Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

92

Robin Milner

A Calculus of Communicating Systems

Springer-Verlag
Berlin Heidelberg New York

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

92

Robin Milner

A Calculus of Communicating Systems

Springer-Verlag
Berlin Heidelberg New York 1980

Author

Robin Milner
University of Edinburgh
Dept. of Computer Science
James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh EH9 3JZ
Great Britain

AMS Subject Classifications (1979): 68-02
CR Subject Classifications (1974): 4.30, 5.20, 5.22, 5.24

ISBN 3-540-10235-3 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-10235-3 Springer-Verlag New York Heidelberg Berlin

Library of Congress Cataloging in Publication Data. Milner, Robin. A calculus of communicating systems. (Lecture notes in computer science; 92) Bibliography: p. Includes index. 1. Machine theory. 2. Formal languages. I. Title. II. Series. QA267.M53. 511.3 80-21068

Printed in Germany

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

ACKNOWLEDGEMENTS

This work was mainly done during my six-month appointment, from August 1979 to January 1980, at the Computer Science department in Aarhus University, Denmark. Although much of the ground work had been done previously it was mainly in response to their encouragement (to make the theory more accessible and related to practice), and to their informed criticism, that the material reached a somewhat coherent form. I am deeply grateful to them and their students for allowing me to lecture once a week on what was, at first, a loosely connected set of ideas, and for the welcoming environment in which I was able to put the ideas in order. I also thank Edinburgh University for awarding me five months sabbatical leave subsequently, which helped me to complete the task in a reasonable time.

The calculus presented here grew out of work which was inspired by Dana Scott's theory of computation, though it has since diverged in some respects. At every stage I have been influenced by Gordon Plotkin; even where I cannot trace particular ideas to him I have been greatly illuminated by our discussions and by his chance remarks, and without them the outcome would certainly be less than it is. I would also like to thank others with whom I have worked: George Milne, with whom I worked out the Laws of Flow Algebra; Matthew Hennessy, with whom the notion of observation equivalence developed; and Tony Hoare, whose parallel work on different but strongly related ideas, expressed in his "Communicating Sequential Processes", has been a strong stimulus.

Many people have given detailed and helpful criticisms of the manuscript, and thus improved its final form. In particular I thank Michael Gordon and David MacQueen, who went through it all in detail in a Seminar at the Information Sciences Institute, University of California; this not only exposed some mistakes and obscurities but gave me more confidence in the parts they didn't criticise.

Finally, I am very thankful to Dorothy McKie and Gina Temple for their patience and skill in the long and involved task of typing.

CONTENTS

0. Introduction 1
Purpose - Character - Related Work - Evolution - Outline.

1. Experimenting on Nondeterministic Machines 9
Traditional equivalence of finite state acceptors - Experimenting upon acceptors - Behaviour as a tree - Algebra of RSTs - Unobservable actions.

2. Synchronization 19
Mutual experimentation - Composition, restriction and relabelling - Extending the Algebra of STs - A simple example: binary semaphores - The ST Expansion Theorem.

3. A case study in synchronization and proof techniques 33
A scheduling problem - Building the scheduler as a Petri Net - Observation equivalence - Proving the scheduler.

4. Case studies in value-communication 47
Review - Passing values - An example: Data Flow - Derivations - An example: Zero searching.

5. Syntax and Semantics of CCS 65
Introduction - Syntax - Semantics by derivations - Defining behaviour identifiers - Sorts and programs - Direct equivalence of behaviour programs - Congruence of behaviour programs - Congruence of behaviour expressions and the Expansion Theorem.

6. Communication Trees (CTs) as a model of CCS 84
CTs and the dynamic operations - CTs and the static operations - CTs defined by recursion - Atomic actions and derivations of CTs - Strong equivalence of CTs - Equality in the CT model - Summary.

7. Observation equivalence and its properties 98
Review - Observation equivalence in CCS - Observation congruence - Laws of observation congruence - Proof techniques - Proof of Theorem 7.7 - Further exercises.

