Measuring Concurrency in CCS

Vashti Christina Galpin

Degree awarded with distinction 6 May 1993

A research report submitted to the Faculty of Science,
University of the Witwatersrand, Johannesburg,
in partial fulfilment of the requirements for
the degree of Master of Science

January 1993

Abstract

This research report investigates the application of Charron-Bost's measure of concurrency m to Milner's Calculus of Communicating Systems (CCS). The aim of this is twofold: first to evaluate the measure m in terms of criteria gathered from the literature; and second to determine the feasibility of measuring concurrency in CCS and hence provide a new tool for understanding concurrency using CCS. The approach taken is to identify the differences between the message-passing formalism in which the measure m is defined, and CCS; and to modify this formalism to enable the mapping of CCS agents to it. A software tool, the Concurrency Measurement Tool, is developed to permit experimentation with chosen CCS agents. These experiments show that the measure m, although intuitively appealing, is defined by an algebraic expression that is ill-behaved. A new measure is defined and it is shown that it matches the evaluation criteria better than m, although it is still not ideal. This work demonstrates that it is feasible to measure concurrency in CCS and that a methodology has been developed for evaluating concurrency measures.

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other university.

[illegible]

Vashti Christina Galpin

28th day of January, 1993

To my parents, Daphne and Paul, and my sister, Karen.

Preface

This research report constitutes the research requirement for the degree of Master of Science by coursework (60%) and research report (40%).

The reason for undertaking this research was to investigate measurement of concurrency in Milner's Calculus of Communicating Systems (CCS) and to address the issue of evaluation of measures of concurrency. CCS provides a formalism that describes concurrency, and additional tools, such as measures of concurrency can aid in the task of understanding concurrency. A number of measures of concurrency have been presented in the literature; however, they are generally not accompanied by a full evaluation. This is partly a result of the fact that there is not yet an accepted set of criteria against which to evaluate concurrency measures. It is important that research is performed to address both of these issues.

The preliminary literature survey for this research was conducted to November 1991 from January 1992. The theoretical work was done in April 1992 and the development of and experimentation with the Concurrency Measurement Tool was done in May and June 1992.

I would like to thank my supervisor Scott Hazelhurst for his guidance, insight, time and encouragement. Thanks to the research seminar—Conrad Mueller, Ian Sanders, Philip Machanick, Sean Pyott, Philip Green and András Salamon—for their constructive criticism. I would also like to thank Colin Fidge for answering my questions, members of the Internet community for responding to my postings, and Veda Hakizumwami for his help with the translation of Charron-Bost's thesis. Thanks to Sheila Rock for her encouragement and to my friends and family for the supportive environment that they provided. Finally, I would like to thank the Foundation for Research and Development for their financial support.

Contents

Abstract ii

Declaration iii

Preface v

Contents vi

List of Tables x

List of Figures xi

1 Introduction 1

1.1 Problem motivation and analysis 1

1.2 Statement of problem and objectives of the research 3

1.3 Method of investigation 4

1.4 Organisation of the research report 5

2 Concurrency : measures and formalisms 7

2.1 Introduction 7

2.2 Evaluation criteria for measures of concurrency 7

2.2.1 Intuitive understanding of the measure 9

2.2.2 Being well behaved for small examples 9

2.2.3 Compatibility with operators on computations 9

2.2.4 Usability and applicability 13

2.2.5 Ability to calculate measure for a specific event 14

2.2.6 Expense of computation in terms of both time and space . . 14

2.2.7 Stability with respect to granularity 14

2.3 Measures of concurrency 14

2.3.1 The message-passing formalism 15

2.3.2 Charron-Bost's measure of concurrency ω 19

2.3.3 Charron-Bost's measure of concurrency m 20

2.3.4 Charron-Bost's measure of parallelism ρ 23
2.3.5 Fidge's measure of concurrency β 24
2.3.6 Raynal, Mizuno and Neilsen's measure of concurrency α . . . 26
2.3.7 Other measures 28
2.4 CCS 32
2.4.1 Operators 32
2.4.2 Algebraic calculi of process and partial orders 33
2.4.3 Time in CCS 34
2.4.4 Measuring concurrency in CCS 35
2.5 Choice of measure 36
2.6 Summary 38

3 Applying the measure to CCS **39**
3.1 Introduction 39
3.2 The message-passing formalism and CCS 39
3.3 Discussion 42
3.4 New message-passing formalism 43
3.4.1 Computation of a distributed system 43
3.4.2 Cuts 46
3.4.3 Clocks 48
3.4.4 Measure of concurrency 51
3.4.5 Calculating the number of consistent cuts 53
3.5 Expressive power 54
3.6 Translation of CCS 58
3.6.1 Discussion 58
3.6.2 Translation algorithm 62
3.7 Summary 63

4 The Concurrency Measurement Tool **64**
4.1 Introduction 64
4.2 Description of approach 64
4.2.1 Data structures 65
4.2.2 Program overview 67
4.3 Translation procedure 67
4.4 Algorithm for adding vector clocks 68
4.5 Algorithm for counting consistent cuts 70

4.5.1 Analysis of algorithm 73
4.5.2 Other algorithms 75
4.6 Use of the Concurrency Measurement Tool 76
4.6.1 Experimental error 77
4.7 Performance of Concurrency Measurement Tool 77
4.8 Extensions to the Concurrency Measurement Tool 78
4.9 Calculation of other measures 79
4.10 Summary 80

5 Results and evaluation **81**
5.1 A review of evaluation criteria 81
5.2 The experiments 82
5.2.1 Experiment *A*: Simple example 82
5.2.2 Experiment *B*: Simple example 82
5.2.3 Experiment *C*: Composition operator 83
5.2.4 Experiment *D*: Prefix operator 85
5.2.5 Experiment *E*: *Par* operator 85
5.2.6 Experiment *F*: *Before* operator 87
5.2.7 Experiment *G*: Dining philosophers 87
5.2.8 Experiment *H*: Dining philosophers 88
5.3 Discussion and evaluation of *m* 89
5.3.1 Evaluation of *m* with respect to the criteria 93
5.4 A new measure 95
5.4.1 Experiment *A*: Simple example 95
5.4.2 Experiment *B*: Simple example 96
5.4.3 Experiment *C*: Composition operator 96
5.4.4 Experiment *D*: Prefix operator 96
5.4.5 Experiment *E*: *Par* operator 96
5.4.6 Experiment *F*: *Before* operator 96
5.4.7 Experiment *G*: Dining philosophers 97
5.4.8 Experiment *H*: Dining philosophers 97
5.5 Discussion and evaluation of the new measure 97
5.5.1 Evaluation of the new measure with respect to the criteria . . 99
5.6 Evaluation of the measurement of concurrency in CCS 100
5.6.1 Measuring concurrency in CCS 100
5.6.2 A methodology for the evaluation of concurrency measures . 100

5.7 Summary 101

6 Conclusions and further research 102

6.1 Outline 102

6.2 Summary of work done 102

6.3 Conclusions 104

6.4 Further research 106

6.4.1 Improving the measure based on consistent cuts 106

6.4.2 Further investigation of the algorithm to count consistent cuts 107

6.4.3 Comparison of measures applied to CCS 107

6.4.4 A measure based on linear extensions 107

6.4.5 Extension of the measure to algorithms 108

6.4.6 Extension of the message-passing formalism 108

6.4.7 Transition graphs of CCS agents 109

6.4.8 Extension of the measure of concurrency on CCS 110

6.4.9 Application of concurrency measures to other formalisms . . 110

6.4.10 Partial ordering semantics for CCS 111

6.5 Conclusion 112

A An overview of CCS 113

A.1 The basic language 113

A.1.1 Syntax and notation 113

A.1.2 Operational semantics 114

A.2 Strong congruence, observational equivalence and observational congruence 115

A.3 Determinism and confluence 116

B Proofs 119

C Agents for the dining philosophers problem 122

References 128

List of Tables

4.1 Comparison of actual and theoretical times. 78
5.1 The nth root of m 92

List of Figures

2.1 (a) Concatenation. (b) Fusion. 11

2.2 A distributed computation expressed as a space-time diagram. . . . 16

2.3 A distributed computation with linear logical clocks. 17

2.4 A distributed computation with vector logical clocks. 19

3.1 Example of a distributed computation. 44

3.2 An example of a distributed computation with synchronous communication. 46

3.3 (a) An example of a cut in Charron-Bost's formalism. (b) An example of a cut in the new formalism. 47

3.4 A distributed computation with vector clocks. 49

3.5 (a) $a.0 \mid b.0$. (b) $a.b.0 + b.a.0$. 62

4.1 An example computation. 76

5.1 Experiment A. 84

5.2 Experiment B. 84

5.3 Experiment C. 86

5.4 Experiment D. 86

5.5 The agent for Experiments E and F. 86

5.6 The agent E_1'. 98

A.1 The confluent agent $a.0 \mid b.(c.0 \mid d.0)$ [50, p. 238]. 117

1. Introduction

As Hoare has noted 'Concurrency remains one of the major challenges facing Computer Science, both in theory and in practice' [50, Foreword]. For this reason, the investigation of issues that relate to concurrency is an important area of research, as such investigations can further our understanding of concurrency.

1.1 Problem motivation and analysis

Algebraic calculi of processes provide an approach to modelling communication and concurrency by describing the behaviour of concurrent communicating systems in terms of an operational semantics. Robin Milner's CCS (Calculus of Communicating Systems) is an example of this type of calculus. CCS provides for the definition of agents using a small number of operators and allows for the description and specification of concurrent systems, and hence for an understanding of the behaviour of such systems and the comparison of different systems. CCS also allows for specifications that abstract from the internal details; and for partial specifications that use agent variables. Any new tools developed for use with CCS will further the understanding of concurrent systems.

In the literature, there are a number of approaches to analysing the behaviour and performance of concurrent systems. Lamport and Lynch [47] note that there are two basic complexity measures for distributed algorithms: time complexity and message complexity. Time complexity measures the amount of time taken by message passing in a particular algorithm, since the time taken for message passing is assumed to be much longer than that required for computation. Message complexity is based on the number of messages transmitted during a computation, although it can also be based on the total number of bits transmitted. Often with distributed algorithms there is a tradeoff between time and message complexity—an algorithm can be 'improved' by decreasing its message complexity with a resulting increase in time complexity. Parallel programs also exhibit concurrency and their performance can be measured by speedup; the ratio of time taken to execute the program on one processor and the time taken on n processors [57].

However, as Bernadette Charron-Bost has noted [15], none of these measures assess the structure of the computation. She gives the example of a multibroadcast problem with two solutions—the solution with the lower number of messages works in an essentially less concurrent manner and is less resistant to failure than the solution with the higher number of messages. She also notes that message complexity can be misleading since the length of messages is not taken into account, although this can be ameliorated by using bit complexity.

The issue of message and time complexity in distributed systems will not be discussed further in this document; however, speed-up in parallel systems will be discussed in Chapter 2 because this gives an indirect indication of the amount of parallelism present in a computation.

Recently in the literature, the notion of a concurrency measure has been proposed by a number of authors [2, 3, 4, 5, 14, 15, 32, 36, 38, 44, 45, 59]. This notion is based on the concept of a quantitative measure of the amount of concurrency in a computation or algorithm. These measures are defined either on $[0,1]$ or $[0,\infty)$, and they are usually based on an intuitively appealing feature of the theoretical framework within which they are defined.

Although there have been a number of different measures proposed, there has been a lack of experimental results for these measures, and the majority require further evaluation. Generally, it appears that researchers present measures but little further research is done to evaluate the usefulness of the defined measures for dealing with concurrency in more practical situations. One aspect of this research is to remedy this situation by first investigating one specific measure, and second by proposing a general approach for evaluating measures.

To evaluate measures of concurrency, it is necessary to determine which features or characteristics are generally desirable. Dispersed throughout the literature are a number of different criteria for evaluating measures of concurrency. These different approaches will be drawn together in this research and used to evaluate the measure of concurrency to be investigated. Evaluation criteria can be divided into two categories—those that are subjective and those that are objective. The three objective criteria are those that relate to the performance of the algorithm used to calculate the measure, to whether a measure of concurrency can be determined for a specific event, and to the stability of the measure with respect to granularity. (These terms will be defined and explained in Chapter 2.)

The remaining criteria can be described as subjective. The intuitive understandability of the measure and its behaviour on small examples are subjective because they relate to individuals' perceptions of concurrency. These two criteria are often used in the original design of the measure.

Another important criterion is that of compatibility with operators. Operators on concurrent computations allow computations to be 'joined' in some sense. Examples of such operators are parallel concatenation and sequential concatenation. When two computations are combined by an operator, it would be desirable that the measure of concurrency for the combined computations can be explained in terms of the type of operator. For example, when computations are combined in parallel, it is reasonable to expect the measure to increase as the computation has in some sense become more parallel. There are a number of different approaches to operators in the literature which will be discussed in this document. Compatibility with operators is an important criterion as it relates to the different levels of abstraction that are used in specifications, and to partial specifications. If the effect of an operator can be explained, then the effect on the amount of concurrency when exchanging components or instantiating variables can be determined. This will lead to a better understanding of concurrent systems.

Finally there are two further criteria, which are also subjective, that relate to the functionality of the measure—usability in analysing distributed algorithms and applicability to real situations. These last two criteria obviously are important in determining the usefulness of the measure.

1.2 Statement of problem and objectives of the research

CCS is a formalism for the investigation of concurrency that provides an understanding of concurrent behaviour and any tools that can be used with CCS will advance this understanding. An example of this type of tool for investigating concurrency is the concept of a measure of concurrency. A number of these measures are presented in the literature; however, most of these have not been fully evaluated. The objective of this research is to address these issues by using CCS as a framework in which to evaluate measures of concurrency.

The focus of this research therefore will be to apply a measure of concurrency to CCS. As will be shown in Chapter 2, Charron-Bost's measure of concurrency m [14] has been chosen as the measure to be studied in this research, as it is one of a number appearing in the literature that require investigation. This measure of concurrency is based on the concept that 'the less the stopping of one of the processes blocks the other processes, the more concurrent is the computation' [14, p. 45].

By applying a measure to CCS, a new approach to investigating aspects of concurrency will be gained and a new methodology for evaluating concurrency measures in CCS will be defined, although in this research only one measure will be investigated.

The main objectives of this research are to:

1. Evaluate m as a measure of concurrency by applying it to CCS and using the evaluation criteria that are found in the literature.
2. Investigate the feasibility of measuring concurrency in CCS.
3. Develop and evaluate a tool to measure the concurrency of CCS agents.

The outcome of achieving these objectives will be:

1. A measure of concurrency defined for CCS will provide a new tool for the investigation of concurrency and a framework for the measurement of concurrency in CCS.
2. The measure of concurrency m will be evaluated, first in terms of its applicability and second in terms of its validity as a measure.
3. An approach to evaluating measures of concurrency will be developed.

1.3 Method of investigation

The research will be conducted in the following manner:

1. A literature survey of evaluation criteria for measures of concurrency, measures of concurrency and relevant issues in CCS will be presented and the choice of the measure m will be justified.
2. The differences between the message-passing formalism in which the measure is defined, and CCS will be identified.

3. The message-passing formalism will be redefined so that CCS agents can be translated into this formalism while ensuring that the justification for the measure of concurrency remains the same. It will be shown that the existing results still hold in the redefined model.

4. A translation algorithm will be developed to map CCS agents into the message-passing formalism. The algorithm will deal only with a subset of CCS and this subset will be chosen and motivated.

5. An algorithm will be developed to calculate the measure and a theoretical analysis of this algorithm will be presented.

6. The Concurrency Measurement Tool will be implemented to automate the calculation of the measure for CCS agents. The Concurrency Measurement Tool will be used to experiment on chosen CCS agents.

7. The results of the experiments will be used to evaluate of the measure in terms of the criteria presented in the literature survey and to investigate the feasibility of measuring concurrency in CCS

1.4 Organisation of the research report

The report will be structured as follows:

Chapter 1. Introduction

Chapter 2. Concurrency : measures and formalisms This chapter will present the background literature that relates to concurrency measures. Ways to evaluate measures of concurrency will be presented first to set the scene for the discussion of measures of concurrency. These criteria will be described in detail, and a framework will be developed to discuss the interpretation of compatibility with operators. Measures will be grouped in terms of the frameworks within which they are defined. Lamport's message-passing formalism that defines a partial ordering model of a distributed system, will be discussed in some detail, as a number of measures including the measure to be evaluated in this research, are defined in this formalism. A discussion of relevant issues in CCS, such as partial ordering, time and a measure of maximum parallelism will also be presented. The justification of m as the measure for consideration will be given in the final section.

Chapter 3. Applying the measure to CCS In this chapter, the differences between the message-passing formalism that describes a distributed system and CCS will be identified and discussed. After this, the message-passing formalism will be redefined to take into account these differences. This redefinition will be done so that the justification for the measure still holds in the new formalism. A number of results are required to define the measure, and it will be shown that these results are valid in the redefined formalism. Finally, issues relating to the translation procedure for mapping CCS agents into the message-passing formalism will be discussed and the translation algorithm will be presented.

Chapter 4. Concurrency Measurement Tool This chapter will discuss the program written to perform the experiments on CCS agents. The main focus of the chapter will be on the algorithm for counting the number of consistent cuts although other algorithms and an overview of the program will be presented. An analysis of the algorithm for counting consistent cuts will be presented and the performance of the Concurrency Measurement Tool will be related to the theoretical results for the algorithm. Other algorithms will be suggested and analysed.

Chapter 5. Results and evaluation The experiments that were performed using the Concurrency Measurement Tool will be detailed, including a description of the aim of each experiment and the agents used in the experiment. The results of each experiment will be presented and will be used to investigate the measure of concurrency in terms of the criteria presented in Chapter 2. These results will show that m does not meet all the criteria and a modified measure m_{new} will be presented and evaluated. This evaluation will show that m_{new} is better behaved than m. It will be argued that the measurement of concurrency in CCS is feasible and that a methodology has been developed to evaluate concurrency measures.

Chapter 6. Further research and conclusions In this chapter, a summary of the research and the conclusions will be presented. Further research will be outlined.

2. Concurrency : measures and formalisms

2.1 Introduction

In this chapter, background literature will be presented, relating to both measures of concurrency and CCS. Criteria for the evaluation of measures of concurrency will be detailed, and a new framework for the discussion of compatibility with operators will be developed. This will facilitate the discussion of measures of concurrency that will follow. The majority of the measures to be discussed in this chapter are defined in the message-passing model of a distributed system defined by Lamport. This formalism will be described and then the relevant measures will be presented, including Charron-Bost's measure m which is the focus of the research. In the section on CCS, the components of an algebraic calculus will be presented, and partial orders and time in CCS will be discussed (an overview of CCS is presented in Appendix A). A measure of maximum parallelism defined for CCS will also be presented. In the final section, the justification for the choice of Charron-Bost's measure m will be given.

2.2 Evaluation criteria for measures of concurrency

The idea behind a measure of concurrency/parallelism is to obtain a numerical value to indicate the amount of concurrency there may be in a computation or an algorithm. This value will then allow for the comparison of computations or algorithms in terms of the amount of concurrency.

In general, measures of concurrency[1] fall into two distinct groups with respect to the range of values that the measure can obtain. Measures in the first group give a value in the interval $[0,1]$, indicating the amount of concurrency present with 0 indicating no concurrency and 1 indicating some maximum amount of concurrency. Those in the second group return a value in the interval $[0,\infty)$ giving some indication

[1] In the rest of this research report, the term 'measure of concurrency' will be used to indicate both a measure of concurrency and a measure of parallelism.

of the number of concurrent processes.

It is necessary when developing a new tool to perform experiments to ensure that the tool behaves correctly. In most of the literature dealing with measures of concurrency, this experimentation is not present, except for a few small examples. For a measure of concurrency to become useful, experiments and experiences with the measure need to be reported.

The process of defining a measure of concurrency presents difficulties as there is no single objective way to determine if the measure is performing as desired. Consider the following example. If a new technique is being developed for measuring short distances, it can be evaluated and checked for consistency against methods proven to work, for example, tape measures. However, in this situation, there is no well-defined method with which to compare a concurrency measure to determine its correctness. This means that it is necessary to determine a number of criteria that can be used to evaluate the measure. This is complicated by the fact is that some of these criteria are also used in the definition of the measure. For example, a measure can be defined by finding an intuitively appealing explanation for using a particular feature of a formal model, and by using the measure on small examples and with operators.

The criteria that are found in the literature are as follows:

1. Intuitive understanding of the measure [14].
2. Being well behaved for simple examples [14, 15].
3. Compatibility with operators on computations [15, 18, 32].
4. Usability for comparison and analysis of distributed algorithms and applicability to real situations [15, 45].
5. Ability to calculate measure during the computation for a specific event [32, 59].
6. Lack of expense of computation in terms of both time and space. [32, 38, 59].
7. Stability with respect to granularity [16].

These criteria are open to criticism because although they can show that a measure is poor, the process of showing whether a measure is good is more difficult as there is no objective way to do this.

In following sections, each criterion will be explained. The first four items are very subjective criteria although they may not all seem so at first. The next three are more objective and it is possible to check objectively whether a measure of concurrency satisfies them, although there is some debate about whether the final criterion is applicable to all measures.

This section is presented before the review of concurrency measures because the concepts given here are required to discuss the measures. However, some material in Section 2.2.3 relies on definitions that occur later in this chapter, and the reader is advised to read that section briefly at first.

2.2.1 Intuitive understanding of the measure

This criterion relates to being able to understand why the definition of the measure is sensible [14]. For example, Charron-Bost's measure m, produces a value in the range [0,1], where 0 represents total sequentiality and 1 total concurrency. The concepts that are being used to define the measure, namely consistent cuts, are explained as a way of measuring the tolerance of the computation to stopping. This explanation appears to give a good approach to measuring concurrency.

This criterion is, of course, subjective as one cannot objectively determine the intuitive appeal of a definition. It is also conceivable, however, that a measure could be defined in such a way that its definition is non-intuitive or even counter-intuitive, and still satisfy all other criteria.

2.2.2 Being well behaved for small examples

It is desirable that a measure of concurrency will work in a reasonable way on small examples [15]. In fact, as described above, this is often used in defining a measure. Charron-Bost [14] uses these grounds to reject the measure ω. She presents two examples that have the same measure under ω. However, the space-time diagrams of the examples show that one appears more concurrent. This means that this is a subjective criterion, since it involves perceptions of concurrency. In small examples, it would seem fairly easy to obtain some sort of consensus in the area; however, when dealing with larger computations, it becomes more difficult to decide.

2.2.3 Compatibility with operators on computations

Compatibility with operators means that when computations are 'joined' by operators, the amount of concurrency will change in a way that can be explained by the actions of the operators. This issue is discussed both by Charron-Bost [15, 16, 18] and Fidge [32] in their papers on measures of concurrency. This section will be organised as follows: first, the approaches taken in the literature will be discussed in detail, then a framework will be developed to generalise the work that has been presented. The explanation of the operators relies on definitions to be presented later in the chapter and it is suggested that the reader give this section a brief reading at first, returning after the completion of Section 2.3.

Charron-Bost [15] defines two operators—concatenation and fusion. Concatenation is essentially the joining of one computation after another where the computations have the same number of processes[2] (see Figure 2.1a). Fusion occurs when two computations have some process sequences with common prefixes. The processes are merged to obtain the longest computations for each process[3] (see Figure 2.1b). The effect is to obtain a computation from the two computations that is more parallel than either of the component computations.

Charron-Bost's measure of parallelism ρ is evaluated in terms of these two operators [15]. If C is the concatenation of C' and C'', with measures ρ, ρ' and ρ'' respectively, then ρ can be expressed as

$$\rho = w'\rho' + w''\rho'' \text{ where } w' + w'' = 1$$

where w' and w'' correspond to the relative sizes of the computations C' and C''. This implies that

$$\rho \geq \min(\rho', \rho'') \text{ and}$$

$$\rho \leq \max(\rho', \rho'').$$

Since for ρ, 0 represents a totally concurrent computation and 1 a totally serial computation, this shows that the computation formed by concatenation cannot become more parallel than its more parallel component, nor can it become less parallel than its less parallel component.

[2] Consider the computations

$$\begin{array}{llllll} C & = \{C_1, C_2, C_3\} & \text{with} & C_1 = ab & C_2 = cde & C_3 = fghijk \\ C' & = \{C'_1, C'_2, C'_3\} & \text{with} & C'_1 = l & C'_2 = mn & C'_3 = opg \end{array}$$

then

$$\begin{array}{llllll} C'' & = \{C''_1, C''_2, C''_3\} & \text{with} & C''_1 = abl & C''_2 = cdemn & C''_3 = fghjkopg. \end{array}$$

is the concatenation of C and C'.

[3] Consider the computations

$$\begin{array}{llllll} C & = \{C_1, C_2, C_3\} & \text{with} & C_1 = abcde & C_2 = fgh & C_3 = jk \\ C' & = \{C'_1, C'_2, C'_3\} & \text{with} & C'_1 = ab & C'_2 = fgh & C'_3 = jklmno \end{array}$$

then

$$\begin{array}{llllll} C'' & = \{C''_1, C''_2, C''_3\} & \text{with} & C''_1 = abcde & C''_2 = fgh & C''_3 = jklmno. \end{array}$$

is defined as the fusion of C and C'. Note that C'_1 is a prefix of C_1 and C_2 is a prefix of C'_2.

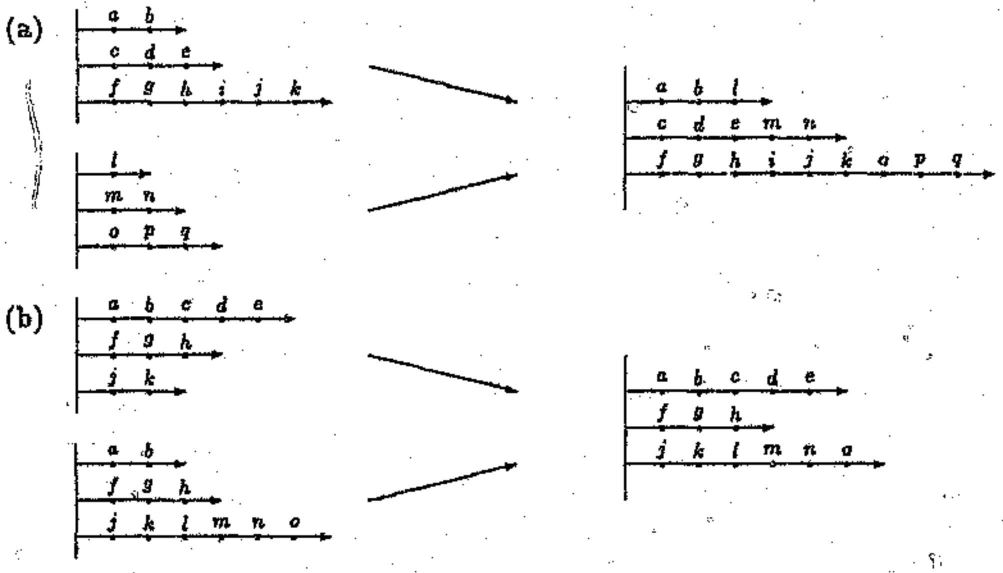

Figure 2.1: (a) Concatenation. (b) Fusion.

Also, if C^m is the concatenation of C m times with measure ρ^m, then

$$\rho^m = \rho.$$

Charron-Bost argues that this shows the behaviour of the measure ρ is correct.

If C is the fusion of C' and C'', with measures ρ, ρ' and ρ'' respectively, then it can be shown that

$$\rho \leq \max(\rho', \rho'').$$

Since for ρ, 0 represents a totally concurrent computation and 1 a totally serial computation, this shows that the computation formed by fusion is more parallel than the lesser of two computations combined to form the fusion.

Fidge [32] has defined a measure β with range $[0,1]$ with 0 representing total sequentiality and 1 representing total concurrency. The following relationship holds for the computations $C_1, \ldots, C_n$

$$\beta(C_1; \ldots; C_n) < \beta(C_1 \mid \ldots \mid C_n)$$

where ; represents the serial concatenation of computations and | represents the parallel concatenation of computations[4]. Also

$$\beta(C_1; \ldots; C_n) \leq \max_{1 \leq j \leq n} \beta(C_j)$$

[4]The definition of ; and | are simpler in Fidge's formalism since nested parallelism is allowed.

$$\beta(C_1 \mid \ldots \mid C_n) > \min_{1\leq j\leq n} \beta(C_j).$$

When computations are concatenated serially then the amount of concurrency cannot become greater than the most concurrent component computation; similarly when computations are concatenated in parallel then the amount of concurrency cannot become smaller than the least concurrent component computation. He also notes that

$$\beta(C_1;\ldots;C_n) = \max_{1\leq j\leq n} \beta(C_j) \text{ when } \beta(C_i) = 0 \,\forall\, i \in \{1,\ldots,n\}$$

although in the general case, β is significantly reduced.

Generalisation

To write these concepts in a generalised form, let $\wr$ represent some generic form of parallel concatenation, $\rightsquigarrow$ some generic form of serial concatenation and MC a generic concurrency measure, defined on $[0,1]$ with 0 representing a totally sequential computation and 1 a totally concurrent computation.

I have chosen the following rules to describe compatibility with operators in the general case for measures defined on $[0,1]$:

$$\begin{aligned} &\mathrm{MC}(C' \rightsquigarrow C'') \leq \max(\mathrm{MC}(C'), \mathrm{MC}(C'')) \\ &\mathrm{MC}(C' \wr C'') \geq \min(\mathrm{MC}(C'), \mathrm{MC}(C'')). \end{aligned}$$

They can be described as follows:

- when dealing with serial concatenation the concurrency cannot become greater than the amount occurring in any component computation;
- when dealing with parallel concatenation the concurrency cannot become less than the amount occurring in any component computation.

These rules have been chosen because they appeal to an intuitive understanding of the application of operators to computations, they are not dependent on any particular measure as both hold for ρ and β, and although they are general, they are not so general as to be meaningless.

Both Charron-Bost[5] and Fidge state that sequential composition should make the resulting computation no more concurrent than its more concurrent component

[5] As the work done by Charron-Bost relates to a measure defined so that 1 represents total sequentiality and 0 represents totally concurrency, it is necessary to 'invert' the results with respect to the generic concurrency measure that is defined so that 1 represents total concurrency and 0 represents total sequentiality, as are most measures presented in the literature.

computation; and that parallel concatenation should make the resulting computation no less concurrent than its least concurrent component computation.

However, they differ with respect to sequential concatenation on two results. First with respect to the condition under which equality should occur; Fidge states that equality only happens when both component computations are sequential and therefore have measure 0, whereas for Charron-Bost equality occurs when the two component computations are the same. Second, they differ on whether the resulting computation can become less concurrent than the component computations; it can be shown from Charron-Bost's work that the resulting computation cannot become less concurrent than the less concurrent of the component computations, whereas Fidge does not draw this conclusion. In the rules given above, I have taken a more general approach that does not exclude any of these interpretations.

2.2.4 Usability and applicability

A measure of concurrency must be usable for comparing and analysing real distributed algorithms [15, 45]. This can be tested by performing the measurement and investigating the results. As noted earlier, it is difficult to have an objective or even intuitive idea about the amount of concurrency present in an algorithm, so this presents problems for evaluating the results of the experiment.

An issue that can also be raised under this heading is the issue of the distribution of values in the range of the measure. If the values obtained for algorithms being measured tend to fall in a small subrange, there are a number of possible reasons:

- The algorithms that are being investigated have some particular characteristic; other algorithms will have different measures.
- The measure is poor and all 'real' algorithms will have a similar measure that does not differentiate between them.
- The measure is in a sense 'non-linear' and all 'real' algorithms will fall into a very small subrange; however the difference in the values of the measures is significant.

Although the last two items have very different implications, it is difficult to determine which one actually holds for a particular measure.

2.2.5 Ability to calculate measure for a specific event

The measure can be calculated for a specific event if it has been defined so that for any given event, a value can be calculated for the computation up to and including this event [32, 59]. This allows for an understanding of how the concurrency changes during the computation, and an indication of where bottlenecks are occurring. Another advantage occurs when these values can be determined during the running of the computation, as opposed to afterwards. This is obviously an objective criterion.

2.2.6 Expense of computation in terms of both time and space

This relates to the algorithm for the calculation of the measure [32, 38, 59]. The questions that need to be asked are as follows:

- Does the algorithm require exponential time?
- If the algorithm is exponential, does it allow the calculation of the measure for reasonably sized examples? What are the constants involved?
- If the algorithm is exponential, are there any algorithms that will allow for an approximate solution?

