AXIOMATISING THE LOGIC OF DISTRIBUTED COMPUTATION

by

Steven G. Cumming

BSc. University of Alberta, 1981

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
August 1986

Approval

Name: Steven G. Cumming

Degree: Master of Science

Title: Axiomatising the Logic of Distributed Computation

Jim Delgrande
Chairman

Tiko Kameda
Senior Supervisor

Hassan Reghbati
Senior Supervisor

Robert Hadley

Raymond Jennings

Steve Thomason
External Examiner

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Axiomatising The Logic of Distributed Computation

Author:

(signature)

S. CUMMING

(name)

Aug 5 1986

(date)

Abstract

Distributed Algorithms may be represented by a finite network of communicating processes. Each process is represented by a set of 'guarded commands' which behaves like a generalised iterator. The distributed execution consists in the non-deterministic interaction of processes. This interaction occurs exclusively by means of message interchange over unreliable communications channels.

A suitable logic for distributed algorithms must at least be capable of describing some aspects of the dynamic behaviour of processes.

We develop a modal logic for processes, in which safety, liveness and fairness properties, as well as partial and total correctness properties, may be derived.

The logic is interpreted in a Kripke-style relational semantics. No higher-order semantical objects such as execution paths or trajectory sets are introduced.

In a concluding section, we indicate how the results may be extended to account for message interchange.

Acknowledgements

A completed Master's thesis is the (anti)-climax of several years activity. It is therefore difficult to single out those most deserving of acknowledgement and thanks. Yet, there are several persons who must be mentioned here.

Foremost, my parents, for making everything possible.

My senior supervisors, Professor Reghbati and Dr. Kameda, have provided long term support. It was Dr. Kameda who incited the research programme culminating in this thesis.

Dr. Jennings, who undertook my education in modal logic, has been, in the course of frequent consultation, the source of much of the substantive content of this thesis.

Dr. Hadley has carefully reviewed much of my writing, and clarified my thinking on several points.

Dr. Delgrande likewise reviewed an early version of this work.

Finally, I thank my external examiner Dr. Thomason, for discussion and encouragement.

This document was prepared using the facilities of the Laboratory for Computer and Communications Research. I wish to thank Mr. Ed Bryant and Ms. Carol Murchison for their considerable assistance.

Partial operating support for this research was provided by NSERC Grants No. A0743 and 5240.

Especially because of the efforts of those who contributed to its development, the author can not escape sole responsibility for the contents of this thesis.

Table of Contents

Preface

This conversation appears in full in [4].

Chang I guess one of the saddest things in logic is that Presburger never got his degree.

Nerode That Presburger never got his degree?

Mostowski Yes, yes. Tarski refused to give him a degree for his paper [36].

Nerode Which is now one of the most cited.

Kleene Why?

Mostowski Because he considered it too simple. He thought that it was not what he wanted.

Sacks What was this? The decidability of ...?

Nerode Addition and Multiplication - separately.

Mostowski No, without multiplication.

Kleene How you can do it with multiplication without addition I do not know, because the usual recursion equations for multiplication have addition in them.

Nerode Sure, but I thought that Presburger had also done the theory of multiplication alone. That was my memory of it. Were you the first to do that?

Mostowski I think Skolem was the first to do that.

Nerode Ah, Skolem!

Mostowski Yes, but I think Skolem was later than Presburger. I am not sure whether it was so. But in Presburger's paper there is only addition.

Sacks So Tarski thought the proof was too simple.

Mostowski Yes. This was a very obvious application of elimination of quantifiers. You know at that time ...

Kleene Presburger - nine pages in 1930.

Chang A nine page thesis!

Mostowski Well, he could have expanded it a bit.

Chapter 1

Introduction

This thesis is about the modal logic of distributed computation. Its principle contributions are extensions of existing work in the logics of programmes. We ought, therefore, to begin by saying what a "logic of programmes" is.

Programming Logics

A logical system for reasoning about the behaviour of computer programmes should be constructed from the following components.

1. A *semantics* i.e. a mathematical abstraction of computer programmes,
2. A *language* in which statements about the behaviour of programmes may be expressed,
3. A notion of *satisfaction*, so that sentences in the language may be interpreted in the semantic idiom and thus said to be true or false,
4. A *deduction system* within which particular sentences in the language may be proved or disproved.

With these pre-formal notions in hand we may sketch the development of programming logics.

One of the first instances of a programming logic is due to Hoare [18]. Hoare's system and its many extensions are characterised by syntactical constructions of the form

$$p\{\alpha\}q\,,$$

which is interpreted

> if p is true when programme α starts, then q will be true if and when α finishes.

Properties of this form are called *partial correctness assertions*, as opposed to *total* correctness assertions, which add that α does in fact finish, or *terminate*. We refer to logics of this ilk as *input/output* or I/O logics. Formulae in I/O logics describe programmes in terms of initial conditions or input (p) and final conditions or output (q).

Hoare systems have been developed for fairly rich programming languages. In particular, Lamport [24] presents a Hoare style deductive system for reasoning about concurrent computations, such as are characteristic of operating systems. The semantics of this and similar systems seems always to be presented informally.

Hoare systems are most used to provide a formal means of reasoning about what have been called *safety* properties. A safety property asserts that certain situations never develop over the course of programme execution. Partial correctness is an example; it is asserted that if p is true when α begins, α can not finish with ψ false.

Total correctness, on the other hand, is an example of a *liveness* property. Liveness properties make some positive assertion about programme behaviour; for instance that α actually does terminate. A logical system for total correctness of concurrent or parallel programmes appears in [11, 12]. It is based upon a Hoare style rule system, augmented by Dijkstra's weakest pre-condition predicates [6].

However, to quote Lamport [29]

> ...programmes are capable of many more sins of omission than just failure to terminate. Indeed for many concurrent programmes - operating systems are a prime example - termination is known by the less flattering name of *crashing*, and we want to prove that it does not happen.

Proving liveness properties requires an ability to reason about the ongoing or dynamic behaviour of programmes. This has led to extensive research into the possible applications of linear and branching time temporal logics to the logics of programmes. Typically, the languages employed prefix a temporal modality such as *henceforth always* or *eventually* to a programme location predicate [29, 32, 21, 33]. The latter papers deal with programme verification - proving that the programme does or does not exhibit certain behaviours. Similar temporal logical schemata have been proposed purely as specification languages, to define explicitly the required behaviour of programmes yet to be written. See, for instance [40, 1]. Algorithms for effectively constructing programmes which realise a given temporal logic specification have also been developed [25, 3]. More abstrusely, much attention has been devoted to complexity and decidability issues which arise from the mere use of temporal logics [8, 46, 31].

In parallel with the research cited above, various modal logics of computation have been developed. It was noticed by Pratt [34] that the proposition

$$p \{\alpha\} q ,$$

has a ready interpretation in a Kripkean relational semantics. This led to the development of Dynamic Logic (DL) [15] and its propositional variant (PDL) [9]. The modality employed in these systems is *after*, as in

$$after(\alpha , p)$$

the intended interpreted of which is "after α terminates, p is true". This corresponds to the Hoare rule

$$\mathbf{true} \{\alpha\} q ,$$

where **true** denotes an input condition to α which is always true.

In the semantics, each programme is assigned a binary relation on a set of states. If the pair $<s,t>$ is in the binary relation for programme α, it means that programme α can terminate in state t when started in state s.

More recently, the logician Robert Goldblatt presented a rigorous and detailed exposition of a Dynamic Logic for a programming language roughly equivalent to PASCAL or ALGOL-60 [14]. It is this work which serves as the foundation for the present thesis. By the criteria above, these various dynamic logics qualify as true logics of programmes, in that they provide a semantical underpinning, in the form of relational semantics, which is a reasonable abstraction of programme behaviour. The existence of an underlying semantics also makes it possible to speak of the soundness and completeness of any proposed deduction system.

The limitation of dynamic logics is that they support *only* the modality *after*. In consequence, it suffers from the same limitations as Hoare logic when it comes to expressing what we have termed liveness properties. This limitation has inspired some very powerful generalisations of DL, which are referred to as Process Logics. The underlying relational interpretation of programmes is altered. Instead of binary relations, programmes are assigned sets of paths, which correspond to possible computations. That is, a path for α is an ordered sequence of states with a first element. Depending upon the language introduced, it is possible to define almost any given property of programme behaviour. For instance, total correctness amounts to the requirement that all execution paths be of finite length. More discussion of the various process logics occurs in §5.

The desire for greater expressive power motivated the development of Process Logics. However, there seems to be general agreement that a programming logic supporting a

fairly restricted set of modalities would be adequate in practice for proving the correctness of most computer programmes. Following Pratt [35], these modalities are

1. *after*(α,p)
2. *throughout*(α,p) - p is true at every point in the computation of α,
3. *during*(α,p) - p is true at least once during the execution of α
4. *preserves*(α,p) - if p ever becomes true during α, then it remains true thereafter.

No logic based on relational semantics has ever been presented which is capable of expressing the property of preservation. Pratt [35] presents such a system adequate for the modalities *during* and *throughout*, with the severe restriction that these modalities can not be nested in any way.

The objective of this thesis is to develop a programming logic capable of representing at least the above list of programme behaviours (indeed, a great deal more.) Furthermore, we aim to accomplish this with out leaving the first order framework of Kripkean relational semantics.

In addition, the notions of non-termination and failure prove capable of interpretation within the formal semantics. This appears to be a new result.

Distributed Computation

Distributed computation refers to problem solving which depends upon the cooperation of loosely coupled, but autonomous computers. We do not consider either concurrent processing, which involves time sharing on a single machine, nor the tightly coupled computations of massively parallel architectures.

The reliable exchange of messages between distant computers involves a form of

distributed computation - there is no central god machine ensuring that the message interchange is successful. At the far end of the scale, it is disturbing to realise that military command and control systems depend upon distributed computation of great sophistication and reliability. We describe a model of distributed computation in the next chapter.

The design of reliable distributed computer programmes is a task of great difficulty, yet no programming logic for distributed computation appears ever to have been advanced.

Outline

Chapter 2 presents and justifies the model of distributed computation which we employ.

Chapter 3 serves as an introduction to modal logic.

Chapter 4 extends the I/O logic of [14] to account for a new programming construct. This construct represents local components within a distributed programme.

Chapter 5 contains the main results of this thesis. A logical system adequate to account for the dynamic behaviour of processes is developed, and partially axiomatised. The system is able to reason about divergence and failure.

Chapter 6 concludes the thesis by indicating how the results may be extended to describe actual distributed computation.

Notational Conventions

In proofs and informal reasoning, the meta-logical connectives

p and q

if p then q (p implies q is synonymous)

p iff q, for 'if and only if'

are used freely. They occur either in roman or italic face as necessary. Punctuation

is provided by the informal use of brackets and commas. The meta-logical connectives take precedence as indicated above.

Chapters are referenced *e.g.* § 3.1. Sections and Subsections are referenced *e.g* § 4.1 and § 5.1.4. Theorems, Lemmas and Definitions are referenced *e.g* Thm. (5.2.15). The first two digits denote the Chapter and Section within which the Theorem or Definition is stated.

Chapter 2

A Model of Distributed Computation

2.1. Parallel Event Servers

In this section we introduce the paradigm for distributed computation which has shaped this research.

Distributed computation is characterised by the following properties:

1. The distributed programme is distributed into a fixed number of *processes*.
2. Each process executes on a dedicated *processor*.
3. Processes communicate by means of a fixed set of unreliable *channels* connecting pairs of processors. Channels are the medium by which *messages* are transmitted between processes.
4. Processes have disjoint sets of variables (i.e. there are no shared variables.)
5. Processes execute asynchronously.
6. Execution at both the local and global level is non-deterministic.
7. In general, the programme does not (is not supposed to) terminate.

An *event* is something which occurs intermittently to some process at some processor. The idea is that when an event occurs, the process will serve it, doing whatever is required to be done, and then be ready to correctly serve the next event. Service of an event will often cause another event to occur later, either at the same process, or somewhere else.

We sometimes use the term *Parallel Event Servers* to describe a programme distributed in this way.

The structured way to represent a process is to list the set of detectable events, and associate with each of them the piece of code which serves the particular event. The programme scheme outlined below illustrates the syntax which we use to represent parallel programmes.

```
parbegin
        P1:
          rep
                G11 → S11
                .
                .
                G1n → S1n
          per
        □
        .
        .
        .
        □
        Pn:
          rep
                Gn1 → Sn1
                .
                .
                .
                Gnm → Snm
          per
parend
```

This scheme is a variation of the *guarded commands* introduced by Dijkstra in [6]. The variant, due to Flon and Suzuki [11, 12], represents the following behaviour.

The P_i are process labels. The S_{ij} are sequential, deterministic programmes (referred to throughout as *sequential components*.) The G_{ij} are the *guards*, which enable the

sequential components. A guard is some truth condition on local P_i variables and locally testable events.

In addition, we must introduce two primitive commands which are used to read from and write to channels [18]. Let C be the name of a channel, and let x be a variable. Then the command $C!x$ causes the current value of the variable x to be transmitted over C. The command $C?x$ is somewhat more complex. It acts as a test to determine whether the channel may be read. If so, the variable x is assigned the current value which the channel is transmitting. Thus $C?x$ acts both as a boolean expression and an assignment statement. A possible formal semantics for these commands will be described in §§ 6.1.

A process may begin executing by evaluating its guards. From among those which are true, one is selected and the corresponding sequential component is executed. Upon termination, the guards are re-evaluated, and a new choice is made. If the attempt to evaluate one of the guards fails then the process *fails* or *aborts*.

The local non-determinism consists in the fact that several guards may be true during the interval when evaluation occurs. The global non-determinism arises in consequence both of this fact, because guard selection affects which events may occur later at other processes, and also because the relative speed of different processes and lossyness of different channels is not specifiable (in the proposed language). The lossyness of a communications link is a measure of the probability that a message is transmitted without error.

A process is *blocked* if none of its guards are true at evaluation. If none of the guards may eventually become true, then the process is *deadlocked*. The entire

programme is deadlocked if all guards G_{ij} become permanently false. A process is *live* if one or more guards is, or will become, true.

The existence of non-determinism gives rise to the issue of *fairness*. An execution is said to be fair if it does not discriminate between equally valid choices. The most usual and simplest fairness condition is this: if a guard is infinitely often true, it will be selected infinitely often. This and many other conditions for fair execution are studied in [22].

Actual termination of a **rep - per** block is accommodated in [11] by an explicit exit command. This feature does not appear to be helpful for the sorts of programmes under consideration here.

Following [11], we have required that all sequential components be deterministic. This implies that an S_{ij} may not spawn other parallel programmes. It may however effectively invoke them by sending messages which result in the activation of parallel components on other machines.

Should a particular application require non-deterministic behaviour, the methods of [11, 12] may be employed. The sequential components containing non-deterministic programming constructs may be effectively decomposed into a guarded command schema. To guarantee that the result of the decomposition executes without the interleaved execution of other sequential components, some additional flags must be incorporated into each of the process's guards. These flags would be used to disable the selection of sequential components not involved in the non-deterministic computation.

Due to the nature of the programmes under consideration, a further simplification is possible: the S_{ij} are to be free of procedure and subroutine calls, and function invocations. The technical machinery needed to account for these (useful) features adds considerable complexity to a programming logic. The study of parallel programmes is largely a study of process interaction. What the sequential components actually do is not that interesting. Formal treatment of the semantics of guarded commands may be found in [6, 11, 12, 14, 43]. Discussion of a more general case, in which processes can spawn other processes (accomplished by allowing **parbegin parend** blocks as general statements) may be found in [48].

In the next Section, some examples of distributed algorithms will be presented. These examples are all expressable in the programming schema given in §§ 2.1.

2.2. Programme Examples

We now illustrate the application of the programming constructs introduced in the previous section, moving from a very general specification schema to a specific distributed algorithm. By demonstrating a reasonably broad range of applicability, this section serves as the justification for introducing the parallel event server as a paradigm for distributed computation.

2.2.1. ISO Protocol Specification Language

At the time of writing an extension to the PASCAL language was being developed [19] to facilitate the specification and implementation of communications protocols.

A communications protocol is an inherently distributed algorithm which enables processes on different computers to exchange messages. A minimal requirement of

any protocol is that it overcome the limitations of the physical communications medium, or channel. These limitations include a tendency to lose messages completely, or to corrupt them. The protocol should present to the two parties using it a virtual channel of higher performance than the physical one.

An example of a simple communications protocol is presented in §§ 2.2.3

In the proposed standard, processes are represented by extended finite state machines (FSMs). The extension is to provide local variables and associate with each state transition a piece of sequential deterministic PASCAL code. State transitions are enabled by a combination of

- EVENTSs : input signals from other FSMs.
- PROVIDEDs: boolean predicates on local variables.

Transitions occur between *major states* which are abstractions of the current state of the FSM, excluding the local variables. Transitions may result in message transmission to other FSMs, which may in turn result in a distant input event.

Procedures are available insofar as the state transition code is written in PASCAL. This is more for expressive convenience than necessity.

The ISO standard also provides for DELAY operators. These operators specify the amount of delay permissible between the enabling of a transition and its *firing* or execution. Process delay will not be dealt with here. Some results in the formal treatment of delay may be found in [17, 28].

The standard specifies that non-deterministic choice be made between simultaneously enabled transitions.

The standard allows a single process to realise several instances of a protocol. That is, a single system process could provide protocol connections to many user level processes. This is accomplished by means of context variables which distinguish the various protocol connections being served. Local variables are then arrays indexed by the context variable. Formal treatment of arrays would only complicate the logical system under development - so we ignore it. The concerned reader is referred to [14§5].

The only difficulty in translating algorithms encoded in extended Pascal into the language of the previous section consists in the construction of the guards. Input events are things of a different type than boolean predicates. How to recast them into tests for "the recent occurrence of an input event" is considered in the Conclusion.

2.2.2. Fault Tolerant Networks

In [5] Dijkstra proved the existence of distributed algorithms which can recover from errors. He calls these algorithms 'self-stabilising', because continued execution of process steps is guaranteed to transform the system from a failure or *illegitimate* state to a *legitimate* state. Consideration of the work cited engendered the research culminating in this thesis.

In Dijkstra's model, processes are finite state machines, and the distributed programme is a sparsely connected network of processes. Each process has a local state variable, and has instantaneous access to the states of adjacent processes. Processes execute by evaluating guards just as above. The only difference is that the selection of the next process step is not made locally, but by a central daemon. In consequence, only one state transition occurs at a time. This means that the system executes synchronously.