8. Some proofs about Data Structures 111
Introduction - Registers and memories - Chaining operations - Pushdowns and queues.

9. Translation into CCS 126
Discussion - The language P - Sorts and auxiliary definitions - Translation of P - Adding procedures to P - Protection of resources.

10. Determinacy and Confluence 138
Discussion - Strong confluence - Composite guards and the use of confluence - Strong determinacy: Confluent determinate CCS - Proof in DCCS: the scheduler again - Observation confluence and determinacy.

11. Conclusion 158
What has been achieved? - Is CCS a programming language? - The question of fairness - The notion of behaviour - Directions for further work.

Appendix: Properties of congruence and equivalence. 166

References 169

CHAPTER 0

Introduction

0.1 Purpose

These notes present a calculus of concurrent systems. The presentation is partly informal, and aimed at practice; we unfold the calculus through the medium of examples each of which illustrates first its expressive power, and second the techniques which it offers for verifying properties of a system.

A useful calculus, of computing systems as of anything else, must have a high level of articulacy in a full sense of the word implying not only richness in expression but also flexibility in manipulation. It should be possible to describe existing systems, to specify and program new systems, and to argue mathematically about them, all without leaving the notational framework of the calculus.

These are demanding criteria, and it may be impossible to meet them even for the full range of concurrent systems which are the proper concern of a computer scientist, let alone for systems in general. But the attempt must be made. We believe that our calculus succeeds at least to this extent: the same notations are used both in defining and in reasoning about systems, and - as our examples will show - it appears to be applicable not only to programs (e.g. operating systems or parts of them) but also to data structures and, at a certain level of abstraction, to hardware systems. For the latter however, we do not claim to reach the detailed level at which the correct functioning of a system depends on timing considerations.

Apart from articulacy, we aim at an underlying theory whose basis is a small well-knit collection of ideas and which justifies the manipulations of the calculus. This is as important as generality - perhaps even more important. Any theory will be superseded sooner or later; during its life, understanding it and assessing it are only possible and worthwhile if it is seen as a logical growth from rather few basic assumptions and concepts. We take this further in the next section, where we introduce our chosen conceptual basis.

One purpose of these notes is to provide material for a graduate course. With this in mind (indeed, the notes grew as a graduate course at Aarhus University in Autumn 1979) we have tried to find a good expository sequence,

and have omitted some parts of the theory - which will appear in technical publications - in favour of case studies. There are plenty of exercises, and anyone who bases a course on the notes should be able to think of others; one pleasant feature of concurrent systems is the wealth and variety of small but non-trivial examples! We could have included many more examples in the text, and thereby given greater evidence for the fairly wide applicability of the calculus; but, since our main aim is to present it as a calculus, it seemed a good rule that every example program or system should be subjected to some proof or to some manipulation.

0.2 Character

Our calculus if founded on two central ideas. The first is observation; we aim to describe a concurrent system fully enough to determine exactly what behaviour will be seen or experienced by an external observer. Thus the approach is thoroughly extensional; two systems are indistinguishable if we cannot tell them apart without pulling them apart. We therefore give a formal definition of observation equivalence (in Chapter 7) and investigate its properties.

This by no means prevents us from studying the structure of systems. Every interesting concurrent system is built from independent agents which communicate, and synchronized communication is our second central idea. We regard a communication between two component agents as an indivisible action of the composite system, and the heart of our algebra of systems is concurrent composition, a binary operation which composes two independent agents, allowing them to communicate. It is as central for us as sequential composition is for sequential programming, and indeed subsumes the latter as a special case. Since for us a program or system description is just a term of the calculus, the structure of the program or system (its intension) is reflected in the structure of the term. Our manipulations often consist of transforming a term, yielding a term with different intension but identical behaviour (extension). Such transformations are familiar in sequential programming, where the extension may just be a mathematical function (the "input/output behaviour"); for concurrent systems however, it seems clear that functions are inadequate as extensions.