Again, this criterion can be classed among those that can be objectively determined.

2.2.7 Stability with respect to granularity

For some measures, it is necessary to count events that have occurred in specific processes. Therefore the definition of what constitutes an event becomes important, and as this definition becomes finer or coarser, there is an increase or decrease in the number of events. Charron-Bost [16] defines this as granularity. However, she expresses reservations about whether it is necessarily desirable for all measures to be stable with respect to granularity.

2.3 Measures of concurrency

Under this heading, the measures of concurrency found in the literature will be presented. It is only relatively recently that the concept of a measure of concurrency has been proposed, and discussed in detail. In some references, the concept is developed as minor part of a broader topic, and in others, the focus of the article is on a particular measure of concurrency. The different approaches that appear in the literature have not yet been compared, related or fully evaluated and the aim of this research is to contribute towards this.

2.3.1 The message-passing formalism

The measures of concurrency described in the following sections are defined in the framework of space-time diagrams of Lamport [46], also referred to as the message-passing formalism. These diagrams define a formal model of a distributed system and allow for the definition of a partial ordering on the events in the distributed system, called the 'happened before' relation. These measures of concurrency rely on the definitions of logical clocks that capture the partial ordering of events in the distributed computations.

The 'happened before' relation

Lamport [46] introduced the 'happened before' relation that gives a partial order of events in a distributed computation. This framework has been formalised by Charron-Bost [14] as follows:

A distributed system consists of a finite set of sequential processes $\{P_1, \ldots, P_n\}$. A process P_i is characterised by a set of sequences of events. Each set P_i is prefix closed, namely for any sequence in P_i, all prefixes of the sequence are also in P_i. Events fall into three classes:

1. an internal event,
2. the sending of a message to another process,
3. the receipt of a message from another process.

All events and messages are different and can be distinguished from each other by some means, for example unique subscripts. Messages are sent from one process P_i and received by another process P_j, so a message cannot be sent by multiple processes or received by multiple processes.

For each i, let C_i be one of the sequences defining P_i. The relation $\prec$ is defined on the set of events obtained from the set of sequences[6] $C_1 \cup \cdots \cup C_n$ as the smallest transitive relation satisfying the following: (1) if a and b occur in the same process and a comes before b, then $a \prec b$, (2) if a is a sending of a message m, and b is the receipt of m, then $a \prec b$. This relation, known as the 'happened before' relation

[6] In [16], a sequence of events is defined as a totally ordered set of events.

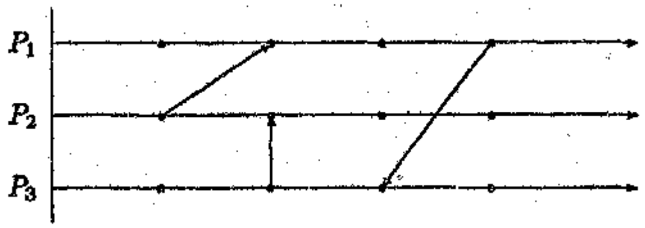

Figure 2.2: A distributed computation expressed as a space-time diagram.

forms a partial order on the set of events, and was first defined by Lamport [46]. It captures the causal relationship between events.

$C = C_1 \cup \cdots \cup C_n$ is called a computation of $\{P_1, \ldots, P_n\}$ if it satisfies the following: (1) $(C, \prec)$ contains no cycles (2) for every receipt of the message m, there is a single sending of m.

$\preceq$ is defined as: $a \preceq b$ iff $a \prec b$ or $a = b$. Events a and b are concurrent (a co b) if $\neg(a \preceq b)$ and $\neg(b \preceq a)$. Concurrent events are those that are causally independent of each other.

The happened-before relationship can be captured in the space-time diagrams introduced by Lamport [46] (see Figure 2.2) and such diagrams entirely define the computation [14]. In Figure 2.2, the horizontal lines represent processes, the time axis runs from left to right, and send and receive events are joined by a directed line.

Logical clocks

The reason for defining clocks on a distributed computation is to provide a tool to determine if two events are concurrent with respect to the partial order defined above [14]. This can be done by assigning a timestamp or date to each event, and comparing these dates to see if two events are concurrent.

There are three different approaches to logical clocks as surveyed by Raynal [58]:

1. linear logical time – each date is represented by a single integer,

2. vector logical time – each date is represented by an n-dimensional vector,

3. matrix logical time – each date is represented by an $n \times n$ matrix.

Only the first and second have been used for measures of concurrency and they will be described here.

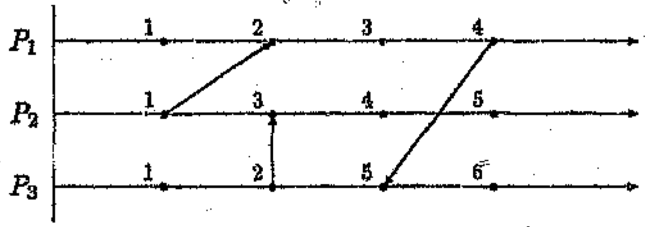

Figure 2.3: A distributed computation with linear logical clocks.

Linear logical time Linear logical time was defined by Lamport [46] in 1978. A clock c_i is defined for each process P_i which assigns a number $c_i(a)$ to each event a. The entire system of clocks is defined by c which assigns to event b the number $c(b)$ where $c(b) = c_j(b)$ if b is in process P_j. The implementation rules for the clocks are as follows:

1. each clock c_i is initialised to 0,
2. when process P_i executes an internal event c_i is incremented by 1,
3. when process P_i executes a send event, c_i is incremented by 1 and the message m contains a timestamp $T_m = c_i(a)$,
4. if event b is the receipt of a message m by process P_j, then c_j is set to $\max(T_m, c_j(b))$ and incremented by 1.

These clocks satisfy the condition

$$a \preceq b \Rightarrow c(a) \leq c(b).$$

Note that the converse implication cannot hold, as it would imply that $c(a) = c(b)$ if a and b are concurrent [14]. Therefore the relationship between events cannot be determined from the dates of events. Linear vector clocks, however, give a total ordering of the events and the time at any event gives the number of events (including that event) in the longest causal chain preceding it.

In Figure 2.3, the distributed computation of Figure 2.2 is shown with linear logical clocks.

Vector logical time Vector logical time was developed independently in 1988 by Fidge [30, 33], Mattern [49] and Schmuck [62][7].

[7][30, 62] are cited in Raynal's survey article on logical time [58].

Partially ordered logical clocks are represented as vectors in $\mathbb{N}^n$:

- If $v \in \mathbb{N}^n$ then $v[i]$ represents the ith component of v.
- $\mathbb{N}^n$ is partially ordered by $\leq$; $a \leq b$ iff $a[i] \leq b[i]$ for each index i.
- If $u, v \in \mathbb{N}^n$, then $w = \sup(u, v)$ is defined as follows: for each index i, $w[i] = \max(u[i], v[i])$.

Intuitively, the ith position in the vector clock of process P_i represents the time in P_i and the jth position ($j \neq i$) represents the most recent knowledge that P_i has of the time in process P_j. Clocks are applied to processes as follows:

1. the initial value of the clock Θ_i is $(0, \ldots, 0)$,
2. when process P_i executes an internal event, $\Theta_i[i]$ is incremented by 1,
3. when process p_i executes a send event, $\Theta_i[i]$ is incremented by 1 and the message m contains a timestamp $T_m = \Theta_i(a)$,
4. if event b is the receipt of a message m by process P_j, Θ_j is set to $\sup(T_m, \Theta_j)$, and $\Theta_j[j]$ is incremented by 1.

   Note that this last rule is equivalent to saying increment $\Theta_j[j]$ by one and set Θ_j to $\sup(T_m, \Theta_j)$, since $\Theta_j[j]$ always contains the most recent knowledge about the logical time in process P_j. So a more general rule for vector clocks states increment $\Theta_i[i]$ by 1 before any event in process P_i. This is the formulation that Charron-Bost uses in her work. However, this cannot be applied to linear logical time if the logical clock time is to represent the length of the longest causal chain of events before a given event (as is required for two of the concurrency measures presented below).

For any event a that occurs in process P_i, its vector time is $\Theta(a) = \Theta_i(a)$.
These clocks satisfy the condition

$$a \preceq b \Leftrightarrow c(a) \leq c(b).$$

This means that vector clocks characterise concurrency—from the dates of two events it is possible to determine whether the events are concurrent[8]. It has been

[8]There are other ways to characterise concurrency. For example, the absence of a path between two events in the directed graph formed by the space-time diagram [44].

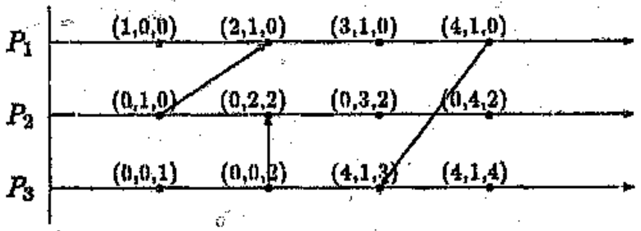

Figure 2.4: A distributed computation with vector logical clocks.

shown by Charron-Bost that for n processes, clocks of size n are optimal—the partial ordering of events cannot be captured with smaller clocks [17].

In Figure 2.4, the distributed computation of Figure 2.2 is shown with vector logical clocks.

Fidge [31][9] extends vector clocks to deal with nested parallelism where processes can be dynamically created and terminated.

In the following sections, I will present the five measures of concurrency defined in the above described framework. Each measure is defined for a specific computation of an algorithm, where a computation is defined to be a specific execution of an algorithm. Charron-Bost notes [15, 16] that to obtain a measure for an algorithm the average of the measures of all the computations with a certain probability must be calculated. This is not explicitly stated for all measures defined in this framework, but is taken as given. At the end of each section, any existing work on the evaluation of each measure will be presented.

2.3.2 Charron-Bost's measure of concurrency ω

Charron-Bost [14] suggests a measure of concurrency ω based on counting concurrent pairs of events—the more concurrent pairs, the more concurrency in the computation—defined as

$$\omega(C) = \frac{\left|\{(a,b) \mid a \text{ co } b\}\right|}{\left|\{(a,b) \mid a \in C_i, b \in C_j \text{ and } i \neq j\}\right|}.$$

If each C_i contains q_i events, then there are $\sum_{1 \leq i < j \leq n} q_i q_j$ concurrent pairs in total and this occurs when the computation is totally concurrent. If the computation is entirely sequential then there are no pairs of concurrent events. Hence the range

[9] Cited in [32].

of this measure is $[0, 1]$, where 0 represents totally serial, and 1 totally concurrent, namely with no communication between processes.

Charron-Bost notes that this is not a very accurate measure [14], and presents an example to show where the measure does not behave in an intuitive way on a simple example. The measure ω does not allow the calculation of a measure for a specific event. Fidge [32] has presented an algorithm to calculate the measure, but there is no analysis of its complexity. Kim et al [44] suggest an approximation to ω, W that is cheaper to calculate. The measure has not been evaluated in terms of any other criteria.

2.3.3 Charron-Bost's measure of concurrency m

The measure m [14] will be the focus of this research, and therefore it will be presented in detail. The reasons for the choice are presented in Section 2.5. The measure m is based on counting the number of consistent cuts in a computation. The concept behind this is that the more processes wait, the less concurrency the computation exhibits. Since the ability of the computation to be cut consistently is related to its tolerance to stopping, a computation's concurrency can be measured in terms of its number of consistent cuts.

A cut of a distributed computation can be defined as follows: Consider the sets $\{x \in C_i, x \preceq a_i\}$, where a_i is an event in C_i, and their union

$$C = \bigcup_{i \in \{1,\dots,n\}} \{x \in C_i \mid x \preceq a_i\}.$$

This is called a cut. It is not a computation, as it may contain the receipt of a message but not the sending of that message.

A consistent cut is a computation and it can be viewed as a global state of the computation where causality is not violated [33], hence the consistent cut cannot contain the receipt of a message without its sending. However, this global state does include messages that are in transit, namely those messages that have been sent but not yet received [49].

A consistent cut can be defined formally as follows: a consistent cut of a distributed computation C is a cut C' that is left closed by the causality relation. This means that for any $a, b \in C$, if $b \in C'$ and $a \preceq b \Longrightarrow a \in C'$. An example of a consistent cut is the past of an event a, $(\downarrow a)_C$ or $(\downarrow a)$ defined by $(\downarrow a) = \{x \in C, x \preceq a\}$.

Results from combinatorics are used to define the measure. If it is assumed that each C_i contains q_i events, then the following values can be determined analytically:

- The value μ^s represents the number of consistent cuts in a totally sequential computation. A totally sequential computation is one in which no events can happen concurrently and it can be conceptualised as each process completing before the next process begins. Hence, there are q_i events in each process at which the computation can be cut consistently. Together with the empty cut that is also consistent, it follows that $\mu^s = 1 + q_1 + \cdots + q_n$.
- The value μ^c represents the number of consistent cuts in a totally concurrent computation. This is defined to be a computation where there is no communication between processes. For each process, there are q_i events at which a consistent cut can occur plus the start of the process when no event has yet occurred. Because all the processes can run concurrently, this means that $\mu^c = (1 + q_1) \cdots (1 + q_n)$.

The measure m is defined as

$$m(C) = \frac{\mu - \mu^s}{\mu^c - \mu^s}$$

where μ is the number of cuts in the computation under consideration. The measure takes values in the interval $[0, 1]$ with 1 for a concurrent computation and 0 for a sequential computation.

Charron-Bost presents two methods to determine μ.

1. A cut

   $$C = \bigcup_{i \in \{1,\ldots,n\}} \{x \in C_i \mid x \preceq a_i\}$$

   is consistent if and only if

   $$\sup(\Theta(a_1), \ldots, \Theta(a_n)) = (\Theta(a_1)[1], \ldots, \Theta(a_n)[n]).$$

   Hence each cut can be checked for consistency using the vector clocks.

2. The number of antichains (or independent subsets)[10] of a partially ordered set of events is equal to the number of consistent cuts in that distributed

[10] An antichain or independent subset has the property that no pairs of elements are related by the partial order relation. In this case, all the events in an antichain will be concurrent.

computation. Therefore each ~~subset~~ can be checked to see if it is an antichain. Each pair of elements in the subsets can be checked for concurrency using the vector clocks.

Charron-Bost also presents a geometric interpretation of the measure. This interpretation allows for the identification of the events that contribute to a reduction in concurrency.

Charron-Bost presents a comparison of ω and m and she notes that ω is a special case of m. Let k-antichains denote antichains of size k, then ω investigates the number of 2-antichains, whereas m investigates the total number of k-antichains for $2 \leq k \leq n$. Hence $\omega = m$ when there are 2 processes. She notes that there exist computations C and C' such that $\omega(C) < \omega(C')$ and $m(C) > m(C')$.

Charron-Bost [16] has compared two algorithms for calculating m, based on the two methods of characterising consistent cuts. She notes that the algorithm that counts the number of antichains is efficient when the computation under consideration contains few 2-antichains. This would occur when there is little concurrency in the computation, namely when there are few concurrent events. Kim et al [44] have presented an analysis of the antichain algorithm that suggests it requires exponential time. No further analysis of these algorithms has been presented.

Raynal [59] describes m as a good characterisation of the degree of concurrency in a computation, but he notes that the computation of μ is not feasible. Fidge [32] notes that the calculation of μ using vector clocks requires $\sum_{1 \leq i \leq n} nq_i$ integers to be stored and that m can only be calculated after the computation has terminated.

Recent work by Kim et al [44] suggests an approximation to m, denoted M that is applied to formal protocol specifications. A computation is decomposed into concurrency blocks and a measure is calculated for each block based on the antichain approach. They show that the algorithm is faster than Charron-Bost's antichain algorithm; however they do not evaluate M in terms of any of the other criteria or show how the values returned by the approximation differ from those returned from the measure.

It has not been shown that there does not exist a fast algorithm to calculate μ, although any algorithm using vector clocks would have some expense in terms of space, because of the necessity of storing the vector clock information and time, because of the procedure for adding the vector clocks. It is possible to compare events for concurrency without using vector clocks by showing that there is no directed path in the graph between the events [44]. This results in a tradeoff between space and time, since once the vector clocks have been added, it is faster to check for concurrency using the vector clocks of two events than to show the absence of a path in the graph of events.

Charron-Bost [16] has shown that m does not give the expected results when used with the operators concatenation and fusion. She describes the following: if $C^{(l)}$ is the computation created from l copies of C concatenated together, and C has measure m and $C^{(l)}$ measure $m^{(l)}$. Then

$$m^{(l)} = m/l \text{ and } l \to \infty \Rightarrow m^{(l)} \to 0.$$

She notes that this is counter-intuitive, as it would be expected that $m^{(l)} = m$. However, within the generalisation for compatibility with operators presented in Section 2.2.3, this would be an acceptable result for an operator such as concatenation. She also notes that with fusion some concurrency is guaranteed when two very concurrent operations are joined by fusion. Finally, she states that there may be other operators that are better adapted for use with m, but she is not aware of any that are easily studied from the point of view of combinatorics.

Charron-Bost [16] notes that the measure is not stable with respect to granularity but argues that the causal relationship cannot remain the same as granularity changes. Therefore this criterion is not applicable to a measure based on the causal dependence and independence of events.

It is not possible to calculate the measure at a specific event in a computation, although the measure can be interpreted geometrically which allows the bottlenecks in the computation to be identified. The measure has not been evaluated in terms of applicability to real examples.

Hence it appears that there is conflicting evidence regarding m. It has been described as a good measure of concurrency, although it is computationally expensive and is not compatible with operators; however, further work is required in these areas.

2.3.4 Charron-Bost's measure of parallelism ρ

In other papers [15, 18], Charron-Bost presents a measure of parallelism, which relies on the idea that the less processes wait, the more concurrent the computation, as does m. This measure is presented in the above-mentioned framework with the following assumptions:

1. all processes start together,

2. the time for an internal event in P_i is constant and equal to δ_i,

3. sending and receipt of messages is instantaneous,

4. the time taken to deliver a message is constant and equal to δ_0.

A set of rules is defined for recursively determining the time $T(a)$ at which event a occurs. T_i is defined as the time of the last event in process P_i. Let p_i be the number of internal events in C_i and p_0 the number of messages in C. Two measures are defined. The first is $(\rho_1, \ldots, \rho_n)$ where

$$\rho_i = \frac{T_i - p_i\delta_i}{\sum_{j\in\{0,\ldots,n\}\setminus\{i\}} p_j\delta_j}.$$

The measure ρ_i is called a coefficient of coupling. This describes the interactions between P_i and the other processes. It is one when the computation is sequential and zero when it is totally concurrent and the last event of C occurs in C_i. The measure $(\rho_1, \ldots, \rho_n)$ presents problems for comparison as $\mathbb{R}^n$ is not totally ordered.

A less accurate measure that allows for comparison is defined as

$$\rho = \frac{\sum_{i=1}^{n}(T_i - p_i\delta_i)}{\sum_{i=1}^{n}(\sum_{j\in\{0,\ldots,n\}\setminus\{i\}} p_j\delta_j)}.$$

This measure takes on values in $[0, 1]$ and equals zero if entirely concurrent; however, it does not necessarily equal one if sequential. The measure ρ is called the mean coefficient of coupling.

Charron-Bost shows that this measure is compatible with the operators concatenation and fusion on computations, it is stable with respect to granularity, and notes that it is well behaved for simple examples of computations. It has not been evaluated in terms of any other criteria.

2.3.5 Fidge's measure of concurrency β

Fidge's aims in designing a measure of concurrency are that it should be more accurate than ω but less computationally expensive than m, should yield useful results during the computation, and should be general enough to deal with 'nested parallelism', namely where processes can be created and can terminate dynamically [32]. The measure is defined at an event a in the computation as

$$\beta = \frac{|\mathcal{P}| - \mathcal{T}}{|\mathcal{P}| - 1}$$

where $|\mathcal{P}|$ is the number of events that have occurred so far and is calculated by summing the event-counts for each process in the clock for a

$$|\mathcal{P}| = \sum_{j=1}^{n} \Theta(a)[j].$$

This value gives the number of events that occur before event a in terms of the 'happened before' relation. Other events may have occurred in the computation, but there is no knowledge of them at a. $\mathcal{T}$ is the linear logical time and represents the minimum number of logical time steps necessary to reach a (this is the same as the longest causal chain)

$$\mathcal{T} = c(a).$$

The measure is based on the idea that as totally ordered time $\mathcal{T}$ increases in proportion to the number of events that have occurred $|\mathcal{P}|$ the amount of concurrency decreases. The numerator denotes the amount of time 'saved' by the use of concurrency. The denominator represents the amount of time that could be 'saved' if there were unlimited computing resources, assuming that it is not possible to execute it in less than one logical time unit.

To calculate the observed concurrency for the whole computation, it is assumed when the 'outer level processes' terminate, the logical clock is integrated as for subprocesses, and the measure can be calculated.

The range of the measure is $[0,1]$, where 0 represents a totally sequential computation and 1 a totally concurrent computation. In this context, a totally concurrent computation would be one in which there is a process for each event in the computation, since new processes can always be created. Therefore, Fidge's totally concurrent computation is more concurrent than Charron-Bost's definition where there are a fixed number of processes and generally more than one event per process.

Fidge notes that there are similarities between β and ρ, as ρ measures the additional time introduced into the computation by concurrency. β tends to return lower figures than ω or m, since it assumes that more processes can be created. Fidge suggests a more realistic measure that takes number of processors d into account

$$\beta = \frac{|\mathcal{P}| - \mathcal{T}}{|\mathcal{P}| - |\mathcal{P}|/d}.$$

He also shows that β is well behaved with respect to the operators ; (sequential concatenation) and | (parallel concatenation). It has not been evaluated in terms of any other criteria, such as applicability to real systems or usability for comparing distributed algorithms. Calculation of the measure would seem not to be computationally intensive since the measure can be calculated from the last known values of the linear clock and the vector clock.

2.3.6 Raynal, Mizuno and Neilsen's measure of concurrency α

Raynal et al [59] present two measures; $\alpha_e(e)$ that measures how much time is wasted in synchronisation delay to produce event e and $\alpha(C)$ that measures how much time is wasted by synchronisation delay in the whole computation.

Two abstractions CONE and CYLINDER are described. A CONE refers to the events that causally precede a given event. It is a partially ordered set and can be defined in terms of the past of an event

$$\text{CONE}(e) = (\downarrow e) \setminus \{e\} = \{b \mid b \in C, b \preceq e\} \setminus \{e\}.$$

A CYLINDER refers to the partially ordered set associated with a given computation.

There are three values associated with each of the abstractions: **volume, weight** and **height**. The calculation of these measures requires an additional counter to be introduced. Each process P_i contains a counter $W_i \in \mathbb{N}^n$. $W_i[i]$ stores c_i, the value of the linear logical clock; and $W_i[j], i \neq j$ stores the last known c_j value of process P_j. The following rules are used:

1. W_i is initialised to $(0, \ldots, 0)$,

2. before process P_i executes an event, $W_i[i]$ is incremented by one,

3. when a message is sent, it carries the value W_i,

4. when a process P_i receives a message from P_j containing W_j

   (a) $W_i[k]$ is set to $\max(W_i[k], W_j[k])$ for $1 \leq k \leq n$ ($k \neq i$)

   (b) $W_i[i]$ is set to $\max(W_i[i], W_j[j])$ and $W_i[i]$ is incremented by 1.

Also define $\Theta_i(C)$ (respectively, $W_i(C)$) to denote the values of Θ_i (respectively, W_i) when the computation C terminates.

The **weight**(CONE(e)) represents the number of events that causally precede e and the **weight**(CYLINDER(C)) represents the total number of events that occurred in the computation.

$$\textbf{weight}(\text{CONE}(e)) = (\textstyle\sum_{j=1}^{n} \Theta_i(e)[j]) - 1 = |\mathcal{P}|_e - 1$$
$$\textbf{weight}(\text{CYLINDER}(C)) = \textstyle\sum_{j=1}^{n} \Theta_j(C)[j] \;\; (= \sum_{j=1}^{n} q_j)$$

The **volume**(CONE(e)) represents the maximum number of events that could possibly have occurred before e, taking into account the 'holes' or unused time slots formed by synchronisation delays. The **volume**(CYLINDER(C)) represents the maximum number of events that could have occurred in the computation.

$$\textbf{volume}(\text{CONE}(e)) = (\textstyle\sum_{j=1}^{n} W_i(e)[j]) - 1$$
$$\textbf{volume}(\text{CYLINDER}(C)) = n \times \textbf{height}(\text{CYLINDER}(C))$$

The **height**(CONE(e)) represents the number of events on the longest causal path of events leading to e. The **height**(CYLINDER(C)) represents the greatest logical time that occurred in the computation.

$$\textbf{height}(\text{CONE}(e)) = W_i(e)[i] - 1 = \mathcal{T} - 1$$
$$\textbf{height}(\text{CYLINDER}(C)) = \max_{1 \le j \le n} W_j(C)[j]$$

The concurrency measures are defined as

$$\alpha_e(e) = 1 - \frac{\textbf{volume}(\text{CONE}(e)) - \textbf{weight}(\text{CONE}(e))}{\textbf{volume}(\text{CONE}(e)) - \textbf{height}(\text{CONE}(e))},$$

$$\alpha(C) = 1 - \frac{\textbf{volume}(\text{CYLINDER}(C)) - \textbf{weight}(\text{CYLINDER}(C))}{\textbf{volume}(\text{CYLINDER}(C)) - \textbf{height}(\text{CYLINDER}(C))}.$$

The first measure produces a value for a specific event, and the second for an entire computation. The numerator denotes the total synchronisation delay that occurred. The denominator denotes the maximum synchronisation delay possible in the computation. The measures are not defined when the denominator is zero which occurs when only one process is involved in the computation. Raynal et al [59] also define other measures that quantify interaction between specific processes.

The measure allows for the calculation of a measure for a single event, but has not been evaluated in terms of any other criteria. Calculation would seem not to be computationally intensive since the measure can be calculated from the last known values of the clocks.

2.3.7 Other measures

In this section, short summaries of other measures of concurrency will be presented.

Formal languages

Two concurrency measures have been developed for the model defined by Arnold and Nivat [1][11] which deals with regular languages and uses the concept of the homogeneous (Cartesian) product of automata to define a model of concurrent processes. They are based on the concept that the less the processes wait, the more concurrency the computation exhibits. A special letter is introduced into the automata to indicate waiting and the concurrency measure is defined in terms of the number of times this letter occurs, namely the amount of waiting that occurs. The measure of Geniet et al [37] is a probabilistic extension of Beauquier, Bérard and Thimonier's measure [5][12] and is based on a mapping from automata to Markov chains.

Arques et al [2] present a measure of the degree of authorised parallelism for database concurrency control algorithms based on combinatorics of words. It is defined as the ratio of all possible serializable executions to the number of serializable executions permitted by a specific concurrency control algorithm.

Françon [36] presents a measure of parallelism for three different approaches to mutual exclusion. The measure is defined on the behaviours of the system that are defined as words over the alphabet of 'allocate' and 'deallocate'. The three approaches to mutual exclusion allow a number of different behaviours and these are used to calculate the measure.

The measures defined by Françon and by Arques et al (and to some extent the measures of Bérard et al and Geniet et al) represent an interleaved approach (see Section 2.4.2) to concurrency, as they depend on the concept of a shuffle over words[13]. The behaviours of processes are represented as words of actions and the measure of Arques et al compares the maximum number of behaviours with the ones that actually occur.

This relates to an approach suggested by Charron-Bost [14] of counting the number of linear extensions[14] of a given partial order of events. However, she rejects this on the grounds that the number of linear extensions of a particular partial

[11]Cited in [37].

[12]Cited in [37].

[13]The shuffle of two words, w_1, w_2 consists of the set of words obtained by interleaving the letters of w_1 and w_2 while retaining the order inside w_1 and w_2. This can be defined recursively [35] as: $(av_1 \sqcup\!\sqcup\, bv_2) = a(v_1 \sqcup\!\sqcup\, bv_2) \cup b(av_1 \sqcup\!\sqcup\, v_2)$ and $v \sqcup\!\sqcup\, \epsilon = \epsilon \sqcup\!\sqcup\, v = \{v\}$ where ϵ is the empty word.

[14]A linear extension of a partial order is a total order that extends the partial order.

order can not be determined analytically from combinatorics, the complexity of counting linear extensions is not known, and an interleaving approach is not suitable for distributed systems. Pruesse and Ruskey [54] have presented an algorithm to generate linear extensions that runs in constant amortised time[15]; however, in the worst case there is an exponential number of linear extensions. This will be discussed further in Chapter 6.

Graph theoretical approach

Barbosa [4] investigates the effect of k-capacity channels on the amount of concurrency using a graph theoretical approach. The reciprocal of the multi-chromatic index of the graph G is defined to be the measure of concurrency. Barbosa shows that the gain in concurrency obtained by replacing 0-capacity channels with k-capacity channels can be described by a bound that is usually small and tends to 1 as the system grows.

Queuing theoretical approach

Bambos and Walrand [3] define a degree of parallelism within the area of queuing theory. They assume an infinite number of processors; however, there are precedence constraints on the jobs being processed that produce a queuing phenomenon. The degree of parallelism is defined to be the asymptotic average number of processors that work concurrently and it is equal to the average quantity of work that enters the system in one period.

Execution of program statements

Kumar [45] notes that many measures relate specifically to the architecture they are being applied to and they measure the parallelism that is present in that specific architecture, and frequently only the reduction in completion time as a function of the number of processes is used to characterise parallelism. He has developed a software tool COMET to determine the maximum possible concurrency in a FORTRAN program. It keeps track of the earliest time that a statement can be executed by determining when the values that it depends on are computed. The measure is expressed in terms of the number of FORTRAN statements that can be executed

[15] A generation algorithm runs in constant amortised time if it runs in $O(N)$ where N is the number of objects generated

at one time. This value changes during the execution of the program, and the author notes that the instant parallelism can be several orders of magnitude larger or smaller than the average parallelism.

Parallel program execution times

This section discusses those measures that are associated with parallel program performance. An important measure for parallel performance is speedup which is a ratio of the time taken to execute a program sequentially to the time taken to execute it in parallel. It measures concurrency indirectly because it only provides an idea of the amount of concurrency within a program without taking the structure of the computation into account. It is debatable whether these performance measures achieve the same aims as concurrency measures.

Quinn [57, Chapter 2] defines the speedup S for a parallel algorithm running on a parallel computer with n processors as

$$S = \frac{T_1}{T_n}$$

where T_1 is the time taken for the fastest sequential algorithm to run on that computer using one processor and T_n is the time for the parallel algorithm to run on n processors. Other definitions take T_1 to be the time taken for the parallel algorithm to run on one processor, or the time taken for the fastest sequential algorithm to run on the fastest sequential computer. Thus speedup indirectly measures the average amount of parallelism by comparing the difference in execution times.

Eager et al [29] defines the average parallelism to be the average number of processors that are busy during the execution of the program, given an unlimited number of processors and this is equivalent to speedup with unlimited processors. This can be seen as an indirect measure of the maximum concurrency.

Sevcik [63] investigates a number of ways to characterise parallelism and investigates their use in scheduling. He defines the following parameters:

- f : the fraction sequential, i.e. the fraction of the program that must be executed sequentially.
- A : average parallelism as defined by Eager [29].
- m : the minimum parallelism, i.e. the minimum number of processors used during the computation.

- M : the maximum parallelism, i.e. the maximum number of processors used during the computation.
- F : the fraction spent at maximum parallelism.
- V : the variance in parallelism.

His research has shown that it is not sufficient to use a single parameter in characterising a parallel program for scheduling purposes. It should be noted that none of the parameters considered fully capture the structure of the computation in a general sense, although the concept of using a number of parameters to measure concurrency has merit.

An approach to determining execution time that deals with task precedence and random task execution times has been suggested by Robinson [61]. A task graph consists of nodes that represent tasks and arcs that describe the precedence constraints between tasks; the task graph describes a partial order of tasks. Bounds on the expected execution time can be determined from the random variables associated with each task. The expected execution time can then be used to determine an expected speedup. This approach moves closer to the idea of a concurrency measure, but its focus remains on comparing execution times.

Amdahl's Law [57] although not a measure, is an important result that gives a bound on the speedup as

$$S \leq \frac{1}{f + (1 - f)/n}$$

where f represents that proportion of the program that is inherently sequential, also known as the serial fraction [43] or the fraction sequential [63]. This law describes how one aspect of the structure of the computation can limit the speedup, although it is not general enough to take into account the whole structure.