The global state is the cross product over all processes of the set of guards. Some subset of this is said to be legitimate. In the cases considered in [5] the legitimate states are just those in which exactly one guard is somewhere true.

The global criteria for self stabilisation are

1. in each legitimate state at least one guard is true,
2. no transition moves from a legitimate to an illegitimate state,
3. each guard is true in some legitimate state,
4. there is a path (sequence of transitions) between any two legitimate states.

These conditions roughly correspond to deadlock freedom (1), a global safety condition (2), a liveness property for each guard (3) and a global liveness property (4) which, assuming a fair daemon, would result in each process step being repeatedly executed.

A system is self-stabilising iff

- at least one guard is always true
- an illegitimate state will always be transformed into a legitimate state in a finite number of transitions.

Here is a self-stabilising algorithm for N machines with $K > N$ distinct states. The machines are connected in a cycle and numbered from 0 to N-1. So, the neighbours of machine n are n-1 and n+1, mod N. Note that process P_0 changes states by performing addition mod K on its state variable S. For process i, S_{i-1} denotes the current value of i's left neighbour. The reader should verify that this structure satisfies the required conditions.

```
parbegin
  P0:
        rep  if S(N-1) = S0 then
                          S0 := (S+1)mod K fi per
  P1:
        rep  if S0 ≠ S1 then S1 := S0 fi per

  .
  .
  .
  Pi:
        rep  if S(i-1) ≠ Si then Si := S(i-1) fi per
  .
  .
  .
  PN-1:
        rep if  S(N-2) ≠ S(N-1) then S(N-2) := S(N-2) fi per
parend
```

The pseudo-code above may be translated into the language of guarded commands. Where, above, neighbours states were treated as variables, we now write R (L) to indicate the channel connecting process S_i to its right (left) neighbour.

```
parbegin
  .
  .
  Pi:
        rep
          L? → temp := L;
          temp ≠ Si  → Si := temp;
          true → R! Si;
        per
parend

where Si holds the value of the current state of Process i,
      temp is a local variable of type state, and
the guard L? is a test for the availability
    of neighbour (i-1)'s state information.
the command  R! Si transmits the value of
    Si on channel R.
```

Notice that the standard communication primitive has been broken down into two components. In the first guarded command, we see the guard *L?*, which is true

exactly when a message may be read from the channel connecting S_i and S_{i-1}. Within the component itself, we see the state variable S_{i-1} on the left hand side of an assignment operator. This assignment should be interpreted as a *read* command. We will argue in §§ 6.1 that this is a better representation of channel communications. We will also present a clearer syntax.

The behaviour of the algorithm is now slightly different. Now, P_i must explicitly receive the state value of its left neighbour, and send an updated version of its state to its right neighbour. By virtue of the third guarded command, it does so frequently.

This recasting raises some interesting questions. Until P_i actually receives a message from its neighbour(s) what is the value of S_{i-1} or temp? Without an initialisation phase, the system might well be inconsistent. But since all transitions are supposed to preserve system legitimacy, the occurrence of a fault in a self-stabilising system corresponds to the system starting in an illegitimate state. The usual definitions of parallel algorithms always include a specification of the initial conditions. To the extent that these conditions affect the truth value of guards, deviation from them could result in the system starting in an illegitimate state. Dijkstra has established that ultimately correct behaviour can be independent of the initial state of a distributed system.

2.2.3. A Version of the Alternating Bit Protocol

The alternating bit protocol (ABP) is a simple though non-trivial communications protocol. As a final programming example we illustrate a simplified instance of this protocol and give an explanation of its behaviour.

The protocol assumes that a channel may garble or lose but not reorder messages.

That is, a message may be physically distorted, or lost entirely, but messages are received, if at all, in the same order as they were transmitted.

The following programme is a fragment of the full alternating bit protocol as specified in [19]. The representation is largely that of Kroeger in [21] although state variables are used as in the ISO standard.

Channel names are IN,OUT, and D. When referenced as "D?" in a guard, a test is performed to see if the channel holds a value. On the right hand side of an assignment they function as variables. D!(*exp*) sends the value of the expression *exp* onto the channel.

Major Programme states are READY, SEND, and ACK_WAIT. When they appear in guards they function as a test on the current global state. In statements they are an assignment to the state variable.

The Sender's channel IN is connected to some other process which at intervals wants to send messages across the faulty channel D. We can regard IN as a queue. Sender **S** takes items from IN one at a time, associates a sequence number with them, and sends a composite message to receiver **R**. When **S** eventually receives an ACK from **R** containing the same sequence number as that of the last message sent, it assumes that the message has reached **R** intact. It is then ready to send the next message, if any.

Now to define the programme variables:

- d_r,d_s: current message for the receiver (sender)
- *ls*: sequence number of last message sent

- nr: expected sequence number of next message
- a: sequence number acknowledged

```
initial conditions: nr = 1, ls = a = 0;
parbegin
S:
      rep
          READY and IN? →
                d_r := IN;
                ls := ls ⊕ 1;
                SEND;

          SEND →
                D!(ls,d_r);
                ACK_WAIT;

          ACK_WAIT and D? →
                a := D;
                if ls = a then READY else SEND fi
      per
R:
      rep
          D? →
                (mn,d_s) := D;
                if mn = nr then OUT!(d_s);
                                nr := nr ⊕ 1;
                fi

          true →
                D!(nr ⊕ 1);
      per
parend
```

R continuously sends acknowledge messages, but will not change the sequence number it sends until it gets a message from **S** with the sequence number it is expecting. This forces **S** to keep sending duplicate copies of a message until **R** gets one intact. The reader may care to verify that the correct pattern of sequence numbers will be followed no matter in which state process **S** begins.

This simple example lacks some obvious refinements. There is no error checking

done on incoming messages by either **R** or **S**, other than to verify the sequence number. In practice the total message traffic would be reduced by the addition of a Timer process. The Timer could interrupt **R** at intervals, causing it to send an ack if appropriate at the time (e.g. if a message had arrived) and not 'as fast as it can'. Equally, **S** could interact with a timer process, so as not to respond to *every* false ack with a new copy of the current message. However, the example given here is complex enough.

This protocol is often employed as a vehicle for demonstrating logical specification languages; see, for instance [48, 40, 21].

In this section, a fairly broad range of distributed algorithms has been presented. All of them can be written in an algorithmic language using the **rep per** construct provided a suitable set of communications primitives is given. This suggests that Parallel Event Servers are suitable to represent distributed algorithms.

In the case of self-stabilising computations, several examples are given in [5] but no techniques are provided for either proving that the property holds in a given case, or for deriving algorithms which exhibit it. In [19] only the syntax of a formal specification language is given. In itself, this does not assist anyone in determining if the specifications are correct. Nor does syntactic correctness guarantee the correctness of the algorithm.

The correctness of both types of algorithm has to do with their behaviour over time, not only with the way that particular events are dealt with. A logic for partial correctness, as described in the Introduction appears inadaquate for the task.

We conclude that a programming logic permitting the derivation of correctness properties directly from the syntactic representation of programmes written in the language under consideration would be useful in many applications. The point of the preceding two sections is that such a logic would qualify as a logic of distributed computation.

Chapter 3

Introduction to Modal Logic

The bulk of this chapter is devoted to a tutorial development of the well known modal systems K and K4. We employ a simple propositional language L_K and the usual Kripkean relational semantics. Proof of completeness is by means of the now standard Henkin technique. Readers familiar with these topics may skip the first section without loss. A system similar to K resurfaces in § 5.

In the the concluding sections, we illustrate some of the many extensions of K which have been advanced as suitable programming logics. These extensions are obtained by adding to L_K various powerful modal operators, which we introduce and define. Finally, we show how a simple system, such as K, is capable of representing those dynamic properties of programmes which were introduced in the preceding chapter.

3.1. The Modal Systems K and K4

In this section we develop a simple modal logic, for the purpose of illustrating the fairly abstract approach taken throughout this thesis. We begin with an intuitive discussion of some fundamental notions.

A *language* is just a set of symbols which may be strung together according to specified *formation rules* to make *well formed formulae*, or *wffs*. Some of these will be true in a particular *interpretation* or *semantics*, others false. For instance, the English

wff "If she weighs the same as duck, she's made of wood" is probably false. By *semantics* we mean a characteristic class of *models* and a collection of rules for interpreting formulae in models. A *model* is a mathematical structure which is an abstraction of the concept or process about which we wish to reason. When a formula is said to be *true*, it is always with respect to a particular model in a given semantics. A formula is said to be *valid* if it it is true in every model allowable in the semantic idiom.. Of course it is often controversial to claim that "truth about a model" implies truth about anything else.

An important task for the logician, which is sometimes even possible, is to find a distinguished subset of the *wffs* of a language, called *axioms*, and a set of closure conditions on sets of *wffs*, called *inference rules* such that the closure of the set of axioms under the rules of inference is *sound* and *complete* in a given semantics. Such a set of formulae is called a *logic*. A member of the set is a *theorem* of the logic. A logic is sound is all of it's theorems are valid. A logic is complete if every valid formula is a theorem. The process of finding such a list of axioms and inference rules is called *axiomatisation*. A famous and unsuccessful example of this venture is Peano's axiomatisation of a first order logic of arithmetic..

We now proceed with this task for a simple propositional modal logic. The development below is essentially a connected series of excerpts from [20] and [14]. For the sake of legibility we forego item by item references. The notation employed is somewhat nonstandard; it is used in the interest of consistency with later chapters.

We begin by defining a language L_K. Then a set of *wffs* is defined, using the primitive symbols of L_K and various syntactic *formation rules*. We then provide a semantics by defining a model (or rather, class of models) and rules for interpreting

formulae of L_K. The next step is to present a well known axiomatisation, namely that of [2], and show it to be sound. Finally, a Henkin style completeness proof is illustrated.

Let $L_K = \{\rightarrow, \Box, \mathbf{false}, P\}$,

where $P = p,q,r \ldots$ is a countably infinite set of elementary propositions (called the *non-logical symbols* of L_K.)

Call the set of well-formed formulae of L_K *Fma*, and define it thus:

- $\mathbf{false} \in Fma$
- if $p \in P$ then $p \in Fma$
- if $\phi, \psi \in Fma$ then $\phi \rightarrow \psi \in Fma$
- if $\phi \in Fma$ then $\Box\phi \in Fma$
- that's all!

The other logical connectives can be defined using $\rightarrow$. In particular $\neg\phi =_{df} (\phi \rightarrow \mathbf{false})$ and $\mathbf{true} =_{df} \neg\mathbf{false}$. The modal operator $\Box$ has a dual, which we define

$$\Diamond\phi =_{df} \neg\Box\neg\phi$$

In passing, we note that *Fma* is relative to a particular language, and to be explicit, should be written Fma_{L_K}. To 'avoid avoidable subscripts' we will ignore this nicety when no ambiguity results. In general more than one class of wff's may be defined for any language. Throughout this thesis however, every language will have exactly one set *Fma* associated with it. This enables us to utter without ambiguity phrases such as "ϕ is an L_K-formula."

The only major difference between L_K and the language of the propositional calculus (PC) is the presence of the symbol $\Box$. Since *Fma* is merely a set of uninterpreted strings of symbols generated by the 'grammar' above, $\Box$ should be no more (or less) mysterious than, for instance, 'p'.

To motivate the construction of the class of models below we briefly discuss some possible intended interpretations of $\Box$. One example is a preference relation over a collection of conceivable states of affairs, or "possible worlds". One thing about preference relations is that, like 'later than' or 'greater than', they are transitive. In other words, if the intended interpretation of $\Box\phi$ is

any state of affairs preferable to the actual is such that ϕ

we might expect that any yet more desirable state of affairs also satisfies ϕ. Equally, if a 'state' is a moment in time and $\Box\phi$ is intended to mean "ϕ tomorrow and thereafter", we would expect that also "tomorrow $\Box\phi$" be true. Thus $\Box\phi$ means (in this interpretation) "henceforth always ϕ" while the dual operator $\Diamond$ means "eventually" i.e. "at *some* later time." "Preference" and "Henceforth" are examples of *modalities*. A symbol such as $\Box$ used to represent them is a *modal operator*. For our purposes, the intended interpretation of $\Box$ is the *temporal* modality "henceforth always". With this interpretation in mind, we can proceed to define a semantics for L_K.

A *model* M is a structure (S,v,R), where S is a set of states, R is a binary relation $R \subseteq S \times S$ and v is an operator which associates with every $s \in S$ a valuation function $v_s : P \rightarrow \{0,1\}$, where 0 and 1 are the synonyms for **false** and **true**. This is sometimes expressed as $v : S \rightarrow 2^P$. The meaning is that v assigns to each state the set of propositions which hold in that state. Members of S are denoted by the lower case italic letters s,t,u,v,w, etc.

Interpretation of members of *Fma* is accomplished by defining the *satisfaction relation* $\models$. We write $M \models_s \phi$ to say that ϕ is satisfied at s by M. Synonyms for satisfaction are *true* or *holds*. Validity in a model M is denoted $M \models \phi$. Formulae which are valid *per-se*, such as propositional tautologies, may be written simply $\models \phi$, indicating that ϕ is true at all states in all models. The relation $\models$ is defined inductively for all members of *Fma*.

1. for $p \in P$ $M \models_s p$ iff $v_s(p) = 1$
2. $M \models_s (\phi \rightarrow \psi)$ iff $M \models_s \phi$ implies $M \models_s \psi$
3. $M \models_s \neg\phi$ iff not $M \models_s \phi$
4. $M \models_s \Box\phi$ iff $\forall t\,(sRt$ implies $M \models_t \phi)$

We write $M \not\models_s \phi$ for not $M \models_s \phi$.

It is easy to see that v_s could be extended to all formulae ϕ, so that $M \models_s \phi$ iff $v_s(\phi) = 1$.

The meaning of the $\models$ relation is that a formula is true in consequence of the structure of S induced by R, the functions v and the meaning of the truth functional connectives and the modal operator $\Box$.

It remains to restrict attention to a class of models appropriate to the intended interpretation of $\Box$. This is done by establishing some conditions on M. We call these *standard model conditions*. The appropriate condition for the interpretations of $\Box$ which we have considered is *transitivity*. The condition is imposed by requiring that

$$\forall u,v,w \in S\,,\ uRv \text{ and } vRw \text{ implies } uRw$$

Henceforth we restrict our attention to standard models. Thus the expression $\models \phi$ is interpreted to mean that ϕ is valid for the class of all transitive models.

We now turn our attention to axiomatising the formulae of L_K which are valid for the class of transitive models. Below and throughout, we use "PC" to refer to the Propositional Calculus.

Our axioms Ax are:

1. any set of axioms adequate for PC
2. $\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$ *K*
3. $\Box\phi \to \Box\Box\phi$ *4*

The symbols *K* and *4* are commonly used to name the indicated axioms.

The rules of inference are

MP: modus ponens

RN: from ϕ derive $\Box\phi$

A *logic* is a subset Λ of *Fma* which contains all instances of schemata Ax1, Ax2 and Ax3, and which is closed under the inference rules MP and RN. That is,

$$\phi \in \Lambda \text{ and } (\phi \to \psi) \in \Lambda \text{ implies } \psi \in \Lambda$$

and

$$\phi \in \Lambda \text{ implies } \Box\phi \in \Lambda.$$

This system of axioms and inference rules is known in the literature as System K4 [2]. The system is named after its distinguishing modal principles. When our axiom 3 is dropped, the result is the minimal modal logic K.

A Λ-*theory* or *theory* is a subset Γ of *Fma* which contains Λ and which is closed under the rule MP. The intersection of any collection of Λ-theories is a Λ-theory itself. It follows that there is a smallest Λ-theory, namely Λ.

The difference between a logic and one of its theories is this. A logic is intended to be those formulae which are true of every state in a model, while a theory, as an extension of a logic, will be true only at some states. The additional formulae in a theory may be regarded as the consequences of certain assumptions, say, about the definition of v at a particular point. This explains why theories are not closed under the rule RN. The fact that ϕ is the case at some state does not imply that ϕ is true at all related states. For instance, it does not *logically* follow that it will rain tomorrow in Vancouver just because it is raining today.

We now define the *deducibility* relation $\vdash_\Lambda$ for a login Λ. We use Σ to represent an arbitrary subset of *Fma*.

$$\Sigma \vdash_\Lambda \phi \text{ iff for all } \Lambda\text{-theories } \Gamma \text{ such that } \Sigma \subseteq \Gamma,\ \phi \in \Gamma$$

That is, $\Sigma \vdash_\Lambda \phi$ if ϕ belongs to every Λ-theory which is an extension of Σ. The reader should verify that $\{\phi : \Sigma \vdash_\Lambda \phi\}$ is a Λ-theory. We write $\vdash_\Lambda \phi$ to mean $\phi \in \Lambda$. It is easy to check that $\vdash_\Lambda \phi$ iff $\varnothing \vdash_\Lambda \phi$, where $\varnothing$ is the empty set.

We say that Σ is consistent if $\Sigma \nvdash_\Lambda \mathbf{false}$. Since $(\mathbf{false} \rightarrow \phi)$ is a tautology, this is equivalent to the requirement that $\Sigma \nvdash_\Lambda \phi$ for at least one formula ϕ.

A *maximal consistent Λ-theory* is a consistent Λ-theory Γ with the property that, for each formula ϕ, either $\phi \in \Gamma$ or $\neg\phi \in \Gamma$.

Before we proceed with the proofs of soundness and completeness, we prove a few results about Λ-theories.

3.1.1 the □-Deduction **Theorem**

$$\text{If } \Sigma \vdash_{\Lambda} \phi \text{ then } \{\Box\psi : \psi \in \Sigma\} \vdash_{\Lambda} \Box\phi$$

Proof:

Let Γ be any any Λ-theory containing $\{\Box\psi : \psi \in \Sigma\}$. Define $\Delta = \{\psi : \Box\psi \in \Gamma\}$. We need to show that $\Box\phi \in \Gamma$, i.e. that $\phi \in \Delta$. By our hypothesis, it is enough to show that Δ is a Λ-theory containing Σ.

If $\psi \in \Sigma$ then $\Box\psi \in \Gamma$ by choice of Γ. Then, by definition of Δ $\psi \in \Delta$.