These two central ideas are really one. For we suppose that the only way to observe a system is to communicate with it, which makes the observer

and system together a larger system. The other side of this coin is that to place two components in communication (i.e. to compose them) is just to let them observe each other. If observing and communicating are the same, it follows that one cannot observe a system without its participation. The analogy with quantum physics may or may not be superficial, but the approach is unifying and appears natural.

We call the calculus CCS (Calculus of Communicating Systems). The terms of CCS stand for behaviours (extensions) of systems and are subject to equational laws. This gives us an algebra, and we are in agreement with van Emde Boas and Janssen [EBJ] who argue that Frege's principle of compositionality of meaning requires an algebraic framework. But CCS is somewhat more than algebra; for example, derivatives and derivations of terms play an important part in describing the dynamics of behaviours.

The variety of systems which can be expressed and discussed in CCS is illustrated by the examples in the text: an agent for scheduling task performance by several other agents (Chapter 3); 'data flow' computations and a concurrent numerical algorithm (Chapter 4); memory devices and data structures (Chapter 8); semantic description of a parallel programming language (Chapter 9). In addition, G. Milne [Mln 3] modelled and verified a peripheral hardware device - a cardreader - using an earlier version of the present ideas.

After these remarks, the character of the calculus is best discovered by a quick look through Chapters 1-4, ignoring technical details. §0.5 (Outline) may also help, but the next two sections are not essential for a quick appraisal.

0.3 Related Work

At present, the most fully developed theory of concurrency is that of Petri and his colleagues. (See for example C.A. Petri, "Introduction to General Net Theory" [Pet], and H.J. Genrich, K. Lautenbach, P.S. Thiagarajan, "An Overview of Net Theory" [GLT].) It is important to contrast our calculus with Net Theory, in terms of underlying concepts.

For Net Theory, a (perhaps the) basic notion is the concurrency relation over the places (conditions) and transitions (events) of a system; if two events (say) are in this relation, it indicates that

they are causally independent and may occur in either order or simultaneously. This relation is conspicuously absent in our theory, at least as a basic notion. When we compose two agents it is the synchronization of their mutual communications which determines the composite; we treat their independent actions as occurring in arbitrary order but not simultaneously. The reason is that we assume of our external observer that he can make only one observation at a time; this implies that he is blind to the possibility that the system can support two observations simultaneously, so this possibility is irrelevant to the extension of the system in our sense. This assumption is certainly open to (extensive!) debate, but gives our calculus a simplicity which would be absent otherwise. To answer the natural objection that it is unwieldy to consider all possible sequences (interleavings) of a set of causally independent events, we refer the reader to our case studies, for example in Chapters 3 and 8, to satisfy himself that our methods can avoid this unwieldiness almost completely.

On the other hand, Net Theory provides many strong analytic techniques; we must justify the proposal of another theory. The emphasis in our calculus is upon synthesis and upon extension; algebra appears to be a natural tool for expressing how systems are built, and in showing that a system meets its specification we are demanding properties of its extension. The activity of programming - more generally, of system synthesis - falls naturally into CCS, and we believe our approach to be more articulate in this respect than Net Theory, at least on present evidence. It remains for us to develop analytic techniques to match those of Net Theory, whose achievements will be a valuable guide.

As a bridge between Net Theory and programming languages for concurrency, we should mention the early work of Karp and Miller [KM] on parallel program schemata. This work bears a relation to Net Theory in yielding an analysis of properties of concurrent systems, such as deadlock and liveness; it also comes closer to programming (in the conventional sense), being a generalisation of the familiar notion of a sequential flow chart.

In recent proposals for concurrent programming languages there is a trend towards direct communication between components or modules, and away from communication through shared registers or variables. Examples are:

N. Wirth "MODULA: A language for modular multiprogramming", [Wir]; P. Brinch Hansen "Distributed Processes; a concurrent programming concept", [Bri 2]; C.A.R. Hoare "Communicating Sequential Processes", [Hoa 3]. Hoare's "monitors" [Hoa 2] gave a discipline for the administration of shared resources in concurrent programming. These papers have helped towards understanding the art of concurrent programming. Our calculus differs from all of them in two ways: first, it is not in the accepted sense an imperative language - there are no commands, only expressions; second, it has evolved as part of a mathematical study. In the author's view it is hard to do mathematics with imperative languages, though one may add mathematics (or logic) to them to get a proof methodology, as in the well-known "assertion" method or Hoare's axiomatic method.