Other authors [40, 43] have disagreed with the extremely limiting nature of this law and suggested other approaches. Gustafson [40] notes that f decreases as n increases, because in real applications the parallel part of the computation grows with the number of processors but the sequential component does not grow. Karp and Flatt [43] suggest that a new metric for parallel programs should be defined in terms of a serial fraction which is determined experimentally.

However, neither of these suggestions for modification of Amdahl's Law remove the fact that this law is based on timing and architecture. Amdahl's law and modifications of it therefore differ from the concept of a concurrency measure where the

- M : the maximum parallelism, i.e. the maximum number of processors used during the computation.
- F : the fraction spent at maximum parallelism.
- V : the variance in parallelism.

His research has shown that it is not sufficient to use a single parameter in characterising a parallel program for scheduling purposes. It should be noted that none of the parameters considered fully capture the structure of the computation in a general sense, although the concept of using a number of parameters to measure concurrency has merit.

An approach to determining execution time that deals with task precedence and random task execution times has been suggested by Robinson [61]. A task graph consists of nodes that represent tasks and arcs that describe the precedence constraints between tasks; the task graph describes a partial order of tasks. Bounds on the expected execution time can be determined from the random variables associated with each task. The expected execution time can then be used to determine an expected speedup. This approach moves closer to the idea of a concurrency measure, but its focus remains on comparing execution times.

Amdahl's Law [57] although not a measure, is an important result that gives a bound on the speedup as

$$S \leq \frac{1}{f + (1-f)/n}$$

where f represents that proportion of the program that is inherently sequential, also known as the serial fraction [43] or the fraction sequential [63]. This law describes how one aspect of the structure of the computation can limit the speedup, although it is not general enough to take into account the whole structure.

Other authors [40, 43] have disagreed with the extremely limiting nature of this law and suggested other approaches. Gustafson [40] notes that f decreases as n increases, because in real applications the parallel part of the computation grows with the number of processors but the sequential component does not grow. Karp and Flatt [43] suggest that a new metric for parallel programs should be defined in terms of a serial fraction which is determined experimentally.

However, neither of these suggestions for modification of Amdahl's Law remove the fact that this law is based on timing and architecture. Amdahl's law and modifications of it therefore differ from the concept of a concurrency measure where the aim is to evaluate the concurrent structure of algorithms independently of execution times on specific architectures.

2.4 CCS

In this section, work will be presented that pertains to Milner's Calculus of Communicating Systems [50]. The operators that will be used in evaluating the measure of concurrency m will be discussed and the definition of an algebraic calculus of processes will be given to facilitate the discussion of partial orders and time in CCS.

An overview of CCS is presented in Appendix A. This appendix is presented to give an overview of the topic for the reader with knowledge of CCS, and to detail the CCS notation that will be used in this research report.

2.4.1 Operators

As discussed earlier, a criterion for the evaluation of concurrency measures is that of compatibility with operators. CCS offers two combinators or operators in the basic set of combinators that can be used for obtaining new agents [50]:

- Composition which allows the combination of two agents in parallel,
- Prefix which allows the prefixing of an agent by an action or sequence of actions. This operator differs from the other types of sequential operators that were presented earlier, since it cannot be used to put two agents in sequence.

Milner has defined a further two operators [50, Chapter 8]. They are *Before* and *Par* and have the following definitions:

Definition 1

$$\begin{aligned} P\ \textit{Before}\ Q &\stackrel{\text{def}}{=} (P[b/\text{done}] \mid b.Q)\backslash b \\ P\ \textit{Par}\ Q &\stackrel{\text{def}}{=} (P[d_1/\text{done}] \mid Q[d_2/\text{done}] \mid \\ &\qquad (d_1.d_2.\text{done}.0 + d_2.d_1.\text{done}.0))\backslash\{d_1, d_2\} \end{aligned}$$

These definitions are only meaningful when P and Q are well-terminating agents, because they rely on the presence of a $\overline{\text{done}}$ label in the case of an agent that terminates

Definition 2 *P is* well-terminating *if, for every derivative P' of P, $P' \xrightarrow{\text{done}}$ is impossible and also if $P' \xrightarrow{\overline{\text{done}}}$ then $P' \sim \overline{\text{done}}.\mathbf{0}$.*

These four operators will be used in the evaluation of the chosen measure.

2.4.2 Algebraic calculi of process and partial orders

The components of an algebraic calculus are defined in [8] as follows:

- a *syntax* consisting of a family of operators defined over a set of labels,
- an *operational semantics* specifying the behaviour of the operators which results in a *transition system* (it is also possible to give a denotational semantics, for example, Hoare's CSP [11] and Hennessy's ATP [41]),
- the semantics given by *equivalences* compatible with the algebraic structure. These equivalences equate processes with similar behaviours.

Algebraic calculi for representing concurrency and nondeterminism can be divided into two main groups [27]: those that represent concurrency by interleaving and those that represent concurrency by some form of 'true concurrency', such as the partial ordering of events.

CCS is an example of the first group—concurrency is represented by the fact that events can occur in any order, and this means that the parallel operator | is not primitive with respect to observational congruence, as shown by the Expansion Law. Any agent containing | can be equated with respect to behaviour to another agent expressed in terms of the other operators, for example,

$$a.b.\mathbf{0} + b.a.\mathbf{0} = a.\mathbf{0} \mid b.\mathbf{0}.$$

These formalisms assume the existence of a global clock and global states [27]. Advocates of the interleaving approach admit that for some applications this assumption is not realistic; however, all formalisms have some simplification and this particular simplification allows for mathematical models of large concurrent systems that are amenable to specification and verification [66].

Calculi[16] in the second group represent concurrency by an absence of ordering—two events are concurrent if they are causally independent, and it is argued that

[16]The calculi based on a 'true concurrency' approach will be referred to as having a causal semantics, because the causality between events is described in such a formalism. This term includes 'partial ordering semantics' and 'noninterleaving semantics'.

sequential nondeterminism should be distinguished from concurrency to understand properties such as liveness in concurrent systems [27]. There is no assumption of a global clock or linear timescale in these models and behaviours of the system are expressed as causal relations between actions performed by distributed subprocesses [27, 39].

A number of different approaches have been taken in providing CCS with a causal semantics. Best and Devillers [7], Goltz and Mycroft [39] and Hirschfield et al [42] have used Petri nets to investigate causal models for CCS. Castellani and Hennessy [13] define a distributed labelled transition system that distinguishes the local and global results of a transition and from this they define a distributed bisimulation that distinguishes concurrent processes from nondeterministic sequential processes. Their approach is based on an observational view similar to Milner's but assumes a number of observers where each can only see the actions in a specific subprocess. Boudol and Castellani define algebras [9, 10], where labels on the transitions allow for the unique identification of each transition and hence allow for the definition of a concurrency relation. Degano, De Nicola and Montanari [22, 23, 24, 25, 26] present a number of approaches to providing CCS with a causal semantics. In their latest work, they define a new operational semantics based on a partial ordering derivation relation [27]. Many of these approaches involve decomposing a CCS agent into a set of sequential subprocesses and the transition relation describes how these sets are transformed.

The issues discussed above relate to the semantics of CCS, which I will not be dealing with in this research. However, it may be possible for vector clocks to be used to define a partial ordering semantics for CCS, and this will be discussed further in Chapter 6.

2.4.3 Time in CCS

The concept of logical time has not been applied to CCS. Tofts [65] investigates a temporal model that is an extension of CCS that is more concerned with real-time issues and ensuring that events occur at certain instants than issues of partial ordering. Transitions are divided into action transitions that represent actions of no duration, and temporal transitions that represent the passage of time. Two extensions to CCS are defined—weakly timed CCS and strongly timed CCS. A new equivalence, time-equivalence $\sim_T$ is defined for strongly timed CCS.

Quemada and Fernandez [56][17] introduce time into LOTOS, a data communication protocol specification language based on CCS. T'Hooft [64] demonstrated that Timed LOTOS was not suitable for the description of timers and time out mechanisms. This attempt at introducing time seems mainly aimed at timing events.

Other work in the area of time and concurrency theory includes the timed model of CSP presented by Reed and Roscoe [60], the real-timed concurrency theory of Murphy [53], and the real-time consistent equivalence of van Glabbeek and Vaandrager [66].

2.4.4 Measuring concurrency in CCS

Moller [52] in his PhD thesis, presents a measure of concurrency for CCS. He considers the subset of CCS containing **0**, Prefix, Summation and Composition and attempts to define a normal form for process terms as a product (parallel composition) of terms

$$\prod_{1 \leq i \leq n} P_i \ (n \geq 0)$$

where each P_i represents a prime process and is in some prime normal form such as

$$\sum_{1 \leq i \leq n} a_i.p_i \ (n > 0)$$

where each p_i is in normal form. He shows that a unique decomposition theorem can be proved for this subset of CCS with or without communication, with respect to strong and weak observational congruence. Each agent can be expressed a unique product of primes

$$P = \prod_{1 \leq i \leq m} \Big(\sum_{1 \leq j \leq n_i} a_{ij}.p_{ij} \Big).$$

This allows for the following definition of a measure of the measure of maximum parallelism as

$$\mathrm{maxpar}(P) = \sum_{1 \leq i \leq m} \Big(\max_{1 \leq j \leq n_i} \mathrm{maxpar}(p_{ij}) \Big).$$

This concept is presented at the beginning of Moller's thesis on axioms for concurrency and to the best of my knowledge has not been investigated further.

[17] Cited in [64].

2.5 Choice of measure

In this section, the justification for the measure chosen to be applied to CCS will be presented. The focus in this chapter has been on measures defined in Lamport's message-passing formalism of a distributed system for the reason that this formalism seems more applicable to CCS. The other frameworks in which measures have been defined are less suitable for a number of reasons which will be discussed below after which the measures defined in Lamport's message-passing formalism will be discussed.

Formal languages

The measures based on the Arnold-Nivat model [1] represent concurrency by an m-tuple of actions occurring in one timestep and therefore these measures would be more applicable to the synchronous calculus defined by Milner [50, Chapter 9]. Degano et al [25] have noted that the multiset labels found in Synchronous CCS give a direct representation of the amount of parallelism.

The other two measures based on formal languages relate to specific areas (database concurrency control and mutual exclusion) and are not obviously generalisable to a wider setting.

Graph theoretical approach

Barbosa's work [4] relates specifically to the issue of size of channel and its effect on concurrency, therefore this measure is difficult to generalise to allow the investigation of other aspects of concurrent systems.

Queuing theoretical approach

Bambos and Walrand [3] deal with queuing theory which is not applicable to CCS, mainly because of the emphasis on service times that are associated with each job. In CCS, the only timing assumption is that 'concurrent agents proceed at indeterminate relative speeds' [50, p. 195]. Time has been applied to CCS as mentioned in Section 2.4.3, but these variants of CCS are not under consideration in this research, as the aim is to investigate a measure of concurrency for CCS in its original form.

Execution of program statements

Kumar's concurrency measurement tool [45] is applicable to FORTRAN programs and looks at the number of statements that can be executed at once. Again this approach seems more applicable to Synchronous CCS, as the number of program statements relates to the number of multiset labels as described by Degano et al [25].

Parallel program execution times

The work that relates to performance measures for parallel programs [57, 29, 40, 43, 63] is not applicable to CCS for two reasons; first, as mentioned above, the only timing assumption in CCS is that nothing can be said about the relative speeds of processes; and second, speedup does not take into account the structure of the computation because it measures concurrency only indirectly by comparing execution times.

Moller's measure of maximum parallelism

In Section 2.4.4, Moller's measure of concurrency in CCS is defined. The investigation of this measure relates to finding an algorithm to decompose CCS agents into normal forms—a research issue which I have chosen not to pursue, although the application of this measure remains an unexplored area of research. A second reason for not choosing this measure is the fact that it measures the maximum parallelism that an agent can have without taking into account the effect of communication and therefore it can not be used to evaluate realistic algorithms.

Measures defined in the message-passing formalism

In Section 2.3.1, a number of measures were presented in Lamport's framework. All of these measures have not yet been fully evaluated and therefore it would be worthwhile investigating any of them to determine their strengths and weaknesses. As will be shown, it is relatively easy to map from CCS to the message-passing formalism. There is a good match between the two models; it is easy to identify events and processes in both, and neither model makes assumptions about the relative speeds of processes.

Charron-Bost's measures ρ_i and ρ [15] assume real times associated with each event, and for similar reasons given above, these measures were not chosen for the focus of this research. Charron-Bost's measure ω [14] has not been chosen because it is described as inaccurate and because m is an extension of it.

Three measures are therefore available for evaluation by application to CCS. They are α, the measure of Raynal et al [59]; β, Fidge's measure [32]; and Charron-Bost's measure m [14].

There is conflicting evidence about m. It is described as expensive to compute [59, 44]; however it has recently been used in the literature as a model of measuring concurrency [44]; hence it is important to fully evaluate m and determine if it is a suitable measure of concurrency. By applying the measure to CCS, more operators will become available to evaluate the compatibility of m with these operators. Finally, by testing m for usability and applicability, the worth of m for use with real examples can be determined.

The approach taken in this research will be general and will allow for the future evaluation and comparison of other measures defined in the same framework.

2.6 Summary

In this chapter, a number of measures of concurrency were presented. The majority are defined in the message-passing formalism that describes distributed systems in terms of the partial ordering of events in processes. Amongst these is the measure m which will be investigated in this research. Other measures that were described fall into the models of concurrent systems based on formal languages, graph theory, queuing theory and parallel program execution times.

A number of criteria for the evaluation of measures of concurrency were presented and discussed. They were divided into two groups—objective and subjective criteria. An important criterion is that of compatibility with operators and a framework was developed to discuss the different approaches presented in the literature.

Work relating to partial ordering and time in CCS was described and a measure of maximum parallelism for CCS agents was detailed. This measure is defined for agents in a specific normal form.

Justification was given for the choice of m, although it was noted that there are a number of measures that require an evaluation.

In the next chapter, the message-passing formalism will be redefined to accommodate synchronous communication and nested parallelism, so that m can be applied to CCS agents.

3. Applying the measure to CCS

3.1 Introduction

The aim of this chapter is to develop the necessary theory that will allow for the application of the measure m to CCS. This will involve the identification of differences between the message-passing formalism in which the measure is defined and CCS. Once these differences have been detailed, it will be possible to redefine the message-passing formalism so that the justification for the measure remains and so that CCS can be translated into the redefined formalism. This process of redefinition will involve changing some basic features of the formalism and proving that the results still hold in this new formalism. Finally an algorithm can be developed to translate CCS into the new formalism. This will involve determining which subset of CCS can be measured.

3.2 The message-passing formalism and CCS

In this section, the existing formalism and CCS will both be briefly presented as well as a discussion of the differences. Then an outline of the necessary steps to apply the measure to CCS will be given.

As discussed in Chapter 2, Charron-Bost[1] describes the following framework:

- A formal definition of a distributed system and computation.
- A partial ordering of events in a distributed computation.
- The concept of a consistent cut.
- Vector clocks applied to the distributed system.
- The definition of a measure of concurrency in terms of consistent cuts.
- A combinatorial characterisation of the number of consistent cuts in a totally sequential computation and in a totally concurrent computation.

[1]Charron-Bost [14] will be used as my main reference although the material on logical time and vector clocks has been presented in [33, 46, 49, 62].

- Two methods for calculating the consistent cuts—one relating to conditions on the vector clock and the other relating to antichains of the partial order.

Milner [50] provides the following[2]:

- A syntax of operators over a set of events.
  - Nil **0**, Prefix $a.$, Summation $+$, Composition $|$, Restriction $\backslash a$, Relabelling $[f]$, Constant (or Recursion $\mathbf{fix}(X = E)$).
- Operational semantics, specifying the behaviour of operators which results in a transition system, namely the transition rules:
  - **Act**, **Sum**, **Com**, **Res**, **Rel**, **Con**.
- Semantics given by an equivalence, for example strong equivalence and observation equivalence.

In this research, a measure of concurrency for CCS will be defined and it will involve mapping the syntax and indirectly the operational semantics to a formal model of a distributed system. However, it will not involve the second level of semantics.

First it is necessary to determine the differences between the message-passing formalism that Charron-Bost presents and the model of CCS. The major differences can be detailed as follows:

1. Charron-Bost's formalism assumes a fixed number of processes that all effectively start at the same time, and that no new processes are created during the computation. In CCS, certain agents can be interpreted as involving process creation. For example $a.b.(c.\mathbf{0} \mid d.\mathbf{0})$, starts as one process that performs a and b, and then 'branches' into two processes, one that performs c and one that performs d. Therefore, in each case, there are a fixed number of processes, however, some processes can only start when other processes have finished.

2. Charron-Bost's formalism assumes asynchronous point-to-point communication, so each message has a send process P_i and receipt process P_j associated with it, and no other process can be involved in the communication. In CCS, communication occurs between a port a and its complement $\overline{a}$ and this communication is synchronous. As in Charron-Bost's formalism, only two partic-

[2]See Appendix A for an overview of CCS

ipants are permitted. Note that in both cases, the content of the messages is ignored[3].

3. There is a difference in expressive power with respect to computations. The message-passing formalism represents one computation of an algorithm whereas a CCS agent can describe an entire algorithm. Charron-Bost is not clear about how computations relate to algorithms and in her presentation she deals with individual distributed computations that are composed of individual sequential computations. This lack of clarity extends to defining the measure for an algorithm; she suggests briefly that a measure can be determined for an algorithm by calculating the average across the values found for each computation.

The steps in the process of applying Charron-Bost's measure of concurrency to CCS are as follows:

1. Redefine Charron-Bost's message-passing formalism to take into account process creation and synchronous communication.

2. Prove all proofs with necessary modifications as given by Charron-Bost in the new formalism.

3. Show that intuitively the basic idea of a measure of concurrency still holds in this formalism.

4. Discuss the difference in expressive power between the message-passing formalism and CCS, and determine how this affects the measurement of concurrency.

5. Present an algorithm for translating CCS into this message-passing formalism. This involves investigating various issues that relate to measuring concurrency in CCS, for example:

   - What is a totally sequential agent?
   - What is a totally concurrent agent?
   - Can all CCS agents be measured or only a subset. If so, which subset?

6. Use the algorithmic translation as a basis for the implementation of the concurrency measurement tool.

[3]This research deals with the basic calculus of CCS, not the value-passing calculus. This is not a major issue as it can be shown that the two are equivalent, so the method developed here will extend to the value passing calculus.

3.3 Discussion

In this work, I will take a different approach to defining synchronous communication in the message-passing formalism to that presented in the literature.

Charron-Bost et al and Fidge [19, 33] define synchronous communication in the message-passing formalism in a manner which I have found to be problematic when using consistent cuts. The concept of consistent cuts has not been dealt with in the framework of synchronous communication except in [44] where it is dealt with briefly.

The definition for synchronous communication given in the literature results in the two events that form a synchronous communication event being concurrent, which means that they are not related by the partial ordering relation. As described in Chapter 2, a consistent cut is defined to be the left closure of a cut with respect to the partial order of events. When this definition of a consistent cut is applied to the situation where synchronous communication events are defined to be concurrent, it is possible for a cut to be consistent but to only include one of the two events. This contradicts the intuitive idea that a consistent cut represents a consistent global state in the computation [33]. (Note that in the case of asynchronous communication, the send event can occur in a consistent global state without the corresponding receive event because the send event is dependent on the receive event; however for synchronous communication, both events must be in the global state for it to be consistent because the two events are mutually dependent.)

Another anomaly occurs in the situation described above. The partial order information about pairs of synchronous communication events describes them as concurrent, however their vector clock information is identical [34]. This means that the vector clock characterisation of consistent cuts given by Charron-Bost [14] identifies different consistent cuts to those identified by the left closure definition in the synchronous case. It also implies that the vector clocks contain more information of the relationship between events than does the partial order.

My approach will be to equate the two events that form a synchronous communication event. With this definition the partial ordering of events is retained, however the left closure definition of a consistent cut together with the equating of communication events means that if one event occurs in the cut, the other event must also appear in the cut. This approach will also result in the vector clocks containing only the information of the partial order, and hence the vector clock characterisation of consistent cuts will identify the same cuts as the left closure definition.

The idea of equating the events in synchronous communication was partly derived from the fact that in CCS, when communication occurs between two agents that are concurrent, their two complementary actions are transformed to a single τ action. Taking the approach that synchronous communication represents a single event therefore allows a closer match between CCS and the message-passing formalism. The idea also arose from the work of Charron-Bost et al [19] where although they use the definition that results in the two events involved in a synchronous communication event being concurrent, they also make statements such as '...in the synchronous case a pair of corresponding send-receive events can be regarded as a single combined communication event...' [19, p. 25] and '...they behave as if they are glued together and can only occur together' [19, p. 15].

This approach has one disadvantage in that the value of the number of cuts in a totally sequential computation must be redefined to take into account that some pairs of events are reduced to single events. This will discussed further in Section 3.4.4.

3.4 New message-passing formalism

In this section, a new message-passing formalism will be defined to take into account the differences mentioned above. This exposition will have the same structure as Charron-Bost's paper [14]. A summary of the original theoretical framework that Charron-Bost uses was presented in Chapter 2, in the Sections titled **The 'happened before relation', Logical clocks** and **Charron-Bost's measure of concurrency m.**

3.4.1 Computation of a distributed system

A computation of a distributed system consists of a finite set of processes which are denoted $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$. In Charron-Bost's formalism, each P_i is a set of finite sequences C_i of events and is prefix closed. In this new formalism, it will be assumed that each P_i represents only one specific finite sequence of events. This means that the property of prefix closure is no longer applicable to the P_i's. However, this does not affect the definition of the concurrency measure, and allows for a simpler definition of process creation.

$S \subseteq \mathcal{P}$ gives the processes that start at the beginning of the computation. All processes that are in $\mathcal{P} \backslash S$ are caused by other processes and the relation $\mapsto$ is used

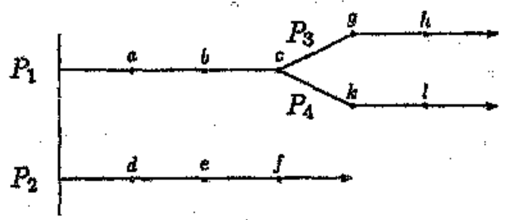

Figure 3.1: Example of a distributed computation.

to describe this. The fact that there are a number of processes that can proceed simultaneously allows for the expression of concurrency in this formalism.

The notation $P_i \mapsto P_j$ is used when process P_i causes process P_j, $(i \neq j)$. Each process in $\mathcal{P} \backslash \mathcal{S}$ is caused by exactly one process[4]. However, each process may cause none, one or many processes. Define $\mathcal{F} \subseteq \mathcal{P}$, the set of processes that do not cause any processes, i.e. the 'last' processes.

For example, let $\mathcal{P} = \{P_1 \ldots, P_7\}$, with $\mathcal{S} = \{P_1, P_2\}$ and $\mathcal{F} = \{P_4, P_6, P_7\}$, then if P_1 causes P_3 and P_4, P_2 causes P_5, P_3 causes P_6, and P_5 causes P_7, then this could be expressed as

$$P_1 \mapsto P_3,\ P_1 \mapsto P_4,\ P_2 \mapsto P_5,\ P_3 \mapsto P_6,\ P_5 \mapsto P_7.$$

Each P_i is a finite sequence of events. Let $a, b, c, \ldots$ denote events. It is assumed that events can be distinguished from each other. The set of events that occur in specific set of processes $\mathcal{P}$ or a specific process P_i is denoted by $E(\mathcal{P})$ and $E(P_i)$ respectively. The number of events in the sequence P_i is denoted q_i, i.e. $\mid E(P_i) \mid = q_i$.

The relation $\prec$ can be defined on the set of events $E(\mathcal{P})$ for a specific set of processes $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ as follows:

1. if process $P_i = a_1 a_2 \ldots a_{q_i}$ and a_j occurs before a_k in the sequence, i.e. $j < k$, then $a_j \prec a_k$,

2. if process $P_i = a_1 a_2 \ldots a_{q_i}$ and $P_j = b_1 b_2 \ldots b_{q_j}$, and $P_i \mapsto P_j$ then $a_{q_i} \prec b_1$.

Define $\preceq' = (\prec \cup =)^*$, the transitive closure of $\prec$ and $=$. Then $\preceq'$ defines a partial order[5] on $E(\mathcal{P})$. (See Appendix B for the proof of this.) The relation $\preceq'$ captures the causality between events; if $a \preceq' b$ then b is causally dependent on a.

[4]This means that each process can only occur once in a computation.

[5]A partial order is a relation that is reflexive, antisymmetric and transitive [20]. Antisymmetry has the following definition: if $a \preceq' b$ and $b \preceq' a$ then $a = b$.

Consider the following: $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$, $\mathcal{S} = \{P_1, P_2\}$, $\mathcal{F} = \{P_2, P_3, P_4\}$ and $P_1 \mapsto P_2$, $P_1 \mapsto P_3$. Each process is defined as a sequence of events: $P_1 = abc$, $P_2 = def$, $P_3 = gh$, $P_4 = kl$. This is expressed diagrammatically in Figure 3.1. The diagram captures the information of the partial order $\preceq'$; if there is a line going from left to right between a process e_1 and e_2 then $e_1 \preceq' e_2$. In the example, $a \preceq' g$.

As yet the formalism described does not allow for communication between processes. In this formalism, only synchronous communication will be represented. Certain pairs of events represent synchronous communication events (this communication can only occur between pairs of events) and an event can only participate in one such communication. Note also that the events must occur in different processes. The relation $\leftrightarrow$ will be used to describe this. If a and b are the pair of events in a synchronous communication, then $a \leftrightarrow b$. This relation is symmetric and has the property that

$$\forall a, b, c \in E(\mathcal{P}), c \neq a, c \neq b,\ a \leftrightarrow b \Rightarrow \neg(a \leftrightarrow c) \wedge \neg(b \leftrightarrow c).$$

It is necessary to redefine $\preceq'$ to include communication. This will be done by the following rule

$$\forall a, b \in E(\mathcal{P}),\ a \leftrightarrow b \Rightarrow a = b.$$

This will retain the partial ordering information of $\prec$ and add the new information given by the $\leftrightarrow$ relation by equating events involved in communication.

Then the relation $\preceq$ which includes communication can be defined as $(\prec \cup =)^*$ or the transitive closure of $\prec$ and $=$; however $=$ now includes the information from $\leftrightarrow$. The relation $\preceq$ gives a partial order on $E(\mathcal{P})$. (See Appendix B for the proof of this.) The transitivity captures the causality that results from communication. For example, if $a, b \in E(P_i)$ and $c, d \in E(P_j)$, with $a \preceq b$ and $c \preceq d$, then if $b \leftrightarrow c$, b and c are equated and $a \preceq d$.

$\mathcal{P}$ is called a computation when $(E(\mathcal{P}), \preceq)$ as defined by $\mapsto$ and $\leftrightarrow$ contains no cycles. (Charron-Bost includes in her definition of a computation the additional condition that for each receipt of a message m there is a (single) sending of m. In the new formalism, this has been dealt with above in the definition of $\leftrightarrow$, so that by definition the two components of a synchronous communication event are included in a computation.)

Two events are concurrent when they are not related by $\preceq$. Formally, a and b are concurrent (a co b) if $\neg(a \preceq b)$ and $\neg(b \preceq a)$. Note that this definition implies

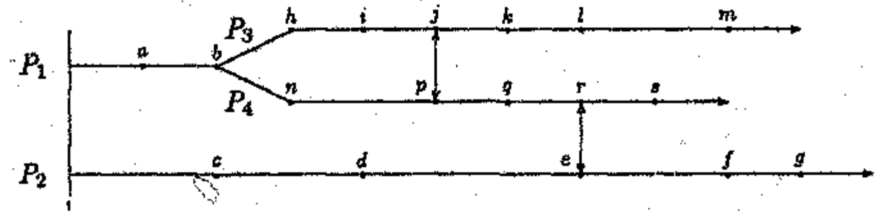

Figure 3.2: An example of a distributed computation with synchronous communication.

that a and b are not in the same process, or in different processes that are related by $\mapsto$.

For example, consider the following: $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$, $\mathcal{S} = \{P_1, P_2\}$, $\mathcal{F} = \{P_2, P_3, P_4\}$ and $P_1 \mapsto P_2$, $P_1 \mapsto P_3$. Each process is defined as a sequence of events: $P_1 = ab$, $P_2 = cdefg$, $P_3 = hijklm$, $P_4 = npqrs$ and the communication events are $j \leftrightarrow p$ and $r \leftrightarrow e$. This is shown in Figure 3.2. This diagram captures partial order $\preceq$; if there is a directed path joining event e_1 and e_2, then $e_1 \preceq e_2$. For example, $i \preceq s$ and $i \preceq e$. Events i and d are concurrent.

3.4.2 Cuts

The measure m is based on the number of consistent cuts in a computation and for this reason the definition of a cut and a consistent cut are required. This definition has a similar intuitive basis as in the asynchronous case; however, for consistency to hold, both events that take part in a synchronous communication event must be in the cut.

Charron-Bost defines a cut with respect to all processes: for each index i, let a_i be an event in P_i, then

$$C = \bigcup_{i \in \{1,\ldots,n\}} \{x \in E(P_i) \mid x \preceq a_i\}$$

is a cut of the computation.

There is a problem with this definition in that it does not allow a cut that excludes some processes, i.e. processes where no events have occurred at the place of the cut, for example, the empty cut.

For example, if $P_1 = ab, P_2 = cd$, and $P_3 = ef$, then it should be possible to obtain the cut $\{c, e, f\}$ from the definition ($\{c, e, f\} = \{x \in E(P_1) \mid x \preceq c\} \cup \{x \in E(P_2) \mid x \preceq f\}$) (see Figure 3.3(a)). However, this is not possible because the

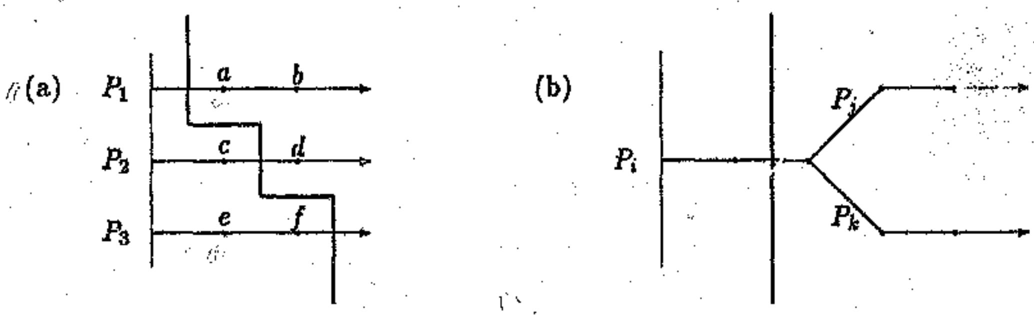

Figure 3.3: (a) An example of a cut in Charron-Bost's formalism. (b) An example of a cut in the new formalism.

definition requires that an event is chosen from each process. This is only a minor problem in Charron-Bost's work[6], but it becomes important when introducing the causal relation between processes, $\mapsto$. For example, given $P_i \mapsto P_j$ and $P_i \mapsto P_j$, it should be possible to cut at an event in P_i and at no event in P_j or P_k, because this would satisfy the intuitive idea of a [illegible] see Figure 3.3(b)).

The definition can be modified [illegible]ows: let $N = \{1, \ldots, n\}$, and let $E_C \subseteq N$. E_C will contain the indices of the processes that do not contribute any events to the cut C. Then for each $i \in N \setminus E_C$, choose an $a_i \in E(P_i)$, then

$$C = \bigcup_{i \in N \setminus E_C} \{x \in E(P_i) \mid x \preceq a_i\}$$

is a cut of the computation.

A consistent cut C of a computation is defined as follows: C is a consistent cut if it is a cut that is left closed by $\preceq$, namely, if $a, b \in E(\mathcal{P})$ such that $b \in C$ and if $a \preceq b$ then $a \in C$.

The past of an event a, is defined as follows: $(\downarrow a) = \{x \in E(\mathcal{P}) \mid x \preceq a\}$. It is a consistent cut. The following notation will be used for the past of event a in process P_i: $(\downarrow a)_i = (\downarrow a) \cap E(P_i) = \{x \in E(P_i) \mid x \preceq a\}$. Note that is not necessarily a consistent cut. Note, however, that it is a totally ordered set, as all events come from one process, and are ordered by $\prec$.

The definition of a cut given above can therefore be rewritten as

$$C = \bigcup_{i \in N \setminus E_C} \{x \in E(P_i) \mid x \preceq a_i\} = \bigcup_{i \in N \setminus E_C} (\downarrow a_i)_i.$$

[6]Charron-Bost's proof of the result concerning the checking of a cut for consistency using vector clocks is not complete as it omits the cuts that are not defined by n events.