If $\vdash_{\Lambda} \psi$ then, by RN, $\vdash_{\Lambda} \Box\psi$, so $\Box\psi \in \Gamma$ and $\psi \in \Delta$.

Finally we show that Δ is closed under the rule MP. If $\psi \rightarrow \theta$, $\psi \in \Delta$ then $\Box(\psi \rightarrow \theta)$, $\Box\psi \in \Gamma$. Since Γ is a Λ-theory, it contains the Ax2 instance $\Box(\psi \rightarrow \theta) \rightarrow (\Box\psi \rightarrow \Box\theta)$. By MP-closure of Γ, $\Box\theta \in \Gamma$ whence $\theta \in \Delta$, as required. ■

This somewhat indirect proof is a consequence of our abstract definition of deducibility. An alternate definition is

> ϕ is deducible from Σ iff there exists a finite ordered list of formulae whose last member is ϕ with the property that every entry is an axiom, a member of Σ, or follows from earlier entries by application of MP.

Such a list is called a *proof* or *derivation* of ϕ from the *assumptions* Σ. Given such a definition of $\vdash_{\Lambda}$, the proof of the theorem would consist of an inductive *construction* of a derivation of $\Box\phi$ given $\{\Box\psi : \psi \in \Sigma\}$ from a derivation of ϕ. Given such a definition of theoremhood, the set of Λ-theorems is recursively enumerable (which is why such definitions are used.) The snag consists in the assumption that inference rules may refer only to a finite number of antecedents. For the *infinitary logics* we consider in later chapters, this is not the case. Proofs appealing to the structures of derivations would require the techniques of trans-finite induction. We achieve the same results using only the most elementary set theory.

The following Corollary is used later in the completeness proof.

3.1.2 **Corollary** If Γ is a Λ-theory, then

$$\Box\phi \in \Gamma \text{ iff } \Box(\Gamma) \vdash_{\Lambda} \phi,$$
$$\text{where } \Box(\Gamma) = \{\phi : \Box\phi \in \Gamma\}$$

Proof:

Necessity: if $\Box\phi \in \Gamma$ then $\phi \in \Box(\Gamma)$, and so $\Box(\Gamma) \vdash_{\Lambda} \phi$.

Sufficiency: if $\Box(\Gamma) \vdash_{\Lambda} \phi$ then, by the Lemma, $\Box\phi$ is -deducible from the set

$$\{\Box\psi : \psi \in \Box(\Gamma)\} = \{\Box\psi : \Box\psi \in \Gamma\} \subseteq \Gamma.$$

Since Γ is a Λ-theory, $\Gamma \vdash_{\Lambda} \Box\phi$. It follows from the definition of $\vdash_{\Lambda}$ and the fact that Γ is a Λ-theory containing itself that $\Box\phi \in \Gamma$. ■

3.1.3 The Deduction **Theorem**

$$\Sigma \cup \{\phi\} \vdash_{\Lambda} \psi \text{ iff } \Sigma \vdash_{\Lambda} (\phi \rightarrow \psi)$$

Proof:

Sufficiency: immediate from the observation that deducibility is preserved by set inclusion and the fact that $\{\phi : \Sigma \vdash_{\Lambda} \phi\}$ is a Λ-theory.

Necessity: assume $\Sigma \cup \{\phi\} \vdash_{\Lambda} \psi$. Let

$$\Gamma = \{\theta : \Sigma \vdash_{\Lambda} (\phi \rightarrow \theta)\}.$$

We wish to show that $\psi \in \Gamma$. By the hypothesis we have to show that Γ is a Λ-theory containing Σ and ϕ. From here the proof proceeds as with (3.1.1). ■

3.1.4 **Theorem** For any maximal consistent Λ-theory Γ,

$$(\phi \rightarrow \psi) \in \Gamma \text{ iff}$$
$$\phi \in \Gamma \text{ implies } \psi \in \Gamma.$$

Proof:

Necessity: is just the MP-closure of Γ.

Sufficiency: suppose $\phi \in \Gamma$ implies $\psi \in \Gamma$. If $\phi \in \Gamma$ then MP closure applied to the propositional tautology $\psi \rightarrow (\phi \rightarrow \psi)$ yields the result. If $\phi \notin \Gamma$, then by maximality, $\neg\phi \in \Gamma$. In that case, MP closure of Γ applied to the tautology $\neg\phi \rightarrow (\phi \rightarrow \psi)$ again gives $(\phi \rightarrow \psi) \in \Gamma$. ■

The object of a completeness proof is to connect the purely syntactic notion of deducibility with the semantic notion of truth in a model.

The theorem we wish to prove is that, for any $\phi \in Fma$,

$$\vdash_{K4} \phi \text{ iff } \models \phi,$$

that is, that ϕ is a Λ-theorem if and only if ϕ holds in each state of every standard model.

The easy direction of this biconditional is the Soundness part from left to right.

Soundness is demonstrated by showing that the axioms of Λ are valid in standard models, and that the rules of inference preserve validity. Since all propositional tautologies are valid due to our standard interpretation of implication and negation, we need only be concerned with Ax2 and Ax3.

To see that Ax2 is valid, assume the contrary, i.e. that it is false at some state u in a model M. Applying the truth condition for $\rightarrow$ we see that this means

$$M \models_u \Box(\phi \rightarrow \psi) \text{ and } M \models_u \Box\phi \text{ and } M \not\models_u \Box\psi$$

iff

$$M \models_u \Box(\phi \rightarrow \psi) \text{ and } M \models_u \Box\phi \text{ and } \exists \text{ v s.t. u}R\text{v and } M \not\models_v \psi.$$

But, by the truth condition for $\Box$, $M \models_v (\phi \rightarrow \psi)$ and $M \models_v \phi$, which in turn, by the truth condition for $\rightarrow$, means that $M \models_v \psi$, which is a contradiction. It follows that Ax2 is valid.

To see the soundness of Ax3, we again proceed by *reductio*. Suppose that

$$M \not\models_u \Box\phi \rightarrow \Box\Box\phi$$

then,

$$M \models_u \Box\phi \text{ and } \exists \text{ v s.t. u}R\text{v and } M \not\models_v \Box\phi,$$

which means, in turn, that $\exists$ w s.t. vRw and $M \not\models_w \phi$. But, since R is transitive, uRw holds, and so $M \models_w \phi$ which is the desired contradiction.

It is obvious that the rule MP preserves validity by the truth condition on $\rightarrow$.

To show that RN is sound, again assume otherwise. That is, assume that for some valid ϕ there is a state u in a model M such that $M \not\models_u \Box\phi$. But then there must be a state v in M with uRv and $M \not\models_v \phi$, which contradicts the assumed validity of ϕ.

These arguments complete the soundness proof for K4.

Completeness is shown by means of a *canonical* model construction. The canonical Λ model for a logic Λ is the structure $M_\Lambda = (S_\Lambda, v, R_\Lambda)$, where

S_Λ is the set of all maximal consistent Λ-theories,

R_Λ is defined $\forall \Gamma,\Delta \in S_\Lambda$ by

$$\Gamma R_\Lambda \Delta \text{ iff } \forall \phi \in Fma\,(\Box\phi \in \Gamma \text{ implies } \phi \in \Delta)$$

and v is defined such that $\forall p \in P, v_\Gamma(p) = 1$ iff $p \in \Gamma$.

Before using the canonical model, we must check to see that it is a standard model. That is, we must show that R_Λ is a transitive relation. Suppose, on the contrary, that there exist states x, y and z in S_Λ with $xR_\Lambda y$, $yR_\Lambda z$, but not $xR_\Lambda z$. The last condition means that for some formula ϕ, $\Box\phi \in x$ but that $\phi \notin z$ which in turn means, since z is maximal, that $\neg\phi \in z$. Since x is a Λ-theory, the Ax2 instance

$$\Box\phi \rightarrow \Box\Box\phi$$

belongs in x and so by MP closure, $\Box\Box\phi \in x$. Then by the definition of R_Λ, $\Box\phi \in y$ and so again by the definition $\phi \in z$. But then, z is inconsistent. This assures us the M_Λ is indeed a standard model.

The next result assures us that the states of M_Λ provide an adequate interpretation of deducibility. Specifically, we prove that if $\Sigma \nvdash_\Lambda \phi$ then for some maximal consistent extension Λ of Σ, $\phi \notin \Lambda$. It is not necessary to attempt to construct a canonical model which has states corresponding to every Λ-theory.

3.1.5 **Theorem** For any $\Sigma \subseteq Fma$,

$$\Sigma \vdash_{\Lambda} \phi \text{ iff } \forall u \in S_{\Lambda}(\Sigma \subseteq u \text{ implies } \phi \in u)$$

Proof:

Necessity is immediate from the definition of $\Sigma \vdash_{\Lambda} \phi$, as each u is a Λ-theory.

to prove sufficiency, we assume that $\Sigma \nvdash_{\Lambda} \phi$. A maximal consistent set is constructed which contains the set $\Sigma \cup \{\neg \phi\}$. We will not carry this proof out here; the technique is illustrated in § 5. ■

An important consequence of (3.1.5) is that every consistent set of formulae has a maximal consistent extension, in particular, a member of S_{Λ}. To see that this is the case, note that $\Sigma \nvdash_{\Lambda} \phi$ for some formula ϕ, or else Σ is not consistent. The constructive part of the proof then yields a maximal consistent set which contains Σ. (3.1.5) is actually Lindenbaum's Lemma for the logic Λ.

We employ the canonical model to prove the

Fundamental Theorem for Λ: $\forall \phi \in Fma$ and $\forall\ u \in S_{\Lambda}$,

$$M_{\Lambda} \models_{u} \phi \text{ iff } \phi \in u$$

■

With the Fundamental Theorem in hand, completeness for the logic Λ is obtained by the following argument. We first establish

The Completeness Theorem for Λ

$$\vdash_{\Lambda} \phi \text{ iff } M_{\Lambda} \models \phi$$

Proof:

Assume $\nvdash_{\Lambda} \phi$. Then the set $\{\neg\phi\}$ is consistent by a simple corollary of the Deduction Theorem. By (3.1.5), $\Lambda \cup \{\neg\phi\}$ has a maximal consistent extension which must be some state u of the Canonical Model. By the Fundamental Theorem,

$M \models_u \neg\phi$ and so $\not\models \phi$ by definition. Contraposition yields the completeness result. ■

Using the completeness result for an arbitrary logic Λ, the completeness theorem for K4 follows easily. The logic K4 is, by definition, the smallest logic. It is easy to show that, for any standard model M, the set $\{\phi : M \models \phi\}$ is a logic, and so contains K4. Thus all K4 theorems are valid in M. On the other hand, if ϕ is valid in all standard models, it is valid in the canonical K4 model. The previous theorem then yields that $\vdash_{K4} \phi$.

We now undertake the proof of the Fundamental Theorem. The proof is by induction on the complexity of formulae.

If ϕ is an propositional letter, the result is from the definition of v.

Assume the result for ϕ and ψ.

If $M_\Lambda \models_u \neg\phi$ then $M_\Lambda \not\models_u \phi$ iff $\phi \notin u$ by the hypothesis of induction. Maximality of u tells us that this is equivalent to $\neg\phi \in u$.

If $M_\Lambda \models_u (\phi \rightarrow \psi)$ the result follows by the truth condition on $\rightarrow$ and Theorem (3.1.4)

Of course, the hard part is the induction step for $\Box\phi$. We prove first that

$$\Box\phi \in u \text{ implies } M_\Lambda \models_u \Box\phi.$$

> Suppose otherwise, i.e. $\Box\phi \in u$ and $M_\Lambda \not\models_u \Box\phi$. Then there is a state v for which $uR_\Lambda v$ and $M_\Lambda \models_v \neg\phi$. By the induction hypothesis, $\neg\phi \in v$. But since $uR_\Lambda v$, the assumption on $\Box\phi$ requires that $\phi \in v$, contradicting the consistency of v.

To complete the proof, we show that

$$M_\Lambda \models_u \Box\phi \text{ implies } \Box\phi \in u.$$

Let $u(\Box) = \{\psi : \Box\psi \in u\}$.

Suppose $u(\Box) \subseteq v \in S_\Lambda$. Then, by definition, $uR_\Lambda v$. The truth condition on $\Box\phi$ gives us that $M_\Lambda \models_v \phi$ and so by the induction hypothesis, $\phi \in v$. Thus, ϕ belongs to every maximal consistent theory that contains $u(\Box)$ and so, by (3.1.5), $u(\Box) \vdash_\Lambda \phi$. (3.1.2) proves that $\Box\phi \in u$.

This completes the proof of the Fundamental Theorem.

3.2. Other Temporal Modalities

This section contains a definition and discussion of other modalities which are commonly encountered in programming logics. These logics are often referred to as *temporal logics*. Strictly speaking, this is in most cases a misnomer. As may now be clear a logic is not *about* a structure (*e.g.* time) merely because the language contains certain modal symbols. The axioms describing the behaviour of the operators are what really tell us what the logic is referring to. Temporal logics have suggested themselves to computing scientists because the sequence of states which constitutes the execution of a programme is a partial order (for non-deterministic programmes) or a linear order (for deterministic programmes.) However, the "temporal" logics of computing science mostly lack the axioms which logicians employ to characterise the temporal order.

Below, we employ the '$<$' symbol for a generic binary relation. If s is a state of model M for which $<$ is discrete, s' is the next state under $<$. That is, $s < s' \wedge \neg\exists t\,(s < t < s')$

The *Until* operator, written $(\phi U \psi)$, is read "ϕ is true (at least) *until* ψ becomes true". It is used in the logic of programmes to represent programme invariants.

The **atnext** operator is a dual of *Until* which has been investigated in [21]. It is intended to mean "in the next state in which ψ, also ϕ". Both *Until* and **atnext** are well defined only in linear orders.

The $\bigcirc$ operator is used to refer to the *next* state under $<$. So, $\bigcirc\phi$ is true if ϕ is true *next*. This particular operator is used when the existence of a unique next state is guaranteed.

The C^* or *chop* operator, written $(\phi C^* \psi)$, has been used in [32] to construct axioms for loops. The meaning is that ϕ is true at zero or more successive states but that eventually ψ becomes true. ψ represents a condition holding at the termination of a loop, while ϕ represents an initial condition for each iteration.

These operators may be interpreted in our semantics as follows.

$M \models_s (\phi \mathbf{U} \psi)$ iff $\exists t(s < t$ and $M \models_t \psi$ and $\forall t'(s < t' < t$ implies $M \models_{t'} \phi))$

$M \models_s (\phi \mathbf{atnext} \psi)$ iff $\forall t(s < t$ implies $M \models_t \neg\psi) \vee \mu t(s < t \wedge M \models_t \psi)(M \models_t \phi)$

$M \models_s \bigcirc\phi$ iff $M \models_s \phi$

$M \models_s \phi \mathbf{C}^* \psi$ iff

either, for some n $>$ 0 there is a finite ordered set of states $t_0, t_1 \ldots t_n$ s.t. $t_i < t_{i+1}$ and $s = t_0$ and $\forall i ((0 \leqslant i < n)$: $M \models_{t_i} \phi)$ and $M \models_{t_n} \psi$

or else $M \models_s \Box\Diamond\phi$

The *satisfiability problem* for a logic Λ is: for an arbitrary formula ϕ, is ϕ true at some state in some model? A logic is said to be *decidable* if the satisfiability problem is decidable.

Two techniques for proving decidability exist. The first is to demonstrate an algorithm which is capable of constructing a model for ϕ. If model construction admits to an effective procedure, it is natural to inquire of its computational complexity. Just how long does it take to build? Equivalently, how many states must the model have for a given formula?

The second technique is to demonstrate that any instance of the satisfaction problem may be effectively reduced to an instance of some other problem of known decidability and complexity. In his classic paper [37] Rabin showns that the decision problem for the weak second order theory of n-successors (SnS) is decidable. Translating, quantifiers over finite sets and monadic set predicates are allowed in the language, while models are (infinite) trees where each node has at most n immediate successors. The logic K4 is proven decidable by such a reduction in [13]. Results dependent upon reduction to SnS are small cause for wild excitement: the complexity of Rabins' algorithm is not elementary recursive [26]. It could be worse. Logics have been demonstrated [44] which are decidable but for which there is no recursive bound on the size of the models which are required for the algorithm.

In the case of K4, an exponential upper bound is known, since K4 is no more complex than the logic considered in [31].

The decision problems for languages containing **U**, **atnext** and the 'chop operator' $\mathbf{C}^*$ are inherently more difficult than those containing only $\Box$. These binary operators are not finitely expressible in the language L_K above. That is, no K4-formula of finite length is equivalent to a nontrivial instance of a formula containing U. An example of a 'trivial instance' is the formula $\phi U \mathbf{false}$.

The operators U, **atnext** and C^* are essentially second order predicates because they assert the existence of a connected interval of points with some property. We can 'get away' without an explicitly second order definition only because we are assuming that $<$ is a transitive relation. In the first order meta-language we are using to define truth conditions, these operators are not expressible using a single quantifier. For instance, $p \mathbf{U} q$ translates more or less as

$$\exists y \ \forall x \ (Q(y) \ \textbf{and} \ (now < x < y \rightarrow P(x)))$$

while $\Box q$ is

$$\forall x \, (x > now \rightarrow P(x)).$$

These two formulae belong to different levels in the arithmetical hierarchy [39]. As a result, one would expect the decision problem for languages containing *Until* and *chop* operators to be harder than for simpler languages, as is indeed the case. Some programming logics containing the equivalent of the *Until* operator are known to be decidable by reduction to SnS. The Process Logics of [47, 16, 30] are examples. Some logics with *chop* are known to be undecidable [16]. The point is that the complexity of the decision problem for logics of programmes arises already in languages which have no syntactic representations of programmes at all.

The difficulty of the decision problem is intimately connected to the expressive power of the language used. Decidability is not an issue in this thesis; the logic under development will turn out to be undecidable. Nevertheless, it seems worthwhile to keep the pieces as manageable as possible.

