave Q_2 holding of the variables it accesses. Thus when we come to perform the output $\overline{put}(\tilde{z})$ in the above, $Q_1 \wedge Q_2$ must hold of those values.

With respect to time; R_1 and R_2 hold whenever $P_1 \wedge P_2$ holds; because they have no common variables and $\underline{time} = max(t_1, t_2)$ holds since (from Proposition 9.2.9),

$$< C_1 \ \ PAR \ \ C_2 >_T \models_T [[get(\tilde{y}), \overline{put}(\tilde{z}), t+1]]_T true$$

with $t = dur(C_1 \;\; PAR \;\; C_2) = max(dur(C_1), dur(C_2))$ as required.

6. Let: a simple extension of the proof of sequence suffices.

7. Iteration:

$$\frac{\begin{array}{l}[P(z+1) \wedge E]\, C : R' \,[P(z) \wedge Q']_{\tilde{X}\cup\{z\}} \qquad P(0) \models \neg E \\ P(z) \wedge \neg E \wedge I_{\tilde{X}} \models Q \qquad Q' Q \models Q \\ P(z) \wedge \neg E \wedge dur^{+}(E) \models R \quad dur^{+}(E) \oplus R \oplus (Q' R') \models R\end{array}}{[\exists z . P(z)]\; WHILE \;\; E \;\; DO \;\; C : R\, [Q]_{\tilde{X}}} \text{ with } z \notin \tilde{X}.$$

We proceed by induction and shall show that; if $P(z)[k/z]$ holds for some k with some set s of variable values then Q and R will hold of the output values.

- $n = 0$ then $P(0)$ holds and hence $\neg E$ so we want to show that

$$\begin{aligned}< WHILE \;\; E \;\; DO \;\; C >_T \models_T P(0)[\tilde{y}/\tilde{X}] \wedge \\ [[get(\tilde{y}_1), \overline{put}(\tilde{z}), t+1]]_T true \supset \\ S[\tilde{z}/\tilde{X}] \wedge \; R[\tilde{y}/\tilde{X}, t/\underline{time}]\end{aligned}$$

  but in this case t is precisely $dur(E)$ and from $P(0) \wedge \neg E \wedge I_{\tilde{X}} \models Q$ we have that $Q[\tilde{z}/\tilde{X}]$ holds; furthermore $P(0) \wedge \neg E \wedge dur(E) \models R$ implies that $R[\tilde{z}/\tilde{X}, t/\underline{time}]$ holds.

- Assume that the rule is sound for $P(z)[k/z]$ we shall prove it sound for $k+1$. Given $P(k+1)$ holds of the current values in the variables. If $\neg E$ holds we proceed as before and get soundness trivially. So we assume that is not the case. From

$$[P(z+1) \wedge b]\, C : R' \,[P(z) \wedge Q]_{\tilde{X}}$$

  we have at some point a collection of variable values where $P(z) \wedge Q$ will hold. We know that state can be reached from the initial state, in a time satisfying $dur^{+}(E) \oplus R'$, but the inductive hypothesis states

$$[P(k)]\; WHILE \;\; E \;\; DO \;\; C : R\, [Q]_{\tilde{X}}$$

  and the process $[\triangleleft WHILE \;\; E \;\; DO \;\; C \triangleright]$ in this case is precisely

$$IF \;\; E \;\; THEN \;\; C; WHILE \;\; E \;\; DO \;\; C \;\; ELSE \;\; SKIP.$$

  From the proof of alternation this costs

$$dur^{+}(E) \oplus R' \oplus (R' Q')$$

but $dur^{+}(E) \oplus R' \oplus (R^{\cdot}Q') \models R$ from the assumptions thus

$$[P(k+1)]\ \mathit{WHILE}\ \ E\ \ \mathit{DO}\ \ C : R\,[Q]_{\tilde{X}}\ .$$

8. Consequence: follows immediately from logical considerations.

9. Invariance: as consequence.

□

9.4 Conclusions.

The process based semantics has been extended to include a notion of time and thereby permit reasoning about exact complexity. We have demonstrated a proof system for reasoning over this extended semantics. Unfortunately we do not yet have a completeness proof for this system. If we were to add an *AWAIT THEN* command and work with atomic assignments in the style of Owicki-Gries this could probably be achieved.