One of the main encumbrances to proof in imperative languages is the presence of a more-or-less global memory (the assignable variables). This was recognized by Hoare in "Communicating Sequential Processes"; although CSP is imperative Hoare avoids one aspect of global memory which makes concurrent programs hard to analyse, by forbidding any member of a set of concurrent programs to alter the value of a variable mentioned by another member. This significant step brings CSP quite close to our calculus, the more so because the treatment of communication is similar and expressed in similar notation. Indeed, algorithms can often be translated easily from one to the other, and it is reasonable to hope that a semantics and proof theory for CSP can be developed from CCS. Hoare, in his paper and more recently, gives encouraging evidence for the expressiveness of CSP.

We now turn to two models based on non-synchronized communication. One, with strong expressive power, is Hewitt's Actor Systems; a recent reference is [HAL]. Here the communication discipline is that each message sent by an actor will, after finite time, arrive at its destination actor ; no structure over waiting messages (e.g. ordering by send-time) is imposed. This, together with the dynamic creation of actors, yields an interesting programming method. However, it seems to the author that the fluidity of structure in the model, and the need to handle the collection of waiting messages, poses difficulties for a tractable extensional theory.

Another non-synchronized model, deliberately less expressive, was first studied by Kahn and reported by him and MacQueen [KMQ]. Here the intercommunication of agents is via unbounded buffers and queues, the

whole being determinate. Problems have arisen in extending it to non-determinate systems, but many non-trivial algorithms find their best expression in this medium, and it is an example of applicative (i.e. non-imperative) programming which yields to extensional treatment by the semantic techniques of Scott. Moreover, Wadge [Wad] has recently shown how simple calculations can demonstrate the liveness of such systems.

A lucid comparative account of three approaches - Hewitt, Kahn/ MacQueen and Milner - is given in [MQ].

In Chapter 9 of these notes we show how one type of concurrent language - where communication is via shared variables - may be derived from or expressed in terms of CCS. This provides some evidence that our calculus is rich in expression, but we certainly do not claim to be able to derive every language for concurrency.

A rather different style of presenting a concurrent system is exemplified by the path expressions of Campbell and Habermann [CaH]. Here the active parts of the system are defined separately from the constraints (e.g. the path expressions) which dictate how they must synchronize. More recent work by Lauer, Shields and others - mainly at Newcastle - shows that this model indeed yields to mathematical analysis. A very different example of this separation is the elegant work of Maggiolo-Schettini et al [MWW]; here the constraints are presented negatively, by stating what conjunctions of states (of separate component agents) may not occur. This approach has an advantage for systems whose components are largely independent (the authors call it "loose coupling"), since then only few constraints need to be expressed.

This section has shown the surprising variety of possible treatments of concurrent systems. It is nothing like a comprehensive survey, and the author is aware that important work has not been mentioned, but it will serve to provide some perspective on the work presented here.

0.4 Evolution

This work evolved from an attempt to treat communication mathematically. In Milner : "Processes: a mathematical model of computing agents" [Mil 1] a model of interacting agents was constructed, using Scott's

theory of domains. This was refined and grew more algebraic in G. Milne and Milner: "Concurrent Processes and their syntax" [MM]. At this point we proposed no programming language, but were able to prove properties of defined concurrent behaviours. For example, Milne in his Ph.D. Thesis "A mathematical model of concurrent computation" [Mln] proved partial correctness of a piece of hardware, a card-reader, built from four separate components as detailed in its hardware description. Our model at this stage drew upon Plotkin's and Smyth's Powerdomain constructions, [Plo 1, Smy]. which extended Scott's theory to admit non-determinism. Part of our algebra is studied in depth in [Mil 2].