3.3. System K as a Programming Logic

We conclude this chapter by showing how those properties of programme execution advanced in the preceding chapter may be expressed in a language as simple as L_K. Interpret the following formulae as if the model's relation coincided with the steps in the execution of a programme.

throughout:	$\Box\phi$
during:	$\Diamond\phi$
preserves:	$\Box(\phi \rightarrow \Box\phi)$
liveness:	$\Box(\phi \rightarrow \Diamond\psi)$

(i.e. if ϕ, then sometime later ψ.)

fairness: if ϕ represents a guard and if ψ represents some consequence of the execution of the associated sequential component, then

$$\Box\Diamond\phi \rightarrow \Box\Diamond\psi .$$

Interpret the modality $\Box\Diamond$ as *always eventually*, or *infinitely often.*

after: $\Box false \rightarrow \phi$

The representation of *after* uses the modal formula $\Box false$. This formula is true only at states which are related to no other states. In our informal interpretation, such a state is the last state in a programme. So if ϕ is supposed to be true *after* programme α has finished, the correspondence becomes clear.

In the next Chapter we present a programming logic in which the *after* modality is fundamental.

Chapter 4

The I/O Logic of Processes

Robert Goldblatt in [14] presents a sound and complete axiomatisation of the input/output behaviour for a class of programmes which contains conditional and iterative instructions. In this chapter, these results are extended by adding a new programming construct to the syntactic class of well formed programmes. This new construct is a syntactic representation of a Process, or set of guarded commands. We dub this representation of processes *generalised iteration*.

The development closely parallels that of §§ 3.1. We define a language $L_{I/O}$ which contains an infinite set of modal operators; one, in fact, for each programme. We define a model for the language, and give standard model conditions such that the meaning of the modalities corresponds to the aspects of programme behaviour being modelled. A set of valid formula schemata are introduced as axioms. Finally, the inferential closure of the set of axioms is shown to be complete with respect to the given semantics.

Of necessity, many of the definitions, and some of the commentary, are appropriated from Goldblatt's work. For the sake of brevity, only those theorems which required extension are presented. All theorems which are straightforward extensions of results in [14] are marked by an italicised reference to the corresponding theorem. The reference given is the number of the result in [14]. References to theorems which hold without modification are made *e.g.* (G2.4.3(1)), again using Goldblatt's numbering. The statements of all such theorems appear in Appendix A.

4.1. Syntax

In this section we present the formal language $L_{I/O}$ which is used to construct formulae describing the input/output behaviour of sequential programmes and processes.

4.1.1. Boolean Expressions

Let

$$Bvp = \{p_0, p_1, p_2, \ldots\}$$

be a denumerable set of *boolean variables*. The set Bxp of *Boolean expressions* is defined inductively:

1. The constant symbol *false* is in Bxp,
2. Bvp is a subset of Bxp,
3. If ϵ, δ, ξ are in Bxp, then so is $(\epsilon \supset \delta, \xi)$,
4. If ϵ, δ are in Bxp, so is $(\epsilon = \delta)$.

The constant expression *true* is introduced as an abbreviation for $(false = false)$.

The *value* of a Boolean expression is the member of the data type

$$\mathbb{B} = \{0, 1\}$$

which it denotes. Boolean variables denote some state dependent member of $\mathbb{B}$, while constant symbols denote fixed members $\mathbb{B}$. The constant symbol *false* names the element 0.

The symbol $\supset$ is the logical *conditional* connective. The expression $(\epsilon \supset \delta, \xi)$ is read

if ϵ then δ, else ξ.

We indicate below how the conditional may be used to define the logical operations common in the conditional expressions of programming languages. The expression $(\epsilon = \delta)$ takes the value 1 when ϵ and δ denote the same member of $\mathbb{B}$, and 0 otherwise. The set $\mathbb{B}$ the function symbols *false* and $\supset$, together with equality constitute a Boolean Algebra.

Boolean expressions serve two roles in our system. An expression functions as a *term*, denoting a *member* of $\mathbb{B}$, and as an *atomic formula* in the *language* of $\mathbb{B}$.

4.1.2. Programmes

Let $\pi_0, \pi_1, \ldots$ be a denumerable list of *programme letters*. The set *Cmd* of *commands* is defined by induction.

1. **skip** and **abort** are in *Cmd*.
2. Each programme letter π is in *Cmd*,
3. If α, β are in *Cmd*, then so is $(\alpha;\beta)$,
4. If α, β are in *Cmd* and ϵ is in *Bxp* then $(\epsilon \Rightarrow \alpha, \beta)$ is in *Cmd*,
5. If α is in *Cmd* and ϵ is in *Bxp* then $(\epsilon \# \alpha)$ is in *Cmd*.

Programme letters are uninterpreted place holders for actual instructions. The *skip* command is a No-Operation or null command. *abort* is defined by Dijkstra thus

> When invoked, the mechanism named "abort" will fail to reach a final state: its attempted activation is interpreted as a symptom of failure.

The command $(\alpha;\beta)$ is the conjunction of the commands α and β. It represents the programmes "do α and then do β". The command $(\epsilon \Rightarrow \alpha,\beta)$ is the conditional (*if* ϵ *then* α *else* β) and $(\epsilon \# \alpha)$ is the iterative (*while* ϵ *do* α).

Processes

We now define a new set *Proc*, the set of Processes.

1. If ϵ_j is in *Bxp* and α_j is in *Cmd* for $(1 \leqslant j \leqslant k)$ then $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$ is in *Proc*,

2. If Π is in *Proc* and α is in *Cmd*, then $(\alpha ; \Pi)$ is in *Proc*.

Processes represent the sets of guarded commands defined in §§ 2.1. The commands α_i are the sequential components. The boolean expression ϵ_i is the guard for α_i. Clause 2 in the definition of *Proc* allows that a process have a sequential initialisation phase which is executed before execution of the iterative phase begins. A process may not contain further processes embedded within it, as discussed in §§ 2.1. However, it is easy to see how to extend the inductive definition to allow this. Due both to the syntactic and semantic resemblance between processes and iterative commands, we sometimes call processes *generalised iterators*.

Finally, define the set of Programmes *Pgm* to be the union of *Cmd* and *Proc*.

4.1.3. Formulae

We now define the class *Fma* of well formed formulae of the language $L_{I/O}$.

1. $Bxp \subseteq Fma$,

2. If ϕ, ψ are in *Fma*, so is $(\phi \rightarrow \psi)$,

3. If ϕ is in *Fma* and α is in *Pgm*, then $[\alpha]\phi$ is in *Fma*.

Using the connective $\rightarrow$, the other logical connectives are defined in the usual way. In particular, negation is defined

$$\neg\phi \ =_{df} \ (\phi \rightarrow \mathit{false})$$

Since *Fma* is a denumerable set, we may assume the existence of a fixed enumeration of its members.

Throughout, unless otherwise specified, ϵ, δ and ξ denote members of *Bxp*, α and β denote members of *Cmd*, Π_k denotes some member $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$ of *Proc* with k components and ϕ, ψ, θ and χ denote members of *Fma*. These designators may be subscripted where convenient.

The intended meaning of the modal formula $[\alpha]\phi$ is:

all terminating computations of α terminate in a state in which ϕ holds

That is, $[\alpha]$ is the *after* modality for programme α.

4.2. Semantics

4.2.1. Models

A Model for the language $L_{I/O}$ is a structure $M = (S, v, [\![\ ]\!])$ where

1. S is a non-empty set of states;
2. v is an operator which associates with every state $s \in S$ a valuation, i.e. a function $v_s : Bvp \rightarrow \mathbb{B}^+$, where $\mathbb{B}^+ = \mathbb{B} \cup \{\omega\}$;
3. $[\![\cdot]\!]$ is an operator that associates with each α in *Cmd* and each Π in *Proc* a binary relation $[\![\alpha]\!]$ or $[\![\Pi]\!]$ on S.

A member ϵ of *Bvp* takes the value ω in $\mathbb{B}^+$ if ϵ is undefined in $\mathbb{B}$. For each $s \in S$, the valuation v_s extends canonically to all of *Bxp* by induction on the formation rules for Boolean expressions:

1. $v_s(false) = 0$

2. If $v_s(\epsilon) = \omega$ then $v_s(\epsilon \supset \delta, \xi) = \omega$. Otherwise,

$$v_s(\epsilon \supset \delta, \xi) = \begin{cases} v_s(\delta) & \text{if } v_s(\epsilon) = 1 \\ v_s(\xi) & \text{if } v_s(\epsilon) = 0 \end{cases}$$

3. If either $v_s(\epsilon)$ or $v_s(\delta)$ is ω, then $v_s(\epsilon = \delta) = \omega$. Otherwise,

$$v_s(\epsilon = \delta) = \begin{cases} 1 & \text{if } v_s(\epsilon) = v_s(\delta) \\ 0 & \text{if } v_s(\epsilon) \neq v_s(\delta) \end{cases}$$

Writing $D\epsilon$ as an abbreviation for $(\epsilon = \epsilon)$, we find from clause 3 that

$$v_s(D\epsilon) = \begin{cases} \omega & \text{if } v_s(\epsilon) = \omega \\ 1 & \text{if } v_s(\epsilon) \neq \omega \end{cases}$$

and hence

$$v_s(true) = v_s(D\,false) = 1$$

The definition of $v_s(\epsilon \supset \delta, \xi)$ is intended to represent the way an optimising compiler would parse such a conditional. By *evaluation* we mean those steps a computer would actually execute in order to determine the value of an expression. If $v_s(\epsilon) = \omega$ then the entire expression takes the value ω, meaning that the evaluation fails to terminate. In that case, conditional, iterative and (as we will prove) guarded commands attempting the evaluation will abort.

If the value of ϵ is 1, the result of the expression is $v_s(\delta)$, independent of the value or undefinedness of ξ. $(\epsilon \supset \delta, \xi)$ may be used to define the logical *and*, *or* and all other logical operators employed in conditional expressions. In all cases, the definition is such that only those terms needed to determine the value of the expression are evaluated. For instance if $v_s(\epsilon) = 1$ and $v_s(\delta) = \omega$, $v_s(\epsilon \vee \delta)$ is 1, not ω. We explicitly define only one logical connective other than $\supset$,

$$not{-}\epsilon =_{df} (\epsilon = false).$$

Then *not–false* is *true* and *not–*ϵ is undefined only if ϵ is.

Let $A_1 \ldots A_{n+1}$ be arbitrary data types (i.e. algebras like $\mathbb{B}$.) A_i^+ denotes the data type which has an element ω adjoined with the elements of A_i. A function f of the form

$$f : A_1 \times \ldots \times A_n \rightarrow A_{n+1}$$

may be extended canonically to one of the form

$$A^+_1 \times \ldots \times A^+_n \rightarrow A^+_{n+1}$$

by putting $f(x_1, \ldots, x_n) = \omega$ whenever one or more of its arguments is ω. The idea is that a function is undefined whenever one or more of its arguments is. However, the operations on $\mathbb{B}^+$ definable using $(\epsilon \supset \delta, \xi)$ are not in general just the canonical extensions of the same operations on $\mathbb{B}$.

It is worth pointing out that any two place operator on $\mathbb{B}$ is definable using only $\supset$ and the constant symbol *false*. For instance, the implication of the Propositional Calculus may be defined

$$(\epsilon \supset \delta, \mathit{true}).$$

4.2.2. Satisfaction

We now define the *satisfaction* relation "ϕ *holds* at s in M", written

$$M \models_s \phi.$$

We sometimes omit the prefix M when no ambiguity results. ϕ is *valid* in M, written $M \models \phi$ if ϕ is satisfied at every state in M. ϕ is valid *per se* if it is valid in all models. The relation $\models$ is defined inductively for all formulae ϕ.

Where $\epsilon \in Bxp$, $\phi, \psi \in Fma$ and $\alpha \in Pgm$:

1. $M \models_s \epsilon$ iff $v_s(\epsilon) = 1$.

2. $M \models_s (\phi \rightarrow \psi)$ iff $M \models_s \phi$ implies $M \models_s \psi$.

3. $M \models_s [\alpha]\phi$ iff $\forall t \in S$ ($s [\![\alpha]\!] t$ implies $M \models_t \phi$).

In referring to these rules, we will use expressions such as 'by the semantic clause for $\rightarrow$' and the like.

The implication $\rightarrow$ and the Boolean operator $\supset$ should not be confused. The former acts as the material implication between members of *Fma*. The latter may be used only to construct complex boolean expressions. The formula $(\epsilon \rightarrow \delta)$ is *not* member of *Bxp*.

We illustrate the above rules by interpreting the formula $\neg [\alpha]false$:

$M \models_s \neg [\alpha]false$

iff $M \models_s [\alpha]false \rightarrow false$ iff — by def'n of negation

iff $M \not\models_s [\alpha]false$ iff — since $M \not\models false$

iff $\neg \forall t (s [\![\alpha]\!] t$ implies $M \models_t false)$ iff — by clause(3) above

iff $\exists t (s [\![\alpha]\!] t$ and $M \not\models_t false)$ iff — by meta-logical manipulation

iff $\exists t (s [\![\alpha]\!] t$ and $v_t(false) \neq 1)$ — by clause (1)

iff $\exists t (<s,t> \in [\![\alpha]\!])$ — since $v_t(false) = 0$ by definition.

which is to say, there is at least one terminating computation of α starting from s.

We conclude this section by clarifying the semantics of $\neg$, *not*— and *D*. An *n-ary* relation R on a set A may be identified with its characteristic function $d_R : A^n \rightarrow \mathbb{B}$ and then canonically extended to a function mapping $(A^+)^n$ to $\mathbb{B}^+$. If $n = 1$ and $R = A$ we define this canonical extension $d_A : A^+ \rightarrow \mathbb{B}^+$

$$d_A(a) = \begin{cases} \omega \text{ if } a = \omega \\ 1 \text{ if } a \in A \end{cases}$$

d_A represents the "defined" elements of A^+ (i.e. the members of A). Because

$$d_A(v_s(\epsilon)) \;=\; v_s(D\epsilon)$$

we may interpret $D\epsilon$ as "ϵ is defined". In fact,

$$M \models_s D\epsilon \quad \text{iff} \quad v_s(\epsilon) \neq \omega$$

However the interpretation of the Boolean expression $D\epsilon$ as "ϵ is defined" is an 'external' matter. The expression $D\epsilon$ can not be used as a test for the definedness of ϵ, since $D\epsilon$ is undefined whenever ϵ is. Thus a command such as $[D\epsilon \# \alpha]$ will abort if ϵ is undefined, rather than terminate.

We can abbreviate $not\text{—}(\epsilon = \delta)$ by $(\epsilon \neq \delta)$. This expression on $\mathbb{B}$ has an extension to $\mathbb{B}^+$, which turns out to be the canonical extension. Notice that $(\epsilon \neq \delta)$ is not equivalent to the formula $\neg(\epsilon = \delta)$, since the latter will hold (because $(\epsilon = \delta)$ does not) when either ϵ or δ is undefined, whereas $(\epsilon \neq \delta)$ can hold (have value 1) only when both ϵ and δ are defined. Similarly, $\neg\epsilon$ and $not\text{—}\epsilon$ differ in that

$$M \models_s not\text{—}\epsilon \;\text{ iff }\; M \models_s (\epsilon = false) \;\text{ iff }\; v_s(\epsilon) = 0$$

while

$$M \models_s \neg\epsilon \;\text{ iff }\; M \not\models_s \epsilon \;\text{ iff }\; v_s(\epsilon) \neq 1 .$$

4.2.3. Standard Models

As in the development of the modal logic in §§ 3.1, it will be necessary to restrict the relation $[\![\alpha]\!]$ so that the *meaning* of α is conveyed. As before,

> a model in which the properties of the relations $[\![\alpha]\!]$ reflect the intended meanings of commands will be called standard [14, p. 47].

In order to define standardness, we need to introduce some notation and operators for binary relations. We use R,P to refer to arbitrary subsets of $S\times S$. The expressions sRt and $<s,t> \in R$ are used interchangeably. We often write $sRuPt$ in place of sRu and uPt.

Define

Equality: $E_S = I = \{<s,s> : s \in S\}$

Restrictions:
$$\phi\backslash R = \{<s,t> : sRt \text{ and } \models_s \phi\}$$
$$[\![\phi\backslash\alpha]\!] = \phi\backslash[\![\alpha]\!]$$
$$[\![\phi\backslash\psi\backslash\alpha]\!] = (\phi \wedge \psi)\backslash[\![\alpha]\!]$$
$$R/\phi = \{<s,t> : sRt \text{ and } \models_t \phi\}$$
$$[\![\alpha/\phi]\!] = [\![\alpha]\!]/\phi$$

Composition: $P \cdot R = \{<s,t> : \exists u\,(sRuPt)\}$

Iteration:
$$R^0 = I$$
$$R^{n+1} = R^n \cdot R$$

Closure: $R^\infty = \bigcup_{n \in \mathbb{N}} R^n$

Recall that Π_k is an abbreviation for a Process $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$ with k components. We introduce $D\vec{\epsilon}_k$ as a definitional abbreviation for $(D\epsilon_1 \wedge, \ldots, \wedge D\epsilon_k)$. $D\vec{\epsilon}_k$ asserts that all k of Π_k's guards are defined. We remark that

$$not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow D\vec{\epsilon}_k$$

is a valid formula.

A model M for the language L is called *standard* if the operator $[\![\ ]\!]$ satisfies the following conditions.

a. $[\![\mathbf{skip}]\!] = I$

b. $[\![\mathbf{abort}]\!] = \varnothing$

c. $[\![\alpha\,;\beta]\!] = [\![\beta]\!]\ [\![\alpha]\!]$

i.e. $s\ [\![\alpha\,;\beta]\!]\ t$ iff $\exists u$ s.t. $s\ [\![\alpha]\!]\ u\ [\![\beta]\!]\ t$

d. $[\![\epsilon \Rightarrow \alpha,\beta]\!] = [\![\epsilon \backslash \alpha]\!] \cup [\![not\text{–}\epsilon \backslash \beta]\!]$

i.e. $s[\![\epsilon \Rightarrow \alpha,\beta]\!]t$ iff

$v_s(\epsilon) = 1$ and $s[\![\alpha]\!]t$ or
$v_s(\epsilon) = 0$ and $s[\![\beta]\!]t$

e. $[\![\epsilon\#\alpha]\!] = [\![\epsilon \setminus \alpha]\!]^{\infty} / not\text{–}\epsilon$

i.e. $s[\![\epsilon\#\alpha]\!]t$ iff for some $n \in \mathbb{N}$, $s[\![\epsilon \setminus \alpha]\!]^n t$ and $v_t(\epsilon) = 0$
iff for some n and some $s_0 \ldots s_n$,
$s = s_0, t = s_n$ and for $0 \leqslant i < n$
$v_{s_i}(\epsilon) = 1$, $s_i[\![\alpha]\!]s_{i+1}$ and $v_t(\epsilon) = 0$.

f. $[\![\Pi_k]\!] = (D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])^{\infty} / not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$

i.e. $s[\![\Pi_k]\!]t$ iff

$s (D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])^{\infty} t$ and $\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$

iff for some $n \in \mathbb{N}$,
$s[\![\setminus \Pi_k]\!]^n t$ and $v_t(not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)) = 1$

iff for some n and some $s_0 \ldots s_n$, $s = s_0, t = s_n$,
for each i, $0 \leqslant i < n$ ϵ_j is defined for $0 \leqslant j \leqslant k$,
$v_t(\epsilon_j) = 0$ for $0 \leqslant j \leqslant k$ and
for each i , $0 \leqslant i < n$ there is a j $0 \leqslant j \leqslant k$
such that $s_i v_{s_i}(\epsilon_j) \setminus [\![\alpha_j]\!] s_{i+1}$.