The time in this semantics was defined by putting duration into the variable access. We can choose many different notions of duration by placing the time dependencies in other structures within the semantics. This requires that the definition of function $dur(C)$ must be changed but the rest of the proofs/reasoning system should remain unchanged.

Chapter 10

Conclusions and Further Work.

10.1 Conclusions.

Whilst we can define language syntax that admits general parallelism, we may not obtain programs that are comprehensible, or genuinely execute in parallel. Our study of the language CIMP has shown that an unconstrained parallel construct allows the introduction of non-determinism. Although this is a useful technique for modelling concurrency or giving specifications, it is a very unpleasant property for a program to have. In programming terms we gain little from the presence of non-determinism, and pay a considerable price in clarity.

The temporal properties of the execution of a program are harder to reason about. But if we assume that physical limitations prevent us from accessing the same memory location more than once within some arbitrary period of time, then we can demonstrate that in general programs will not execute in parallel: they become sequentialised by the exclusions caused by variable accesses. These effects jointly lead us to conclude that, to program effectively in a shared variable language like CIMP, we must work with some restricted form of parallel composition which avoids the problems of non-determinism and sequentialisation. This restricted language we called TIMP.

The restriction applied to the parallel construction is syntactic, therefore it can be implemented simply within any compiler/interpreter as an extra syntactical requirement on the language for parallel commands to be well formed. To the programmer these restrictions may seem to complicate the task of program construction, (We have a constructor that does not admit all well formed commands as arguments, and this generally leads to an increase in syntactical mistakes by programmers), but the resulting determinism will greatly simplify program correction.

The use of the independence condition as a programming discipline is not particularly onerous, and is in fact a property that programmers are aware of as they are producing concurrent programs. The author knows of no programming application of internal non-determinism, i.e. the undirected choice of a computation path that will change the outcome of the program execution. So restricting to a deterministic concurrent programming language retains all the properties we desire.

Deterministic programs are easy to understand. It is hard for a programmer to detect all the possible execution paths of a concurrent program when it is written. Moreover since the execution of a program is implementation dependent, a non-deterministic program may execute correctly under one implementation, whilst executing incorrectly under another. The respective implementations may always choose different execution paths for the same concurrent commands. However, a deterministic program will retain correctness when moved between correct implementations of the underlying programming language. Determinism in concurrent programming is very similar to side effect freedom in sequential programming.

The semantics of deterministic programs are straightforward. The complexity of the semantics of general concurrent programs is a direct result of the presence of non-determinism it is not a result of the attempt to compute concurrently. In the absence of non-determinism programs have a simple functional nature and are therefore much easier to comprehend.

The knowledge that a program is deterministic greatly simplifies the task of proving that it has certain behaviours. The determinism condition has also been shown to be closely related to a very simple discipline for the construction of correct concurrent programs.

We can translate sequential programs into deterministic concurrent programs without affecting program behaviour. As almost all programs have been and are still written for sequential execution, without some automatic methods for the transformation of sequential programs into concurrent programs the availability of concurrent, and hence fast, programs will be limited.

The calculus sTCCS, could prove the basis for the exploration of the effects of temporal knowledge on the derivable behaviours of concurrent processes. For a sequential process the order of events is all that is important; in a concurrent system the order relative to time is important. Consider for example the following;

$$((4)a.P \mid (6)b.Q \mid \delta\overline{a}.\delta\overline{b}.\delta)\backslash\{a,b\}$$

which can evolve to $(P \mid Q)\backslash\{a,b\}$ whilst,

$$((4)a.P \mid (2)b.Q \mid \delta\overline{a}.\delta\overline{b}.\delta)\backslash\{a,b\}$$

cannot evolve at all after a time 2 has passed. Yet all that was altered between the two processes was the relative time at which the actions a and b can occur; we introduced no other dependency between them. The study of such properties of timed processes should prove rich.