Consider the example given in Figure 3.2 (p. 46)

- Let $E_C = \{3,4\}$, $a_1 = b$, $a_2 = c$, then $C = \{a,b,c\}$ and it is a consistent cut.
- Let $E_C = \emptyset$, $a_1 = b$, $a_2 = e$, $a_3 = l$, $a_4 = s$,
  then $C = \{a,b,c,d,e,h,i,j,k,l,n,p,q,r,s\}$. It is a consistent cut.
- Let $E_C = \{1\}$, $a_2 = d$, $a_3 = i$, $a_4 = q$, then $C = \{c,d,h,i,n,p,q\}$ is a cut. It is not a consistent cut since $b \preceq h$ and $b \notin C$.

As further examples, $(\downarrow q) = \{a,b,h,i,j,n,p,q\}$, $(\downarrow q)_1 = \{a,b\}$, $(\downarrow q)_2 = \emptyset$,
$(\downarrow q)_3 = \{h,i,j\}$, $(\downarrow q)_4 = \{n,p,q\}$.

3.4.3 Clocks

In this section, the theoretical background for vector logical clocks will be presented. First the clocks will be defined, then some results will be proved. The first result shows that the vector clock of an event gives the number of events that have happened in that process before the event under consideration. The next two results show how vector clocks can be used to characterise concurrency.

Partially ordered logical clocks are represented as vectors in $\mathbb{N}^n$:

- If $v \in \mathbb{N}^n$ then $v[i]$ represents the ith component of v.
- $\mathbb{N}^n$ is partially ordered by $\leq$; $a \leq b$ iff $a[i] \leq b[i]$ for each index i.
- If $u, v \in \mathbb{N}^n$, then $w = \sup(u,v)$ is defined as follows: for each index i, $w[i] = \max(u[i], v[i])$.

Clocks are applied to the message-passing formalism described above as follows:

- Each process P_i has a clock Θ_i that takes values in $\mathbb{N}^n$.
- $\Theta_i(a)$ represents the time of event a in process P_i.
- Initial values of clocks
  - If $P_i \in S$, then the initial value of the clock Θ_i is $(0,\ldots,0)$.
  - If $P_i \in \mathcal{P} \backslash S$ and $P_k \mapsto P_i$, then let a be the last event in P_k, clock Θ_i takes the value of the clock $\Theta_k(a)$.
- Updating values of clocks

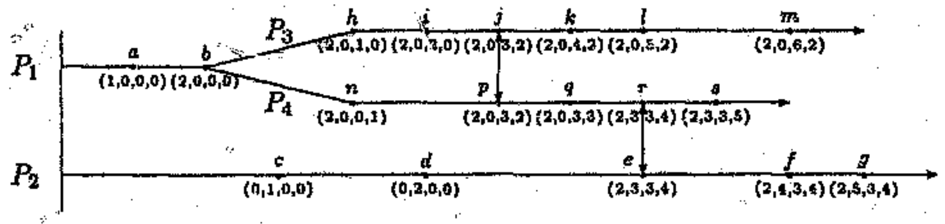

Figure 3.4: A distributed computation with vector clocks.

  - For each event in P_i, $\Theta_i[i]$ is incremented by 1.
  - If $a \leftrightarrow b$ is a synchronous communication event and a occurs in P_i and b in P_j, then both Θ_i and Θ_j are set to $\sup(\Theta_i, \Theta_j)$ when this event occurs[7].

- For any event a that occurs in process P_i, its vector time is $\Theta(a) = \Theta_i(a)$.

Consider the example given previously. In Figure 3.4, vector clocks have been added to the computation. Note that communication is treated as an event and therefore the clock is incremented. At event j, the third position has been changed to 3, and the fourth to 2.

In the next part of this section, some results are proved that relate to vector clocks. The statements of these proofs do not differ from those stated by Charron-Bost. To understand why this is so, consider the following case: if $P_i \mapsto P_j$ then by the rules given above, Θ_j contains full knowledge of Θ_i, so it is possible to present the proof statements without recourse to the $\mapsto$ relation. Note also that the communication information is contained in the partial order and therefore does not need to be dealt with explicitly.

Proposition 1 *For any event a, $\Theta(a)[i]$ equals the number of the events of P_i that belong to the past of a : $\Theta(a)[i] = \mid E(P_i) \cap (\downarrow a) \mid = \mid (\downarrow a)_i \mid$.*
Proof: If $P_i \in \mathcal{S}$, then Θ_i is initialised to $(0, \ldots, 0)$ and on each event in P_i, $\Theta_i[i]$ is incremented by one, so this value represents the number of events up to and including a in P_i.

If $P_i \in \mathcal{P} \backslash \mathcal{S}$ then the following situation exists for some $P_{j_1}, \ldots, P_{j_m}$

$$P_{j_1} \mapsto P_{j_2} \mapsto \ldots \mapsto P_{j_m} \mapsto P_i \text{ with } P_{j_1} \in \mathcal{S}.$$

When P_{j_1} starts, Θ_{j_1} is initialised to $(0, \ldots, 0)$, so $\Theta_{j_1}[i] = 0$. As P_i cannot be started before P_{j_1} it is not possible for it to communicate any timestamp to P_{j_1}

[7]This approach to vector clocks in the case of synchronous communication is presented by Fidge in [33]

(either directly or indirectly). Therefore at the last event ~~in~~ P_{j_1}, $\Theta_{j_1}[i] = 0$ still holds and therefore holds at the start of P_{j_2}. This argument holds for each P_{j_k} hence when P_i starts the value of $\Theta_i[i]$ is zero and so by a similar argument to above, on event a in P_i, $\Theta(a)[i]$ will represent the number of events up to and including a in P_j. (In the case of a communication event, say $a \leftrightarrow b$ with $a \in E(P_i)$, $b \in E(P_j)$, then a remains in P_i and the count of the number of processes remains the same.)

$\square$

The following result shows that the vector clocks totally capture the information of the partial order. This is in contrast to linear vector clocks as mentioned in Chapter 2.

Theorem 1 *For any events a and b of a distributed computation the following holds*

$$a \preceq b \iff \Theta(a) \leq \Theta(b).$$

Proof:

$\Rightarrow$: Since $\preceq$ is transitive, $c \preceq a$ implies that $c \preceq b$, hence for each i, $c \in \{x \in E(P_i) \mid x \preceq a\} \Rightarrow c \in \{x \in E(P_i) \mid x \preceq b\}$, i.e. $\{x \in E(P_i) \mid x \preceq a\} \subseteq \{x \in E(P_i) \mid x \preceq b\}$. Therefore $(\downarrow a)_i \subseteq (\downarrow b)_i$, so from Proposition 1, $\Theta[i](a) \leq \Theta[i](b)\ \forall i \in \{1, \ldots, n\}$, therefore $\Theta(a) \leq \Theta(b)$.

$\Leftarrow$: Assume the converse; there are two cases:

1. $b \preceq a$; by the first part of the proof it can be shown that $\Theta(b) \leq \Theta(a)$—contradiction.

2. a co b; therefore $a \in E(P_i)$ and $b \in E(P_j)$ with $i \neq j$, since if a and b are in the same process, then they are not concurrent. Consider $(\downarrow b)_i = \{x \in E(P_i) \mid x \preceq b\}$. There are two cases.

   (a) $(\downarrow b)_i = \emptyset$ and hence $(\downarrow b)_i \subset (\downarrow a)_i$.

   (b) $(\downarrow b)_i \neq \emptyset$, $(\downarrow b)_i$ is totally ordered and has a largest element c, say. Note that $c \prec a$, otherwise if $a \preceq c$ then since $\preceq$ is transitive and $c \preceq b$, $a \preceq b$ which would be a contradiction of a co b. Therefore $(\downarrow c)_i \subset (\downarrow a)_i$. However, $(\downarrow b)_i = (\downarrow c)_i$ since c is the largest element of $(\downarrow b)_i$ and hence $(\downarrow b)_i \subset (\downarrow a)_i$.

   In either case, $(\downarrow b)_i \subset (\downarrow a)_i$. Hence by Proposition 1 $\Theta(b)[i] < \Theta(a)[i]$. Therefore it is not true that $\Theta(a) \leq \Theta(b)$—contradiction.

□

The final result in this section presents a method for determining if two events are concurrent, by investigating the values of the vector clocks.

Proposition 2 *Let a and b be two events that belong respectively to P_i and P_j. Then*

$$a \text{ co } b \iff \big(\, \Theta(b)[i] < \Theta(a)[i] \text{ and } \Theta(a)[j] < \Theta(b)[j] \,\big).$$

Proof:

$\Rightarrow$: Assume a co b, with $a \in E(P_i)$ and $b \in E(P_j)$. Consider $(\downarrow b)_i = \{x \in E(P_i) \mid x \preceq b\}$. There are two cases, as in Theorem 1 and it can be shown by a similar argument that $(\downarrow b)_i \subset (\downarrow a)_i$. Hence by Proposition 1, $\Theta(b)[i] < \Theta(a)[i]$. By a symmetrical argument, it can be shown that $\Theta(a)[j] < \Theta(b)[j]$.

$\Leftarrow$: $\Theta(b)[i] < \Theta(a)[i]$ and $\Theta(a)[j] < \Theta(b)[j]$ show that $\Theta(a) \not\leq \Theta(b)$ and $\Theta(b) \not\leq \Theta(a)$. Therefore from Theorem 1, $\neg(a \preceq b)$ and $\neg(b \preceq a)$, hence a co b. □

These results show how vector clocks can fully describe the partial order $\preceq$. This allows for a characterisation of concurrency as shown in the last proposition.

3.4.4 Measure of concurrency

Charron-Bost motivates the measuring of concurrency in terms of consistent cuts by arguing that the tolerance of a computation to stopping relates to its ability to be cut in a consistent manner. This argument still holds in the new formalism. For example, consider Figure 3.4 (p. 49). If process P_3 is stopped (or cut) at i, then process P_4 can be stopped (or cut) at n, however, it cannot be cut at p because event j has not occurred, because of the cut at i. If the communication event $j \leftrightarrow p$ were not present, more cuts would be possible, for example at q.

Recall that the measure of concurrency for the computation C is defined by Charron-Bost [14] as

$$m(C) = \frac{\mu - \mu^s}{\mu^c - \mu^s}.$$

By definition, $m(C)$ should fall in the interval $[0, 1]$. As a result of the redefinition of the formalism, there are now a few choices for μ^s and μ^c.

- μ^s represents the number of consistent cuts in a totally sequential execution of C. It is determined by summing the number of events in C. There are two possible values

  $$\mu^s_{max} = 1 + q_1 + \cdots + q_n$$

  $$\mu^s_{min} = \mu^s_{max} - |\leftrightarrow| .$$

  The first value counts each event separately, so each pair of communication events is counted as two events. In the second case, each pair of communication events is counted as one event, to mirror the equating of communication events. This value is obtained by subtracting the cardinality of the communication relation. To ensure that the measure remains in the range $[0, 1]$, it was decided that the second value μ^s_{min} would be used. This can be explained by the fact that for some computations $\mu < \mu^s_{max}$ and hence $m < 0$. This occurs because when μ is calculated, pairs of communication events are viewed as one event, whereas for μ^s_{max} they are viewed as two.

- In the original formalism, a totally concurrent computation was one with no communication constraints and therefore $\mu^c = (1 + q_1)(1 + q_2)\ldots(1 + q_n)$. In the new formalism, there are two factors that constrain the amount of concurrency—communication and the causality between processes. It was decided that a totally concurrent agent in the new formalism should be one in which there is no communication and in which the process structure is retained. The first condition is transferred from the original formalism. The second condition can be explained as follows: removing communication indicates how concurrent the agent under consideration can possibly become; however, if the process structure was removed, it would result in the concurrency of the agent under consideration being compared to the concurrency of a different agent.

  Because the causality of processes is being taken into account, a more complex equation is required to characterise the number of consistent cuts μ^c in a totally concurrent computation in terms of q_i

  $\mu^c = \prod_{P_i \in \mathcal{S}} \text{cuts}(P_i)$ where

  $$\text{cuts}(P_i) = \begin{cases} q_i + \prod\limits_{\{P_j | P_i \mapsto P_j\}} \text{cuts}(P_j) & \text{if } P_i \notin \mathcal{F} \\ q_i + 1 & \text{if } P_i \in \mathcal{F}. \end{cases}$$

- The number of consistent cuts in a given computation μ can be calculated by methods that will be discussed in the next section.
- The original measure is defined for a specific computation of an algorithm. In this research I will extend the measure to algorithms by defining an ordered pair as the measure of an algorithm. More specifically two values will be chosen to represent the measure of concurrency of an algorithm A:
  - $\min\{m(C) \mid C \text{ is a computation of } A\}$
  - $\max\{m(C) \mid C \text{ is a computation of } A\}$.

  This makes it more diffic[illegible] to compare algorithms because $\Re^2$ is not totally ordered. However, it seems realistic that when taking into account all computations of an algorithm, there may be algorithms that cannot easily be compared.

  Another possible approach suggested by Charron-Bost [15, 16] which will be discussed further in Chapter 6 is that of determining probabilities for computations, and using these probabilities a[illegible] weights to determine a measure for the algorithm from the measures of each computation. This approach would generate a total [illegible]ering of algorithms.

3.4.5 Calculating the number of consistent cuts

Charron-Bost presents a result for determining whether a cut defined by $a_1, \ldots, a_n$ is consistent by checking a specific condition on the vector clocks

$$\sup(\Theta(a_1), \ldots, \Theta(a_n)) = (\Theta(a_1)[1], \ldots, \Theta(a_n)[n]).$$

The idea is that consistency occurs when the clock of a process P_i has the most up-to-date knowledge about its own 'time', and all other processes have the same or older knowledge about its time.

As the notion of a cut has been redefined, this result needs to be modified. First, to recap, the definition of a cut is as follows: let $N = \{1, \ldots, n\}$, and let $E_C \subseteq N$. E_C will contain the indices of the processes that do not contribute any events to the cut C. Then for each $i \in N \setminus E_C$, choose an $a_i \in E(P_i)$, then

$$C = \bigcup_{i \in N \setminus E_C} \{x \in E(P_i) \mid x \preceq a_i\} = \bigcup_{i \in N \setminus E_C} (\downarrow a_i)_i$$

is a cut of the computation.

Additional notation is required because of the new definition. Let C be a cut defined by E_C and a_i for $i \in N \backslash E_C$. Define $T_i \in \mathbb{N}^n$ as $\Theta(a_i)$ if $i \in N \backslash E_C$ otherwise T_i is defined as $(0, \ldots, 0)$. The value $T_i[j]$ will denote the jth component of T_i. The following result can now be proved.

Proposition 3 *The cut C defined by E_C and a_i for each $i \in N \backslash E_C$,*

$$C = \bigcup_{i \in N \backslash E_C} \{x \in E(P_i) \mid x \preceq a_i\} = \bigcup_{i \in N \backslash E_C} (\downarrow a_i)_i$$

is a consistent cut if and only if

$$\sup(T_1, \ldots, T_n) = (T_1[1], \ldots, T_n[n]).$$

Proof: The proof of this proposition appears in Appendix B. □

Charron-Bost presents a second approach to counting consistent cuts, whereby it is shown that the number of consistent cuts in a computation is equal to the number of antichains[8]. This result has not been proved in the redefined formalism.

Finally, it has been suggested that the number of consistent cuts in a computation is equal to the number of nodes in the (finite) transition graph of a confluent CCS agent. As yet, this result has not been proved and an outline of a possible proof is presented in the section on further work in Chapter 6.

3.5 Expressive power

The message-passing formalism has now been redefined to deal with the first two differences described in Section 3.2. The remaining difference to be discussed is that of expressive power. As noted earlier, the message-passing formalism represents one particular behaviour or computation of an algorithm, whereas CCS is a description of the behaviour of an algorithm. In the following section, some background material on computations and nondeterminism will be presented, after which it will be discussed how the difference in expressive power affects the measurement of concurrency.

A computation is defined to be a specific behaviour or execution of an algorithm or program. Therefore for a sequential algorithm, there are a number of different computations of that algorithm, each relating to different inputs. It is generally

[8]An antichain is a subset of a partially ordered set with no pair of elements comparable. An antichain is also called an independent subset. In this context, it means that each pair of elements is concurrent.

expected that sequential programs or specifications will be deterministic; that is for a given input, the program will behave in a predictable way. Obviously it is possible for a construct that causes nondeterminism to be introduced into a sequential language. However, the issue of nondeterminism becomes more important when dealing with concurrency as it is necessary to model the nondeterminism that occurs in the real world because of relativistic effects of time—these effects are the result of a lack of a global or centralised clock.

Nondeterminism can be described as follows: an algorithm exhibits nondeterministic behaviour if for a subset of its inputs, two or more different behaviours are possible for each input in this subset. So for each input, there are one or more possible computations.

The message-passing formalism does not explicitly express nondeterminism as it describes individual computations, except for that nondeterminism which results from the interleaving of events from different processes.

When dealing with a concurrent algorithm specified in CCS, nondeterminism is introduced by a number of factors and these result in different computations:

- Summation, for example
  - $a.\mathbf{0} + b.\mathbf{0}$ can perform the action a or the action b
- Composition
  - $a.\mathbf{0} \mid b.\mathbf{0}$ can perform a then b, or b then a
  - $a.\mathbf{0} \mid \overline{a}.\mathbf{0}$ can perform as a first action a, $\overline{a}$ or τ

There is a special class of nondeterministic agents called confluent agents (see Appendix A). Although these agents generate a number of computations, they have the characteristic that performing an action does not preclude the later occurrence of a different action that could have occurred at that time. These are a useful class of agents since the property of confluence is desirable when specifying concurrent systems [50]. Nondeterminism can occur in confluent agents as follows:

- Confluent Summation[9], for example

[9]For $\alpha_1, \ldots, \alpha_n \in Act, n \geq 0$, the Confluent Sum $(\alpha_1 \mid \ldots \mid \alpha_n).P$ is defined recursively as follows

$$\begin{aligned} ().P &\overset{\text{def}}{=} P \\ (\alpha_1 \mid \ldots \mid \alpha_n).P &\overset{\text{def}}{=} \sum_{1 \leq i \leq n} \alpha_i.(\alpha_1 \mid \ldots \mid \alpha_{i-1} \mid \alpha_{i+1} \mid \ldots \mid \alpha_n).P \quad (n > 0) \end{aligned}$$

  - $(a \mid b).\mathbf{0} \overset{\text{def}}{=} a.b.\mathbf{0} + b.a.\mathbf{0}$ can perform a and then b or alternatively b and then a.

- Confluent Composition[10], for example

  - $a.\mathbf{0} \mid_{\emptyset} b.\mathbf{0} \overset{\text{def}}{=} (a.\mathbf{0} \mid b.\mathbf{0})\backslash\emptyset$ can perform a then b, or b and then a.
  - $a.\mathbf{0} \mid_{\{a\}} \overline{a}.\mathbf{0} \overset{\text{def}}{=} (a.\mathbf{0} \mid \overline{a}.\mathbf{0})\backslash a$ can only perform τ, so this agent does not display nondeterminism.

Note that nondeterminism can no longer occur because of potential communication across the Composition operator since all communication is restricted by the set used in the definition of Confluent Composition. (See Appendix A.)

It is necessary to determine what effect the difference in expressiveness has on applying the measure to CCS. The most general approach is to consider each trace[11] of actions as a separate computation. However, in this research it will be ensured that at least the confluent agents can be measured.

The simplest way to deal with Confluent Summation is to consider each occurrence of the Summation operator as causing two different computations. Therefore, in the example given above, there are two computations ab and ba. This approach can be applied more generally to any occurrence of Summation, where each occurrence of this operator is treated as the point where the computation becomes two computations. For example, given $a.\mathbf{0} \mid (b.c.\mathbf{0} + d.e.\mathbf{0})$, there are two computations $a.\mathbf{0} \mid b.c.\mathbf{0}$ and $a.\mathbf{0} \mid d.e.\mathbf{0}$.

The type of nondeterminism allowed by Confluent Composition can be considered as basic to the message-passing formalism since it is the nondeterminism that occurs because of interleaving of events. Hence $(a.\mathbf{0} \mid b.\mathbf{0})\backslash\emptyset$ can be regarded as two processes, one of which performs a and the other b. In a Confluent Composition communication is 'forced', that is the set of actions in the Restriction is defined in such a way that communication has to happen if there is the potential for it, namely if an action and its complement appear on opposite sides of the Composition. This maps conveniently to the relation $\leftrightarrow$ which defines the communication events.

To generalise to Composition, consider the differences between Composition and Confluent Composition. There are two major differences; first, the two agents involved in the Composition must be confluent and second, nondeterminism as a result

[10]For $L \subset \mathcal{L}$, $P_1 \mid_L P_2 \overset{\text{def}}{=} (P_1 \mid P_2)\backslash L$ is a Confluent Composition if $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$ and $\overline{\mathcal{L}(P_1)} \cap \mathcal{L}(P_2) \subseteq L \cup \overline{L}$.

[11]A trace is the sequence of actions generated by a CCS agent.

of communication is removed. For the purposes of resolving the differences between the formalisms, the second is more important, since it permits the definition of the communication event relation $\leftrightarrow$ and it would be desirable to retain this in an attempt to make the approach more general. For example, given $a.0 \mid \overline{a}.0$ there is the potential of a, $\overline{a}$ or τ. To retain a simple translation to $\leftrightarrow$, the Composition operator will be redefined so that communication can only appear when the action and its complement are bound by Restriction. This involves combining **Com**$_3$ with **Res** as follows

$$\mathbf{Com}'_3 \quad \frac{E \xrightarrow{\ell} E' \quad F \xrightarrow{\overline{\ell}} F'}{(E \mid F)\backslash L \xrightarrow{\tau} (E' \mid F')\backslash L}(\ell \in L \cup \overline{L}) \quad .$$

This rule can also be viewed in terms of a condition on **Com**$_3$ stating that it cannot be applied if **Com**$_1$ or **Com**$_2$ can be applied. There are also similarities between the Composition operator as redefined by **Com**$'_3$ and Hoare's Conjunction and Hiding operators [[illegible]o, p. 194].

Note that communication in Confluent Composition will satisfy this rule, as communication always falls in the scope of a Restriction. This redefinition does affect the agents that can be described in CCS using Composition; however, it does not have a serious impact because in general communication is restricted when specifying concurrent systems in CCS.

To understand how Composition works when **Com**$_3$ replaces **Com**$'_3$, consider the following; if two complementary actions fall in the scope of a Restriction, these actions will be the events that are paired in the $\leftrightarrow$ relation and will be represented by one τ action—this is the same as in standard CCS. However, the difference occurs in the case where there is no Restriction. In standard CCS, the occurrence of the action, its complement or a communication action is permitted; however for the purposes of this research the ability to perform a communication action will be precluded.

For example, in the agent $(a.0 \mid \overline{a}.0)$ only the actions a and $\overline{a}$ will be permitted, and in the agent $(a.0 \mid \overline{a}.0)\backslash a$ only the τ action is permitted, as in standard CCS.

This discussion also raises the issue of measuring the concurrency of an algorithm. As noted in Section 3.4.4, the measure of an algorithm will be determined by the minimum and maximum values found over all computations of that algorithm.

3.6 Translation of CCS

3.6.1 Discussion

The message-passing formalism has now been extended to deal with synchronous communication and process creation, and can be applied to CCS. There are a number of questions that must be answered before the translation of CCS agents into the redefined formalism is presented. These relate to motivating the translation and to determining which subset of CCS should be modelled. The following points describe informally the intended translation process and are presented here to set the scene for the discussion.

- CCS actions will be mapped to events in the message-passing formalism, and actions and their complements will be used to identify communication event pairs. When a communication event occurs, the two actions are replaced by a single τ action. For example, consider the following agent $(a.\mathbf{0} \mid \overline{a}.\mathbf{0})\backslash a$. The communication event that occurs between a and $\overline{a}$ will be represented by the single action τ.
- An agent of the form $a_1.\dots.a_n.Q$ where Q is a CCS agent that does not start with a prefixed action, will be identified as a process. So, for example, the agent $a.b.(c.d.\mathbf{0} \mid e.f.\mathbf{0})$ will be decomposed into three processes—$P_1 = ab$, $P_2 = cd$ and $P_3 = ef$, with $P_1 \mapsto P_2$ and $P_1 \mapsto P_3$.
- As discussed in the previous section, the Composition operator will be modified by the replacement of the transition rule $\mathbf{Com}_3$ by $\mathbf{Com}_3'$ to allow simple translation from CCS to the $\leftrightarrow$ relation. When two complementary actions fall in the scope of a Restriction, these actions will be the events that are paired in the $\leftrightarrow$ relation and will be represented by one τ action. However, when two complementary actions are not bound by a Restriction, they are not considered as communication events, and therefore are not related by the $\leftrightarrow$ relation.

The approach that will be taken here is that CCS agents can be broken down into a number of computations, each of which will be expressed in CCS notation.

Events and actions

In the message-passing formalism, processes are composed of events. It is necessary to translate the actions of CCS to events. In CCS, there is the set of labels $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$, from which the set of actions $Act = \mathcal{L} \cup \{\tau\}$ is defined. τ is a distinguished action that represents communication. The translation of actions to events can be done as follows:

- Each element of $\mathcal{L}$ represents an event.
- A communication pair is indicated by an action a and its complement $\overline{a}$. When the communication occurs, it is indicated by a single event which is represented by the τ action.

Processes

In the formalism, processes are described as a finite sequence of events. Therefore, to translate CCS into this model, it would be reasonable to identify sequences of action Prefixes as events. For example, in the agent $a.b.c.d.e.\mathbf{0}$, the process would consist of the events $abcde$. In general, an agent of the form $a_1.\ldots.a_n.Q$ where Q is a CCS agent that does not start with a prefixed action, will be identified as a process.

Uniqueness of events and messages

In the message-passing formalism, events and messages are assumed to be distinguishable from each other. In CCS, actions which will be treated as events, do not have this property when considered as elements of Act. However, each action will be mapped to a specific point in the time-space diagram, and from the point of view of the Concurrency Measurement Tool, this will be sufficient to distinguish actions. A more rigorous approach to the unique identification of actions is to label them with the transition in which they occurred. For example, consider the agent $a.b.\mathbf{0} \mid a.c.\mathbf{0}$. There are two possible actions both of which are a actions. They can be distinguished as:

- the a action that causes the transition $a.b.\mathbf{0} \mid a.c.\mathbf{0} \xrightarrow{a} b.\mathbf{0} \mid a.c.\mathbf{0}$ and
- the a action that causes the transition $a.b.\mathbf{0} \mid a.c.\mathbf{0} \xrightarrow{a} a.b.\mathbf{0} \mid c.\mathbf{0}$.

Messages can be distinguished similarly by associating them with the transition where the communication occurred. This approach, however, is not required for the correct implementation of the Concurrency Measurement Tool.

Communication

In the message-passing formalism, communication is defined by the $\leftrightarrow$ relation which defines pairs of events between which communication occurs. As discussed in detail above, this set of events will be identified by using a redefined form of Composition where Restriction is used to permit communication. The communication events that can occur are those that are present in the Restriction set.

Totally concurrent agents and totally sequential agents

As mentioned in Section 3.2, it is necessary to investigate what agents will be interpreted as totally concurrent agents and totally sequential agents. These agents are required to calculate the measure.

As described in Section 3.4.4, there are two factors that constrain the amount of concurrency in a computation in the redefined formalism; communication and process causality. In line with the discussion given in that section, a totally concurrent agent will be defined as one [illegible] no communication, so each process is not blocked by communication with other processes; however, the process causality will be retained in the definition of a totally concurrent agent. This means that delays that occur because of the structure of processes will contribute to the measure. For example, consider the agent $a.b.(c.d.\mathbf{0} \mid \overline{c}.e.\mathbf{0})\backslash c$. The totally concurrent form of this agent will be $a.b.(c.d.\mathbf{0} \mid \overline{c}.e.\mathbf{0})$ where no communication can occur. So the effects of the communication on the concurrency are not taken into account; however, the effects of the fact that events c, $\overline{c}$, d and e can only occur after a and b are taken into account.

A totally sequential agent is the agent where each pair of communication events has been reduced to one event, and where all events must occur one after another. For example, consider $a.b.(c.d.\mathbf{0} \mid \overline{c}.e.\mathbf{0})\backslash c$ again. A totally sequential form of this agent will be one interleaving of possible events, for example $a.b.\tau.d.e.\mathbf{0}$ or $a.b.\tau.e.d.\mathbf{0}$, with the pair of communication events reduced to one τ event, as discussed in Section 3.4.4. In effect, the agent that represents the sequential computations in the example is $a.b.\tau.d.e.\mathbf{0} + a.b.\tau.e.d.\mathbf{0}$, the agent that is defined by the Expansion Law.

Finite agents

The formalism assumes that processes eventually terminate and therefore the measure can only be applied to a set of processes with a finite number of events. For that reason, only finite agents will be used. Milner [50, p. 160] defines finite agents as follows:

> An agent expression is finite if it contains only finite Summations and no Constants (or Recursions).

Subset of CCS

The subset of CCS that can be dealt with contains the following operators:

> **0**, Prefix, (finite) Summation, Composition, Restriction and Relabelling

and excludes:

> infinite Summation and Constant (or Recursion).

However, as discussed above the message-passing formalism does not allow the Summation operator, so the agents that are used for input will be divided into the different computations that would result from the presence of a Summation. Relabelling will be treated in a similar way—an agent for input will be in a form where all Relabellings have been applied.

The last question to be answered is the suitability of the subset of CCS. As stated above, this subset includes (finite) confluent agents which are defined by:

> **0**, (finite) Confluent Summation (which includes Prefix), Composition, Restriction and one-to-one Relabelling.

Milner has noted in his book [50] that confluence is a desirable property for ensuring well-behaved specifications. This would indicate that although it would be preferable to capture the whole of CCS, the subset that has been dealt with is, in fact, significant. Possible extensions to the subset will be discussed in Chapter 6.

Use of CCS notation

Note that the different syntactic forms of agents are used to distinguish the different forms these agents take in the message-passing formalism even though these different syntactic forms may be equated by the equivalence semantics ($\sim$, $\approx$ or $=$) of CCS. For example, consider Figure 3.5, (a) represents diagrammatically, the one computation that is possible in the message-passing formalism by the agent $a.\mathbf{0} \mid b.\mathbf{0}$, and (b) represents diagrammatically the two computations that are possible in the new formalism by the agent $a.b.\mathbf{0} + b.a.\mathbf{0}$, although from the Expansion

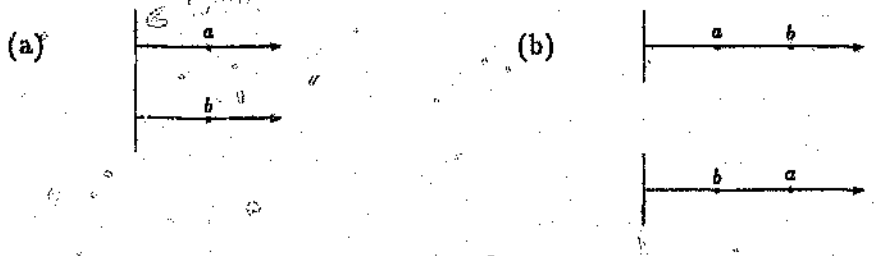

Figure 3.5: (a) $a.0 \mid b.0$. (b) $a.b.0 + b.a.0$.

Law, $a.0 \mid b.0 = a.b.0 + b.a.0$. Note that (a) is the totally concurrent form of the agent $a.0 \mid b.0$ and (b) is its totally sequential form.

This use of notation does not impose any restrictions on the agents that can be translated. However, this illustrates that the research deals with the syntax and operational semantics of CCS, but not the equivalence semantics of CCS.

3.6.2 Translation algorithm

The algorithm presented is a static algorithm to translate a CCS agent into the new formalism. It is static in the sense that the decomposition of the agent into processes is based only on the syntax of the agent. The input is a CCS agent and the output is a set of processes $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, consisting of sequences of events, a relation that describes the causal links between processes $\mapsto$, and a relation that describes the communication pairs $\leftrightarrow$. Note that this translation algorithm maps CCS agents into the message-passing formalism, whereas the algorithm used in the Concurrency Measurement Tool which will be presented in the next chapter, maps CCS agents into the data structures that represent the message-passing formalism.