We usually write $[\![\setminus \Pi_k]\!]^n$ for the cumbersome $(D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])^n$.

In standard models, the relation assigned to a composite programme accurately reflect the programme's actual behaviour. For instance, $s[\![\alpha ; \beta]\!]t$ if and only if there is some u such tha $s[\![\alpha]\!]u[\![\beta]\!]t$, which means that the programme $[\alpha ; \beta]$ executed by first doing α, and then β.

One of the key restrictions on the relation $[\![\Pi_k]\!]$ is that $s[\![\Pi_k]\!]t$ implies $\models_s D\vec{\epsilon}_k$, which is to say, every guard must be defined initially, and *every* time guard evaluation occurs. When that is not the case, Π_k effectively aborts, since the set of states Π_k-related to s is empty.

Where $\models_s not-(\epsilon_1 \vee \ldots \vee \epsilon_k)$, all the relations $[\![\epsilon_j \setminus \alpha_j]\!]$ evaluate to the empty set, because $v(\epsilon_j) = 0$. Thus the entire expression $(D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])$ evaluates to the empty set. Then, by the definition of infinite closure, $(D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])^{\infty} = I$.

If $s[\![\Pi_k]\!]t$ then either $s = t$ and Π_k does not execute, or else t was reached by executing a finite sequence of *enabled* sequential components α_i. In either case, t is a final state because all the guards ϵ_i are false.

All this is to say that, in standard models, Processes execute in the way described in §§ 2.1

An important property of Process execution is fairness. A fair execution sequence can not be specified in the language L used in this chapter. This is because a fairness specification has to say something about the relative frequency of execution of each sequential component. The fairness condition introduced in §§ 2.1 was

> If a component is infinitely often enabled, then it is infinitely often executed

We are presently considering only terminating computations; that is, Process executions in which only a finite number of sequential components are executed. But under this rather liberal definition of fairness, any *finite* execution sequence is a fair one. In § 5, equipped with a stronger language, we will present formulae which specify fair execution.

It may appear that a fairness condition is covertly introduced by the requirement that all guards be defined at guard selection time. This requires that the 'selection daemon' at least examine all of the guards, no matter what scheduling algorithm it

employs. Fair schedules exist for which this need not be the case. There indeed are non-standard models in which any given scheduler would be 'fortunate enough' to find a particular guard defined when it required evaluation: and far more non-standard models in which this would fail to be the case. Short of encoding a particular scheduler into the standard model conditions, there is no alternative to the present restriction.

4.2.4. Analyses of Generalised Iteration

In the remainder of this section, we establish some results about the semantics of Processes. The results establish (for standard models) the validity of the axioms for Processes to be introduced in §§ 4.3.1.

4.2.1 **Theorem** *In any standard model M,* *[G2.3.4]*

$$[\![\Pi_k]\!] \;=\; (D\vec{\epsilon}_k \setminus \cup_{j=1,k} ([\![\Pi_k]\!] \cdot [\![\epsilon_j \setminus \alpha_j]\!])) \;\cup\; not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \setminus [\![\mathbf{skip}]\!]$$

Proof:

Suppose $s\,[\![\Pi_k]\!]\,t$. Then for some $n \in \mathbb{N}$

$$s\,(D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])^n\, t \text{ and } \models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$$

If $n = 0$ then $s = t$ and so $s\,(not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \setminus [\![\mathbf{skip}]\!])\,t$.

Otherwise, for some j,

$$s\,(\,(D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])^{n-1} \cdot D\vec{\epsilon}_k \setminus [\![\epsilon_j \setminus \alpha_j]\!]\,)\,t$$

so that, for some state u,

$$s\,(D\vec{\epsilon}_k \setminus [\![\epsilon_j \setminus \alpha_j]\!])\,u\,(D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])^{n-1}\, t.$$

But then, $u([\![\setminus \Pi_k]\!]^{n-1})t$ and so $u\,[\![\Pi_k]\!]\,t$. Thus we have $s\,[\![\alpha_j]\!]\,u\,[\![\Pi_k]\!]\,t$ and $\models_s \epsilon_j \wedge D\vec{\epsilon}_k$, which gives $s\,(D\vec{\epsilon}_k \setminus \cup_{j=1,k} ([\![\Pi_k]\!] \; [\![\epsilon_j \setminus \alpha_j]\!]))\,t$.

Conversely, suppose that $<s,t>$ belongs to the relation on the right side of the statement of the theorem. There are two cases.

Case 1:
$s [\![\mathbf{skip}]\!] t$ and $\models_s not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$

Then, $s = t$, so $s (D\vec{\epsilon}_k \setminus \cup_{j=1,k} [\![\epsilon_j \setminus \alpha_j]\!])^0 t$ and $\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$, giving $s [\![\Pi_k]\!] t$.

Case 2:
For some j and some state u, $\models_s \epsilon_j \wedge D\vec{\epsilon}_k$, $s [\![\alpha_j]\!] u$ and $u [\![\Pi_k]\!] t$. Then for some n, $u [\![\setminus \Pi_k]\!]^n t$ and $\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$. Then since $s (D\vec{\epsilon}_k \setminus [\![\epsilon_j \setminus \alpha_j]\!]) u$, $s [\![\setminus \Pi_k]\!]^{n+1} t$, which gives $s [\![\Pi_k]\!] t$ again.

■

The scary equation in the statement of the theorem is essentially a recurrence relation in $[\![\Pi_k]\!]$. It says that a pair $<s,t>$ is in the relation $[\![\Pi_k]\!]$ if and only if

1. $s = t$ and all the guards are false at s, (the meaning of the subformula "$not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \setminus [\![\mathbf{skip}]\!]$"), or else

2. all the guards are defined at s, $s [\![\alpha_j]\!] u$ for some enabled sequential component $[\alpha_j]$ and $u [\![\Pi_k]\!] t$.

There is a connection between this representation of the relations for iterative commands and the denotational semantics of Dana Scott. In denotational semantics, $[\![\Pi_k]\!]$ would be defined to be the least fixed point of a certain relational operator.

4.2.2 **Corollary** *The schema*

$$[\Pi_k]\phi \rightarrow (D\vec{\epsilon}_k \rightarrow \wedge^k_{i=1} (\epsilon_i \rightarrow [\alpha_i][\Pi_k]\phi))$$

is valid in all standard models.

Proof:

If not, then there must be a state s in some model M such that

$$M \models_s [\Pi_k]\phi \text{ and } M \not\models_s D\vec{\epsilon}_k \rightarrow (\epsilon_j \rightarrow [\alpha_j][\Pi_k]\phi)$$

for some $j, 1 \leqslant j \leqslant k$. Thus, $M \models_s \epsilon_j \wedge D\vec{\epsilon}_k$ and $M \not\models_s [\alpha_j][\Pi_k]\phi$ i.e. for some t such that $s[\![\Pi_k]\!] \cdot [\![\alpha_j]\!] t$, $M \not\models_t \phi$. But by the Theorem, $s[\![\Pi_k]\!] t$, *contra* the hypothesis that $[\Pi_k]\phi$. ■

4.2.3 **Corollary** *The schema*

$$not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow (\phi \leftrightarrow [\![\Pi_k]\!]\ \phi)$$

is valid in all standard models.

Proof:

Suppose $M \models_s not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$. By the Theorem, $s[\![\Pi_k]\!] t$ *iff* $s = t$.
Then $\models_s [\Pi_k]\phi$ *iff* $\models_s \phi$. ■

The formula $[\epsilon \Rightarrow \alpha,\beta]\phi$ is equivalent in standard models to

$$(\epsilon \rightarrow [\alpha]\phi) \wedge (not\text{–}\epsilon \rightarrow [\beta]\phi).$$

For non-iterative programmes, any modal formula may be reduced in like fashion to a single formula containing only the operators **[skip]**, **[abort]** and $[\pi]$ where π is a programme letter. The only 'reduction' possible with Processes, as with iterative programmes, is one which generates an infinite set of formulae whose modal operators involve just the sequential components α_j. We now define this reduction. If $\phi \in Fma$ and $[\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k] \in Proc$, define the series $\phi_0(\Pi_k) \ldots \phi_n(\Pi_k) \ldots$ inductively by letting

$$\phi_0(\Pi_k) =_{df} (not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow \phi)$$

and

$$\phi_{n+1}(\Pi_k) =_{df} D\vec{\epsilon}_k \rightarrow \wedge^k_{i=1} (\epsilon_i \rightarrow [\alpha_i]\phi_n(\Pi_k)).$$

This sequence reduces to the similar sequence for iterative programmes for processes with only one component, namely

$$\phi_0(\epsilon,\alpha) =_{df} not\text{–}\epsilon \rightarrow \phi$$

$$\phi_n(\epsilon,\alpha) =_{df} \epsilon \rightarrow [\alpha]\phi_n(\epsilon,\alpha).$$

In this case the requirement $D\epsilon$ is redundant, since $(\epsilon \rightarrow D\epsilon)$ is a valid formula.

4.2.4 **Theorem** *In any model M, for any* $n \in \mathbb{N}$ *[G 2.3.6]*

$$\models_s \phi_n(\Pi_k) \text{ iff } (s[\![\setminus \Pi_k]\!]^n t \text{ and } \not\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)) \text{ implies } \models_t \phi$$

Proof:

The result is shown for all s in M, by induction on n.

When n=0, we must show that

$$\models_s \phi_0(\Pi_k) \text{ iff } (s = t \text{ and } \not\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)) \text{ implies } \models_t \phi.$$

i.e. that $\models_s not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow \phi$ iff $\models_s not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$ implies $\models_s \phi$ which is just the semantic clause for $\rightarrow$.

Now assume the result for n. Let $\models_s \phi_{n+1}(\Pi_k)$. Then, if $s[\![\setminus \Pi_k]\!]^{n+1} t$ and $\not\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$, for some j and some state u

$$s(D\vec{\epsilon}_k \setminus [\![\epsilon_j \setminus \alpha_j]\!])u[\![\setminus \Pi_k]\!]^n t.$$

But then, $\models_s \epsilon_j \wedge D\vec{\epsilon}_k$ and $s[\![\alpha_j]\!]u$. By the definition of $\phi_{n+1}(\Pi_k)$ and the semantic clauses for $\wedge$ and $\rightarrow$, we find that $\models_s [\alpha_j]\phi_n(\Pi_k)$ and since $s[\![\alpha_j]\!]u$, $\models_u \phi_n(\Pi_k)$. By the induction hypothesis on n applied to u and t, we get $\models_t \phi$ as desired.

On the other hand, suppose $\not\models_s \phi_{n+1}(\Pi_k)$. Then for some j, $\models_s \epsilon_j \wedge D\vec{\epsilon}_k$ and $\not\models_s [\alpha_j]\phi_n(\Pi_k)$. It follows from the semantic clause for $[\alpha_j]$ that for some u, $s[\![\alpha_j]\!]u$ and $\not\models_u \phi_n(\Pi_k)$. By the induction hypothesis, there exists a t such that $u[\![\setminus \Pi_k]\!]^n t$ and $\not\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$ but $\not\models_t \phi$. Then, since $s(D\vec{\epsilon}_k \setminus [\![\epsilon_j \setminus \alpha_j]\!])u$, we have $s[\![\setminus \Pi_k]\!]^{n+1} t$, which establishes the Theorem. ■

4.2.5 **Corollary** *In any standard model M,* *[G 2.3.7]*

$$\models_s [\Pi_k]\phi \text{ iff } \forall n \in \mathbb{N},\ \models_s \phi_n(\Pi_k).$$

Proof:

Suppose $\models_s [\Pi_k]\phi$. For any n, if $s [\![\backslash \Pi_k]\!]^n t$ and $\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$, we have $s [\![\Pi_k]\!] t$ by the standard model condition, and so $\models_t \phi$. Hence, by the Theorem, $\models_s \phi_n(\Pi_k)$.

On the other hand, if $\not\models_s [\Pi_k]\phi$ then for some t, $s [\![\Pi_k]\!] t$ and $\not\models_t \phi$. Since the model is standard, there is some $n \in \mathbb{N}$ for which $s [\![\backslash \Pi_k]\!]^n t$ and $\models_t not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$. The Theorem then shows that $\not\models_s \phi_n(\Pi_k)$. ■

The result of the Corollary may be extended. We define a class of *admissible forms* whose members are strings of the form

$$\psi_1 \rightarrow [\beta_1](\psi_2 \rightarrow \ldots(\psi_m \rightarrow [\beta_m]\#)\ldots)$$

(in which not all the ψ_i's and β_j's need be present.) These forms are not really formulae because of the single occurrence of the symbol #. They become formulae when # is replaced by a member of *Fma*.

To make this precise, Goldblatt defines the class *Afm* of admissible forms as follows:

1. $\# \in Afm$
2. If $\Phi \in Afm$ and $\beta \in Cmd$ then $[\beta]\Phi \in Afm$
3. If $\Phi \in Afm$ and $\psi \in Fma$ then $(\psi \rightarrow \Phi) \in Afm$.

We denote by $\Phi(\phi)$ the formula obtained by substituting $\phi \in Fma$ for the occurrence of # in Φ.

We may regard an admissible form $\Phi(\Pi)$ as specifying initial conditions (the ψ_j's)

and intitialisation steps (the $[\beta_j]$'s) for the process Π. We must convince ourselves that the following theorem still holds now that the new Process modality is a possible substituend in an admissible form.

4.2.6 **Theorem** In any standard model, for any $\Phi \in Afm$, *[G 2.3.8]*

$$\models_s \Phi([\Pi_k]\phi) \text{ iff } \forall n \in \mathbb{N}, \models_s \Phi(\phi_n(\Pi_k)).$$

Proof:

By straightforward induction on the formation rules for the class *Afm*. The basis step is (4.2.5). ■

4.3. Proof Theory

In this section we define a logic Λ and characterise its deducibility relation $\vdash_\Lambda$. As in §§ 3.1, this relation is intended to provide a syntactic characterisation of validity.

The following list of axioms is from [14, §2.4].

Axioms

Tautologies

A1 $\phi \rightarrow (\psi \rightarrow \phi)$
A2 $(\phi \rightarrow (\psi \rightarrow \theta) \rightarrow ((\phi \rightarrow \psi) \rightarrow (\phi \rightarrow \theta))$
A3 $\neg\neg\phi \rightarrow \phi$

Termination

A4 $[\alpha](\phi \rightarrow \psi) \rightarrow ([\alpha]\phi \rightarrow [\alpha]\psi)$

Structured Commands

A5 $[\textbf{skip}]\phi \leftrightarrow \phi$
A6 $[\textbf{abort}]\mathit{false}$
A7 $[\alpha ; \beta]\phi \leftrightarrow [\alpha][\beta]\phi$

A8 $[\epsilon \Rightarrow \alpha,\beta]\phi \leftrightarrow ((\epsilon \rightarrow [\alpha]\phi) \wedge (not{-}\epsilon \rightarrow [\beta]\phi))$
A9 $not{-}\epsilon \rightarrow \neg[\epsilon\#\alpha]false$
A10 $[\epsilon\#\alpha]\phi \rightarrow (\epsilon \rightarrow [\alpha][\epsilon\#\alpha]\phi)$

Boolean Expressions

A11 $(\epsilon = \delta) \rightarrow (\partial \rightarrow \partial')$ where ∂' differs from ∂ only in having δ in one or more places where ∂ has ϵ.
A12 $true$
A13 $D\epsilon \leftrightarrow (\epsilon \vee not{-}\epsilon)$
A14 $\epsilon \rightarrow (\epsilon = true)$
A15 $\epsilon \supset \delta, \xi \leftrightarrow ((\epsilon \wedge \delta) \vee (not{-}\epsilon \wedge \xi))$
A16 $D(\epsilon \supset \delta, \xi) \leftrightarrow ((\epsilon \wedge D\delta) \vee (not{-}\epsilon \wedge D\xi))$
A17 $D(\epsilon = \delta) \leftrightarrow (D\epsilon \wedge D\delta)$

To Goldblatt's axioms we add two more to describe generalised iteration.

Processes

AP1 $[\Pi_k]\phi \rightarrow D\vec{\epsilon}_k \rightarrow \wedge_{i=1}^{k} (\epsilon_i \rightarrow [\alpha_i][\Pi_k]\phi)$
AP2 $not{-}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow (\phi \leftrightarrow [\Pi_k]\phi)$

It is worthwhile to verify that A10 is a special case of AP1, for processes having only one component.

Rules of Inference

Modus Ponens
MP: *From* $(\phi \rightarrow \psi)$ *and* ϕ, *infer* ψ.

Termination
TR: *From* ϕ *infer* $[\alpha]\phi$.

Omega-Iteration
OI: *From* $\Phi(\phi_n(\epsilon,\alpha))$ *for all* $n \in \mathbb{N}$, *infer* $\Phi([\epsilon\#\alpha]\phi)$.

One further rule is required by the presence of Processes in our language.

Generalised Omega-Iteration

GOI: *From* $\Phi(\phi_n(\Pi_k))$ *for all* $n \in \mathbb{N}$, *infer* $\Phi([\Pi_k]\phi)$.

All instances of axioms A1 through A17 are shown to be valid in standard models [14, §2] The validity of AP2 is proven by (4.2.3), while the result for AP1 is (4.2.2). That the rule GOI preserves validity is shown by (4.2.6).