At this point there were two crucial developments. First - as we had hoped - our behaviour definitions looked considerably like programs, and the resemblance was increased by merely improving notation. The result, essentially the language of CCS, is reported in [Mil 3] and was partly prompted by discussions with Hoare and Scott. (For completeness, two other papers [Mil 4,5] by the author are included in the reference list. Each gives a slightly different perspective from [Mil 3], and different examples.) The second development was to realise that the resulting language has many interpretations; and that the Powerdomain model, and variants of it, may not be the correct ones. A criterion was needed, to reject the wrong interpretations. For this purpose, we turned to observation equivalence; two behaviour expressions should have the same interpretation in the model iff in all contexts they are indistinguishable by observation.

It now turns out that a definition of observation equivalence (for which admittedly there are a few alternatives) determines a model, up to isomorphism, and moreover yields algebraic laws which are of practical use in arguing about behaviours. We have strong hope for a set of laws which are in some sense complete; in fact the laws given in Chapters 5 and 7 have been shown complete for a simplified class of finite (terminating) behaviours. In this case, "complete" means that if two behaviour expressions are observation-equivalent in all contexts then they may be proved equal by the laws; this completeness is shown in [HM].

0.5 Outline

In Chapter 1 we discuss informally the idea of experimenting on, or observing, a non-deterministic agent; this leads to the notion of

synchronisation tree (ST) as the behaviour of an agent. Chapter 2 discusses mutual experiment, or communication, between agents, and develops an algebra of STs. In Chapter 3 we do a small case-study (a scheduling system) and prove something about it, anticipating the formal definition of observation equivalence and its properties to be dealt with fully in Chapter 7.

Chapter 4 enriches our communications - up to now they have been just synchronizations - to allow the passing of values from one agent to another, and illustrates the greater expressive power in two more examples; one is akin to Data Flow, and the other is a concurrent algorithm for finding a zero of a continuous function. The notion of derivative of a behaviour is introduced, and used in the second example.

In Chapter 5 we define CCS formally, giving its dynamics in terms of derivations (derivative sequences). This yields our strong congruence relation, under which two programs are congruent iff they have essentially the same derivations, and we establish several laws obeyed by the congruence. In Chapter 6 we present communication trees (CTs, a generalisation of STs) as a model which obeys these laws; this model is not necessary for the further development, but meant as an aid to understanding.

Chapter 7 is the core of the theory; observation equivalence is treated in depth, and from it we gain our main congruence relation, observation congruence, under which two programs are congruent iff they cannot be distinguished by observation in any context. Having derived some properties of the congruence, we use them in Chapter 8 to prove the correct behaviour of two further systems, both to do with data structures.

In Chapters 9 and 10 we look at some derived Algebras. One takes the form of an imperative concurrent programming language, with assignment statements, "cobegin-coend" statements, and procedures. In effect, we show how to translate this language directly into CCS. The other is a restriction of CCS in which determinacy is guaranteed, and we indicate how proofs about such programs can be simpler than in the general case.

Finally, in Chapter 11 we try to evaluate what has been achieved, and indicate directions for future research.

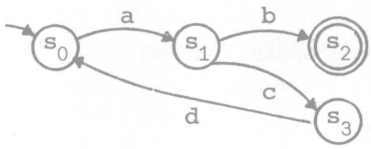
CHAPTER 1

Experimenting on nondeterministic machines

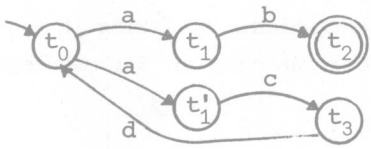
1.1 Traditional equivalence of finite state acceptors

Take a pair S,T of nondeterministic acceptors over the alphabet $\Sigma = \{a,b,c,d\}$:

S



T



The accepting states of S and T are s_2 and t_2 respectively; in <u>nondeterministic</u> acceptors we can always make do, as here, with a single 'dead' accepting state.