In the following, *agent* will be used to indicate the CCS agent and i, j, k will indicate process numbers.

```
translate($agent$ , $i$)
    if $agent$ has the form $a.E$ then
        add $a$ to $P_i$
        translate($E, i$)
    if $agent$ has the form $E \mid F$ then
        get new process numbers $j$ and $k$
        create empty processes $P_j$ and $P_k$
        add $(P_i, P_j)$ and $(P_i, P_k)$ to $\mapsto$
        translate($E, j$)
        translate($F, k$)
    if $agent$ has the form $\mathbf{0}$ then
        do nothing
    if $agent$ has the form $E \backslash a$
        add the action $a$ to the restriction information about $E$
        translate($E, i$)
```

After this has been done, the communication pairs $\leftrightarrow$ can be determined from the restriction information that was recorded. A communication pair occurs whenever a label a and its complement $\overline{a}$ occur within two distinct processes in the scope of a Restriction on a.

3.7 Summary

In this chapter, the differences between the message-passing formalism of a distributed system and CCS were identified. The major differences are asynchronous versus synchronous communication, and nesting of processes, and expressive power. The formalism was redefined to allow for these differences, and it was shown that the original results are valid in a modified form in the new formalism, and that the justification for the measure still holds. The difference in expressive power between CCS and the formalism was resolved by defining how agents in CCS will be measured. It was determined which subset of CCS will be used in the research, and it was shown that this subset includes confluent agents. Finally, a translation algorithm was presented to map CCS agents into the redefined formalism.

In the next chapter, the Concurrency Measurement Tool which was developed to measure the concurrency of CCS agents, will be discussed. This discussion will include the algorithms used in calculating the measure, and the performance of the Concurrency Measurement Tool.

4. The Concurrency Measurement Tool

4.1 Introduction

In this chapter, the Concurrency Measurement Tool will be described and discussed. The Concurrency Measurement Tool is a program that provides for the measurement of the concurrency of agents in a specific subset of CCS, as detailed in the previous chapter. First, an overview of the program will be given, with discussion of the approach taken, the data structures used and the algorithms implemented. The two major algorithms—that for updating vector clocks and that for calculating μ—will be discussed. The second algorithm comprises the main work in determining the measure and a theoretical analysis of its performance will be given. This analysis will be compared with experimental results that were obtained. Suggestions for optimising the algorithm will also be presented. Finally, extensions to the Concurrency Measurement Tool and the application of it to other measures will be discussed.

4.2 Description of approach

Since there is no known method for analytically determining the number of consistent cuts in an arbitrary computation, it is necessary to generate cuts and check them for consistency. As there are a large number of possible cuts to be checked, it becomes necessary to automate the process. Therefore, as part of this research, a program was implemented to calculate the measure for a given CCS agent, subject to the limitations described in Chapter 3. The program provides the ability to perform experiments on CCS agents and hence to evaluate the measure and the feasibility of applying a measure of concurrency to CCS.

The program receives as input a CCS agent, and maps the agent in a data structure that is a representation of the message-passing formalism as redefined in the previous chapter. As the algorithm requires vector clocks to determine if a cut is consistent, vector clocks must be added to the events. Once this is done, the number of consistent cuts μ can be counted. The values μ^c and μ^s can be determined analytically and the measure can be calculated.

4.2.1 Data structures

The basic data structure is that of the event. Events are combined into a tree structure that represents the possible causation of one process by another. Although in Chapter 3, the presentation is at the level of processes, in the implementation of the Concurrency Measurement Tool, events cannot be represented as a sequence of labels attached to each process because of the amount of data associated with each event. It is sometimes necessary to have dummy events that represent no action, as only binary branching is allowed. These dummy events are ignored when the measure is calculated.

In the following, the type declarations (in C) for the data structures are presented. Both the action type and the restriction type contain label fields. Instead of each action being represented by a single character as in the previous chapter, an action label consists of a string. This allows for the use of meaningful names when defining CCS agents. The action structure has a label field and a field to indicate whether the label is complemented. The Restriction operator defines a subset of actions that are precluded, and these actions are stored in a linked list. It is not necessary to store the complement of an action, since if L is the set that is specified by the operator, then $L \cup \overline{L}$ is the set of actions that are prevented by the Restriction.

```
struct action_t                  /* action type */
{
 labelstr label;                 /* label name */
 int bar;                        /* complement indicator */
};

struct restrict_t                /* restriction type */
{
 labelstr res;                   /* label name */
 struct restrict_t *next_res; /* pointer to next restriction in list */
};
```

The event structure has two pointers to the two next events, a pointer to a possible communication event and an action structure as described above. Next there is a pointer to the variable that contains the vector clock, and a counter to record the latest clock update. Other fields contain the event number and the process number in which the event occurs. There is a pointer to a structure of Restrictions that captures all Restrictions that apply to that event and hence to all events further down the tree.

```
struct event_t                 /* event type */
{
 struct event_t *l_event;     /* pointer to next (left) event */
 struct event_t *r_event;     /* pointer to next (right) event */
 struct event_t *comm;        /* pointer to event involved in communication
                                 (may be NULL) */
 struct action_t act;         /* action information (may be empty) */
 int *clock;                  /* pointer to vector clock */
 int update;                  /* most recent clock update */
 int event_num;               /* number of event */
 int process_num;             /* number of process that event occurs in */
 struct restrict_t *next_res; /* pointer to list of restrictions
                                 (may be NULL) */
};
```

Finally, there is a structure to keep track of process information This structure duplicates some of the information that can be obtained from the event tree.

```
struct process_t                /* process type */
{
 struct event_t *first_event;  /* pointer to first event in process */
 int num_events;               /* number of events in process */
 int first;                    /* first (non-empty) event in process */
 int last;                     /* last (non-empty) event in process */
 int left;                     /* number of next (left) process */
 int right;                    /* number of next (right) process */
                               /* for convenience, this type contains info
                                  about which event (if any) in the process
                                  is contributing to the current cut during
                                  the cut checking process */
 struct event_t *cut;          /* pointer to event in current cut
                                  (may be NULL) */
 int enable;                   /* indicator of whether events in process
                                  can contribute to current cut */
};
```

This structure captures the $\mapsto$ relation in the fields `left` and `right`. It also contains the information about the event (if any) in the process is contributing to the current cut being checked. Finally, the enable field indicates whether the process is currently available to contribute events to the cut. This will be discussed in more detail later.

4.2.2 Program overview

There are two main sections in the program. The first relates to creating an event tree, finding communication events and adding the vector clock. The second consists of the procedures to calculate μ, μ^c and μ^s and from these to determine the value of m. The structure of the program is as follows:

```
input the agent
parse agent and create event tree
find pairs of communication events and return count
add vector clocks to events
calculate $\mu$ by generating all cuts and checking them for consistency
calculate $\mu^c$ and $\mu^s$ analytically from the number of events in each
    process
calculate measure and print results
```

The agent is input via a file, in the form specified in the previous chapter. To find pairs of communication events, the restriction lists are used to determine what subtrees to search. Vector clocks are updated by the rules presented in Chapter 3. Dummy events receive the same time stamp as the event preceding them. The procedures used to find μ will be discussed in more detail in the next section, and μ^s and μ^c can both be calculated by knowing the number of events in each process, the structure of the processes and the number of communication events.

The output includes the measure, μ, μ^s, μ^c, the number of events, number of processes, number of communication events and the time taken to obtain μ.

4.3 Translation procedure

The translation procedure takes CCS agents and translates them into the data structures given above. A recursive descent parser together with a number of semantic routines is used to create the event structure. The grammar is shown below, including the action symbols indicating the semantic routines, which are the functions that create the event tree:

```
<..> indicates nonterminals
{..} indicates optional elements
#..  indicates action symbols

<agent>    -->  Nil <atail> #fNil
                LPar <agent> RPar <atail> #fPar
                Action #StoreAction Dot <agent>
<atail>    -->  Comp <agent> <atail> #fComp
                Res <set> <atail> #fRes
                empty string
<set>      -->  Action #StoreRestrict
                LBrace <setelem> RBrace
<setelem>  -->  Action #StoreRestrict { Comma <setelem> }

terminals
 Nil      0    LPar     (    RPar      )
 LBrace   {    RBrace   }    Dot       .
 Comma    ,    Comp     |    Res       \
 Action   {'}X where X is a string consisting of one or more lower
          case letters.

semantic routines associated with action symbols
 StoreAction   - record details of action for use later in the
                 translation procedure
 StoreRestrict - add new action for restriction to list of actions for
                 later use
 fNil, fPar,
 fAction, fComp,
 fRes          - (in general) create new events and add them to event tree
```

As opposed to the conceptual algorithm presented in Chapter 3, the grammar and the semantic routines presented here embody the concrete algorithm that is used to translate CCS agents into the data structures that are themselves a concretisation of the message-passing formalism.

4.4 Algorithm for adding vector clocks

The goal of the algorithm is to add vector clocks to the event tree using the rules presented in Chapter 3, which are presented here in a less formal form:

- Initial values of clocks
  - If an event occurs at the beginning of a process that is not caused by another process, then the initial value of the clock is $(0, \ldots, 0)$.
  - If an event occurs at the beginning of a process that is caused by another process, then the clock is set to the value of the clock of the last event in the other process.
- Updating values of clocks
  - For each event that occurs in process P_i, the ith position of the vector clock is incremented by 1.
  - If two events are partners in synchronous communication then their clocks are set to the maximum of the values of their clocks after the previous step has been done.

A counter is used to indicate if the update of the ith position has been performed and this is used to indicate which events (nodes) have been visited. The algorithm traverses the tree depth-first adding vector clocks. If the beginning of a new process is found, the clock for the first event is copied from the last event in the previous process. In a similar fashion, an event that is not at the beginning of a process takes a copy of the clock from the previous event. Then regardless of whether the event is at the beginning of a process, a communication event or any other event, the ith position is incremented. However, if the event is one of a pair of communication events then it is necessary to find the maximum of the two clocks. This can only be done once both clocks have had the position in the vector that correspond to their process indices updated. If the partner has not been updated yet, the subtree beneath the event cannot be updated and is left until the partner is updated and then vector clocks are added to the subtrees beneath both events.

As each event is visited at most twice, the algorithm takes $O(q)$ time where $q = \sum_{j=1}^{n} q_j$, the total number of events.

```
add_clocks to event e
  if e is not a communication event
     copy clock from previous event
     if e is not a dummy event
       increment position i of the clock, where e is in process i
     set update counter to new value
     add_clocks to the left child of e
     add_clocks to the right child of e
  else   /* e is a communication event */
     if  e's communication partner has not been updated
        copy clock from previous event
        increment position i, where e is in process i
        set update counter to new value
     else    /* e's has been updated */
        copy clock from previous event
        increment position i of the clock, where e is in process i
        take max of e's clock and e's partners clock
        set update counter to new value
        add_clocks to the left child of e
        add_clocks to the right child of e
        add_clocks to the left child of e's partner
        add_clocks to the right child of e's partner
```

4.5 Algorithm for counting consistent cuts

The algorithm for counting the number of consistent cuts in an arbitrary computation is necessary to determine μ. The approach taken is based on Proposition 3 which presents a condition for determining whether a cut is consistent.

This cut depends on the vector times of the events that defined the cuts. Let C be a cut defined by E_C, and a_i for each $i \in N \setminus E_C$, $C = \bigcup_{i \in N \setminus E_C} (\downarrow a_i)_i$, and let $T_i \in \mathbb{N}^n$ be defined as $\Theta(a_i)$ if $i \in N \setminus E_C$ otherwise $(0, \ldots, 0)$. If the following condition holds for a specific cut then the cut is consistent

$$\sup(T_1, \ldots, T_n) = (T_1[1], \ldots, T_n[n]).$$

This result suggests a simple way for determining the number of consistent cuts, first generate a cut, then use the condition to check if the cut is consistent.

There are two functions—gen_cuts and consistent_cut—for implementing this algorithm. The first is a recursive procedure to generate all cuts and the second checks whether a cut is consistent or not. These algorithms will be discussed in detail below.

Function gen_cuts

In the original message-passing formalism, there were $(q_1+1)(q_2+1)\ldots(q_n+1)$ cuts to be checked, namely the number of consistent cuts in a totally concurrent version of the computation. However, in the new formalism, because of process causality, this is described by[1]

$$\mu^c = \prod_{P_i \in \mathcal{S}} \text{cuts}(P_i) \quad \text{where} \quad \text{cuts}(P_i) = q_i + \prod_{\{P_j \mid P_i \mapsto P_j\}} \text{cuts}(P_j).$$

Hence, to prevent extra work, the causality structure of the processes needs to be taken into account. This will mean that the actual number of cuts as described by the above equation, will be checked for consistency, as opposed to an inflated value given by $(q_1+1)(q_2+1)\ldots(q_n+1)$.

In order to generate only those sets of events that form cuts as defined by the causality of processes, it is necessary to allow processes to be 'enabled' and 'disabled'. If $P_i \mapsto P_j$ then no events of process P_j can contribute to a cut unless the final event of process P_i is in the cut. Therefore, process P_j will be marked as disabled until the final event of P_i has been put in the cut. The algorithm proceeds by working through each process, adding events (in order of occurrence) to the cuts. After each process has had the opportunity to contribute an event, the cut is defined and can be checked for consistency. The procedure is started by the call `gen_cuts(0)`.

[1]This is the same as the more detailed equation presented in Section 3.4.4 since if $P_i \in \mathcal{F}$, $\{P_j \mid P_i \mapsto P_j\} = \emptyset$ and $\prod_\emptyset = 1$.

```
gen_cuts(int pr)
  if pr = total_number_of_processes      /* if each process has
                                            contributed an event */
    if consistent_cut()                  /* check cut for consistancy */
      increment consistent_cut counter

  else  /* still processes to contribute */

    gen_cuts(pr + 1)   /* with no event contributing from process pr */

    if process pr is enabled
      while not last event in process pr   /* add each event in turn
                                              from process pr to the */
        add next event to cut              /* cut, except last one */
        gen_cuts(pr + 1)

      enable next processes
      gen_cuts(pr + 1)            /* with last event from process pr */
      disable next processes
```

Function consistent_cut

As described above, the vector clocks of the events that form a cut can be examined to determine whether the cut is consistent. The condition that is required to hold on the vector clocks can be rewritten as follows:

$$\begin{aligned} & \sup(T_1,\ldots,T_n) = (T_1[1],\ldots,T_n[n]) \\ \Leftrightarrow\ & \forall p \in \{1,\ldots,n\}\ \max_{1\leq j\leq n} T_j[p] = T_p[p] \\ \Leftrightarrow\ & \forall p,j \in \{1,\ldots,n\}\ T_j[p] \leq T_p[p] \end{aligned}$$

Consider j and p; there are four distinct scenarios that can occur with respect to membership of E_C:

- $j, p \in E_C$, then $T_j[p] = 0 = T_p[p]$, and the condition holds,
- $j \in E_C$ and $p \notin E_C$, then $T_j[p] = 0 \leq T_p[p]$, and the condition holds,
- $j \notin E_C$ and $p \in E_C$, then $T_j[p] > T_p[p] = 0$, and the condition does not hold,
- $j, p \notin E_C$, then $T_j[p] > 0$ and $T_p[p] > 0$ and these two values must be explicitly checked to determine if the condition holds.

To minimise the time involved in accessing complex data structures, the above scenarios were used to derive an algorithm that first determines if a process has contributed an event to the cut before explicitly checking the clock values. As soon as a counterexample is found, it can be indicated that the cut is not consistent. However if a cut is consistent it requires $O(n^2)$ operations to show this. The algorithm can be described as follows:

```
for( $p$ = 0; $p$ < total_number_of_processes; $p$++ )
  for( $j$ = 0; $j$ < total_number_of_processes; $j$+ )
    if $j \in E_C$
      /* do nothing since $T_j[p] = 0 \leq T_p[p]$ */
    else if $p \in E_C$
      return(false)      /* condition fails */
    else if $T_j[p] > T_p[p]$
      return(false)      /* condition fails */
return(true)      /* condition true, cut is consistent */
```

4.5.1 Analysis of algorithm

In this section, I will present a worst-case analysis of the algorithm used to check for consistent cuts. This analysis is an extension of an analysis presented by Charron-Bost [16]. The number of cuts to be checked can be described by the following formula

$$\mu^c = \prod_{P_i \in \mathcal{S}} \text{cuts}(P_i) \quad \text{where} \quad \text{cuts}(P_i) = q_i + \prod_{\{P_j | P_i \mapsto P_j\}} \text{cuts}(P_j).$$

The worst case occurs for a given n where n is the number of processes when there is no causality between processes, namely when

$$\forall P_i \in \mathcal{P},\ P_i \in \mathcal{S} \text{ and } P_i \in \mathcal{F}.$$

then there are $\mu^c = \prod_{i=1}^{n}(q_i + 1)$ cuts to be checked.

To analyse this algorithm the following inequality which is derived from the relationship between geometric and arithmetic means [21] is required

$$\sqrt[n]{\prod_{i=1}^{n}(q_i+1)} \leq \frac{\sum_{i=1}^{n}(q_i+1)}{n} = \frac{q}{n} + 1 \quad \text{where} \quad q = \sum_{j=1}^{n} q_j,$$

therefore

$$\prod_{i=1}^{n}(q_i+1) \leq (\frac{q}{n}+1)^n.$$

From this, μ^c can be expressed as

$$\mu^c = \prod_{i=1}^{n}(q_i + 1) \leq (q/n + 1)^n.$$

Note that in the case where all the q_i's are equal, namely

$$\forall i \in \{1, \ldots, n\},\ q_i = q/n$$

then

$$\mu^c = \prod_{i=1}^{n}(q_i + 1) = (\frac{q}{n} + 1)^n$$

and hence the upper bound is achieved in this case.

Also in the worst case it takes n^2 steps to check a cut for consistency which therefore gives an upper bound of $n^2(q/n + 1)^n$ steps to generate and check all cuts for consistency.

This gives an algorithm of $O(q^n/n^{n-2})$ where q is the total number of events in the computation and n is the number of processes. Although it has not been shown that this is a tight upper bound for solving this problem, it is the tightest upper bound known.

It would be difficult to perform an average case analysis as this would require the knowledge of the probabilities associated with the occurrence of certain computation structures. However, an approach to the average case can be demonstrated as follows. Assume that on average, when a cut is not consistent, it will require half the time used for determining if a cut is consistent to discover this, and let t be the time required in the worst case, then time taken in an average case can be described as

$$\frac{\mu}{\mu^c}t + (\frac{\mu^c - \mu}{\mu^c})\frac{t}{2} = (\frac{\mu + \mu^c}{2\mu^c})t \in \Theta(t).$$

This shows that even in an average case, the algorithm remains exponential.

Fidge [32] notes that the calculation of μ requires $\sum_{1 \leq i \leq n} nq_i$ integers to be stored. Using q as defined above, this can be written as

$$\sum_{i=1}^{n} nq_i = n\sum_{i=1}^{n} q_i = nq.$$

In the implementation of the Concurrency Measurement Tool, assuming an average number of restrictions per event, let the number of bytes required to store an event be c_1. The total number of events is defined above by q, and n is the total number of processes. Dummy events are required in some circumstances, but more than q dummy events are not possible. Sc the space required for the event tree is $2c_1qn$. If each process requires c_2 bytes to be stored, then c_2n bytes are required to store all processes in the process structure. This gives a total of $n(2c_1q + c_2)$ bytes for these two data structures, or space complexity of $O(qn)$ which correlates with Fidge's result.

These results show that the algorithm implemented is expensive. In the next section, other approaches will be discussed after which experimental timings for the program be will compared to the above theoretical result.

4.5.2 Other algorithms

The research done here did not focus on finding the most efficient algorithm to perform the counting of consistent cuts, so in some sense the algorithm used is naive. In the literature, however there are indications that an efficient algorithm has not been found. Raynal et al [59] note that the calculation of μ is not feasible but do not give any further details. Fidge [32] notes that the space requirements for calculating μ are expensive.

Charron-Bost [14, 16] has derived an algorithm based on her result that relates the number of consistent cuts to the number of antichains in the partial order. The algorithm first determines all 2-antichains and from these, builds the k-antichains for $k \geq 3$. She notes that the algorithm is efficient only when there are few 2-antichains in the computation under consideration. This occurs because if there are few 2-antichains, there are few k-antichains for $k > 2$. This is the same as saying that the algorithm is efficient when there is little concurrency in the computation. The analysis of this algorithm is complex and requires determining the number of k-antichains for each k in terms of the number of 2-antichains. I have investigated this algorithm further and found it to be $O((nq)^{n+1})$.

Kim et al [44] present an approach whereby they divide computations into concurrency blocks, and count the number of antichains in each block. This results in an approximation to m that is faster to compute; namely $O((nq)^n/b^{n-1})$, where b is the number of concurrency blocks.

The algorithm that is used in the Concurrency Measurement Tool also exhibits that property that the less concurrent the computation, the faster it is to count consistent cuts, as the procedure to check for consistency needs to find one index pair for which the condition does not hold to show that a cut is not consistent,

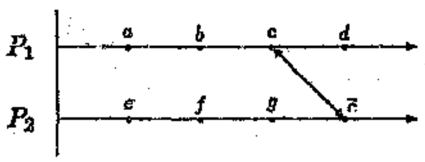

Figure 4.1: An example computation.

although as shown above it does not affect the complexity of the algorithm.

It would seem therefore that all algorithms proposed in the literature are exponential. However, in terms of the algebraic expressions found by the analyses, Charron-Bost's antichain algorithm is the more expensive. Depending on the relative sizes of n and b, the analyses of algorithm that I have implemented and the algorithm proposed by Kim et al are similar, but the latter algorithm is less desirable because of the inaccuracy involved in the approximation of m.

A few different approaches have been suggested that attempt to take the structure of the graph into account, however their worst case analysis appears to be the same as the algorithm given above. A basic approach to improving the algorithm is to eliminate the generation of cuts that are not consistent. This can be done by 'knowing' where the communication events are in a graph. Consider Figure 4.1—let the current cut be defined by $(\downarrow d)_1 \cup (\downarrow e)_2$. It is not consistent and in fact it will only be consistent once the algorithm has reached the cut $(\downarrow d)_1 \cup (\downarrow \bar{c})_2$. If the communication events can be located, then it should be possible to avoid checking the cuts created by adding f and g. In some sense, it is necessary for the algorithm to determine why the cut is not consistent, namely in which processes the inconsistency occurs and how to skip it. This may require an additional data structure to retain the communication structure, but this is an issue for further research.

4.6 Use of the Concurrency Measurement Tool

As mentioned earlier in this chapter, the CCS agent is input via a file, the calculation is performed, and m, μ, μ^s, μ^c, the number of events, the number of communication events, and the time taken to obtain μ are output.

To determine the maximum and minimum values for the measure of a specific CCS agent, it is necessary to determine if there is any Summation in the agent that may cause a number of distinct computations. If this is the case, the CCS agent must be decomposed into a number of agents that represent the different computations that can occur. These agents are measured individually by distinct executions of the Concurrency Measurement Tool and the maximum and minimum values for the original agent can be determined from the values obtained. An area of further research is to automate the decomposition and the presentation of the maximum and minimum values.

4.6.1 Experimental error

As there is no possibility of different values for the measure being obtained from different experiments on the same agent, the issue of experimental error does not play a major role in this research. The execution times presented later in this chapter form the only opportunity for experimental error; however they do not form the crux of this research.

4.7 Performance of Concurrency Measurement Tool

The Concurrency Measurement Tool consists of 1740 lines of optimised ANSI C code, and was executed on a Silicon Graphics Indigo workstation with on a 50MHz MIPS R4000 processor. In the experiments that were performed the number of cuts checked per second ranged from 83,000 to 168,000. This would indicate that although the algorithm is exponential in the number of processes, it is still feasible to calculate the measure for CCS agents as long as the number of events and processes remains within a reasonable range. For example, in the largest experiment performed, there were 60 events, 23 processes and 1.8×10^9 cuts to be checked. Of these, 11,836 were consistent and it took 2.2×10^4 CPU seconds to check the cuts and calculate the measure.

Table 4.1 presents a comparison of theoretical and actual execution times across a number of experiments for different values of q and n. The first two columns give the values of n and q respectively. The third column gives the number of CPU seconds taken to calculate the number of consistent cuts, and the fourth column gives the predicted number of operations obtained from the analysis of the algorithm as given in Section 4.5.1. The final column presents the ratio of the third column to the fourth column. These figures in the final column range from 2.07×10^6 to 112.16×10^6. This range indicates that the number of operations given by the analysis does to a fairly large extent predict the CPU time required to calculate the

n	*q*	*seconds*	*operations* $(n^2(q/n+1)^n)$	*operations/ second*
4	20	0.01	2.07×10^4	2.07×10^6
5	28	0.06	3.13×10^5	5.22×10^6
8	40	12.91	1.07×10^8	8.33×10^6
8	48	33.68	3.69×10^8	10.95×10^6
8	64	144.08	2.75×10^9	19.12×10^6
8	96	885.80	5.22×10^{10}	58.94×10^6
10	56	1027.24	1.57×10^{10}	15.27×10^6
10	64	2138.93	4.92×10^{10}	23.02×10^6
11	56	1910.80	5.18×10^{10}	27.11×10^6
11	64	4100.32	1.79×10^{11}	43.68×10^6
10	80	6900.07	3.49×10^{11}	50.53×10^6
11	80	13398.80	1.50×10^{12}	112.16×10^6
12	60	22366.00	3.13×10^{11}	14.02×10^6

Table 4.1: Comparison of actual and theoretical times.

number of consistent cuts. This confirms the analysis of the algorithm presented in Section 4.5.1.

4.8 Extensions to the Concurrency Measurement Tool

One important extension is the addition of a preprocessor to apply Relabellings and determine the different computations that result from the Summation operator, as currently it is necessary to create individual input files by hand. The program can also be extended to deal with these different files and to print all results including the minimum and maximum of the measures. A preprocessor could also be used in the case of probabilistic CCS [55] where actions are assigned probabilities. Then a weighting can be determined from these probabilities for each possible computation and a measure can be determined for the whole algorithm. This is discussed further in Chapter 6.

Other extensions discussed in Chapter 6 relate to extending the subset of CCS for which the measure is defined. To make these extensions to the measure usable, the Concurrency Measurement Tool would also require modification. In general, the idea would be to extend the map of CCS to the message-passing formalism, and hence the underlying algorithms are unlikely to need much alteration.

4.9 Calculation of other measures

The other measures that have been defined in the message-passing formalism are ω [14], ρ [15], β [32] and α [50] (see Chapter 2). In what follows, modifications required to enable the Concurrency Measurement Tool to calculate the measures will be briefly detailed.

The measure ω This measure is a special case of m and can be calculated by determining the number of concurrent pairs of events in the computation. The data structures would remain the same and a function could be added to perform pairwise comparisons of events. If the antichain algorithm is used to count the number of consistent cuts then finding the number of concurrent pairs will be part of this algorithm as these are just the 2-antichains.

The measure β The calculation of this measure requires two additions. First linear logical clock times are required in the calculation of the measure. The second addition is more complex as it requires changes to the message-passing formalism, since the clocks must be integrated at the end of the computation, namely

$$\Theta_{\text{final}} = \sup_{\{P_i | P_i \in \mathcal{F}\}} \Theta(a_{q_i}).$$

It is necessary to check that this is a reasonable approach when applied to CCS agents. In terms of the Concurrency Measurement Tool, however it is a relatively simple modification.

The measure α This measure requires an additional vector counter W which to be added to the event data structure. As for the measure β, the values of both Θ and W are integrated at the end of the computation. Again the modifications are relatively simple.

The measure ρ This measure is not defined in terms of vector clocks and hence the modifications required are more difficult. ρ is defined in terms of event and message times and the values of the measures can be determined analytically if the number of events and communication events are known. Therefore the data structures and functions used by the Concurrency Measurement Tool would not be required and a new program should be written.

4.10 Summary

In this chapter, the Concurrency Measurement Tool was discussed, including the algorithms used to calculate the measure, the most important of these being the algorithm used to calculate μ. It was shown that this algorithm was exponential in n, however it was possible to calculate the measure for agents of a reasonable size. Other algorithms, extensions to the Concurrency Measurement Tool and the calculation of other measures were discussed. In the next chapter, results of experiments performed using the Concurrency Measurement Tool will be presented and discussed with respect to the criteria used for evaluating the concurrency measure.

5. Results and evaluation

In this chapter, the results of experiments performed on CCS agents using the Concurrency Measurement Tool will be documented and interpreted. First, a review of the evaluation criteria described earlier in this document will be presented after which the experiments that were performed to evaluate m will be discussed in detail. This discussion will include the aim of each experiment, the agents chosen, figures that include the measure for the agent and other relevant information, and a short interpretation of the results. Following this there will be a section that consists of a fuller discussion of the results obtained and an evaluation of the measure in terms of the criteria. The next two sections will present suggestions for an improved measure and a discussion of this new measure in terms of the criteria.

Finally, the application of a measure of concurrency to CCS will be discussed and it will be argued that this has resulted in new methodology for the evaluation of concurrency measures defined in the message-passing formalism.

5.1 A review of evaluation criteria

This section presents a summary of the criteria presented in Chapter 2.

- Intuitive understanding of the measure (p. 9) [14]
- Being well behaved for small examples (p. 9) [14, 15]
- Compatibility with operators on computations (p. 9) [15, 18, 32]
- Usability and applicability (p. 13) [15, 45]
- Ability to calculate measure for a specific event (p. 14) [32, 59]
- Expense of computation in terms of both time and space (p. 14) [32, 38, 59]
- Stability with respect to granularity (p. 14) [16]

5.2 The experiments

The Concurrency Measurement Tool will be used to perform a number of experiments that have been designed to evaluate m with respect to the issues discussed above. The presentation format for each experiment contains the CCS agents or a description of them, a short description of the aim of the experiment, the results of the experiment and a brief interpretation of the results.

5.2.1 Experiment A: Simple example

The agents :

$$\begin{aligned}
A_1 &\overset{\text{def}}{=} (c.a_1.a_2.a_3.a_4.0 \mid \overline{c}.b_1.b_2.b_3.b_4.0)\backslash c \\
A_2 &\overset{\text{def}}{=} (a_1.c.a_2.a_3.a_4.0 \mid b_1.\overline{c}.b_2.b_3.b_4.0)\backslash c \\
A_3 &\overset{\text{def}}{=} (a_1.a_2.c.a_3.a_4.0 \mid b_1.b_2.\overline{c}.b_3.b_4.0)\backslash c \\
A_4 &\overset{\text{def}}{=} (a_1.a_2.a_3.c.a_4.0 \mid b_1.b_2.b_3.\overline{c}.b_4.0)\backslash c \\
A_5 &\overset{\text{def}}{=} (a_1.a_2.a_3.a_4.c.0 \mid b_1.b_2.b_3.b_4.\overline{c}.0)\backslash c
\end{aligned}$$

The aim : To investigate the behaviour of the measure on a simple example.

The results :

	μ^c	μ^s	μ	m
A_1	36	10	26	0.62
A_2	36	10	20	0.38
A_3	36	10	18	0.31
A_4	36	10	20	0.38
A_5	36	10	26	0.62

Interpretation : In Figure 5.1, the space-time diagrams are presented for Experiment A. The results show that there is a symmetry displayed about the middle of the computation, which was not expected. The closer the communication is to the middle of the computation the less concurrency there is. This can easily be explained by the fact that communication in the middle of the computation can affect a greater number of events, therefore the processes in this computation are more susceptible to blocking.

5.2.2 Experiment B: Simple example

The agent : $B_n \overset{\text{def}}{=} (c.a_1.\cdots.a_n.0 \mid \overline{c}.b_1.\cdots.b_n.0)\backslash c$

The aim : To see how the measure changes as the number of events in each process increases.

The results :

	μ^c	μ^s	μ	m
B_1	4	1	1	0
B_2	9	4	5	0.2
B_4	25	8	17	0.53
B_{10}	121	20	109	0.8
B_{100}	10201	200	10001	0.98

Interpretation : In Figure 5.2 the space-time diagrams are presented for Experiment B. The measure seems to extend across the whole of $[0,1)$ for increasing values of n. It should never reach 1 unless the single communication event in the graph is removed. The fact that the amount of concurrency increases is in accordance with the expected behaviour of such an example, because as the number of events increases, a single communication event has less effect on the concurrency of the computation.