We now define a logic to be a subset Λ of *Fma* which contains all instances of the axioms A1 through A17, AP1 and AP2, and which is closed under the inference rules MP, TR, OI and GOI.

As before, we write $\vdash_{\Lambda}\phi$ to mean that $\phi \in \Lambda$. The members of Λ are called the *theorems* of Λ.

By PC, abbreviating 'Propositional Calculus', we refer to the set which contains axiom instances A1, A2 and A3 and which is closed under modus ponens. PC contains all propositional tautologies. When a step in a theorem follows from the predicate calculus, we write "by PC" or some similar formulation.

The intersection of any set of logics is itself a logic. We call the smallest logic PL, for Programming Logic. In the remainder of this chapter, we prove that

$$\vdash_{PL}\phi \quad \textit{iff} \quad \models\phi$$

that is, that the PL-theorems are exactly those formulae valid in all standard models. This result is the completness theorem for the logic PL.

The next two theorems establish some necessary results about the deducibility of

formulae containing Process modalities. First, we demonstrate the deducibility of the analogue for Processes of schema A9.

4.3.1 **Theorem** $\vdash_{\Lambda} not{-}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow \neg[\Pi_k]false$

Proof:

By an AP2 instance and PC, $\vdash_{\Lambda} not{-}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow ([\Pi_k]false \rightarrow false)$. From the definition of negation, $\vdash_{\Lambda} not{-}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow \neg[\Pi_k]false$ ■

4.3.2 **Theorem** $\vdash_{\Lambda} \Phi([\Pi_k]\phi) \rightarrow \Phi(\phi_n(\Pi_k))$ *[G 2.4.3(5)]*

Proof:

We prove the result by induction on the complexity of members of *Afm*.

If the result holds for an admissible form Φ, then it holds for the form $[\alpha]\Phi$ by the rule TI (*G*2.4.1(1)), and for the form $\phi \rightarrow \Phi$ by PC. It is enough, then, to show the result for the admissible form # i.e.

$$\vdash_{\Lambda} [\Pi_k]\phi \rightarrow \phi_n(\Pi_k)$$

We proceed by induction on n.

From AP2 and PC we derive the basis step

$$\vdash_{\Lambda} [\Pi_k]\phi \rightarrow (not{-}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow \phi).$$

Assuming the result for n and applying TI we have, for each α_i,

$$\vdash_{\Lambda} [\alpha_i][\Pi_k]\phi \rightarrow [\alpha_i]\phi_n(\Pi_k)$$

Applying AP1 and PC,

$$\vdash_{\Lambda} [\Pi_k]\phi \rightarrow D\vec{\epsilon}_k \rightarrow \wedge^k_{i=1}(\epsilon_i \rightarrow [\alpha_i]\phi_n(\Pi_k))$$

which is the result for n+1. ■

4.3.1. Theories

A Λ-theory for a logic Λ is any subset Γ of *Fma* which contains Λ and which is closed under the rules MP, OI and GOI.

The intersection of any set of Λ-theories is itself a Λ-theory, and so there is a smallest Λ-theory, namely Λ.

The deducibility relation $\Sigma \vdash_{\Lambda} \phi$ is defined just as in §§ 3.1, except of course that the Λ theories in question are based on a different language and different sets of axioms and closure conditions.

The reader should verify that

1. $\{\phi : \Sigma \vdash_{\Lambda} \phi\}$ is a Λ-theory, for any $\Sigma \subseteq Fma$.
2. If M is a standard model, then

$$\Lambda_M = \{\phi : M \models \phi\}$$

   is a logic, and

3. If M is standard, then for each state s in M, $\{\phi : M, s \models \phi\}$ is a Λ_M-theory, where Λ_M is as defined above.

> The difference between a logic and one of its theories parallels the difference between the set of formulae valid in a model, and the set of formulae that hold at some particular state in that model [14, p. 67].

The preceding remark can be best understood by reference to the suggested exercises.

Below, we prove the Deduction Theorem for Λ, and the generalisation of the $\Box$-Deduction Theorem of TL. This generalisation shows that the result holds for all of the infinitely many modalities $[\alpha]$ and $[\Pi_k]$.

4.3.3 The Deduction **Theorem** *[G 2.4.7]*

$$\Sigma \cup \{\phi\} \vdash_{\Lambda} \psi \text{ iff } \Sigma \vdash_{\Lambda} \phi \rightarrow \psi .$$

Proof:

If $\Sigma \vdash_{\Lambda} (\phi \rightarrow \psi)$, then by (G2.4.4(2)), $\Sigma \cup \{\phi\} \vdash_{\Lambda} (\phi \rightarrow \psi)$. From (G2.4.4(1)), $\Sigma \cup \{\phi\} \vdash_{\Lambda} \phi$ and the fact that $\{\phi : \Sigma \vdash_{\Lambda} \phi\}$ is a Λ-theory and so MP-closed, $\Sigma \cup \{\phi\} \vdash_{\Lambda} \psi$.

Conversely, suppose $\Sigma \cup \{\phi\} \vdash_{\Lambda} \psi$. Let

$$\Gamma = \{\theta : \Sigma \vdash_{\Lambda} (\phi \rightarrow \theta)\} .$$

We need to show that $\psi \in \Gamma$. By hypothesis, it suffices then to show that Γ is a Λ-theory containing $\Sigma \cup \{\phi\}$. We need only show here that Γ is closed under the rule GOI. (For the remainder of the proof, see the cited theorem.) In order to see this, suppose that

$$\{\Phi(\chi_n(\Pi)) : n \in \mathbb{N}\} \subseteq \Gamma$$

for $\Phi \in Afm$, $\chi \in Fma$, $\Pi \in Proc$. Then

$$\Sigma \vdash_{\Lambda} \phi \rightarrow \Phi(\chi_n(\Pi)) \textit{ for all } n \in \mathbb{N}.$$

Applying GOI closure to the admissible form $(\phi \rightarrow \Phi)$, we find $\Sigma \vdash_{\Lambda} \phi \rightarrow \Phi([\Pi]\chi)$, and so $\Phi([\Pi]\chi) \in \Gamma$. ■

4.3.4 The α-Deduction **Lemma** *[G 2.4.9]*

$$\text{If } \Sigma \vdash_{\Lambda} \phi \text{ then } \{[\alpha]\psi : \psi \in \Sigma\} \vdash_{\Lambda} [\alpha]\phi .$$

Proof:

Let Γ be any Λ-theory containing $\{[\alpha]\psi : \psi \in \Sigma\}$. Put $\Delta = \{\psi : [\alpha]\psi \in \Gamma\}$. We need to show that $[\alpha]\phi \in \Gamma$, i.e. that $\phi \in \Delta$. By hypothesis, if Δ is a Λ-theory which contains Σ, then we are done. It is enough to show here that Δ is closed under the rule GOI.

But, if $\{\Phi(\chi_n(\Pi)) : n \in \mathbb{N}\} \subseteq \Delta$, , then $\{[\alpha]\Phi(\chi_n(\Pi)) : n \in \mathbb{N}\} \subseteq \Gamma$. Applying the GOI closure of Γ to the admissible form $[\alpha]\Phi$, we get $[\alpha]\Phi([\Pi]\chi) \in \Gamma$, and hence $\Phi([\Pi]\chi) \in \Delta$. This establishes the result. ■

4.4. Completeness

If M is a Λ-model, i.e. has

$$M \models \phi \quad \forall \phi \in \Lambda,$$

then each state s in M determines the set

$$\Gamma_s = \{\phi : M \models_s \phi\}$$

which is an extension of Λ closed under MP. If M is standard, then Γ_s will be closed under OI (G2.3.8) and GOI (4.2.6). It will also be consistent, since *false* does not hold at s. Since in general, either ϕ or $\neg\phi$ holds at s, Γ_s is a maximal consistent Λ-theory.

As before, the completeness proof is obtained by of a canonical model based on the set S_Λ of all maximal consistent Λ-theories.

Our first result verifies that, in characterising $\vdash_\Lambda$, we may confine our attention to maximal theories. That is, we show that each consistent set of formulae Σ has a maximal consistent extension.

4.4.1 **Theorem** $\Sigma \vdash_\Lambda \phi$ iff *for all* $\Gamma \in S_\Lambda$ *s.t.* $\Sigma \subseteq \Gamma$, $\phi \in \Gamma$ *[G 2.5.2(1)]*

Proof:

From left to right is immediate by the definition of $\vdash_\Lambda$.

For the converse, suppose that $\Sigma \nvdash_\Lambda \phi$. It is necessary to show that for some maximal consistent theory Γ, $\Sigma \subseteq \Gamma$ and yet $\phi \notin \Gamma$. We therefore construct such a Γ that contains $\Sigma \cup \{\neg\phi\}$.

Let $\phi_0, \phi_1, \phi_2, \ldots$ be an enumeration of *Fma*. The enumeration is used to define an increasing sequence $\Gamma_0 \subseteq \Gamma_1 \subseteq \ldots$ whose least upper bound Γ is a maximal consistent set in S_Λ which contains $\Sigma \cup \{\neg\phi\}$. Of course, this Γ is the one we need. The series is defined inductively

$$\Gamma_0 = \Sigma \cup \{\neg\phi\}$$

Γ_{n+1} is defined according to the following cases.

Case 1:
If $\Gamma_n \vdash_\Lambda \phi_n$, put $\Gamma_{n+1} = \Gamma_n \cup \{\phi_n\}$;
otherwise (i.e. $\Gamma_n \nvdash_\Lambda \phi_n$),

Case 2:
If ϕ_n is *not* an admissible form, put $\Gamma_{n+1} = \Gamma_n \cup \{\neg\phi_n\}$;

Case 3:
If ϕ_n is of the form $\Phi([\epsilon\#\alpha]\psi)$, let

$$\Gamma_{n+1} = \Gamma_n \cup \{\neg\Phi(\psi_j(\epsilon,\alpha)), \neg\phi_n\},$$

where j is the least number such that

$$\Gamma_n \nvdash_\Lambda \Phi(\psi_j(\epsilon,\alpha))$$

There remains a further possibility due to the introduction of generalised iterators, namely

Case 4:
If ϕ_n is of the form $\Phi([\Pi_k]\psi)$, put

$$\Gamma_{n+1} = \Gamma_n \cup \{\neg\Phi(\psi_j(\Pi_k)), \neg\phi_n\},$$

where j is chosen as in *Case 3*.

This completes the definition of Γ_{n+1}. We define $\Gamma = \cup\{\Gamma_n : n \in \mathbb{N}\}$. It remains to show that Γ is a maximal consistent Λ-theory.

Lemma 1. *For all* $n \in \mathbb{N}$, Γ_n *is* Λ*-consistent.*

Proof:

By induction on n. When $n = 0$ the result is (G2.4.8(2)), since $\Sigma \nvdash_\Lambda \phi$.

Assume that Γ_n is consistent. There are four cases for Γ_{n+1}. We need consider here only *Case 4*.

If $\Gamma_{n+1} = \Gamma_n \cup \{\neg\Phi(\psi_j(\Pi)), \neg\phi_n\}$ is not Λ-consistent, by (G2.4.8(2))

$$\Gamma_n \cup \{\neg\Phi(\psi_j(\Pi))\} \vdash_{\Lambda} \phi_n,$$

where ϕ_n is $\Phi([\Pi_k]\psi)$. But $\phi_n \rightarrow \Phi(\psi_j(\Pi))$ is a Λ-theorem by (4.3.2), so

$$\Gamma_n \cup \{\neg\Phi(\psi_j(\Pi))\} \vdash_{\Lambda} \Phi(\psi_j(\Pi)),$$

which makes $\Gamma_n \cup \{\neg\Phi(\psi_j(\Pi))\}$ inconsistent. By (G2.4.8(2)) again, $\Gamma_n \vdash_{\Lambda} \Phi(\psi_j(\Pi))$, which contradicts the choice of j in the definition of Γ_{n+1} ■

Lemma 2.

For any $\phi \in$ Fma, exactly one of ψ and $\neg\psi$ belongs to Γ.

Proof:

Any ψ is ϕ_n for some n. Either $\phi_n \in \Gamma_{n+1} \subseteq \Gamma$ (*Case 1*) or else $\neg\phi_n \in \Gamma_{n+1}$ (*Cases 2, 3 and 4.*) So at least one of $\psi, \neg\psi$ is in Γ. But if both are, Γ_m is inconsistent for some m, contradicting Lemma 1. ■

Lemma 3. *Γ is closed under* MP, OI *and* GOI.

Proof:

We show only that Γ is closed under GOI.

Suppose $\{\Phi(\psi_k(\Pi)) : k \in \mathbb{N}\} \subseteq \Gamma$. Let $\Phi([\Pi_k]\psi) = \phi_n$. Then if $\phi_n \notin \Gamma$, $\Gamma_n \nvdash_{\Lambda} \phi_n$, or else $\phi_n \in \Gamma_{n+1}$.

By *Case* 4 of the definition of Γ_{n+1}, $\neg\Phi(\psi_j(\Pi)) \in \Gamma$ for some j. From Lemma 2 it follows that $\Phi(\psi_j(\Pi)) \notin \Gamma$, contrary to hypothesis. ■

To complete the main result, if $\vdash_{\Lambda} \phi_n$ then, by (G2.4.4(3)), $\Gamma_n \vdash_{\Lambda} \phi_n$, so Γ contains Λ. By Lemma 3 then, Γ is a Λ-theory. By (G2.4.6(4)), Γ is consistent, or else *false* is a member of Γ. But then *false* $\in \Gamma_n$ for some n, contradicting Lemma 1. Maximality of Γ follows by Lemma 2. Since Γ contains Σ and does not contain ϕ, the theorem is established. ■

4.4.2 **Theorem** $\vdash_{\Lambda} \phi$ iff *for all* $\Gamma \in S_{\Lambda}, \phi \in \Gamma$. *[G 2.5.2(2)]*

Proof:

Put $\Sigma = \varnothing$ in (4.4.1). ■

4.4.1. The Canonical Model

The *canonical model* for a logic Λ is the structure

$$M_{\Lambda}=(S_{\Lambda},v,[\![\cdot]\!]),$$

based on the set S_{Λ} of maximal Λ-theories with

(i) for each $p \in Bvb$ and $\Gamma \in S_{\Lambda}$

a. $v_{\Gamma} = 1$ if $p \in \Gamma$

b. $v_{\Gamma} = 0$ if $not{-}p \in \Gamma$

c. $v_{\Gamma} = \omega$ if $\neg D\epsilon \in \Gamma$

(by (G2.5.1(6)), v_{Γ} is well defined)

(ii) the relation $[\![\alpha]\!]$ on S_{Λ} is defined inductively, by putting

$$\Gamma\, [\![\pi]\!]\, \Delta \text{ iff } \{\psi \in Fma : [\pi]\phi \in \Gamma\} \subseteq \Delta,$$

for programme letters π, and then for structured commands by the appropriate standard model conditions.

Thus M_{Λ} is a standard model by definition.

Fundamental Theorem for Λ: *For any* $\phi \in Fma$, *and any* $\Gamma \in S_{\Lambda}$

$$M_{\Lambda} \models_{\Gamma} \phi \text{ iff } \phi \in \Gamma$$

■

The rest of this chapter is devoted to proving the Fundamental Theorem. Before proving it, we show how it yields a solution to the completeness problem for Λ.

Completeness Theorem for Λ

$$\vdash_{\Lambda} \phi \text{ iff } M_{\Lambda} \models \phi$$

Proof:

By the Fundamental Theorem, "$M_{\Lambda} \models \phi$" is equivalent to "for all $\Gamma \in S_{\Lambda}, \phi \in \Gamma$". By (4.4.2), this is equivalent to $\vdash_{\Lambda} \phi$. ■

Completeness Theorem for PL

$$\vdash_{PL} \phi \quad \text{iff} \quad \phi \textit{ is valid in every standard model.}$$

Proof:

Recall that PL is the smallest (i.e the intersection) of all logics.

Now if M is a standard model, then $\{\phi : M \models \phi\}$ is a logic, and so contains PL. Thus all PL theorems are valid in M.

Conversely, if ϕ is valid in all standard models, then in particular, ϕ is valid in M_{PL}. Hence, by the previous result, $\vdash_{PL} \phi$. ■

Each stage in the hierarchical development of a logic of distributed algorithms repeats the following steps.

1. Add new formulae and programme constructing devices to L.
2. Introduce the 'necessary' new axioms and standard model conditions.
3. Extend the definition of the canonical model to account for the new standard model conditions.
4. Extend the Fundamental Theorem to cover the new sorts of formulae.

This strategy or 'completeness proving algorithm' is somewhat misleading. The 'necessary' axioms for each step are just those valid schema which are required to make the Fundamental Theorem work. The conditions on standardness are those

unearthed in the attempt to show that the new axioms are valid. To quote van Benthem [45]

> probably the most interesting, and certainly the most instructive way of discovering the fundamental calculus [for a logic] is by starting a Henkin proof empty-handed, so to speak, writing down necessary axioms and rules of inference along the way. (This heuristic use of proofs is described quite vividly in [23].) Historically however, the outcome was found in advance...

and that is how it is always presented.

We now proceed with the proof of the Fundamental Theorem.

4.4.3 The First α- **Lemma** *In* M_Λ, *for all* $\Gamma, \Delta \in S_\Lambda$, *[G 2.5.6]*

$$\Gamma [\![\alpha]\!] \Delta \text{ implies } \{\phi : [\alpha]\phi \in \Gamma\} \subseteq \Delta$$

Proof:

By induction on the formation of α. We prove the theorem only for the case where α is in *Proc*. What follows is essentially case (f) of the proof cited.

Assume the Lemma for $\alpha_i, i = 1,k$. To prove it for $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$ we require a preliminary result:

Sublemma: *For all* $\Gamma, \Delta \in S_\Lambda$,

$$\text{If } \Gamma [\![\setminus \Pi_k]\!]^n \Delta \text{ and } \models_\Delta not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \text{ , then}$$

$$\phi_n(\Pi_k) \in \Gamma \text{ implies } \phi \in \Delta.$$

Proof:

By induction on n

If $n = 0$, then $\Gamma = \Delta$ and so $\models_\Gamma not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$, whence, by (G2.5.5), $not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \in \Gamma$. Then, if $\phi_0(\Pi_k)$, i.e. $(not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow \phi)$, is in Γ, MP closure of Γ gives $\phi \in \Gamma$.