10.2 Further Work.

10.2.1 Expressive Confluent Operators.

We have a collection of confluence preserving operators, but we cannot yet show that every confluent process can be constructed from either the confluent operators or the semi-confluent operators. The notion of primality [Mol] can be used to produce a structure preserving normal form for CCS processes. We can show that the primes forming the decomposition for any finite CCS process from a derivation closed class of processes are also in that class. To show that a set of operators is expressive for that class, we have to demonstrate that all the primes of that class are constructible using the operators, and that there is a parallel composition equivalent to general parallel composition for that class of processes.

In the case of confluence, the confluent operators can build any confluent prime, but the confluent parallel composition has not been demonstrated general enough.

10.2.2 Extending the Language CIMP.

For practical programming purposes the language CIMP is far too restrictive. We should like to extend it towards a more practical programming language, and obtain results similar to those of this thesis in the presence of the extensions. The natural extensions are.

1. Input and Output: these additions should not prove too difficult and will probably admit a simple syntactic determinism condition.

2. Arrays and Loops: as loops are a restricted class of *WHILE DO* commands their addition should not prove problematic. We can define processes to represent arrays in the same manner as variables, but a syntactic condition to ensure determinacy on arrays will prove far too restrictive. Many programs (especially on vector processing systems) can be organised to only access disjoint parts of an array in each of the separate parallel components. To take account of such methods either we must require that the program should be re-written, or we must use a semantical definition of determinism.

3. Procedures and Functions: Milner [Mil1,Mil6] has produced process representations of such constructs, but we have not yet studied the effect of their presence on determinism and other properties of programs.

It may also be possible to duplicate the work on CIMP simply, for a simple channel based language. Unfortunately similar work [Zwi] on such languages suggests that any determinism condition which is effective will be semantical in nature.

10.2.3 An Order for sTCCS.

In [Cas] a non-interleaving approach to CCS is presented. This is achieved by recording in the evolution of a process not only the new state and the action leading to it, but also what part of a process evolved to produce the new state. Derivations of the following style are used;

$$\frac{P \xrightarrow{a} < P', P'' >}{P \mid Q \xrightarrow{a} < P', P'' \mid Q >}$$

and a (distributed) bisimulation defined over such evolutions. The application of the above technique may enable us to obtain a substitutive order for sTCCS processes. The need for such an order is clear; where timing is concerned we usually wish to show that a solution can execute with greater rapidity than a specification requires.

The production of such an order will permit the study of a much greater range of examples, and in particular, should enable a reasonable notion of observational timed order to be defined. There are many time critical systems in existence. The study and evaluation of such systems may lead to greater ease of construction and, perhaps, a greater degree of reliability and predictability.

10.2.4 Adding Probabilities to sTCCS.

In [Smo] a probabilistic extension to the calculus SCCS is presented. A similar extension to sTCCS may enable us to consider temporal, probabilistic and computational behaviour simultaneously. Such a calculus should have the expressive power to represent any quantum mechanical system with a finite number of eigen-states. The notion of communication which underlies CCS is very similar to the notion of measurement which underlies quantum mechanics, since communication fixes a state much in the same manner as measurement. This implies that a sufficiently expressive version of CCS may prove a useful calculus for the study of quantum systems.

10.2.5 Automatic Transformation Tools.

The conditions we required to be able to transform a sequential command into a parallel command within CIMP, are simple enough to admit automatic execution.

It is not yet clear whether this will remain so when the language CIMP is extended to a more useful programming language. It should however be, possible to automate some of the task of identifying which components can execute in parallel, without affecting program behaviour, and hopefully checking the correctness of such constructions should always be possible. Such a tool could greatly simplify the production of concurrent programs.