5.2.3 Experiment C: Composition operator

The agents :

$$C_1 \overset{\text{def}}{=} (c_1.a_1.c_2.0 \mid \bar{c}_1.c_3.\bar{c}_2.0 \mid a_2.\bar{c}_3.a_3.0)\backslash\{c_1,c_2,c_3\}$$
$$C_2 \overset{\text{def}}{=} (c_1.a_1.c_2.0 \mid \bar{c}_1.c_3.\bar{c}_2.0 \mid a_2.\bar{c}_3.a_3.0)\backslash\{c_1,c_2,c_3\} \mid (c_1.a_1.c_2.0 \mid \bar{c}_1.c_3.\bar{c}_2.0 \mid a_2.\bar{c}_3.a_3.0)\backslash\{c_1,c_2,c_3\}$$

The aim : To investigate the behaviour of the measure when using the Composition operator.

The results :

	μ^c	μ^s	μ	m
C_1	64	7	12	0.0877
C_2	4096	13	144	0.0321

Interpretation : The agents for this experiment are described in Figure 5.3. This experiment is used to determine the action of the measure under the CCS operator Composition. As described in Section 2.2.3, it would be reasonable to expect the amount of concurrency to increase. However, in this experiment there is a decrease in the measured concurrency.

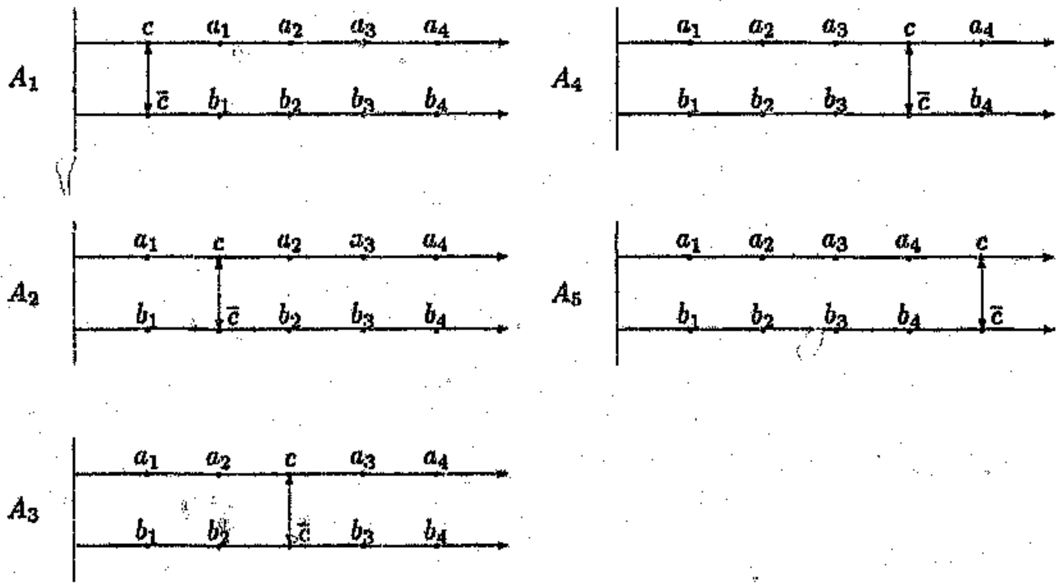
A_1
c a_1 a_2 a_3 a_4
$\overline{c}$ b_1 b_2 b_3 b_4
A_2
a_1 c a_2 a_3 a_4
b_1 $\overline{c}$ b_2 b_3 b_4
A_3
a_1 a_2 c a_3 a_4
b_1 b_2 $\overline{c}$ b_3 b_4
A_4
a_1 a_2 a_3 c a_4
b_1 b_2 b_3 $\overline{c}$ b_4
A_5
a_1 a_2 a_3 a_4 c
b_1 b_2 b_3 b_4 $\overline{c}$

Figure 5.1: Experiment A.

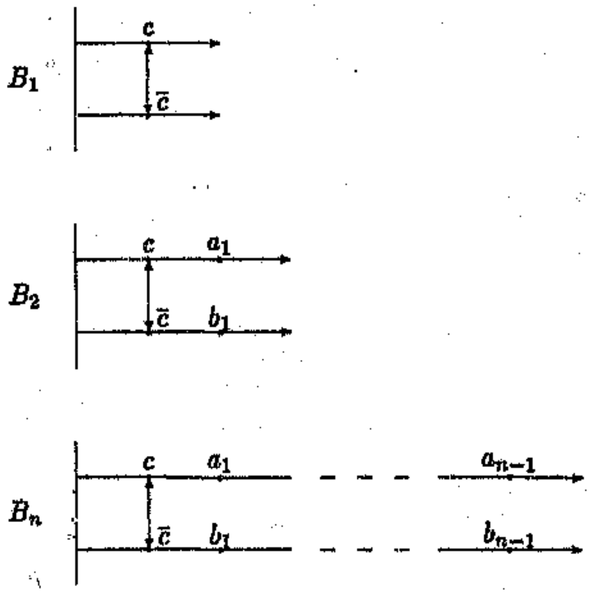

Figure 5.2: Experiment B.

5.2.4 Experiment D: Prefix operator

The agents :

$$\begin{aligned} D_1 &\overset{\text{def}}{=} (c_1.a_1.c_2.\mathbf{0} \mid \bar{c}_1.c_3.\bar{c}_2.\mathbf{0} \mid a_2.\bar{c}_3.a_3.\mathbf{0})\backslash\{c_1, c_2, c_3\} \\ D_2 &\overset{\text{def}}{=} b_1.b_2.b_3.(c_1.a_1.c_2.\mathbf{0} \mid \bar{c}_1.c_3.\bar{c}_2.\mathbf{0} \mid a_2.\bar{c}_2.a_3.\mathbf{0})\backslash\{c_1, c_2, c_3\} \end{aligned}$$

The aim : To investigate the behaviour of the measure when using the Prefix operator.

The results :

	μ^c	μ^s	μ	m
D_1	64	7	12	0.0877
D_2	67	10	15	0.0877

Interpretation : The agent for this experiment is shown in Figure 5.4. In this experiment, the effect of the CCS operator Prefix on the measure was investigated. As can be seen from the above table there was no change in the measure.

5.2.5 Experiment E: *Par* operator

The agents :

$$\begin{aligned} E_1 &\overset{\text{def}}{=} (j.m.p.\mathbf{0} \mid k.\bar{p}.\bar{q}.r.\overline{\text{done}}.\mathbf{0} \mid l.n.q.\mathbf{0})\backslash\{p, q\} \\ E_2 &\overset{\text{def}}{=} E_1 \ \mathit{Par}\ E_1 \\ &\overset{\text{def}}{=} ((j.m.p.\mathbf{0} \mid k.\bar{p}.\bar{q}.r.\overline{d_1}.\mathbf{0} \mid l.n.q.\mathbf{0})\backslash\{p, q\} \mid \\ &\quad b.(j.m.p.\mathbf{0} \mid k.\bar{p}.\bar{q}.r.\overline{d_2}.\mathbf{0} \mid l.n.q.\mathbf{0})\backslash\{p, q\} \mid \\ &\quad d_1.d_2.\text{done}.\mathbf{0} + d_2.d_1.\text{done})\backslash b \end{aligned}$$

The aim : To investigate the behaviour of the measure when using the *Par* operator.

The results :

	μ^c	μ^s	μ	m
E_1	96	10	24	0.1628
E_2	36864	20	554	0.0145

Interpretation : The agent used for this experiment (and for the next) are described in Figure 5.5. This experiment is used to determine the action of the measure under the operator *Par* . As described in Section 2.2.3, it would be reasonable to expect the measure of concurrency to increase, but to a lesser extent than with the Composition operator because an additional agent is introduced by the *Par* operator. As can be seen from the results, the measure decreases.

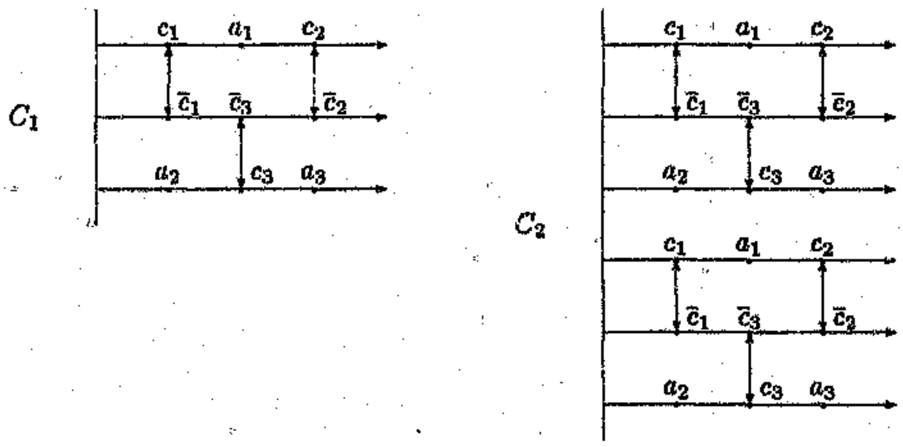

Figure 5.3: Experiment C.

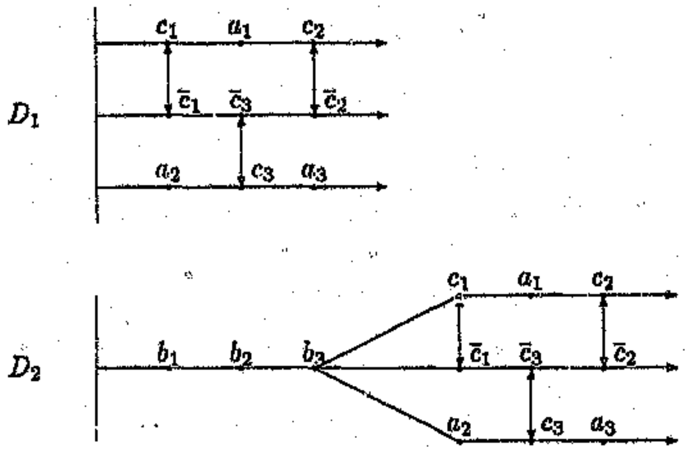

Figure 5.4: Experiment D.

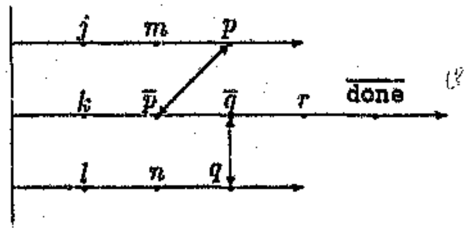

Figure 5.5: The agent for Experiments E and F.

5.2.6 Experiment F: *Before* operator

The agents :

$$
\begin{aligned}
F_1 &\overset{\text{def}}{=} (j.m.p.0 \mid k.\overline{p}.\overline{q}.r.\overline{done}.0 \mid l.n.q.0)\backslash\{p,q\} \\
F_2 &\overset{\text{def}}{=} F_1 \ \textit{Before}\ F_1 \\
&\overset{\text{def}}{=} ((j.m.p.0 \mid k.\overline{p}.\overline{q}.r.\overline{b}.0 \mid l.n.q.0)\backslash\{p,q\} \mid \\
&\qquad b.(j.m.p.0 \mid k.\overline{p}.\overline{q}.r.\overline{done}.0 \mid l.n.q.0)\backslash\{p,q\})\backslash b
\end{aligned}
$$

The aim : To investigate the behaviour of the measure when using the *Before* operator.

The results :

	μ^c	μ^s	μ	m
F_1	96	10	24	0.1628
F_2	9312	19	47	0.0030

Interpretation : This experiment is used to determine the action of the measure under the operator *Before* . As can be seen from the results, the measure decreases significantly.

5.2.7 Experiment G: Dining philosophers

The next set of experiments involves the dining philosophers problem which was first proposed by Dijkstra [28]. This problem is of interest because it is a classic problem in concurrent systems, and involves mutual exclusion with multiple resources. Solutions to the problem must ensure that deadlock does not occur. Two solutions are investigated:

1. The room-ticket solution, where there are $n-1$ room-tickets which ensure that deadlock cannot occur because it only allows $n-1$ philosophers in the room at one time [12].

2. The solution when n is even, where philosophers p_i for $i \bmod 2 = 0$ pick up their right forks first and philosophers p_i for $(i+1) \bmod 2 = 0$ pick up their left forks first. This prevents a situation where all n philosophers are holding a right fork and cannot obtain a left fork. This will be referred to as the odd-even solution [47].

The agents used for these experiments are detailed in Appendix B. To facilitate the comparisons of results the notation $x\mathrm{P}y$ is used, where x indicates the number of philosophers in the experiment and y indicates the number of eat and think actions that occurs. In Experiment G, philosophers alternatively perform one 'eat' action and then one 'think' action. In Experiment H, the number of eat and think actions is increased.

The agents :

G_1	2 philosophers	odd-even solution	2P1
G_2	4 philosophers	odd-even solution	4P1
G_3	6 philosophers	odd-even solution	6P1
G_4	2 philosophers	room-ticket solution	2P1
G_5	4 philosophers	room-ticket solution	4P1

The aim : To test the usability and applicability of the measure for comparison of real distributed algorithms.

The results :

	μ^c	μ^s	μ	m	
G_1	1.22×10^3	1.30×10^1	2.00×10^1	5.78×10^{-3}	2P1
G_2	1.50×10^6	2.50×10^1	1.72×10^2	9.79×10^{-5}	4P1
G_3	1.84×10^9	4.20×10^1	1.18×10^4	6.42×10^{-6}	6P1
G_4	1.01×10^4	1.70×10^1	2.50×10^1	7.92×10^{-4}	2P1
G_5	1.03×10^8	3.30×10^1	2.51×10^2	2.13×10^{-6}	4P1
	1.85×10^8	3.30×10^1	4.49×10^2	2.25×10^{-6}	4P1

Interpretation : The interpretations of experiments G and H are related, therefore the interpretations will be discussed after the presentation of Experiment H. Note, however, that the values are very small and decrease as the size of the example increases. There are two results for G_5 since there are two computations that can occur. The smaller of these values is listed first as the minimum measure for the algorithm, and the larger second as the maximum measure.

5.2.8 Experiment H: Dining philosophers

In the final experiment, the number of think and eat actions was increased for each philosopher, to see what effect a lower ratio of communication events had on the amount of concurrency and to test for stability with respect to granularity.

The agents :

G_2	4 philosophers	odd-even solution	1 eat	1 think	4P1
H_1	4 philosophers	odd-even solution	2 eat	2 think	4P2
H_2	4 philosophers	odd-even solution	4 eat	4 think	4P4
H_3	4 philosophers	odd-even solution	8 eat	8 think	4P8
G_5	4 philosophers	room-ticket solution	1 eat	1 think	4P1
H_4	4 philosophers	room-ticket solution	2 eat	2 think	4P2
H_5	4 philosophers	room-ticket solution	4 eat	4 think	4P4

The aim : To test the usability and applicability of the measure for comparison of real distributed algorithms and to test for stability with respect to granularity.

The results :

	μ^c	μ^s	μ	m	
G_2	1.50×10^6	2.50×10^1	1.72×10^2	9.79×10^{-5}	4P1
H_1	4.10×10^6	3.30×10^1	5.12×10^2	1.17×10^{-4}	4P2
H_2	1.79×10^7	4.90×10^1	2.52×10^3	1.38×10^{-4}	4P4
H_3	1.22×10^8	8.10×10^1	1.94×10^4	1.59×10^{-4}	4P8
G_5	1.03×10^8	3.30×10^1	2.51×10^2	2.13×10^{-6}	4P1
	1.85×10^8	3.30×10^1	4.49×10^2	2.25×10^{-6}	4P1
H_4	4.12×10^8	4.10×10^1	1.03×10^3	2.41×10^{-6}	4P2
	2.29×10^8	4.10×10	6.93×10^2	2.85×10^{-6}	4P2
H_5	1.42×10^9	5.70×10^1	3.95×10^3	2.73×10^{-6}	4P4
	7.91×10^8	5.70×10^1	3.13×10^3	3.88×10^{-6}	4P4

Interpretation The interpretation of these results will be presented in the next section. From the results, it can be seen that additional internal events reduce the amount of concurrency measured. As in the previous experiment, for each of the room-ticket solutions, two values are obtained for the two different computations that can occur. They represent the minimum and maximum values found in each experiment.

5.3 Discussion and evaluation of m

A number of issues arose in the experiments, which will discussed under the following headings:

Behaviour of the measure with respect to parallel operators The behaviour of the measure when using the Composition operator can be explained as follows:

- the number of cuts in the totally concurrent computation is squared
- the number of cuts in the computation itself is squared
- the number of cuts in the sequential computation is doubled and decremented by one.

Considering experiment C, this means that $m(C_2)$ can be expressed as

$$\frac{(\mu_{C_1})^2 - (2\mu^s_{C_1} - 1)}{(\mu^c_{C_1})^2 - (2\mu^s_{C_1} - 1)}$$

where $\mu_{C_1}, \mu^c_{C_1}, \mu^s_{C_1}$ are the relevant measures for computation C_1.

Since $\mu_{C_1}/\mu^c_{C_1} < 1$, $(\mu_{C_1}/\mu^c_{C_1})^2 < \mu_{C_1}/\mu^c_{C_1}$. Assuming that the size of μ^s is small with respect to μ^c and μ, as in this experiment, the value of m must decrease.

The operator *Par* was used in Experiment E and it too can be regarded as a parallel operator. In this experiment, the measured concurrency decreased and this decrease can be explained by a similar argument to that given above. Note, however, that there is a sharper decrease in concurrency. This can be explained by the fact that the *Par* operator defines an extra agent, $d_1.d_2.\overline{\mathbf{done}}.\mathbf{0} + d_2.d_1.\overline{\mathbf{done}}.\mathbf{0}$, to be used for synchronisation purposes.

Both of the results from Experiments C and E contradict the expectation that

$$m(C_2) = m(C_1 l C_1) > m(C_1) \text{ and } m(E_2) = m(E_1 l E_1) > m(E_1)$$

as discussed in Section 2.2.3.

Behaviour of the measure with respect to sequential operators The fact that m does not change when the Prefix operator is used as explained in Experiment D can be explained by the fact that a constant number of cuts is added to each element of the equation. This means that $m(D_2)$ can be expressed as

$$\frac{(\mu_{D_1} + c) - (\mu^s_{D_1} + c)}{(\mu^c_{D_1} + c) - (\mu^s_{D_1} + c)} = \frac{\mu_{D_1} - \mu^s_{D_1}}{\mu^c_{D_1} - \mu^s_{D_1}}$$

where $\mu_{D_1}, \mu^c_{D_1}, \mu^s_{D_1}$ are the relevant number of cuts for computation D_1.

There are two possible explanations for why the measure does not change in the presence of sequential operators as presented in Section 2.2.3; first, if the component computations are the same; and second if the component computations are sequential. As can be seen, neither hold in this case. The Prefix does in a very strong sense make the computation more sequential and the fact that the measure does not change, means that the measure can not express this.

Experiment F deals with a sequential operator which is more similar to the sequential concatenation presented by Fidge and Charron-Bost than the Prefix operator. In the presence of the *Before* operator, the amount of measured concurrency decreases, and this agrees with the expectation that

$$\text{MC}(C' \rightsquigarrow C'') \leq \max(\text{MC}(C'), \text{MC}(C'')).$$

The dining philosophers experiments It was unexpected that the measure would be so small for these examples, and that the measure would become smaller as n increased. Consider, for example, experiment G_4 (2P1) where only one philosopher can eat at one time because there is only one room-ticket. This would indicate that the system is in some sense serial, however this system has a higher concurrency measure than G_5 (4P1) where two philosophers can eat simultaneously (although three philosophers have access to the dining room). In fact, because the values are so small and cover such a small linear range, the use of these values for the comparison of different algorithms is difficult to justify.

There are two basic approaches to comparison; taking the absolute value of the difference of the two values (additive comparison) or taking the ratio of the two values (multiplicative comparison). Because the measure takes on values in the range $[0, 1]$, additive comparison is inappropriate, because the values are bounded. Multiplicative comparison seems more valid to apply to a measure that returns bounded values; and this approach will be taken here.

When G_2 (4P1) and G_5 (4P1) are compared, the ratio of the first to the second is 42.61. Although it can be said that the odd-even solution has less message passing than the room-ticket solution because there are no room-ticket agents, it seems unreasonable to say that the odd-even solution is 43 times more concurrent than the room-ticket approach. Also note the overlap in the room-ticket example—the minimum value for H_5 (4P4) is less than the maximum for H_4 (4P2), so these values do not distinguish between the different algorithms.

It can be seen from the relative sizes of μ^c, μ^s and μ that μ^c dominates the value of m. Hence, the values obtained in the experiment are small because μ^c occurs in the denominator. From this, it can be argued that the algebraic expression for m is

	m	n	$\sqrt[n]{m}$	
G_1	5.78×10^{-3}	4	0.2757	2P1
G_2	9.79×10^{-5}	8	0.3154	4P1
G_3	6.42×10^{-6}	12	0.3692	6P1
G_4	7.92×10^{-4}	5	0.2397	2P1
G_5	2.13×10^{-6}	10	0.2709	4P1
G_5	2.25×10^{-6}	11	0.3066	4P1
H_1	1.17×10^{-4}	8	0.3224	4P2
H_2	1.38×10^{-4}	8	0.3293	4P4
H_3	1.59×10^{-4}	8	0.3351	4P8
H_4	2.85×10^{-6}	10	0.2789	4P2
H_4	2.41×10^{-6}	11	0.3085	4P2
H_5	3.88×10^{-6}	10	0.2877	4P4
H_5	2.73×10^{-6}	11	0.3121	4P4

Table 5.1: The nth root of m

ill behaved.

To support this argument consider Table 5.1, where the nth root of m is presented for both Experiments G and H. As can be seen from this table, $\sqrt[n]{m} \approx 0.3$. If it can be assumed that $\sqrt[n]{m}$ is in some sense constant then this implies that $m \approx 0.3^n$. This has two consequences—first that the value of the measure appears to be related to n and second it would appear by extrapolation that as $n \to \infty$ then $m \to 0$. This also implies that the measure is inversely related to n, namely as n increases, m decreases.

The interpretation given above relates to the experiments performed using the dining philosophers agents. It would be desirable to be able to generalise these results to all algorithms. Unfortunately this is not possible, because of particular features of the dining philosopher problem that may be unique to this algorithm, for example, the fact that the number of processes correlates with the number of communication events. However, this has shown that there are algorithms for which the measure does not behave well.

The effect of changes in granularity Within the agents for each type of solution to the dining philosopher's problem, there is little change in the measure as the number of non-communication events changes. This can be explained by the fact that the addition of non-communication events is causing μ and μ^c grow similar rate. Charron-Bost states that she did not expect m to be stable with respect to granularity as it is a measure based on a causal relationship; this viewpoint can also be justified by the fact that the addition of more non-communication events results in a computation that will be more tolerant to blocking and this approach will be adopted here. Hence, the measure does not display the expected change when there is a change in granularity.

5.3.1 Evaluation of m with respect to the criteria

The measure m will be evaluated in terms of the criteria presented at the beginning of the chapter. A short description will be given for each criterion and a summary of these points and the preceding discussion will be presented afterwards.

Intuitive understanding of the measure The measure has a good intuitive base as described in Section 2.2.1. The concept that the number of consistent cuts in a computation is an indication of its tolerance to the stopping of individual processes is valid.

Being well behaved for simple examples The two experiments (A and B) performed on m would indicate that it is well behaved. The concurrency increases in a way that is expected, although the symmetry found in Experiment A was not expected. However, this symmetry can be explained as follows: the single communication event divides the computation into two parts, and the number of consistent cuts in the computation equals one plus the number of consistent cuts in the first part plus the number of consistent cuts in the second part. Hence, whether the larger part comes first or second does not affect the number of consistent cuts. Note that this only applies in the case of two processes.

Compatibility with operators The four experiments performed to investigate this had mixed results. In the case of the parallel operators, Composition and *Par* the measure decreased—this would indicate that the measure is not compatible with these operators. In the case of the serial operator Prefix, the measure stayed the same, which cannot be explained in terms of the operator. Finally, the *Before* operator yielded the expected decrease in concurrency and hence it would seem that m is compatible with this operator, although this has not been shown that this is true in all cases. By using the CCS operators, it has been possible to corroborate

Charron-Bost's results concerning compatibility with operators and moreover, to show analytically why these results occur.

Usability and applicability The measure was tested on two solutions of the dining philosophers problem. For all examples, the measure produced very small values on a small subset of [0, 1]. This can be explained by the fact that the algebraic structure of the equation defining m, and the manner in which μ^s, μ and μ^c grow, causes μ^c to dominate the resultant value of the measure. The measurement of concurrency for the dining philosophers solutions did not distinguish a more concurrent algorithm.

This has shown the measure does not work in some instances. It may be possible that the example chosen has certain peculiar characteristics that prevent measurement of concurrency, although the analytic results dealing with operators on computations suggest there are more fundamental problems with the measure. However, the dining philosophers example does show that in general, m cannot be used to compare algorithms.

Ability to calculate measure for a specific event The measure m cannot be calculated for a specific event, although it may be possible to extend the measure. However, the geometric interpretation (which has not been dealt with in detail here) can be used to determine which events in the computation are causing a reduction in the amount of concurrency.

Lack of expense of computation In Chapter 4, it was shown that the algorithm used to calculate the number of consistent cuts in any given agent was exponential in the number of processes. It was, however, possible to calculate the measure with the computing power available, with 1.4×10^9 cuts being the largest number checked for consistency in any experiment. The possibility that consistent cut counting is intractable needs further investigation.

Stability with respect to granularity It was shown that the measure showed some stability with respect to granularity, although this contradicts the expectation that the measure would change as the granularity changed.

Summary The more important issues rising out of the evaluation of m are as follows: m is not compatible with a number of operators, it returns very small values, it is expensive to calculate, the value of μ^c appears to dominate the value of m since it grows faster than μ or μ^s. These results show that m does not meet the criteria for evaluation and this contradicts both Charron-Bost [14, 16] and Raynal et al [59].

5.4 A new measure

As shown in the previous sections, the measure m has a number of flaws which can be explained by the fact that μ^c dominates the value of the measure. To solve these problems, m was redefined and the new measure is defined as

$$m_{\text{new}} = \frac{\log \mu - \log \mu^s}{\log \mu^c - \log \mu^s}.$$

This form was chosen because taking logarithms reduces the size of the values, and should thus prevent μ^c from dominating the measure.

Two other algebraic forms were considered, however they did not prove to be as successful, although the results were similar to those of m_{new}. They were

$$m_{\text{new}} = \frac{\sqrt{\mu} - \sqrt{\mu^s}}{\sqrt{\mu^c} - \sqrt{\mu^s}} \text{ and } m_{\text{new}} = \frac{\sqrt[n]{\mu} - \sqrt[n]{\mu^s}}{\sqrt[n]{\mu^c} - \sqrt[n]{\mu^s}}.$$

The results of the experiments with the new measure are presented in the following sections. Refer to Section 5.2 for further details of the experiments.

5.4.1 Experiment A: Simple example

The results :

	μ^c	μ^s	μ	m	m_{new}
A_1	36	10	26	0.62	0.7459
A_2	36	10	20	0.38	0.5411
A_3	36	10	18	0.31	0.4589
A_4	36	10	20	0.38	0.5411
A_5	36	10	26	0.62	0.7459

5.4.2 Experiment B: Simple example

The results :

	μ^c	μ^s	μ	m	m_{new}
B_1	4	1	1	0	0
B_2	9	4	5	0.2	0.2752
B_4	25	8	17	0.53	0.6615
B_{10}	121	20	109	0.8	0.9420
B_{100}	10201	200	10001	0.98	0.9950

5.4.3 Experiment C: Composition operator

The results :

	μ^c	μ^s	μ	m	m_{new}
C_1	64	7	12	0.0877	0.2436
C_2	4096	13	144	0.0321	0.4180

5.4.4 Experiment D: Prefix operator

The results :

	μ^c	μ^s	μ	m	m_{new}
D_1	64	7	12	0.0877	0.2436
D_2	67	10	15	0.0877	0.2132

5.4.5 Experiment E: *Par* operator

The results :

	μ^c	μ^s	μ	m	m_{new}
E_1	96	10	24	0.1628	0.3871
E_2	36864	20	554	0.0145	0.4417

5.4.6 Experiment F: *Before* operator

The results :

	μ^c	μ^s	μ	m	m_{new}
F_1	96	10	24	0.1628	0.3871
F_2	9312	19	47	0.0030	0.1462

5.4.7 Experiment G: Dining philosophers

The results :

	μ^c	μ^s	μ	m	m_{new}	
G_1	1.22×10^3	1.30×10^1	2.00×10^1	5.78×10^{-3}	0.0948	2P1
G_2	1.50×10^6	2.50×10^1	1.72×10^2	9.79×10^{-5}	0.1753	4P1
G_3	1.84×10^9	4.20×10^1	1.18×10^4	6.42×10^{-6}	0.3206	8P1
G_4	1.01×10^4	1.70×10^1	2.50×10^1	7.92×10^{-4}	0.0604	2P1
G_5	1.03×10^8	3.30×10^1	2.51×10^2	2.13×10^{-6}	0.1357	4P1
	1.85×10^8	3.30×10^1	4.49×10^2	2.25×10^{-6}	0.1680	4P1

5.4.8 Experiment H: Dining philosophers

The results :

	μ^c	μ^s	μ	m	m_{new}	
G_2	1.50×10^6	2.50×10^1	1.72×10^2	9.79×10^{-5}	0.1753	4P1
H_1	4.10×10^6	3.30×10^1	5.12×10^2	1.17×10^{-4}	0.2337	4P2
H_2	1.70×10^7	4.90×10^1	2.52×10^3	1.38×10^{-4}	0.3076	4P4
H_3	1.22×10^8	8.10×10^1	1.94×10^4	1.59×10^{-4}	0.3852	4P8
G_5	1.03×10^8	3.30×10^1	2.51×10^2	2.13×10^{-6}	0.1357	4P1
	1.85×10^8	3.30×10^1	4.49×10^2	2.25×10^{-6}	0.1680	4P1
H_4	4.12×10^8	4.10×10^1	1.03×10^3	2.41×10^{-6}	0.2002	4P2
	2.29×10^8	4.10×10^1	6.93×10^2	2.85×10^{-6}	0.1820	4P2
H_5	1.42×10^9	5.70×10^1	3.95×10^3	2.73×10^{-6}	0.2488	4P4
	7.91×10^8	5.70×10^1	3.1[illegible] 10^3	3.88×10^{-6}	0.2435	4P4

5.5 Discussion and evaluation of the new measure

For Experiments A and B, the same trends are shown by m_{new} as by m, although there is a general increase in the values.

When applied to Experiments C and E, m_{new} seems a better measure than m since the value increases as the computation becomes more concurrent through use of the Composition operator and the use of the *Par* operator. However, it is possible to find agents where this does not hold. Consider the agents E_1' and E_2' where the number of internal events has been increased (See Figure 5.6).

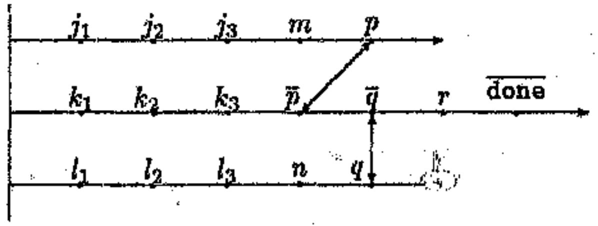

Figure 5.6: The agent E'_1.

$$\begin{aligned}
E'_1 &\overset{\text{def}}{=} (j_1.j_2.j_3.m.p.0 \mid k_1.k_2.k_3.\overline{p}.\overline{q}.r.\overline{\text{done}}.0 \mid l_1.l_2.l_3.n.q.0)\backslash\{p,q\} \\
E'_2 &\overset{\text{def}}{=} E'_1 \; Par \; E'_1 \\
E'_3 &\overset{\text{def}}{=} E'_1 \mid E'_1
\end{aligned}$$

where the measure has the following values

$$m_{\text{new}}(E'_1) = 0.8268,\; m_{\text{new}}(E'_2) = 0.6369,\; m_{\text{new}}(E'_3) = 0.7514$$

. Once again, this contradicts the expectatioa that

$$m(E'_2) \geq m(E'_1) \;\text{ and }\; m(E'_3) \geq m(E'_1).$$

This contradiction can be explained by the fact as μ^c becomes very large, the value of $\log \mu^c$ will still dominate the value of the measure, hence as the agent size increases because of concatenation, the value of m_{new} will decrease in some cases.

Considering the Experiments D and F, the value of m_{new} now decreases, as was originally expected both for Prefix and for *Before* . Note also that the decrease caused by the *Before* operator is not as marked.