Assume the Sublemma for n. Then if $\Gamma [\![\setminus \Pi_k]\!]^{n+1} \Delta$ and $\models_\Delta not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$, for some Δ' and some j we have

$\Gamma\,(D\vec{\epsilon}_k \setminus [\![\epsilon_j \setminus \alpha_j]\!])\,\Delta'\,[\![\setminus \Pi_k]\!]^n\Delta$, by (4.2.1). But then $v_\Gamma = 1$ and $\Gamma\,[\![\alpha]\!]\,\Delta'$. Thus, if $\phi_{n+1}(\Pi_k) \in \Gamma$ i.e.

$$D\vec{\epsilon}_k \rightarrow \wedge^k_{i=1}\,(\epsilon_i \rightarrow [\alpha_i]\phi_n(\Pi_k)) \quad \in \Gamma$$

the fact that $\epsilon_j \in \Gamma$ and $D\vec{\epsilon}_k \in \Gamma$ (G2.5.5) guarantees that $[\alpha_j]\phi_n(\Pi_k) \in \Gamma$. By the main hypothesis on α_j, $\phi_n(\Pi_k) \in \Delta'$. From this, by the induction hypothesis on n, $\phi \in \Delta$. This establishes the result for $n+1$ and so completes the proof of the Sublemma. ■

Returning to the main result for $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$, if $\Gamma\,[\![\Pi_k]\!]\,\Delta$ in S_Λ then, for some n, $\Gamma\,[\![\setminus \Pi_k]\!]^n\Delta$ and $\models_\Delta$ *not*$-(\epsilon_1 \vee \ldots \vee \epsilon_k)$. But if $[\Pi_k]\phi \in \Gamma$, we have $\phi_n(\Pi_k) \in \Gamma$ by (4.3.2) and so $\phi \in \Delta$ by the Sublemma.

This completes the inductive case for Processes, and thus the necessary extension of the First α-Lemma. ■

4.4.4 The Second α- **Lemma** *For all formulae* ϕ *[G 2.5.7]*

$$\text{if } M_\Lambda \models_\Gamma \phi \text{ implies } \phi \in \Gamma \text{ , } \textit{for all } \Gamma \in S_\Lambda,$$

$$\text{then } M_\Lambda \models_\Gamma [\alpha]\phi \text{ implies } [\alpha]\phi \in \Gamma \text{ , } \textit{for all } \Gamma \in S_\Lambda.$$

Proof:

We proceed inductively.

As with the First α-Lemma we need only prove the result for α a Process, extending the result cited.

Assume the result for $\alpha_i, i=1,k$ To prove it for $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$ we again need a preliminary result.

Sublemma:

For all $\Gamma \in S_\Lambda$,

$$M_\Lambda \models_\Gamma \phi_n(\Pi_k) \text{ implies } \phi_n(\Pi_k) \in \Gamma.$$

Proof:

Let $\phi_0(\Pi_k)$, i.e. $(not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \rightarrow \phi)$ hold at Γ. Then, if $not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k) \in \Gamma$, $\models_\Gamma not\text{–}(\epsilon_1 \vee \ldots \vee \epsilon_k)$ and so $\models_\Gamma \phi$ by the semantic clause for $\rightarrow$. By the main hypothesis on ϕ, it follows that $\phi \in \Gamma$.

Assume the result for n, and let $\models_\Gamma \phi_{n+1}(\Pi_k)$, i.e.

$$\models_\Gamma D\vec{\epsilon}_k \rightarrow \wedge^k_{i=1}(\epsilon_i \rightarrow [\alpha_i]\phi_n(\Pi_k)).$$

If $\vdash_\Lambda D\vec{\epsilon}_k$ then $\models_\Gamma D\vec{\epsilon}_k$ and so for each i, $\models_\Gamma (\epsilon_i \rightarrow [\alpha_i]\phi_n(\Pi_k))$. So, if $\epsilon_i \in \Gamma$, applying $(G2.5.5)$ again yields that $\models_\Gamma \epsilon_i$ and so $\models_\Gamma [\alpha_i]\phi_n(\Pi_k)$. By the hypothesis on n for the Sublemma and the main hypothesis on α_i, $[\alpha_i]\phi_n(\Pi_k) \in \Gamma$, for each i. Then $(G2.4.6(5))$ establishes the result for $n+1$. ■

To prove the main result, suppose $\models_\Gamma [\Pi_k]\phi$. By (4.2.6), $\models_\Gamma \phi_n(\Pi_k)$, for each $n \in \mathbb{N}$. By the Sublemma, $\phi_n(\Pi_k) \in \Gamma$. Since Γ is closed under GOI, $[\Pi_k]\phi \in \Gamma$. This concludes the inductive case for $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$ and so completes the extension of the Second α-Lemma. ■

We now finally prove the Fundamental Theorem, i.e.

$$M_\Lambda \models_\Gamma \phi \quad \text{iff} \quad \phi \in \Gamma$$

We proceed inductively:

1. For $\phi \in Bxp$ the result is $(G2.5.5)$.

2. Assume the result holds for ϕ and ψ. The semantic clause for $\rightarrow$ and the fact $(G2.5.1(3))$ that

$$(\phi \rightarrow \psi) \in \Gamma \text{ iff } (\phi \in \Gamma \text{ implies } \psi \in \Gamma)$$

allow us to conclude that the theorem holds for the formula $(\phi \rightarrow \psi)$.

3. Assume the result for ϕ, and let $\alpha \in Pgm$. Then if $\models_{\Gamma} [\alpha]\phi$, the Second α-Lemma gives $[\alpha]\phi \in \Gamma$. Conversely, let $[\alpha]\phi \in \Gamma$. Then, if $\Gamma [\![\alpha]\!] \Delta$, we get $\phi \in \Delta$ by the First α-Lemma, and so $\models_{\Delta} \phi$ by the assumption on ϕ. This shows, by the semantic clause for $[\alpha]$, that $\models_{\Gamma} [\alpha]\phi$, and so establishes the result for the formula $[\alpha]\phi$.

This completes the proof of the Fundamental Theorem, and so our extension of Goldblatt's I/O logic to Processes.

Chapter 5

Logic of Dynamic Behaviour

In the previous chapter we presented the foundations of an existing Programming Logic PL, augmented by a new syntactic construction $[\Pi_K]$, representing Parallel Event Servers. The semantics for PL is fundamentally indistinguishable from the semantics for the Dynamic Logic (DL) of Pratt [35] and the many variants thereof. All are founded on a generalised Kripkean relational semantics, in which each programme is assigned a binary relation.

Henceforth we employ the symbol $L_{I/O}$ to refer to the language defined in § 4, and L_K to denote the language of §§ 3.1. The locution "ϕ is an $L_{I/O}$ (L_K) formula" means that ϕ belongs to the set *Fma* defined in §§ 4.1 (§§ 3.1).

The langauge and semantics so far developed is inadaquate for the purposes of this thesis. Consider, for instance, the two programmes

$[\epsilon, true \# \alpha, \mathbf{abort}]$ and $[\epsilon \# \alpha]$

It turns out that, in standard $L_{I/O}$ models, they are both assigned identical binary relations. It follows from this that any formula ϕ of $L_{I/O}$, or *any* other language, is true (in standard models) of the one programme if and only if it is true of the other. The non-deterministic nature of the generalised iterator effectively disguises the fact that abortion may occur. Goldblatt observes that "our semantics is not finally adequate to describe non-determinism."

It is also unsuited to interpreting assertions about the progressive behaviour of programmes, such as

sometime during α, ϕ is true.

We maintain that this difficulty is not inherent in relational semantics. It is the nature of the binary relations actually employed which places inherent limitations on the expressive power of any language. In PL, $s[\![\alpha]\!]t$ means that a computation of α can end in state t if started in state s. It is not easy to say in PL or DL that α passes through an *intermediate* state u.

5.0.1. Comparison to Previous Work

The best developed system for the problem under study appears to be the Process Logic of Harel *et.al.* [16]. The word 'Process' in the name does not refer to Processes in our sense. Process Logic is built upon Propositional Dynamic Logic (PDL), as in [9]. The key semantical feature is that programmes are characterised by a set of paths. A path is a linearly ordered subset of the set of states in a model. Formulae in the logic are interpreted relatively to paths. The important meta-logical notions are those of the first and last states of a path, and a *suffix of* relation between paths. A valid theorem from PL of the form $[\alpha]\phi$ would be interpreted as follows in Process Logic:

$$\models[\alpha]\phi \text{ iff, for every } \alpha \text{ path } p, \models_p \mathbf{fin} \rightarrow \mathbf{last}\phi$$

where **fin** is true of finite paths, and **last**ϕ asserts that ϕ is true of a path p's final state.

Process Logic is very much more powerful than the logic we develop, since it contains the theory of well-founded linear orderings. Its only drawback is that, in appearance, it has little to do with programming. It was designed so that decidability

could be proved by reduction to *SnS* [37]. The axioms used in the proof of completeness are distant abstractions from the properties of actual computer programmes. This abstraction appears to arise from the fact that paths are taken to be primitive objects.

A less general system was developed by Segerberg [42]. His logic incorporates a version of PDL. In addition there are two dual operators of the form *throughout* and *during*, as introduced by Pratt [35]. These operators are interpreted relatively to paths. A model for Segerberg's logic includes both path and input/output relations for programmes. The difficulty with this system is that the expressive power of its language is very limited. It is not possible to encode liveness or preservation properties in it.

It was observed by Emerson [7] that there is a connection between the paths employed in the semantics of Process Logics and the binary relations employed in the various Dynamic Logics. Emerson identified conditions under which the two are inter-definable. This suggests the possibility of avoiding the explicit use of paths as semantic objects.

In the present chapter we develop a semantics which is capable of supporting all of the durational assertions, such as liveness and fairness which were mentioned in the Introduction. Concurrently, an extension of PL is defined and shown to be sound and complete with respect to the new semantics.

The role of paths is filled by meta-logical objects which, inspired by Harel [15], we dub *computation-trees*. A generalisation of the modal system K4 of §§ 3.1 is then introduced, to allow for the interpretation of **abort**. A new modality is then defined,

which allows formulae in an extension of L_K to be evaluated relatively to a model of the computation tree of a particular programme. The syntax is similar in flavour to that of Nishimura [27]. The semantics is new.

5.1. (Meta-) Logical Preliminaries

5.1.1. Computation Trees

We write $ct^M(\alpha,s)$ to denote the *computation tree* of programme α as started at state s in model M. $ct^M(\alpha,s)$ is a binary relation on the states S of M. It represents a pre-ordering of the states which may be traversed by an execution of α which starts at state s. If $<t,u> \in ct^M(\alpha,s)$ it means that α can pass through state t *before* state u.

We write $Q^M(\alpha,s)$ to denote a subset of the states in $ct^M(\alpha,s)$. We call members of $Q^M(\alpha,s)$ *queer* points. If $t \in Q^M(\alpha,s)$ it means that some α-computation started at s in model M aborts at state t. The notion of queer points is central to the semantics of programme failure motivated in the following pages and formally developed in §§ 5.1.4.

The purpose of defining these structures is to provide a class of binary relations with which to augment the models of § 4. These augmented models will then be capable of interpreting a more powerful language. In particular, to the language $L_{I/O}$ we will later add a class of formulae similar to that defined in §§ 3.1. For the remainder of this section we will freely use modal formulae such as those of L_K. They should be interpreted as follows. Let $M=(S,v,[\![\cdot]\!])$ be a standard $L_{I/O}$ model. To say that a formulae ϕ in the language L_K is true of a computation tree $ct^M(\alpha,s)$ means that ϕ is true at state s in the L_K model $N=(S,v,<)$, where

$t < u$ iff $<t,u> \in ct^M(\alpha,s)$. Formally, this is of course not quite meaningful, as the elementary propositions in the two languages are not the same. We are interested just now only in explaining what computation trees are, and showing that a simple language like L_K is expressive enough to represent certain important properties of these structures.

We now present the formal definition of a computation tree. and of the queer points within it. The symbols s,t,u refer to states in a standard $L_{I/O}$-model M. v and $[\![\ ]\!]$ are M's valuation function and relational operator.

We require two preliminary ideas.

5.1.1 **Definition** Generated Submodels

Let $M = (S,v,[\![\ ]\!]$ be an $L_{I/O}$ model. $M(s) = (S',v',[\![\ ]\!]')$ is an s-generated submodel of M iff S' is the least subset of S such that

(i) $s \in S'$

(ii) $\forall t \in S'$, if $t[\![\alpha]\!]u$ in M for some programme α then $u \in S'$.

while v' and $[\![\ ]\!]'$ are the restrictions of v and $[\![\ ]\!]$ to members of S'. ■

Intuitively, $M(s)$ is that subset of M containing s and closed under reachability by programme execution from s.

Let S be a set of states with a distinguished subset Q of queer points, and R be a binary relation on a set S. In the manner of Fitting [10] we define the relational property of Transitivity based on *normal* states (*i.e* states which are not queer). We also define a restricted transitive closure operator τ.

TRANS*: $\forall s,t,u \in S$, if s,t are not queer and sRt and tRu, then sRu.

Define R^τ to be the TRANS*-closure of the relation R. A modal axiom characteristic of this weakened transitivity is introduced in §§ 5.1.4.

When reading the following definitions, it may help to refer to the inductive definition of the $[\![\ ]\!]$ operator in §§ 4.2.3. Think of the definition of $ct^M(\alpha,s)$ as the 'unwinding' of the definition of $[\![\alpha]\!]$. For instance, $[\![\alpha;\beta]\!]$ was defined as the set of all pairs $<s,t>$ such that for some u, $s[\![\alpha]\!]u$ and $u[\![\beta]\!]t$. In the definition of the computation tree, all the intermediate points such as u are preserved. Recall that the symbol Π_k in clause (6) denotes the process $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$, with k components.

5.1.2 **Definition** Computation Trees $ct^M(\alpha,s)$.

The computation tree $ct^M(\alpha,s)$ is defined by induction on the formation of α:

1. $ct^M(\mathbf{skip},s) = \varnothing$
2. $ct^M(\mathbf{abort},s) = \{<s,t> : t \in M(s)\}$
3. $ct^M(\pi,s) = <s,s> \cup \{<s,t> : s[\![\pi]\!]t\}$, for π an elementary programme letter.
4. $ct^M((\alpha;\beta),s) = (ct^M(\alpha,s) \cup_{t:s[\![\alpha]\!]t} ct^M(\beta,t))^\tau$.
5. $ct^M((\epsilon \Rightarrow \alpha,\beta),s) =$ if $v_s(\epsilon) = \omega$ then $ct^M(\mathbf{abort},s)$, otherwise,

$$\begin{cases} ct^M(\alpha,s) & \text{if } v_s(\epsilon) = 1 \\ ct^M(\beta,s) & \text{is } v_s(\epsilon) = 0 \end{cases}$$

6. $ct^M((\epsilon \# \alpha),s) =$

$$\left(\cup_{i<\omega} \left(\{ct^M((\epsilon \Rightarrow \alpha, \mathbf{skip}),t) : s[\![\epsilon \setminus \alpha]\!]^i t\} \right) \right)^\tau$$

7. $ct^M(\Pi_k,s) =$

$$\left(\cup_{i<\omega} \left(\cup_{1 \leqslant j \leqslant k} \{ct^M((\epsilon_j \Rightarrow \alpha_j, \mathbf{skip}),t) : s[\![\setminus \Pi_k]\!]^i t\} \right) \right)^\tau$$

■

5.1.3 **Definition** Queer Points $Q^M(\alpha,s)$.

The queer points $Q^M(\alpha,s)$ in a computation tree $ct^M(\alpha,s)$ are defined inductively on the structure of α:

1. $Q^M(\mathbf{skip},s) = \varnothing$

2. $Q^M(\mathbf{abort},s) = \{s\}$

3. $Q^M(\pi,s) = \{s\}$ if $[\![\pi]\!] = \varnothing$ and is empty otherwise

4. $Q^M(\,(\alpha;\beta)\,,s) = (Q^M(\alpha,s) \;\cup_{t:s[\![\alpha]\!]t} Q^M(\beta,t)\,)^\tau$

5. $Q^M((\epsilon \Rightarrow \alpha,\beta),s)$ = if $v_s(\epsilon) = \omega$ then $Q^M(\mathbf{abort},s)$, otherwise,

$$\begin{cases} Q^M(\alpha,s) & \text{if } v_s(\epsilon) = 1 \\ Q^M(\beta,s) & \text{is } v_s(\epsilon) = 0 \end{cases}$$

6. $Q^M((\epsilon \# \alpha),s)$ =

$$(\; \cup_{i<\omega} (\,\{Q^M(\,(\epsilon \Rightarrow \alpha\,,\mathbf{skip})\,,t) : s\,[\![\epsilon \setminus \alpha]\!]^i\, t\}\,)\,)^\tau$$

7. $Q^M(\Pi_k,s)$ =

$$(\; \cup_{i<\omega} (\; \cup_{1\le j\le k} \{Q^M(\,(\epsilon_j \Rightarrow \alpha_j\,,\mathbf{skip})\,,t) : s\,[\![\setminus \Pi_k]\!]^i\, t\}\,)\;)^\tau$$

■

Since the definition of $ct^M(\alpha,s)$ employs the τ operator, which in turn makes use of $Q^M(\alpha,s)$, the 'definitions' above should proceed instead by simultaneous induction on α. Yet again, rigour is sacrificed for the sake of exposition.

We usually write "t is *queer* in $ct^M(\alpha,s)$" to mean $t \in ct^M(\alpha,s)$.

To make the relationship between computation trees and paths more explicit, we introduce

5.1.4 **Definition** σ is an α-computation starting at s if

$$\sigma = \{s_i : 0 \leqslant i \leqslant \zeta\}, \zeta \leqslant \omega, \text{ such that}$$

$s_0 = s$ and
σ is a maximal linearly ordered subset of $ct^M(\alpha,s)$.

■

We say that σ is a terminating computation if ζ is finite and $<s_\zeta,s_\zeta> \notin ct^M(\alpha,s)$. It is now clear what it means to say that α always, never, or sometimes terminates.

A point t is an *endpoint* if $<t,u> \in ct^M(\alpha,s)$ implies $t = u$. t is a *final* or *terminal* state if it is an endpoint which is not related to itself in $ct^M(\alpha,s)$. t is an *internal* state if $<t,u> \in ct^M(\alpha,s)$ with $t \neq u$.