10.2.6 General Transformation to Parallelism.

We should like to obtain conditions that a transformation of a pair of sequential commands to a parallel command is correct, within any fully abstract semantics over a concurrent programming language. In other words, (since the semantics is fully abstract) if $[\![C]\!] = [\![C']\!]$ then we can replace C by C' in any context so, what condition do we require on C_1 and C_2 such that,

$$[\![C_1; C_2]\!] = [\![C_1 \ \ PAR \ \ C_2]\!].$$

Given that within the semantics assignment is atomic then the condition that;

$$[\![C_1; C_2]\!] = [\![C_2; C_1]\!]$$

is sufficient to ensure that the replacement of the sequential commands by a parallel command is valid. Unfortunately for a fully interleaving model this condition does not suffice; consider,

X:=X+1;X:=X+1

this is order independent, when executed sequentially, but is non-deterministic when executed in parallel. It may be the case that the conditions

- $[\![C_1; C_2]\!] = [\![C_2; C_1]\!]$ and
- $\mathcal{W}(C_1) \cap \mathcal{W}(C_2) = \emptyset$,

suffice, but this has not yet been proved.

The production of a general semantical condition is challenging, but it is the condition we would ultimately hope to have.

10.2.7 Expression Evaluation.

Most of the execution time of an imperative program is taken in evaluating expressions. Perhaps a fruitful approach to increasing the efficiency of concurrent programs is to attempt to distribute expression evaluation efficiently. In the semantics of CIMP presented in this thesis, we assumed in one case an arbitrary number of processing sites, and in the other we used 2. A study of efficient expression evaluation on an arbitrary *fixed* number of processing sites may lead to a simple method of automating some concurrency in imperative programs.

10.2.8 Functional Programming.

A natural extension of the work done so far, and an attempt to solve the problem stated above, is to examine the execution of functional programs in a concurrent manner. Simple functional programming languages are deterministic, even when executed in parallel; languages like ML [Har] will not be. A study of determinism and efficiency for concurrent functional programming languages with references and higher order types should be, in many respects, more useful than similar results on imperative languages. The abstraction of functional programming, allows the implementor to hide all the complexity of concurrency, whilst allowing the programmer to concentrate on obtaining a correct programming solution.

Chapter 11

Bibliography.

[Ada] The programming language Ada reference manual, LNCS 106, 1981.

[Apt] K. Apt, Ten Years of Hoare's Logic: A Survey, ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, October 1981, pp 431-483.

[Bak] J. de Bakker and J. Zucker, Processes and the Denotatiotal Semantics of Concurrency, Infomation and Control, 54 pp 70-120, 1982.

[Ber] J. A. Bergestra, J. W. Klop, Algebra of Communicating Processes with Abstraction, Theoretical Computer Science, Volume 37, No 1, 1985.

[Car] L. Cardelli, Real Time Agents, Proc. 9th ICALP LNCS 140.

[Cas] I. Castellani and M. Hennessey, Distributed Bisimulation, Report Sussex University 5/87, July 1987.

[Cha] K. M. Chandy and J. Misra, Parallel Program Design, Mcgraw Hill, 1988.

[Chu] A. Church A. and J. B. Rosser, Some properties of conversion, Trans AMS 39 1936 pp 472-482.

[Coo] S. A. Cook, Soundness and completeness of an axiom system for program verification, SIAM J. Computing 7, pp 70-90, 1978.

[Dar] J. Darlington, Program Transformation: an Introduction and Survey, Computer Bulletin, 22-24, 1979.

[Dir] P.A.M. Dirac, Principles of Quantum Mechanics, 4 th. Edition Oxford 1958.

[For] Vs FORTRAN version 2, programming guide release 2, order no SC26-4222-2, 1987

[Hal] B. Hailpern, Temporal Logic, Springer LNCS 129.

[Har] R. Harper, Introduction to Standard ML, LFCS-86-14, University of Edinburgh.

[Hart] A. C. Hartman, A Concurrent Pascal Compiler for Minicomputers, Springer-Verlag, 1977.

[Hen1] M. Hennessey and G. D. Plotkin, Full Abstraction for Simple Parallel Programming Language, LNCS 74, 1979, pp 108-120.

[Hen2] M. Hennessey, Algebraic theory of Processes, M.I.T. press 1988.

[Hoa1] A. Hoare, Communicating Sequential Processes, Prentice Hall '85.