In both Experiments G and H, the values given by m_{new} are much increased on those of m. Also as the number of philosophers increases in Experiments G_1 (2P1), G_2 (4P1) and G_3 (6P1), so does the value of m_{new}, which seems more promising than the values for m. Similar results are shown for the room-ticket example. However, it is still difficult to compare the odd-even and room-ticket solutions. It appears that for the same number of philosophers and the same number of eat and think actions, the room-ticket algorithms are less concurrent than the odd-even algorithms. This can be explained by the communication overhead required by the use of room-tickets.

Similarly as the number of non-communication events increases in Experiments G_1 (4P1), H_1 (4P2), H_2 (4P4) and H_3 (4P8), the measure also increases substantially. This would show that m_{new} is less stable with respect to granularity than m.

5.5.1 Evaluation of the new measure with respect to the criteria

The measure m_{new} will be evaluated in terms of the same criteria used in Section 5.3.1.

Intuitive understanding of the measure The measure m_{new} has a similar intuitive base as m; however the use of logarithms causes it to be less intuitive, especially in the light of the fact that it does not seem intuitive that μ^c dominates m.

Being well behaved for simple examples The two experiments (A and B) performed on m_{new} would indicate that it is well behaved as m for these examples.

Compatibility with operators m_{new} shows more compatibility with Prefix, *Par* and Composition. However, there exist agents for which m_{new} does not behave as expected in the presence of parallel concatenation.

Usability and applicability The measure m_{new} produced better results on the dining philosophers problem than m. The values of the measure were larger and increased as the number of philosophers increased. Different algorithms could be distinguished by m_{new}. Further research is required on other examples.

Ability to calculate measure for a specific event m_{new} cannot be calculated for a specific event, although it may be possible to extend the measure.

Lack of expense of computation In Chapter 4 it was shown that the algorithm used to calculate the number of consistent cuts in any given agent was exponential in the number of processes. Since m_{new}, like m, is based on the number of consistent cuts, it is as expensive to calculate as m.

Stability with respect to granularity The measure m_{new} shows less stability with respect to granularity then m. As discussed earlier, this is the result that is expected because m is based on a causal relationship and because the addition of more non-communication events makes a computation more tolerant to blocking.

Summary The measure m_{new} is better than m although it does not fully solve all problems associated with m.

5.6 Evaluation of the measurement of concurrency in CCS

The focus of this chapter so far, has been on the evaluation of the measure m. Two other important aspects of this research are the feasibility of measuring concurrency in CCS and the development of a methodology for the evaluation of concurrency measures. In the next two subsections these points will be discussed in detail.

5.6.1 Measuring concurrency in CCS

In the course of this work it has been shown repeatedly by the fact that the evaluation of the measure m was possible that the concept of measuring concurrency in CCS is a workable one. Although the measures have been defined for a subset of CCS, it was shown in Chapter 3 that this subset includes confluent agents. An area of research is the extension of the measure to the whole of CCS.

5.6.2 A methodology for the evaluation of concurrency measures

This research has shown that concurrency can be measured in CCS and that it can be used to evaluate concurrency measures. Hence, the approach taken here to evaluate m can be used to evaluate other measures. This will provide both for evaluation of existing measures and the development of new measures; and for the further development of a new tool for the theoretical investigation of concurrency using CCS.

When authors present measures of concurrency in literature, generally they do not present evaluations of these measures. For a measure to be a useful tool, it is essential that it is evaluated with respect to relevant criteria. The research has drawn together a number of criteria and used them in conjunction with the Concurrency Measurement Tool to evaluate m. This approach can also be applied to other measures defined in the message passing formalism.

Therefore, an important outcome of this work has been to produce a methodology for evaluating measures of concurrency. This approach currently can be used for measures defined in the message-passing formalism; however, it may be possible to use CCS with measures defined in other formalisms. It has been shown to be a useful methodology by the fact that it facilitated the discovery that a specific measure was ill behaved and allowed for the definition of a measure using the same basis that fitted better with the evaluation criteria.

5.7 Summary

In this chapter, criteria for the evaluation of concurrency measures were reviewed. The experiments that were performed using the Concurrency Measurement Tool were presented. From these experiments, it was shown that the measure m does not fulfil some of the criteria for evaluation, such as compatibility with CCS operators and applicability and usability. A new measure m_{new} was defined and its values were determined for the same experiments. It was shown that m_{new} although better than m was still problematic with respect to a few criteria.

Through the experiments on both m and m_{new} it was shown that the concept of measuring concurrency can be applied to CCS agents. Finally it was shown that this research has produced a methodology for the evaluation of concurrency measures.

The work presented in this document has resulted in a number of questions for further research and these questions will be discussed in the next chapter together with the final conclusions.

6. Conclusions and further research

In this final chapter, an outline of the research and the conclusions drawn will be presented. After this section, issues that are suitable for further investigation as a result of this research will be discussed. There are a number of directions that include improving m, comparing concurrency measures, defining a measure based on linear extensions, extending the measure to algorithms, proving that there is the same number of consistent cuts in the computation representing a confluent CCS agent as there are nodes in a transition graph of that agent, extending the measure to a larger subset of CCS, applying measures to other algebraic calculi of processes and defining a partial ordering semantics for CCS.

6.1 Outline

The aim of this research was to investigate the measurement of concurrency in CCS and at the same time, to evaluate a particular measure of concurrency.

Measures of concurrency are proposed in the literature as a means of investigating distributed algorithms. These aim to assess the structure of computations, as opposed to time complexity and message complexity which do not do this. CCS is a calculus of processes that allows for the investigation of concurrency. By achieving a fusion of the two, a new tool can be developed for the investigation of concurrency using CCS and comparison of different agents. There are a number of criteria which can be used to evaluate a measure and this work aims to address the fact that measures of concurrency presented in the literature are not fully evaluated.

6.2 Summary of work done

In this section, an overview of the work done in this report will be given.

Literature survey First, the relevant background literature was presented, to draw together the different approaches to concurrency measurement and evaluation of these measures. The choice of model was narrowed down to Lamport's space-time model/message-passing formalism. A number of measures have been defined in this framework and all require some further investigation and evaluation. Charron-Bost's measure of concurrency m was chosen as the measure to be applied to CCS.

Comparison of the message-passing formalism and CCS Once the model and measure had been chosen, it was necessary to compare the message-passing formalism and CCS, and determine any differences that would prevent the direct application of the measure to CCS agents. Three main differences were found—the message-passing formalism provides asynchronous communication and CCS provides synchronous communication; CCS allows for process creation, whereas the message-passing formalism has a fixed number of processes; and CCS has more expressive power than the message-passing formalism.

Redefinition of the message-passing formalism The message-passing formalism was redefined to take into account these differences. This process involved defining a new process structure, whereby a process could 'cause' other processes, and determining the effect that synchronous communication had on the ordering of processes. The result of this was a new partial ordering of events where the events in a process that causes another process, precede the events in the process that was caused and where events that take place in synchronous communications are equated. A number of results were shown to hold in the new formalism. These results and the intuitive basis for the measure (that of consistent cuts being related to the tolerance of a computation to stopping) justify the extension of the measure to the redefined formalism.

Development of the translation algorithm After the redefinition, it was possible to define a translation algorithm to map CCS agents into the message-passing formalism and hence for the measure to be applied to CCS agents. A subset of CCS was chosen to be translated—this subset includes finite agents defined using Prefix, **0**, (finite) Summation, Composition, Restriction and Relabelling, although the Concurrency Measurement Tool is not able to directly handle Summation and Relabelling. It was shown that this subset includes finite confluent agents.

Development of an algorithm to calculate the measure An algorithm was defined to calculate the concurrency measure. The largest part of this algorithm is counting the number of consistent cuts in the agent under consideration, since the number of cuts in the totally concurrent and totally sequential forms can be determined analytically. The algorithm generates all possible cuts, and checks them for consistency. It was shown that the algorithm has a worst case analysis of $O(q^n/n^{n-2})$ where q is the total number of events in the computation and n is the number of processes in the computation. Other algorithms were suggested.

Implementation of the Concurrency Measurement Tool The Concurrency Measurement Tool was implemented to automate the calculation of the measure for CCS agents, using the algorithm that was developed. It was used to experiment with chosen CCS agents to evaluate m and its performance was measured to compare it with the theoretical analysis given above. It was show that the theoretical analysis did largely predict the performance of the program.

Evaluation of experiments A number of experiments were performed to evaluate the measure and to evaluate the measurement of concurrency in CCS. These experiments will be discussed in more detail in the section below. The outcome was that the measure m does not meet the criteria for evaluation, especially with respect to compatibility with operators and applicability to real situations, that a better measure m_{new} can be defined and that the measurement of concurrency in CCS is possible.

6.3 Conclusions

In the following the conclusions of the research will be presented and explained:

1. In the evaluation presented in Chapter 5, it was shown that there are flaws in the measure m. This is demonstrated clearly in the experiments:

    - When two agents are concatenated in parallel, the effect is essentially to square both μ^c and μ. Because of the equational structure of the measure and the fact that μ^s tends to be relatively small, this results in the squaring of the measure and since m is defined on $[0,1]$, $m^2 \leq m$. Therefore the amount of measured concurrency decreases which is contrary to expectation. Similarly, the use of the Prefix operator does not provide the expected results, although the use of *Before* the second sequential operator does result in the expected decrease in concurrency. Therefore the measure, in general, does not fulfil the criterion of compatibility with operators. By applying the measure to CCS, it was possible to confirm

Charron-Bost's results concerning compatibility with operators and to show analytically why these results occur. As discussed earlier, compatibility with operators is an important criterion because CCS allows for abstraction from detail.

- The results of the dining philosopher experiment showed that applying the measure in a real situation does not allow clear comparison of algorithms. The measures calculated fall into a very small subset of [0, 1], in the range of 2×10^{-6} to 5×10^{-3} which raises issues about the interpretation of the measure, and the validity of comparison. The size of the values returned by the measure can be explained by the fact that μ^c dominates the measure m.
- In the dining philosopher experiment, it was shown that as the size of the experiment increased, the measure decreased, which indicates that it was not measuring the actual concurrency in the agents.

Do these experiments show that the approach that was used to apply the measure to CCS was flawed? No, they show that Charron-Bost's measure is problematic. Although the message-passing formalism that was used originally to define the measure was modified, it was done in a consistent way and hence the argument for the motivation of the measure was successfully applied to the new formalism.

The anomalies presented above, can be explained as follows: for reasonable sized systems, the resulting measure will be very small; and μ^c the number of cuts in a totally concurrent computation dominates the value of the measure for a given computation. Hence, it can be claimed that although the justification for the measure is correct, the algebraic expression used to define the measure is ill behaved.

2. A new measure m_{new} using logarithms was proposed and evaluated. It was shown that this measure was more compatible with the four operators, although in the case of the parallel operators the measure is not always compatible. It was shown that m_{new} returned larger, more reasonable values than m. However, for certain computations, it was still possible for $\log \mu^c$ to dominate the value of m_{new}, as in the case of the parallel operators.

3. The algorithm used to calculate μ was exponential in the number of processes; although it was possible to calculate the measure for the dining philosophers

algorithms, an example of applying the measure in a real situation. It has been suggested that the problem is inherently intractable. This issue requires further research.

4. The measure was successfully applied to a subset of CCS that included finite confluent agents. This shows that the measurement of concurrency in CCS is feasible and therefore other measures can be applied to CCS and evaluated. The understanding of concurrency that is provided by CCS allowed for the evaluation of m and will allow for the further development of a tool for the measurement of concurrency which will aid the theoretical investigation of concurrency. Because of the abstraction provided by CCS, compatibility with operators is an important issue in any measure of concurrency for CCS; it is desirable to be able to understand the effects on the amount of concurrency when components are interchanged or variables are instantiated.

5. The fact that it was possible to apply the concurrency measure to CCS and evaluate it in terms of the criteria gleaned from the literature means that this approach can be applied to other measures and can be used in the definition of new measures. The redefined formalism, CCS, the Concurrency Measurement Tool and the criteria provide a methodology for evaluating concurrency measures in a manner that has been absent in previous presentations of concurrency measures.

6.4 Further research

6.4.1 Improving the measure based on consistent cuts

As it was shown that the measure had undesirable properties, an area of further research would be to use the motivation given for the measure m—the fact that the number of consistent cuts is related to how tolerant the computation is to being stopped—and attempt the definition of a better measure based on consistent cuts. A suggestion was advanced in Chapter 5 using logarithms to define the measure m_{new}; however, it did not solve all the problems associated with m.

6.4.2 Further investigation of the algorithm to count consistent cuts

Currently the algorithm used to calculate μ is exponential in the number of processes. Further investigation is required to determine if there exists an efficient algorithm to solve the problem or whether the problem is intractable. Some suggestions were presented in Chapter 4. Another suggestion involves that of space complexity. Fidge [32] has noted that $\sum_{1\leq i\leq n} nq_i$ integers are required to store the vector clock information. It is possible to count consistent cuts without using vectors clocks. The partial ordering of events describes a directed acyclic graph and the consistent cut counting problem can be rephrased in terms of the number of subgraphs there are which have the property that if vertex v is in the subgraph, every vertex on every path from the root to v is also in the subgraph.

6.4.3 Comparison of measures applied to CCS

A number of measures have been defined in the framework that Charron-Bost used. These include ω [14], ρ [15], β [32] and α [59] all of which were described in Chapter 2. These measures can also be translated into the new formalism with the necessary research to ensure they remain consistent and can be motivated. Hence they can also be applied to CCS, and the Concurrency Measurement Tool can be modified to calculate these measures. This modification should be minor as the program already embodies the redefined theoretical basis. For the measures β and α, linear logical clocks need to be added, and for α a second type of vector clock is required—the modification to the program for these is not large. The measure ω is related strongly to m and its calculation should be possible in the existing program; ρ is more complex as it does not use vector clocks and therefore the implementation requires more investigation. This is discussed in more detail in Chapter 4.

Similar experiments can be performed to those done in this research and the results can be used to compare different measures. This work is important as such comparisons have not previously been reported in the literature.

6.4.4 A measure based on linear extensions

Charron-Bost [14] suggests a measure using linear extensions of the partial ordering of events, but rejects it because there is no fast algorithm to count linear extensions. Pruesse and Ruskey [54] have recently presented an algorithm for generation of linear extensions that runs in constant amortized time. A generation algorithm is said to run in constant amortized time if it runs in time $O(N)$ where N is the number of objects generated. In the worst case, this algorithm is exponential since the number of linear extensions is exponential. However, it has been shown that it is practical for certain examples. This algorithm could be used to investigate a concurrency measure based on linear extensions, as it seems as good as the algorithms for counting consistent cuts.

The approach taken in this algorithm revolves around generating each extension by permuting adjacent elements of the previous extension. Another area for further research is to investigate whether this technique can be applied to the counting of consistent cuts, to obtain a constant amortized time algorithm. In both of these cases, this research will be fruitful only if it can be determined that there are interesting subclasses of the partial orders (computations) for which the algorithms become tractable.

6.4.5 Extension of the measure to algorithms

In this research, a measure for an algorithm was determined from the minimum and maximum of the values of the measures of its computations. Another approach would be to determine weights $w_1, \ldots, w_n$ that could be applied to the measures of the computations $C_1, \ldots, C_n$ for an algorithm A as follows:

$$m(A) = \sum_{i=1}^{n} w_i m(C_i).$$

These weights could be determined from the probabilities assigned to actions as discussed by Purushothaman and Subrahmanyam [55]. This work involves assigning probabilities to τ actions in agents of the form $\tau_{p_1}.P_1 + \tau_{p_2}.P_1$ where $p_1 + p_2 = 1$. This determines the probabilities of different actions occurring. These probabilities can be added to the derivation tree of the agent and the result would be a decision tree from which the probability of each computation can be determined.

6.4.6 Extension of the message-passing formalism

In Section 3.6.1, the issue of nondeterminism was discussed and it was shown that the message-passing formalism cannot adequately deal with the nondeterminism caused by the Summation operator, or by the possibility of communication permitted by the Composition operator in the absence of Restriction.

One extension to include the former type of nondeterminism is suggested as follows and a similar approach could be taken for the latter type. This suggestion is to describe the processes that a process P_i *does* cause and the processes that it *may* cause. This relation $\mapsto$ could then be redefined to map from the set of processes to the powerset of processes and thus capture the nondeterminism. For example, consider the agent $a.b.(c.d.0 \mid (e.f.0 + g.h.0))$ with $P_1 = ab$, $P_2 = cd$, $P_3 = ef$ and $P_4 = gh$, then $P_1 \mapsto \{P_2, \{P_3, P_4\}\}$. This indicates that P_1 causes P_2, and P_1 causes one of P_3 and P_4.

A different approach involves finding a topology in which the space-time diagrams can be embedded, so that the concept of nondeterminism can be added to the message passing formalism.

6.4.7 Transition graphs of CCS agents

In Section 3.4.5 on calculating the number of consistent cuts, it was suggested that the derivation graph of an agent could be used. A result that has not yet been proved or disproved is that:

> The number of consistent cuts in the computation of a given confluent CCS agent is equal to the number of nodes in the transition graph of that CCS agent.

If this result is shown to hold, it would then be possible to investigate an algorithm based on creating this graph and counting the nodes.

For a rigorous proof, it needs to be shown that there is a bijection from the set of nodes in the transition graph to the set of consistent cuts (or vice versa). Transition graphs have not been investigated in this research and therefore this proof is beyond the scope of this report.

An informal justification can be explained as follows: A consistent cut represents a 'consistent' state of a computation or in other words, a state where all events that precede an event in the cut are also in the cut. Since communication determines the precedence between events in different processes, a consistent cut takes this communication structure into account. Therefore it is not possible to have a communication event without its partner and all events that preceded the partner, in the cut. On the other hand, a node in a transition graph represents the agent in a particular state. The actions which label the directed edges of the graph correspond to events; and the actions that occur on the preceding edges of the graph must represent those in the cut, since the agent at a node contains the information about what can happen in the future and nothing else.

6.4.8 Extension of the measure of concurrency on CCS

Another area for future research is the extension of the measure to the whole of CCS. The subset currently supported includes

Nil, Prefix, finite Summation, Composition, Restriction and Relabelling

and excludes:

infinite Summation and Constant (or Recursion).

However the Concurrency Measurement Tool does not deal directly with Summation or Relabelling—agents for input can consist only of Nil, Prefix, Composition and Restriction.

Areas for extension include:

- The Concurrency Measurement Tool can be extended to handle agents that include finite Summation and Relabelling, explicitly, without some form of preprocessing by the user. Currently it is necessary to apply Relabellings and decompose the agent into the different computations caused by the occurrence of Summation before using the Concurrency Measurement Tool, then from the values for each computation, the maximum and minimum values for the agent are obtained. This proposed extension would automa :: the work involved, allowing the user to input an agent to the Concurrency Measurement Tool and receive the maximum and minimum values as output. However, this extension will increase the cost of computing the measure considerably.

- The measure could be extended to deal with the whole of CCS. This would involve defining a measure on infinite agents. One approach would be to determine the measure at specific stages during an infinite computation and from these values calculate a final measure for the infinite computation. This could be done by considering the stages of the partial computation as finite and applying a measure defined for finite computations to obtain these values. Two basic approaches can be used to obtain the final values from the intermediate values; averaging across all intermediate values obtained, or viewing the intermediate values as a sequence that converges to a final value, with conditions to ensure convergence.

6.4.9 Application of concurrency measures to other formalisms

In this research, the focus has been on measuring concurrency in CCS. There are other algebraic calculi of processes in the literature for expressing concurrency, and some of them will be discussed briefly here.

- Bergstra and Klop's ACP (Algebra of Communicating Processes) [6] and Hennessy's ATP (Algebraic Theory of Processes) [41] have similarities with CCS at the syntactic and operational level, although different congruences are defined at the semantic level. Each calculus has some operators that are unique, and therefore to apply any measures defined in the message-passing formalism, it is necessary to decide how these operators will be mapped into the message-passing formalism. The issues of differences in expressive power must also be addressed for these calculi. A similar argument as used in Section 2.5 can be used to justify the application of one of the measures from the message-passing formalism.

- Meije [8] and Synchronous CCS (SCCS) [50] are two synchronous process calculi that are similar. In a synchronous calculus, processes proceed in lockstep, and the assumption that nothing is known about the relative speeds of the processes does not hold. As suggested in Section 2.5, these calculi would be more applicable to the measures of concurrency based on formal languages, because actions occur in lockstep and transitions are represented by multisets of labels.

- COSY [48] is an algebraic approach to Petri nets. Paths which represent resources, and processes are the components that make up a COSY program. Processes act independently except where they are restricted by paths. To apply a measure from those defined in the message-passing formalism, it is necessary to determine how to map COSY paths and processes in the message-passing formalism, and to consider issues that relate to the expressive power of the two formalisms, and it appears that this would be more complex than the solution found in this research.

- Hoare's CSP (Communicating Sequential Processes) [11], is defined in terms of a denotational semantics, although it is sometimes informally described operationally. It is not immediately obvious how a measure of concurrency could be applied to CSP processes, and hence this would appear to require a different approach.

6.4.10 Partial ordering semantics for CCS

As discussed in Chapter 2, there is a debate about whether models of concurrency should display interleaved semantics or semantics based on 'true' concurrency. The vector clocks described in this research could be added to CCS to give a partial ordering semantics in a similar way to the work of Degano et al [27]. In this research, CCS was mapped into the message-passing formalism. However, for a partial ordering semantics it would be necessary to add, in some sense, the clocks that define the partial ordering in the message-passing formalism to CCS. The methodology presented here distinguishes between agents that are observationally congruent; however, the methodology does not deal with the semantic properties of an agent, but rather with properties that are defined by the syntax and the transitional rules of CCS.

The general approach to providing CCS with a partial ordering semantics can be described as follows:

- redefine the syntax of the expressions in the language,
- redefine the operational semantics in terms of a new transition relation,
- provide a new equivalence under the redefined expressions. Generally the Expansion Law will not hold under this new equivalence.

This is a major area of research and requires a new approach to the work presented here.

6.5 Conclusion

The aim of the research was to investigate the measurement of concurrency in CCS. This aim was achieved by applying the measure m to CCS agents, and in the process of accomplishing this, criteria for evaluating measures of concurrency were drawn from the literature, the message-passing formalism was redefined to enable the mapping of CCS agents to it, a software tool was developed to calculate the measure, the measure m was shown by evaluation against the criteria to be problematic and a new measure m_{new} was defined which was shown to be better than m. The redefined formalism, CCS, the Concurrency Measurement Tool and the evaluation criteria together form a methodology that permitted the evaluation of m, and this methodology can be applied in the evaluation of other measures.

When measures of concurrency are defined in the literature it is necessary that they are properly evaluated to ensure their validity and usefulness. This research has

A. An overview of CCS

In this appendix, a brief overview of Milner's Calculus for Communicating Systems (CCS) [50, 51] is given. The aim of this presentation is twofold; first to review of CCS, and second to set the notation used in this research report. This is in no way a full presentation of CCS and the reader that has no experience of the subject is referred to Milner's book *Communication and Concurrency* [50]. The last section in this appendix presents confluence which is used in the research report.

A.1 The basic language

The language is defined in terms of expressions, actions and an action relation which combined form a Labelled Transition System.

A.1.1 Syntax and notation

Let $\mathcal{A}$ be an infinite set of names $a, b, c, \ldots$, and $\overline{\mathcal{A}}$ be the set of co-names $\overline{a}, \overline{b}, \overline{c}, \ldots$, with $\overline{\overline{a}} = a$. $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ is the set of labels, ℓ and $\overline{\ell}$ will range over $\mathcal{L}$, and K, L will range over $\mathcal{L}$. Define $\overline{L} = \{\overline{\ell} \mid \ell \in L\}$. τ is a distinguished action such that $\tau \notin \mathcal{L}$, called the silent or perfect action. Let $Act = \mathcal{L} \cup \{\tau\}$, with α, β ranging over Act.

A relabelling function $f : \mathcal{L} \to \mathcal{L}$ is a function such that $f(\overline{\ell}) = \overline{f(\ell)}$. It is extended to Act by defining $f(\tau) = \tau$.

Also assume a countable set $\mathcal{X}$ of process variables $X, Y, \ldots$. An arbitrary (possibly infinite) indexing set will be denoted by I. The set $\mathcal{E}$ of process expressions $E, F, \ldots$, is the smallest set including $\mathcal{X}$ and the following expressions:

- $\alpha.E$, a Prefix ($\alpha \in Act$),
- $\Sigma_{i \in I} E_i$, a Summation (when $I = \emptyset$, the summation is written as $\mathbf{0}$ and represents the agent that can perform no action),
- $E_0 \mid E_1$, a Composition,
- $E \backslash L$, a Restriction ($L \subseteq \mathcal{L}$),
- $E[f]$, a Relabelling (f a relabelling function),

- $\mathbf{fix}_j\{X_i = E_i \mid i \in I\}$, a Recursion ($j \in I$) (or alternatively $A \overset{\mathrm{def}}{=} P$, a Constant, where $P \in \mathcal{P}$, the set of agents which are defined to be expressions that contain no free variables. This characterisation of recursion will be used in the sequel because of its more presentable form.)

Let $\mathcal{L}(E)$ denote the syntactic sort of the agent E [50, p. 52]. A sort of an agent E is a subset of $\mathcal{L}$ containing all the actions that E and its derivatives may perform.

A.1.2 Operational semantics

The operational semantics of CCS are defined in terms of a Labelled Transition System, $(\mathcal{E}, Act, \{\xrightarrow{\alpha} \mid \alpha \in Act\})$, where the relation $\xrightarrow{\alpha}$ are defined to be the smallest relation satisfying the following rules:

$$\textbf{Act} \quad \frac{}{\alpha.E \xrightarrow{\alpha} E}$$

$$\textbf{Sum}_\mathbf{j} \quad \frac{E_j \xrightarrow{\alpha} E_j'}{\sum_{i \in I} E_i \xrightarrow{\alpha} E_j'} \quad (j \in I)$$

$$\textbf{Com}_1 \quad \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$$

$$\textbf{Com}_2 \quad \frac{E \xrightarrow{\alpha} E'}{F \mid E \xrightarrow{\alpha} F \mid E'}$$

$$\textbf{Com}_3 \quad \frac{E \xrightarrow{\ell} E' \quad F \xrightarrow{\bar{\ell}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\textbf{Res} \quad \frac{E \xrightarrow{\alpha} E'}{E \backslash L \xrightarrow{\alpha} E' \backslash L} \quad (\alpha, \overline{\alpha} \notin L)$$

$$\textbf{Rel} \quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$\textbf{Con} \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A \overset{\mathrm{def}}{=} P)$$

Examples Consider the following agents:

- $a.0 + \overline{a}.0$ can perform the action a or the action $\overline{a}$.
- $a.0 \mid \overline{a}.0$ can perform the actions a then $\overline{a}$, or the actions $\overline{a}$ then a, or the perfect action τ.
- $(a.0 \mid \overline{a}.0)\backslash a$ can only perform the perfect action τ.

A.2 Strong congruence, observational equivalence and observational congruence

Strong congruence

Strong congruence $\sim$ is defined as follows.

Definition 3 *$P \sim Q$ iff, for all $\alpha \in Act$,*

(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xrightarrow{\alpha} Q'$ and $P' \sim Q'$
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \xrightarrow{\alpha} P'$ and $P' \sim Q'$.

Because */sim* is a congruence it is substitutive under all operators, namely if $P_1 \sim P_2$, then $\alpha.P_1 \sim \alpha.P_2$, $P_1 + Q \sim P_2 + Q$, $P_1 \mid Q \sim P_2 \mid Q$, $P_1\backslash L \sim P_2\backslash L$, $P_1[f] \sim P_2[f]$; and it is also substitutive under recursive definition.

To define observational equivalence $\approx$, a few definitions are required.

Definition 4 *Let $t = \alpha_1 \ldots \alpha_n \in Act^*$. Then*

(i) $\xrightarrow{t} \overset{\text{def}}{=} \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n}$
(ii) $\hat{t} \in \mathcal{L}^*$ is the result of removing all τ's from t
(iii) $\overset{t}{\Rightarrow} \overset{\text{def}}{=} (\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^*$.

Note that $\hat{\tau} = \varepsilon$, the empty sequence, and $\overset{\varepsilon}{\Rightarrow} = (\xrightarrow{\tau})^*$.

Definition 5 *$P \approx Q$ iff, for all $s \in \mathcal{L}^*$,*

(i) Whenever $P \overset{s}{\Rightarrow} P'$ then, for some Q', $Q \overset{s}{\Rightarrow} Q'$ and $P' \approx Q'$
(ii) Whenever $Q \overset{s}{\Rightarrow} Q'$ then, for some P', $P \overset{s}{\Rightarrow} P'$ and $P' \approx Q'$.

However, $\approx$ is not a congruence, therefore the further definition of observational congruence or equality is required.

Definition 6 *$P = Q$ iff, for all $\alpha \in Act$,*

(i) Whenever $P \overset{\alpha}{\Rightarrow} P'$ then, for some Q', $Q \overset{\alpha}{\Rightarrow} Q'$ and $P' \approx Q'$
(ii) Whenever $Q \overset{\alpha}{\Rightarrow} Q'$ then, for some P', $P \overset{\alpha}{\Rightarrow} P'$ and $P' \approx Q'$.

The following result gives the relationship between the three equivalences.

Proposition 4 *$P \sim Q \Rightarrow P = Q \Rightarrow P \approx Q$.*

Finally, the full form of the expansion law is given in the next proposition.

Proposition 5 *Let $P \equiv (P_1[f_1] \mid \ldots \mid P_n[f_n])\backslash L$, with $N \geq 1$. Then*

$$\begin{aligned} P \;\sim\; & \sum\{f_i(\alpha).(P_1[f_1] \mid \ldots \mid P_i'[f_i] \mid \ldots \mid p_n[f_n])\backslash L \mid \\ & \quad P_i \xrightarrow{\alpha} P_i', f_i(\alpha) \notin L \cup \overline{L}\} \\ + & \sum\{\tau.(P_1[f_1] \mid \ldots \mid P_i'[f_i] \mid \ldots \mid P_j'[f_j] \mid \ldots \mid p_n[f_n])\backslash L \mid \\ & \quad P_i \xrightarrow{\ell_1} P_i', P_j \xrightarrow{\ell_2} P_j',\ f_i(\ell_1) = \overline{f_j(\ell_2)},\ i < j\}. \end{aligned}$$

Example The following agents are equated by the Expansion Law

$a.0 \mid b.0 \sim a.b.0 + b.a.0.$

A.3 Determinism and confluence

Strong and weak determinacy can be defined as follows:

Definition 7 *P is strongly determinate if, for every derivative Q of P and for all $\alpha \in Act$, whenever $Q \xrightarrow{\alpha} Q'$ and $Q \xrightarrow{\alpha} Q''$ then $Q' \sim Q''$.*

Definition 8 *P is (weakly) determinate if, for every derivative Q of P and for all $s \in \mathcal{L}^*$, wheneve $Q \overset{s}{\Rightarrow} Q'$ and $Q \overset{s}{\Rightarrow} Q''$ then $Q' \sim Q''$.*

The concept behind determinacy is predictability. If an experiment is performed on a determinate system, the same result or behaviour should be obtained each time that experiment is performed. It would be desirable to have a set of rules

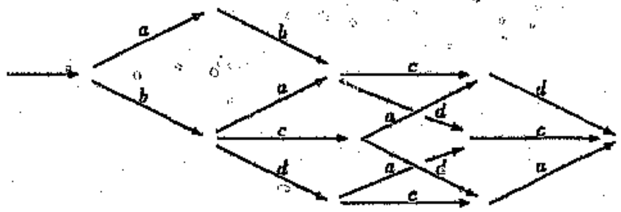

Figure A.1: The confluent agent $a.0 \mid b.(c.0 \mid d.0)$ [50, p. 238].

that allow determinate systems to be constructed from determinate components. However, there is no constrained form of Composition that is determinate and allows communication. Therefore confluence, a stronger form of determinacy, has been proposed.

Definition 9 *r/s, the excess of r over s, is defined as follows*

$$\begin{aligned} \varepsilon/s &= \varepsilon \\ (\ell r)/s &= \begin{cases} \ell(r/s) & \text{if } \ell \notin s \\ r/(s/\ell) & \text{if } \ell \in s. \end{cases} \end{aligned}$$

Definition 10 *P is confluent iff, for all $r, s \in \mathcal{L}^*$, the following diagram can be completed*

$$\begin{tikzcd} P \arrow[r, Rightarrow, "r"] \arrow[d, Rightarrow, "s"'] & & P_1 \arrow[d, Rightarrow, "s/r"] \\ P_2 \arrow[r, Rightarrow, "r/s"] & Q_2 \arrow[r, phantom, "\approx"] & Q_1 \end{tikzcd}$$

The idea behind confluence is that when performing an experiment different actions may occur on different orders however the final result will be the same, or more concisely stated—the occurrence of one action will not preclude the possibility of another. In Figure A.1, an example of a confluent agent is given.