A standard argument that S and T are equivalent, meaning that they accept the same language (set of strings), runs as follows. Taking s_i (resp t_i) to represent the language accepted starting from state s_i (resp t_i), we get a set of equations for S , and for T :

$$s_0 = as_1 \qquad t_0 = at_1 + at'_1$$
$$s_1 = bs_2 + cs_3 \qquad t_1 = bt_2$$
$$s_2 = \varepsilon \qquad t'_1 = ct_3$$
$$s_3 = ds_0 \qquad t_2 = \varepsilon$$
$$t_3 = dt_0$$

Here as usual + stands for union of languages, ε for the language $\{\varepsilon\}$ containing only the empty string, and we can think of the symbol a standing for a function over languages: $as = a(s) = \{a\sigma;\ \sigma \in s\}$.

Now by simple <u>substitution</u> we deduce

$$s_0 = a(b\varepsilon + cds_0) .$$

By applying the <u>distributive law</u> $a(s + s') = as + as'$ we deduce

$$s_0 = ab\varepsilon + acds_0 ,$$

and we can go further, using a standard rule for solving such equations known as Arden's rule, to get

$$s_0 = (acd)^*ab\varepsilon \ .$$

For T it is even simpler; we get directly (without using distributivity)

$$t_0 = ab\varepsilon + acdt_0$$

and the unique solvability of such equations tells us that $s_0 = t_0$, so S and T are equivalent acceptors.

But are they equivalent, in all useful senses?

1.2 Experimenting upon acceptors

Think differently about an acceptor over $\{a,b,c,d\}$. It is a black box, whose behaviour you want to investigate by asking it to accept symbols one at a time. So each box has four buttons, one for each symbol:

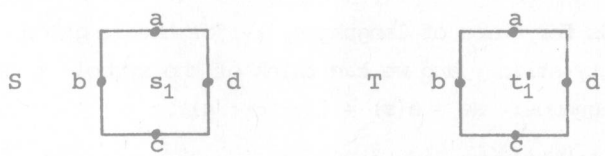
There are four atomic experiments you can do, one for each symbol. Doing an a-experiment on S (secretly in state s_0, but you don't know that) consists in trying to press the a-button, with two possible outcomes in general:

(i) Failure - the button is locked;

(ii) Success - the button is unlocked, and goes down (and secretly a state transition occurs).

In fact we cannot distinguish between S and T, in their initial states, by any single atomic experiment; the a-experiment succeeds in each case, and the other three fail.

After a successful a-experiment on each machine, which may yield



we may try another atomic experiment, in our aim to see if the machines are equivalent or not. Clearly a b-experiment now succeeds for S and fails

for T , though the other three experiments fail to distinguish them. After trying the b-experiment, then, can we conclude that S and T are not equivalent?

No, because S's response to the a-experiment could have been different (for all we know) and locked the b-button, while T's response could have been different (for all we know - and it could indeed!) and unlocked the b-button. Following this argument further, we may feel forced to admit that no finite amount of experiment could prove to us that S and T are, or are not, equivalent!

But suppose

(i) It is the weather at any moment which determines the choice of transition (in case of ambiguity, e.g. T at t_0 under an a-experiment) ;

(ii) The weather has only finitely many states - at least as far as choice-resolution is concerned ;

(iii) We can control the weather .

For some machines these assumptions are not so outrageous; for example, one of two pulses may always arrive first within a certain temperature range, and outside this range the other may always arrive first. (At the boundary of the range we have the well-known glitch problem, which we shall ignore here.)

Now, by conducting an a-experiment on S and T under all weather conditions (always in their start states, which we have to assume are recoverable), we can find that S's b-button is always unlocked, but that T's b-button is sometimes locked, and we can conclude that the machines are not equivalent.

Is this sense of equivalence, in which S and T are not equivalent, a meaningful one? We shall find that we can make it precise and shall adopt it - partly because it yields a nice theory, partly because it is a finer (smaller) equivalence relation than the standard one (which we can always recover by introducing the distributive law used in §1.1), but more for the following reason. Imagine that the b-buttons on S and T are hidden. Then in all weathers every successful experiment upon S unlocks some visible button:

S (with b hidden) is not deadlockable ,