[Hoa2] A. Hoare, An axiomatic basis for computer programming, Comm. Acm 12 1969 pp 576-580, 583.

[Hue] G.Huet, Confluent reductions; abstract properties and applications to term rewriting systems. JACM 27 80 pp 797-821.

[Jon] C.B. Jones, Specification and Design of (Parallel) Programs, Proc. IFIP 9th World Computer Congress, North Holland, Pages 321-332, 1983.

[Jef] A. Jeffrey, Synchronous CSP, Oxford University, to appear.

[Koy1] R. Koymans, J. Vytopil, W.P. de Roever, Real-Time and Asynchronous Message Passing, Technical Report, RUU-CS-83-9, University of Eindhoven,1983.

[Koy2] R. Koymans, Specifying Message Passing and Real-Time Systems with Real-Time Temporal Logic, Technical report, University of Eindhoven, 1987.

[Mil1] R. Milner, A Calculus of Communicating Systems, Springer LNCS vol 92.

[Mil2] R. Milner, Calculi for Synchrony and Asynchrony, Theoretical Computer Science 25(3), pp 267-310, 1983.

[Mil3] R. Milner, A Calculus For Concurrency, To appear in the Handbook of Theoretical Computer Science.

[Mil4] R. Milner, Personal communication.

[Mil5] R. Milner, Interpreting One Concurrent Calculus in Another,

[Mil6] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[Mol] F. Moller, Axioms for Concurrency, PhD. Thesis 1989, University of Edinburgh.

[Nie] H. Nielson, Hoare Logic's for Run Time Analysis of Programs, Thesis Edinburgh University '84.

[Occ] The Occam language reference manual Prentice Hall, 1984.

[Owi] S. Owicki and D. Gries, An Axiomatic Proof Technique for Parallel programs I, Acta Informatica 6:1, Pages 319-340, 1976.

[Par] D. Park, Concurrency and Automata on infinite sequences, Springer LNCS 104.

[Plo1] G. D. Plotkin, A structured approach to operational semantics. Technical report Daimi Fn-19, Computer Science Department, Aarhus University. 1981

[Plo2] G. D. Plotkin, A powerdomain for Countable Nondeterminism, Proc 9th ICALP, LNCS 140, 1982.

[Rei] W. Reisig, Pertri Nets, an Introduction, Springer-Verlag 1985.

[Rem] M. Rem, Concurrent computations and VLSI circuits pp 399-438 in Control flow and Data flow. Ed. M. Broy Springer-Verlag 1986.

[Ros1] G. M. Reed and A. W. Roscoe, A Timed Model for CSP, LNCS 226: ICALP 1986.

[Ros2] Roscoe A.W.& Hoare C.A.R., The laws of OCCAM programming.

[San] M.T. Sanderson, Proof techniques for CCS. PhD. Thesis Edinburgh 1982.

[Sch] D. Schmidt, Denotational Semantics. Allyn and Bacon, 1986

[Shi] L. I. Schiff, Quantum Mechanics, 3 rd. edition, Mcgraw Hill 1982.

[Smo] S. Smolka, B. Steffen and C. Tofts, Full Abstraction in a Calculus of Relatively Frequent Processes, to appear.

[Son] M. Sonnenselien, An extension to the language C for concurrent programming. Parallel Programming, vol 2. 1986,

[Sti1] C.S. Stirling, A Generalisation of Owicki-Gries' Hoare's Logic for a Concurrent While Language, Journal of Theoretical Computer Science, 1988.

[Sti2] C. S. Stirling, Modal Logics for Communicating Systems, CSR-193-85, University of Edinburgh.

[Sto] J. Stoy, Denotational Semantics, M. I. T. Press, 1977

[Tof] C. Tofts, Temporal Ordering for Concurrency, LFCS-49-88, University of Edinburgh.

[Wal] D. Walker, Bisimulation equivalence and divergence in CCS, proceedings LICS 1988.

[Zwi] J. Zwiers, Compositionality, Concurrency and Partial Correctness, PhD. Thesis University of Eindhoven, 1988.