The following results describe which syntactic forms are confluent, and which operators preserve confluence.

Proposition 6 *If P is confluent, then so are the following:*

(i) $\mathbf{0}, \alpha.P$ *and* $P\backslash L$

(ii) $P[f]$, *provided that f is injective on $\mathcal{L}(P)$.*

Definition 11 *For $\alpha_1, \ldots, \alpha_n \in Act, n \geq 0$, the Confluent Sum $(\alpha_1 \mid \ldots \mid \alpha_n).P$ is defined recursively as follows.*

$$\begin{aligned} ().P &\overset{\text{def}}{=} P \\ (\alpha_1 \mid \ldots \mid \alpha_n).P &\overset{\text{def}}{=} \sum_{1\leq i\leq n} \alpha_i.(\alpha_1 \mid \ldots \mid \alpha_{i-1} \mid \alpha_{i+1} \mid \ldots \mid \alpha_n).P \quad (n > 0). \end{aligned}$$

Proposition 7 *If P is confluent then so is $(\alpha_1 \mid \ldots \mid \alpha_n).P$.*

Definition 12 *For $L \subset \mathcal{L}$ the Restricted Composition is defined as*

$$P_1 \mid_L P_2 \overset{\text{def}}{=} (P_1 \mid P_2)\backslash L$$

and it is called a Confluent Composition if $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$ and $\overline{\mathcal{L}(P_1)} \cap \mathcal{L}(P_2) \subseteq L \cup \bar{L}$.

Proposition 8 *Let P_1 and P_2 be confluent. Then if $P_1 \mid_L P_2$ is a Confluent Composition it is confluent.*

Proposition 9 *Let P, and all Constants upon which it depends be defined using only $\mathbf{0}$, one-to-one Relabelling, Confluent Sum, Restriction, Confluent Composition and Constants. Then P is confluent.*

B. Proofs

In this appendix, proofs of propositions and theorems from Chapter 3 will be presented.

Proposition 10 $\preceq'$ *is a partial order*
Proof: To show that $\preceq'$ is a partial order, it must be shown that $\preceq'$ is reflexive, transitive and antisymmetric relation. To recap, $\preceq'$ is equal to the transitive closure of $(\prec \cup =)$.

1. Reflexivity : This follows from equality.
2. Transitivity : This follows from the fact that $\preceq'$ is defined to be the transitive closure.
3. Antisymmetry: Given $(a \preceq' b) \wedge (b \preceq' a)$, there are four cases to be considered.
   (a) $(a \prec b) \wedge (b = a) \Rightarrow (a = b)$
   (b) $(a = b) \wedge (b \prec a) \Rightarrow (a = b)$
   (c) $(a = b) \wedge (b = a) \Rightarrow (a = b)$
   (d) $(a \prec b) \wedge (b \prec a)$. This case cannot occur because of the definition of $\prec$.

□

Proposition 11 $\preceq$ *is a partial order*
Proof: $\preceq$ is defined to be the transitive closure of $(\prec \cup =)$ with the additional rule that

$$\forall a, b \in E(\mathcal{P})\, a \leftrightarrow b \Rightarrow a = b.$$

As can be seen from the previous proof, this is a partial order. □

Proposition 12 *(Proposition 3) The cut C defined by E_C and a_i for each $i \in N \backslash E_C$,*

$$C = \bigcup_{i \in N \backslash E_C} \{x \in E(P_i) \mid x \preceq a_i\} = \bigcup_{i \in N \backslash E_C} (\downarrow a_i)_i$$

is a consistent cut if and only if

$$\sup(T_1, \ldots, T_n) = (T_1[1], \ldots, T_n[n]).$$

Proof:

Note that in the following proof it is not necessary to consider individually the cases $P_i \mapsto P_j$, $P_j \mapsto P_i$ and the case where neither is caused by the other, since this is captured in the relation $\preceq$.

$\Rightarrow$: Assume that C defined by E_C and a_i for $i \in N \setminus E_C$, is a consistent cut, namely $\forall a \in C, b \in E(\mathcal{P})\ (b \preceq a \Rightarrow b \in C)$. It must be shown that

$$\begin{aligned} & \sup(T_1, \ldots, T_n) = (T_1[1], \ldots, T_n[n]) \\ \Leftrightarrow\ & \forall i \in \{1, \ldots, n\}\ \max_{1 \le j \le n} T_j[i] = T_i[i] \\ \Leftrightarrow\ & \forall i, j \in \{1, \ldots, n\}\ T_j[i] \le T_i[i]. \end{aligned}$$

For any i and j, there are four possible cases to consider:

1. $i, j \in E_C$: then $T_j[i] = T_i[i] = 0$ so the condition holds.
2. $i \notin E_C$ and $j \in E_C$: then $T_j[i] = 0 < T_i[i]$.
3. $i \in E_C$ and $j \notin E_C$: hence $T_j[i] > 0$, therefore there exists a communication event from P_i, $b \in E(P_i)$ such that $b \preceq a_j$ and $b \notin C$ since P_i does not contribute any events to C. This contradicts the fact that C is a consistent cut, so this case cannot hold.
4. $i, j \notin E_C$: consider a_i and a_j. They can be related to each other in the following ways:
   (a) a_j co a_i : From Proposition 2, this implies $\Theta(a_j)[i] < \Theta(a_i)[i]$, hence $T_j[i] < T_i[i]$ by the definition of T, so the required condition holds.
   (b) $a_j \preceq a_i$: This implies that $\Theta(a_j) \le \Theta(a_i) \Leftrightarrow \Theta(a_j)[i] \le \Theta(a_i)[i] \Leftrightarrow T_j[i] \le T_i[i]$.
   (c) $a_i \prec a_j$: Therefore $\Theta(a_i) < \Theta(a_j) \Leftrightarrow \Theta(a_i)[i] < \Theta(a_j)[i]$. Then from Proposition 1, $|\ (\downarrow a_i)_i\ | < |\ (\downarrow a_j)_i\ |$. Now since $\preceq$ is a total order on $E(P_i)$, $(\downarrow a_i)_i \subset (\downarrow a_j)_i$. Therefore $\exists\, b \in E(P_i)$, such that $b \in (\downarrow a_j)_i$ but

$b \notin (\downarrow a_i)_i$, therefore $b \notin C$, but $b \preceq a_j$, which contradicts the definition of the cut C, and so this case cannot hold.

So it has been shown that in all cases satisfying the definition of a consistent cut, the required conditions hold.

$\Leftarrow$: Assume the following holds for C, a cut defined by E_C and a_i for $i \in N \backslash E_C$.

$$\begin{aligned} & \sup(T_1, \ldots, T_n) = (T_1[1], \ldots, T_n[n]) \\ \Leftrightarrow \quad & \forall i \in \{1, \ldots, n\} \ \max_{1 \le j \le n} T_j[i] = T_i[i] \\ \Leftrightarrow \quad & \forall i, j \in \{1, \ldots, n\} \ T_j[i] \le T_i[i] \end{aligned}$$

It must be shown that C is a consistent cut, namely $\forall a \in C, b \in E(\mathcal{P}) \, (b \preceq a \Rightarrow b \in C)$. Suppose the converse, $\exists \, a \in C, b \in E(\mathcal{P})$ such that $b \preceq a$ and $b \notin C$. There are two cases to consider:

1. $a, b \in E(P_i) \, (i \in N \backslash E_C)$: $a \in C$ and $a \in E(P_i)$ implies $a \in (\downarrow a_i)_i$ which implies that $a \preceq a_i$, but $b \preceq a$ so by transitivity $b \preceq a_i$ which implies that $b \in (\downarrow a_i)_i$ by definition and hence $b \in C$. Contradiction.

2. $a \in E(P_j), b \in E(P_i) \, (i, j \in N \backslash E_C, i \ne j)$: $a \in C$ and $a \in E(P_j)$ implies that $a \in (\downarrow a_j)_j$ and $a \preceq a_j$. Also $b \notin C$ and $b \in E(P_i)$ implies $b \notin (\downarrow a_i)_i$ and $a_i \preceq b$.

   Since $a_i \preceq b$, $(\downarrow a_i)_i \subset (\downarrow b)_i$ and $\Theta(a_i)[i] < \Theta(b)[i]$ from Proposition 1.

   Also $b \preceq a, a \preceq a_j \Rightarrow b \preceq a_j$, so $\Theta(b) \le \Theta(a_j)$ and $\Theta(b)[i] \le \Theta(a_j)[i]$.

   Therefore $\Theta(a_i)[i] < \Theta(a_j)[i]$ and by the definition of T_i, T_j, $T_i[i] < T_j[i]$ since $i, j \in N \backslash E_C$. Hence $\exists \, i, j \in \{1, \ldots, n\}$ such that $T_i[i] < T_j[i]$. Contradiction.

Therefore it has been proved that C is a consistent cut. $\square$

C. Agents for the dining philosophers problem

In this appendix, the agents used ~~in~~ Experiments G and H in Section 5.2 are presented. The actions are abbreviations for the following:

t	-	think
e	-	eat
gf_i	-	get the ith fork
df_i	-	drop the ith fork
gt	-	get room ticket
[illegible]	-	drop room ticket

G_1: 2 philosophers, odd-even solution

$(\quad t.gf_1.gf_2.e.df_1.df_2.0 \mid$	philosopher$_1$
$t.gf_1.gf_2.e.df_1.df_2.0 \mid$	philosopher$_2$
$\overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid$	fork$_1$
$\overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0$	fork$_2$
$) \setminus \{gf_1, df_1, gf_2, df_2\}$	

G_2: 4 philosophers, odd-even solution

$(\quad t.gf_1.gf_4.e.df_1.df_4.0 \mid$	philosopher$_1$
$t.gf_1.gf_2.e.df_1.df_2.0 \mid$	philosopher$_2$
$t.gf_3.gf_2.e.df_3.df_2.0 \mid$	philosopher$_3$
$t.gf_3.gf_4.e.df_3.df_4.0 \mid$	philosopher$_4$
$\overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid$	fork$_1$
$\overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid$	fork$_2$
$\overline{gf}_3.\overline{df}_3.\overline{gf}_3.\overline{df}_3.0 \mid$	fork$_3$
$\overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.0$	fork$_4$
$) \setminus \{gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\}$	

G_3: 6 philosophers, odd-even solution

$(\quad t.gf_1.gf_6.e.df_1.df_6.0 \mid$	philosopher$_1$
$t.gf_1.gf_2.e.df_1.df_2.0 \mid$	philosopher$_2$
$t.gf_3.gf_2.e.df_3.df_2.0 \mid$	philosopher$_3$
$t.gf_3.gf_4.e.df_3.df_4.0 \mid$	philosopher$_4$
$t.gf_5.gf_4.e.df_5.df_4.0 \mid$	philosopher$_5$
$t.gf_5.gf_6.e.gf_5.gf_6.0 \mid$	philosopher$_6$
$\overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid$	fork$_1$
$\overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid$	fork$_2$
$\overline{gf}_3.\overline{df}_3.\overline{gf}_3.\overline{df}_3.0 \mid$	fork$_3$
$\overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.0 \mid$	fork$_4$
$\overline{gf}_5.\overline{df}_5.\overline{gf}_5.\overline{df}_5.0 \mid$	fork$_5$
$\overline{gf}_6.\overline{df}_6.\overline{gf}_6.\overline{df}_6.0$	fork$_6$
$) \ \backslash\{gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4, gf_5, df_5, gf_6, df_6\}$	

G_4: 2 philosophers, room-ticket solution

This example is more complex because of the introduction of room-tickets and requires the calculation of the measure for two different possible runs of the agent (There are more runs, but these relate to permutations on the order of processes). The first agent involving only two philosophers has only one possible run because of the existence of only one room-ticket.

$(\quad t.gt.gf_1.gf_2.e.df_1.df_2.dt.0 \mid$	philosopher$_1$
$t.gt.gf_2.gf_1.e.df_2.df_1.dt.0 \mid$	philosopher$_2$
$\overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid$	fork$_1$
$\overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid$	fork$_2$
$\overline{gt}.\overline{dt}.\overline{gt}.\overline{dt}.0$	room − tickets
$) \ \backslash\{gt, dt, gf_1, df_1, gf_2, df_2\}$	

G_5: 4 philosophers, room-ticket solution

This situation now involves a number of different computations for the same agent. This occurs, because any one of three room-tickets can generate a new copy of itself to let the fourth philosopher into the room. The agent is described first, and then the two different computations are given. The same applies to Experiments H_4 and H_5, although only the agents for those experiments are given in this appendix.

The agent

$$
\begin{array}{lll}
(& t.gt.gf_1.gf_4.e.df_1.df_4.dt.0 \mid & \text{philosopher}_1 \\
 & t.gt.gf_2.gf_1.e.df_2.df_1.dt.0 \mid & \text{philosopher}_2 \\
 & t.gt.gf_3.gf_2.e.df_3.df_2.dt.0 \mid & \text{philosopher}_3 \\
 & t.gt.gf_4.gf_3.e.df_4.df_3.dt.0 \mid & \text{philosopher}_4 \\
 & \overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid & \text{fork}_1 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid & \text{fork}_2 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid & \text{fork}_3 \\
 & \overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.0 \mid & \text{fork}_4 \\
 & \overline{gt}.\overline{dt}.(\overline{gt}.\overline{dt}.0 + 0) \mid & \text{room} - \text{tickets} \\
 & \overline{gt}.\overline{dt}.(\overline{gt}.\overline{dt}.0 + 0) \mid & \\
 & \overline{gt}.\overline{dt}.(\overline{gt}.\overline{dt}.0 + 0) & \\
) & \backslash\{gt, dt, gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\} &
\end{array}
$$

The first computation

$$\begin{array}{lll}
(& t.gt.gf_1.gf_4.e.df_1.df_4.dt.0 \mid & \text{philosopher}_1 \\
 & t.gt.gf_2.gf_1.e.df_2.df_1.dt.0 \mid & \text{philosopher}_2 \\
 & t.gt.gf_3.gf_2.e.df_3.df_2.dt.0 \mid & \text{philosopher}_3 \\
 & t.gt.gf_4.gf_3.e.df_4.df_3.dt.0 \mid & \text{philosopher}_4 \\
 & \overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid & \text{fork}_1 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid & \text{fork}_2 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid & \text{fork}_3 \\
 & \overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.0 \mid & \text{fork}_4 \\
 & \overline{gt}.\overline{dt}.\overline{gt}.\overline{dt}.0 \mid & \text{room} - \text{tickets} \\
 & \overline{gt}.\overline{dt}.0 \mid & \\
 & \overline{gt}.\overline{dt}.0 & \\
) & \backslash\{gt, dt, gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\} &
\end{array}$$

The second computation

$$\begin{array}{lll}
(& t.gt.gf_1.gf_4.e.df_1.df_4.dt.0 \mid & \text{philosopher}_1 \\
 & t.gt.gf_2.gf_1.e.df_2.df_1.dt.0 \mid & \text{philosopher}_2 \\
 & t.gt.gf_3.gf_2.e.df_3.df_2.dt.0 \mid & \text{philosopher}_3 \\
 & t.gt.gf_4.gf_3.e.df_4.df_3.dt.0 \mid & \text{philosopher}_4 \\
 & \overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid & \text{fork}_1 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid & \text{fork}_2 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid & \text{fork}_3 \\
 & \overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.0 \mid & \text{fork}_4 \\
 & \overline{gt}.\overline{dt}.\overline{gt}.\overline{dt}.0 \mid & \text{room} - \text{tickets} \\
 & \overline{gt}.\overline{dt}.\overline{gt}.\overline{dt}.0 \mid & \\
) & \backslash\{gt, dt, gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\} &
\end{array}$$

H_1: 4 philosophers, odd-even solution, 2 eat, 2 think

$$\begin{array}{lll}
(& t.t.gf_1.gf_4.e.e.df_1.df_4.\mathbf{0} \mid & \text{philosopher}_1 \\
 & t.t.gf_1.gf_2.e.e.df_1.df_2.\mathbf{0} \mid & \text{philosopher}_2 \\
 & t.t.gf_3.gf_2.e.e.df_3.df_2.\mathbf{0} \mid & \text{philosopher}_3 \\
 & t.t.gf_3.gf_4.e.e.df_3.df_4.\mathbf{0} \mid & \text{philosopher}_4 \\
 & \overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.\mathbf{0} \mid & \text{fork}_1 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.\mathbf{0} \mid & \text{fork}_2 \\
 & \overline{gf}_3.\overline{df}_3.\overline{gf}_3.\overline{df}_3.\mathbf{0} \mid & \text{fork}_3 \\
 & \overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.\mathbf{0} & \text{fork}_4 \\
) & \backslash\{gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\} &
\end{array}$$

H_2: 4 philosophers, odd-even solution, 4 eat, 4 think

$$\begin{array}{lll}
(& t.t.t.t.gf_1.gf_4.e.e.e.e.df_1.df_4.\mathbf{0} \mid & \text{philosopher}_1 \\
 & t.t.t.t.gf_1.gf_2.e.e.e.e.df_1.df_2.\mathbf{0} \mid & \text{philosopher}_2 \\
 & t.t.t.t.gf_3.gf_2.e.e.e.e.df_3.df_2.\mathbf{0} \mid & \text{philosopher}_3 \\
 & t.t.t.t.gf_3.gf_4.e.e.e.e.df_3.df_4.\mathbf{0} \mid & \text{philosopher}_4 \\
 & \overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.\mathbf{0} \mid & \text{fork}_1 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.\mathbf{0} \mid & \text{fork}_2 \\
 & \overline{gf}_3.\overline{df}_3.\overline{gf}_3.\overline{df}_3.\mathbf{0} \mid & \text{fork}_3 \\
 & \overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.\mathbf{0} & \text{fork}_4 \\
) & \backslash\{gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\} &
\end{array}$$

H_3: 4 philosophers, odd-even solution, 8 eat, 8 think

$$\begin{array}{lll}
(& t.t.t.t.t.t.t.t.gf_1.gf_4.e.e.e.e.e.e.e.e.df_1.df_4.\mathbf{0} \mid & \text{philosopher}_1 \\
 & t.t.t.t.t.t.t.t.gf_1.gf_2.e.e.e.e.e.e.e.e.df_1.df_2.\mathbf{0} \mid & \text{philosopher}_2 \\
 & t.t.t.t.t.t.t.t.gf_3.gf_2.e.e.e.e.e.e.e.e.df_3.df_2.\mathbf{0} \mid & \text{philosopher}_3 \\
 & t.t.t.t.t.t.t.t.gf_3.gf_4.e.e.e.e.e.e.e.e.df_3.df_4.\mathbf{0} \mid & \text{philosopher}_4 \\
 & \overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.\mathbf{0} \mid & \text{fork}_1 \\
 & \overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.\mathbf{0} \mid & \text{fork}_2 \\
 & \overline{gf}_3.\overline{df}_3.\overline{gf}_3.\overline{df}_3.\mathbf{0} \mid & \text{fork}_3 \\
 & \overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.\mathbf{0} & \text{fork}_4 \\
) & \backslash\{gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\} &
\end{array}$$

H_4: 4 ph[illegible]phers, room-ticket solution, 2 eat, 2 think

The agent

$(\quad t.t.gt.gf_1.gf_4.e.e.df_1.df_4.dt.0 \mid$	philosopher$_1$
$t.t.gt.gf_2.gf_1.e.e.df_2.df_1.dt.0 \mid$	philosopher$_2$
$t.t.gt.gf_3.gf_2.e.e.df_3.df_2.dt.0 \mid$	philosopher$_3$
$t.t.gt.gf_4.gf_3.e.e.df_4.df_3.dt.0 \mid$	philosopher$_4$
$\overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid$	fork$_1$
$\overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid$	fork$_2$
$\overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid$	fork$_3$
$\overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.0 \mid$	fork$_4$
$\overline{gt}.\overline{dt}.\overline{gt}.\overline{dt}.0 \mid$	room $-$ tickets
$\overline{gt}.\overline{dt}.0 \mid$	
$\overline{gt}.\overline{dt}.0$	
$) \setminus \{gt, dt, gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\}$	

H_5: 4 philosophers, room-ticket solution, 4 eat, 4 think

The agent

$(\quad t.t.t.t.gt.gf_1.gf_4.e.e.e.e.df_1.df_4.dt.0 \mid$	philosopher$_1$
$t.t.t.t.gt.gf_2.gf_1.e.e.e.e.df_2.df_1.dt.0 \mid$	philosopher$_2$
$t.t.t.t.gt.gf_3.gf_2.e.e.e.e.df_3.df_2.dt.0 \mid$	philosopher$_3$
$t.t.t.t.gt.gf_4.gf_3.e.e.e.e.df_4.df_3.dt.0 \mid$	philosopher$_4$
$\overline{gf}_1.\overline{df}_1.\overline{gf}_1.\overline{df}_1.0 \mid$	fork$_1$
$\overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid$	fork$_2$
$\overline{gf}_2.\overline{df}_2.\overline{gf}_2.\overline{df}_2.0 \mid$	fork$_3$
$\overline{gf}_4.\overline{df}_4.\overline{gf}_4.\overline{df}_4.0 \mid$	fork$_4$
$\overline{gt}.\overline{dt}.\overline{gt}.\overline{dt}.0 \mid$	room $-$ tickets
$\overline{gt}.\overline{dt}.0 \mid$	
$\overline{gt}.\overline{dt}.0$	
$) \setminus \{gt, dt, gf_1, df_1, gf_2, df_2, gf_3, df_3, gf_4, df_4\}$	

References

[1] A. Arnold and M. Nivat. Comportements de processus. Technical Report 82-12, Laboratoire Informatique Théorique et Programmation, Institut Blaise Pascal, 1982.

[2] D. Arques, J. Françon, M. Guichet, and P. Guichet. Comparison of algorithms controlling concurrent access to a database: a combinatorial approach. In L. Kott (ed), *Proceedings of 13th ICALP*, Lecture Notes in Computer Science **226**, 11–20. Springer-Verlag, 1986.

[3] N. Bambos and J. Walrand. On stability and performance of parallel processing systems. *Journal of the ACM*, **38**(2), 429–452, April 1991.

[4] V. Barbosa. Blocking versus non-blocking interprocess communication: a note on the effect on concurrency. *Information Processing Letters*, **36**, 171–175, 1990.

[5] J. Beauquier, B. Bérard, and L. Thimonier. On a concurrency measure. Technical report, L.R.I., Univ. Paris-Sud, Orsay, 1986.

[6] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, **60**, 109–137, 1984.

[7] E. Best and R. Devillers. Interleaving and partial orders in concurrency: a formal comparison. In M. Wirsing (ed), *Formal Description of Programming Concepts III*, 299–322. North-Holland, 1987.

[8] G. Boudol. Notes on algebraic calculi of processes. In K. Apt (ed), *Logics and models of concurrent systems*, Volume F13 of *Nato ASI Series*, 261–303. Springer-Verlag, 1984.

[9] G. Boudol and I. Castellani. On the semantics of concurrency: partial orders and transition systems. In R. Ehrig, R. Kowalski, G. Levi, and U. Montanari (eds), *Proceedings of CAAP '87*, Lecture Notes in Computer Science **249**, 121–137. Springer-Verlag, 1987.

[10] G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informatica*, **XI**, 433–452, 1988.

[11] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, **31**(3), 560–599, July 1984.

[12] N. Carriero and D. Gelernter. *How to write parallel programs.* MIT Press, 1990.

[13] I. Castellani and M. Hennessy. Distributed bisimulations. *Journal of the ACM*, **36**(4), 887–911, October 1989.

[14] B. Charron-Bost. Combinatorics and geometry of consistent cuts: application to concurrency theory. In J.-C. Bermond and M. Raynal (eds), *Distributed Algorithms*, Lecture Notes in Computer Science **392**, 45–56. Springer-Verlag, 1989.

[15] B. Charron-Bost. Measure of parallelism of distributed computations. In B. Monien and R. Cori (eds), *STACS '89*, Lecture Notes in Computer Science **349**, 434–445. Springer-Verlag, 1989.

[16] B. Charron-Bost. *Mesures de la concurrence et du parallélisme des calculs répartis.* PhD thesis, Informatique, Universite Paris VII, 1989.

[17] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, **39**, 11–16, 1991.

[18] B. Charron-Bost. Coupling coefficients of a distributed execution. Technical Report 92.07, Laboratoire Informatique Théorique et Programmation, Institut Blaise Pascal, Paris, 1992.

[19] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and asynchronous communication in distributed computations. Technical Report 92.77, Laboratoire Informatique Théorique et Programmation, Institut Blaise Pascal, Paris, 1992.

[20] P. Crawley and R. Dilworth. *Algebraic Theory of Lattices.* Prentice-Hall, 1973.

[21] F. Croxton, D. Cowdon, and S. Klein. *Applied General Statistics.* Prentice-Hall, 1967.

[22] P. Degano, R. De Nicola, and U. Montanari. Partial ordering derivations for CCS. In *Proceedings of FCT '85*, Lecture Notes in Computer Science **199**, 520–533. Springer-Verlag, 1985.

[23] P. Degano, R. De Nicola, and U. Montanari. CCS is an (augmented) contact-free C/E system. In M. Venturini Zilli (ed), *Proceedings of the Advanced School on Mathematical Models for the Semantics of Parallelism*, Lecture Notes in Computer Science **280**, 144–165. Springer-Verlag, 1987.

[24] P. Degano, R. De Nicola, and U. Montanari. Observational equivalences for concurrency models. In M. Wirsing (ed), *Formal Description of Programming Concepts III*, 105–132. North-Holland, 1987.

[25] P. Degano, R. De Nicola, and U. Montanari. A distributed operational semantics for CCS based on condition/event systems. *Acta Informatica*, **26**, 59–91, 1988.

[26] P. Degano, R. De Nicola, and U. Montanari. Partial ordering descriptions and observations of nondeterministic concurrent processes. In J. de Bakker, W.-P. de Roever, and G. Rozenberg (eds), *Linear time, branching time and partial orders in logics and models for concurrency*, Lecture Notes in Computer Science **354**, 438–466. Springer-Verlag, 1988.

[27] P. Degano, R. De Nicola, and U. Montanari. A partial ordering semantics for CCS. *Theoretical Computer Science*, **75**, 223–262, 1990.

[28] E. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, **1**, 115–138, 1971.

[29] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, **38**(3), 408–423, March 1989.

[30] C. Fidge. Time stamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, **10**(1), 56–66, 1988.

[31] C. Fidge. *Dynamic analysis of event orderings in message-passing systems*. PhD thesis, Department of Computer Science, Australian National University, 1989.

[32] C. Fidge. A simple run-time concurrency measure. In T. Bossomaier, T. Hintz, and J. Hulskamp (eds), *The Transputer in Australasia (ATOUG-3)*, 92–101. IOS Press, 1990.

[33] C. Fidge. Logical time in distributed computing systems. *Computer*, **24**(8), 28–33, August 1991.

[34] C. Fidge. Personal communication, April 1992.

[35] P. Flajolet, D. Gardy, and L. Thimonier. Random allocations and probabilistic languages. In T. Lepistö and A. Salomaa (eds), *Proceedings of 15th ICALP*, Lecture Notes in Computer Science **317**, 239–253. Springer-Verlag, 1988.

[36] J. Françon. Une approche quantitative de l'exclusion mutuelle (A quantitative approach to mutual exclusion). *Theoretical Informatics and Applications*, **20**(3), 275–289, 1986.

[37] D. Geniet, R. Schott, and L. Thimonier. A Markovian concurrency measure. In A. Arnold (ed), *Proceedings of CAAP '90*, Lecture Notes in Computer Science **431**, 177–190. Springer-Verlag, 1990.

[38] D. Geniet and L. Thimonier. Using generating functions to compute concurrency. In J. Csirik, J. Demetrovics, and F. Gécseg (eds), *FCT '89*, Lecture Notes in Computer Science **380**, 185–196. Springer-Verlag, 1989.

[39] U. Goltz and A. Mycroft. On the relationship of CCS and Petri nets. In J. Paredaens (ed), *Proceedings of 11th ICALP*, Lecture Notes in Computer Science **172**, 196–208. Springer-Verlag, 1984.

[40] J. Gustafson. Re-evaluating Amdahl's law. *Communications of the ACM*, **31**(5), 532–533, May 1988.

[41] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[42] J. Hirshfeld, A. Rabinovich, and B. Trakhtenbrot. Discerning causality in interleaving behaviour. In A. Meyer and M. Taitslin (eds), *Logic at Botik '89*, Lecture Notes in Computer Science **363**, 146–162. Springer-Verlag, 1989.

[43] A. Karp and H. Flatt. Measuring parallel processor performance. *Communications of the ACM*, **33**(5), 537–543, May 1990.

[44] M. Kim, S. Chanson, and S. Vuong. Concurrency model and its application to formal protocol specifications. In *IEEE Infocom '93*, 1993. To appear.

[45] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, **37**(9), 1088–1098, September 1988.

[46] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7), 558–565, July 1978.

[47] L. Lamport and N. Lynch. Distributed computing: models and methods. In J. van Leeuwen (ed), *Handbook of Theoretical Computer Science*, Volume B, 1158–1199. Elsevier Science Publishers, 1990.

[48] P. Lauer, P. Torrigiani, and M. Shields. COSY - a system specification language based on paths and processes. *Acta Informatica*, **12**, 109–158, 1979.

[49] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, P. Quinton, Y. Robert, and M. Raynal (eds), *Parallel and Distributed Algorithms*, 215–226. North-Holland, 1989.

[50] R. Milner. *Communication and concurrency*. Prentice Hall, 1989.

[51] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen (ed), *Handbook of Theoretical Computer Science*, Volume B, 1201–1242. Elsevier Science Publishers, 1990.

[52] F. Moller. Axioms for concurrency. PhD Thesis, Report No. CST-59-89, Department of Computer Science, University of Edinburgh, 1989.

[53] D. Murphy. Approaching a real-timed concurrency theory. In M. Kwiatkowska, M. Shields, and R. Thomas (eds), *Semantics for Concurrency*, 295–310. Springer-Verlag, 1990.

[54] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, to appear.

[55] S. Purushothaman and P. Subrahmanyam. Reasoning about probabilistic behaviour in concurrent systems. *IEEE Transactions on Software Engineering*, **SE-13**(6), 740–745, June 1987.

[56] J. Quemada and A. Fernadez. Introduction of quantitative relative time into LOTOS. In *Proceedings of the Seventh Protocol Specification, Testing and Verification Symposium*, 1987.

[57] M. Quinn. *Designing efficient algorithms for parallel computers*. McGraw-Hill, 1987.

[58] M. Raynal. About logical clocks for distributed systems. *Operating Systems Review*, **26**(1), 41–48, January 1992.

[59] M. Raynal, M. Mizuno, and M. Neilsen. Synchronisation and concurrency measures for distributed computations. Technical Report 1539, INRIA, October 1991. To appear in IEEE International Conference on Distributed Computer Systems 1992.

[60] G. Reed and A. Roscoe. A timed model for communicating sequential processes. In L. Kott (ed), *Proceedings of 13th ICALP*, Lecture Notes in Computer Science **226**, 314–321. Springer-Verlag, 1986.

[61] J. Robinson. Some analysis techniques for asynchronous multiprocessor algorithms. *IEEE Transactions on Software Engineering*, SE-5(1), 24–31, January 1979.

[62] F. Schmuck. The use of efficient broadcast in asynchronous distributed systems. PhD Thesis, TR 88-828, Cornell University, 1988.

[63] K. Sevcik. Characterisations of parallelism in applications and their use in scheduling. *Performance Evaluation Review*, **17**(1), 171–180, May 1989.

[64] G. T'Hooft. Timer description in CCS (Milner), LOTOS (ISO) and Timed lotos (Quemada-Fernadez). *ACM SIGCOMM Computer Communication Review*, 18(1/2), 31–62, January/April 1988.

[65] C. Tofts. Timed concurrent processes. In M. Kwiatkowska, M. Shields, and R. Thomas (eds), *Semantics for Concurrency*, 281–294. Springer-Verlag, 1990.

[66] R. van Glabbeek and F. Vaandrager. Petri net models for algebraic theories of concurrency. In J. de Bakker, A. Nijman, and P. Treleaven (eds), *Proceedings of PARLE '87*, Lecture Notes in Computer Science **259**, 224–242. Springer-Verlag, 1987.