As an example, $ct^M((true\#\mathbf{skip}),s) = \{<s,s>\}$. The only computation σ for this tree is the series $\sigma = \{s\}$. σ is a finite but non-terminating computation. A programme like $(true \# x := x+1)$ possesses an infinite non-terminating computation.

We previously remarked that the semantics of § 4 fails to distinguish a computation which sometimes aborts from one which never does (see the remark on page 73.) In order explain the reasoning behind the definitions of computation trees and queer points, and especially the definition of $ct^M(\mathbf{abort},s)$, we must consider more closely what we intend by the notion of an aborting computation. An appropriate semantics for **abort** is constrained by the distinctions we wish to make between certain general types of programme behaviours. We claim that the following is a minimal list of distinguishable behaviours.

1. The null programme **skip**.
2. Programmes which *sometimes* terminate *e.g.* $(\epsilon \Rightarrow \alpha, (true \,\#\, \mathbf{skip}))$
3. Programmes which *never* terminate *e.g.* $(true \,\#\, \mathbf{skip})$.
4. Programmes which *always* terminate.
5. The aborting programme **abort** itself.
6. Programmes which *sometimes* abort, but which may properly execute some steps first *e.g.* $(true, true \,\#\, \pi, \mathbf{abort})$
7. Programmes which *never* abort.
8. Programmes which *always* abort, but which may properly execute some steps first *e.g.* $(\epsilon \Rightarrow (\alpha\,;\mathbf{abort}), \mathbf{abort})$

We first show how the language L_K can deal with the various sorts of non-aborting computations.

5.1.2. Semantics of Termination and Divergence

The elementary programme **skip** has the simplest computation tree, namely the empty set. The formula $\Box false$ is true of $ct^M(\mathbf{skip},s)$, since $\Box false$ is true at a state s if and only if s has no successors.

To say that α *sometimes* terminates is to say that it has a terminating computation. A terminating computation for α is one which has a final state. Then either s is final, or some later state t is. Suppose that s is not final, i.e. that $\Diamond true$ holds at s. Assuming for the moment that $ct^M(\alpha,s)$ is a transitive relation, that means that for some pair $<s,t> \in ct^M(\alpha,s)$, there is no u such that $<t,u> \in ct^M(\alpha,s)$. Then $\Box false$ is true at t, and so $(\Diamond true \rightarrow \Diamond\Box false)$ is true at s. The analyses of the $L_{I/O}$ formula $\neg[\alpha]false$ given in §§ 4.2 shows that the two formulae are equivalent: both

assert the existence of at least one final state, and thus at least one terminating computation of α. Note that $(\Diamond true \rightarrow \Diamond\Box false)$ holds in $ct^M(\mathbf{skip},s)$ because the antecedent $\Diamond true$ fails.

We next show how a L_K formula can assert that a programme α *always* terminates, something that we have shown can not be said by any $L_{I/O}$ formula. To say that α always terminates when started at s means that every state in $ct^M(\alpha,s)$ is either a final state, or has a successor which is a final state. The appropriate L_K formula for this condition is $\Box(\Box false \vee \Diamond\Box false)$ or, equivalently $\Box(\Diamond true \rightarrow \Diamond\Box false)$.

Next consider a computation which never terminates. That is, s is not a final state in $ct^M(\alpha,s)$, and no successor t is final either. It is easy to see that the appropriate L_K formula for this condition is $(\Diamond true \wedge \Box\Diamond true)$, again assuming transitivity of $ct^M(\alpha,s)$.

Consider the relationships between the preceding formulae. The notions *never terminates* and *sometimes terminates* are dual. That is, it is not the case that α never terminates if and only if α sometimes terminates. Recalling the duality between $\Box$ and $\Diamond$ in L_K, the formula

$$(\Diamond true \wedge \Box\Diamond true) \leftrightarrow \neg(\Diamond true \rightarrow \Diamond\Box false)$$

is just a substitution instance of a tautology.

Equally, if α *always* terminates, we ought to be able to deduce that it terminates *sometime*. In other words, the formula

$$\Box(\Diamond true \rightarrow \Diamond\Box false) \rightarrow (\Diamond true \rightarrow \Diamond\Box false)$$

should be valid. If computation trees were reflexive relations it would be valid,

since it is an instance of the axiom schema T: $(\Box p \rightarrow p)$. It is easy to show that this schema is valid in exactly the class of reflexive models [20]. Inspection of the definition of computation trees shows that they are not reflexive relations. In fact, they are defined such that exactly the terminal points are irreflexive. It is clear from the definitions that computation trees satisfy the following property, which we dub internal reflexivity.

IREF: $\forall x\,(\exists y\,(x < y) \rightarrow x < x)$

This condition says that if a point is related to anything, then it is related to itself. Endpoints are allowed to be either reflexive, or irreflexive. Irreflexive endpoints are what we have called terminal points. As we prove below, the appropriate modal axiom schema for models satisfying IREF is

T*: $\Diamond\, true \rightarrow (\Box p \rightarrow p)$

In any model validating this principle,

$$\Box(\Diamond\, true \rightarrow \Diamond\Box false) \rightarrow (\Diamond\, true \rightarrow \Diamond\Box false)$$

is true at internal points and reflexive endpoints. At any irreflexive endpoint t, $\Box false$ holds. Since, in general $(\Box false \rightarrow \Box\phi)$, $\Box(\Diamond\, true \rightarrow \Diamond\Box false)$ holds at t. Since $\Box false$ is true, $\Diamond\, true$ is false at t and so $(\Diamond\, true \rightarrow \Diamond\Box false)$ holds at t. This reasoning has established that if a programme inevitably terminates, then it terminates sometime.

We can use the formulae developed above to compare the behaviours of certain programmes. For instance, consider **skip** and (*true*#**skip**) A glance at the definitions shows that the computation tree of the former is the empty set, while that of the latter is a single reflexive point. It is easy to see that **skip** always terminates. On the other hand, $\Diamond\, true$ holds at reflexive points, and since $<s,s>$ is the only member of $ct^M(\mathbf{skip},s)$ $\Box\Diamond\, true$ holds at s. It follows that (*true*#**skip**) never terminates.

This illustration explains why the pair $<s,s>$ is included in clauses (2), (5) and (6) of (5.1.2), and why the programme **skip** is assigned the empty computation tree in clause (1). It would be handier, and more elegant, if we could postulate a state "just like, but not equal to s" for **skip** to go to. The machinery to specify such a property is not readily available: we wish to define computation trees only in terms of concepts and semantic objects already available from § 4.

Since $[\![\mathbf{skip}]\!]$ was defined as the identity relation in standard $L_{I/O}$ models, a computation for **skip** starting at s may at most proceed from s back to s. But then, it would be impossible to say in L_K that **skip** terminated. No choice remains but to make $ct^M(\mathbf{skip},s) = \varnothing$.

An inspection of the standard model conditions for iterative programmes reveals that $[\![true\#\mathbf{skip}]\!] = \varnothing$. Thus, in the semantics of § 4, (*true*#**skip**) is equivalent to **abort**. The cost of distinguishing the two in L_K would be too high if we were forced to claim that (*true*#**skip**) was just the same as **skip**. By adding $<s,s>$ to the relevant clauses it becomes possible to tell the two apart in L_K.

Having accounted for clauses (5) and (6), consider clause (2). The computation tree $ct^M(\pi,s)$ for an elementary programme π is defined so that the reflexive point s is related to one or more irreflexive endpoints, as given by the relation $[\![\pi]\!]$. The intended interpretation of a formula of the form $\Box\phi$ is that it is true at s if and only if ϕ is true at all related points. If s were not reflexive in $ct^M(\pi,s)$ then $\Box\phi$ would be true of state s in the tree if and only if $M \models_s [\pi]\phi$ (assuming that π always terminates.) We intend that $\Box\phi$ be true of $ct^M(\alpha,s)$ if and only if ϕ is true of every point in the computation, including the start. Hence the definition of clause (2) requires that $<s,s> \in ct^M(\pi,s)$.

To summarise the results so far, the following modal formulae correspond to programme behaviours (1) through (4) above

1. $\Box false$ for $ct^M(\mathbf{skip},s)$.
2. $(\Diamond true \rightarrow \Diamond\Box false)$ for $ct^M(\alpha,s)$, if α sometimes terminates.
3. $(\Diamond true \wedge \Box\Diamond true)$ for $ct^M(\alpha,s)$, if α never terminates.
4. $\Box(\Diamond true \rightarrow \Diamond\Box false)$ for $ct^M(\alpha,s)$, if α always terminates.

5.1.3. Semantics of Aborting Computations

In this section we present arguments to justify the formal semantics for programme failure already implicit in the definition of $ct^M(\mathbf{abort},s)$.

In dissecting non-termination we found that the modal formula $\Box false$ characterised the simplest terminating programme. We were then able to construct more complex formulae which could distinguish programmes which terminate always from those which terminate sometimes, or never. These formulae relate the idea of programme termination to the existence of final states in a computation tree. If we could identify a property characteristic of the aborting computation and find an L_K formula corresponding to that property, presumably our problem would be solved.

Certainly **abort** can not be the same as any terminating computation. Equally, the mere lack of a final state should not cause us to insist that a programme aborts. Reviewing Definition (5.1.2), we see that a computation of α aborts if and only if it contains **abort** as a sub-programme, or else attempts the evaluation of an undefined boolean expression. There are perfectly good infinite computations to which none of these dreadful conditions applies: **abort** must be distinguishable from all of them.

It may help at this point to review some of what has been said about aborting or *failure* in the literature. Dijkstra [6] considers that

> When invoked, the mechanism named "abort" will fail to reach a final state: it's attempted activation is interpreted as a symptom of failure.

It was this interpretation that led to the definition $[\![\mathbf{abort}]\!] = \varnothing$. We have already explained that merely failing to reach a final state is at least an incomplete characterisation of **abort**. We are left with the notion of *failure* and symptoms of failure.

It is possible to leave the term failure undefined, and cast about instead for a formula which just says that failure occurs. This is the approach taken by Harel [15]. A set of predicates $fail_\alpha$ are introduced. An inductive definition is then given in terms of the structure of α. From the remarks above, we can see how such a definition would go in our language. $fail_{\mathbf{abort}}$ would be true by definition. The predicates for **skip** and elementary programmes π would be false. Then for compound α, the definitions would proceed analogously to the following illustration:

$$fail_{(\epsilon \Rightarrow \alpha,\beta)} \leftrightarrow (\neg D\epsilon \vee (\epsilon \wedge fail_\alpha) \vee (not\text{-}\epsilon \wedge fail_\beta)) .$$

Harel is also able to define *looping* (which we call non-termination) by similar means.

Segerberg in [42] alludes to possible generalisations of Kripke models which are capable of giving some meaning to failure. Each programme is assigned a set of states $F(\alpha)$ from which α has a failing computation. Then the predicate $fail_\alpha$ is defined to be true at a state s if and only if $s \in F(\alpha)$. The models are otherwise similar to our own.

The concept of failure is not explicitly addressed in the literature on Process Logics.

The solution we ultimately adopt is similar in flavour to Segerberg's. We are, however, able to found ours upon at least a sketchy analysis of what failure actually means.

There are two common models of failure in distributed networks. Both model the failure of individual network processes. We do not consider here what 'failure' of an entire network should mean.

Fail-Stop: α reaches an aborting state, and *can not* continue execution. Physically, the operating system recognises that an **abort** has taken place, perhaps by means of a signal from the hardware. The aborting programme is then *prevented* from continuing, either by fiat of the operating system, or an actual "crash" of the supporting hardware. Depending on which agent intervenes, other programmes may be capable of executing correctly. We use the term 'prevented' because if α were allowed to continue execution, the process might exhibit

Byzantine-Failure: a term connoting treachery. α exhibits Byzantine failure if it continues to execute after an **abort** in such a way that its behaviour is independent of its inputs. That is, after failure, programmes no longer behave in the way that the axioms say they should.

We adopt the Fail-Stop model, with the assumption that an abort by α in state s affects only α. Any other programme may begin executing from s without being affected. We choose this model only because the solution is easier. The technique we employ could be extended to account for Byzantine Failure, or for both failure models together.

Intuitively, a programme exhibiting fail-stop failure has been prevented from continuing to execute. If it *had* continued, it could have traversed a large portion of the state space. To see how large, we again appeal to the behaviour of real machines. Presumably there is a limit to the non-determinism of a programme exhibiting Byzantine failure. This limit is defined as follows.

$$<u,v> \in ct^M(\mathbf{abort},s) \text{ implies } \exists \alpha \in Pgm\,(u\,[\![\alpha]\!]\,v)$$

What this expression means is roughly this: an aborting computation can effect a transformation between two states if and only if there exists some terminating programme α which is capable of effecting the same transformation. That is, an aborting computation started at s can reach at most states within the generated submodel $M(s)$. This restriction rules out certain kinds of infinite behaviours. For instance, a programme exhibiting Byzantine Failure is not capable of changing the value of more than finitely many variables.

We may now determine which L_K formulae have to be true at state s in an L_K model corresponding to $ct^M(\mathbf{abort},s)$. It is obvious that $\Box \mathit{false}$ can not be true at s, which means that $\Diamond \mathit{true}$ must hold. But no successor of s can be a final state either,or else **abort** would have a terminating computation, so it seems that $\Box\Diamond \mathit{true}$ must hold at s. Now the logic K has the inference rule RN, namely

$$\text{from } \vdash_{\Lambda} \phi \text{ infer } \vdash_{\Lambda} \Box\phi$$

In the presence of this rule, $\Box\phi$ must hold at s for every theorem ϕ of the underlying logic. It is tempting to characterise s by saying that *no* formulae of the form $\Box\phi$ hold at s unless required to by RN. Modal logics exist which roughly correspond to this notion. It would be possible to introduce yet another modality (call it *) and interpret it in the universal relation on the model for $ct^M(\mathbf{abort},s)$. Assume this * were given the axioms for S5 (see, for instance [2]). Then $*\phi$ is true at any state s if and only if it is true at all states in the model. In that case, the infinite conjunction over all formulae ϕ of $\Box\phi \leftrightarrow *\phi$ could be taken to represent an aborting state. There is at least one problem with this idea. This infinite conjunction would place our programming logic in the nether realm of infinitary logics which allow formulae to contain an infinite number of terms, and an infinite number of

variables. Although first order languages (called L_{ω_1,ω_1}-languages) of this sort have been studied, developing a modal extension of such powerful languages is beyond the scope of this thesis. This approach also requires that, for every non-terminating computation, there exist some non-theorem ϕ such that $\Box\phi$ holds for the computation.

Since there is no obvious (finite) set of $\Box$-formulae which provides a convenient way of picking out aborting states, perhaps aborting states should make all of them false. In order to do this the logic of §§ 3.1 will have to be modified to remove the inference rule RN. Modal logics have been studied which contain no theorems of the form $\Box\phi$. We present such a *Non-Normal* logic suitable for our purposes in the next section.

We can now provide a simple formula characteristic of aborting states. For, if every formula of the form $\Box\phi$ is false at an aborting state, then in particular $\Box$*true* is false, and so $\Diamond$*false* must be true.

Consider a programme α which sometimes aborts, say at state t. Then $\Diamond$*false* is true at t in $ct^M(\alpha,s)$. It follows that $\Diamond\Diamond$*false* must hold at s. Now if computation trees are transitive, they validate the axiom schema $(\Diamond\Diamond\phi \rightarrow \Diamond\phi)$, which is merely the dual of the axiom schema 4 $(\Box\phi \rightarrow \Box\Box\phi)$. In that case, a programme which sometimes aborts is indistinguishable in L_K from the aborting programme itself. This consideration motivated the introduction of the sets $Q^M(\alpha,s)$ and the restricted transitive closure operator τ.

We are now in a position to derive modal formulae which characterise the properties of never, always, and sometimes aborting.

5. $\Diamond$*false* for ct^M(**abort**,s).

6. $(\Diamond\Diamond false)$ for $ct^M(\alpha,s)$ if α sometimes aborts.

7. $\Box\Box true$ for $ct^M(\alpha,s)$ if α never aborts.

8. $(\Diamond false \vee (\Diamond true \wedge \Box\Diamond\Diamond false))$ for $ct^M(\alpha,s)$ if α always aborts.

Case 8 requires a little discussion. If α *always* aborts when started from s, then either it aborts directly, in which case $\Diamond false$ holds at s, or else every successor t of s is related to an aborting point, so $\Diamond\Diamond false$ holds at t. But state s must have at least one successor, or else α would be identical to **skip**. So, if s is not itself an aborting point, $\Diamond true$ must hold at s.

The application of Non-Normal modal logic to the analyses of non-termination and failure is one of the original contributions of this research. The appropriate logic and semantics are formally derived in the next section. The correctness of the formulae listed above may then be verified.

5.1.4. A Non-Normal Logic for Computation Trees

In this section we present a modal system which is adequate for Kripke models whose relation has the structure of a computation tree. These relational structures are characterised by

> The presence of queer points at which the formula $\Box true$ fails to hold. Queer points represent a point in a computation tree for α at which α aborts.
>
> The principles TRANS* and IREF*.

The following exposition of the minimal non-normal logic C (and the name 'C') is derived from [10, 41]. The Logic K introduced in §§ 3.1 is a *normal* logic. It is called normal because of the presence of the inference rule RN. K is the smallest normal logic, in the sense that the theorems of K are a subset of any other normal logic. To aid the development, we present a logic K', which is equivalent to K.

The Logic K′ is the smallest set of formulae which contains all instances of the following schema

1. any set of axioms adequate for PC
2. $\Box(\phi \rightarrow \psi) \rightarrow (\Box\phi \rightarrow \Box\psi)$
3. $\Box true$

and is closed under the inference rules

MP: modus ponens, and

RM: from $\vdash_{\Lambda} (\phi \rightarrow \psi)$ infer $\vdash_{\Lambda} (\Box\phi \rightarrow \Box\psi)$.

To see that these axiomatisations are equivalent, we must show that RM preserves validity in K, that RN preserves validity in K′ and that $\Box true$ is a theorem of K.

Suppose $(\phi \rightarrow \psi)$ is K-theorem. Then RN, the axiom K and MP yield $(\Box\phi \rightarrow \Box\psi)$. If ϕ is a K′-theorem then, by PC and MP, so is $(true \rightarrow \phi)$. By RM, $(\Box true \rightarrow \Box\phi)$ whence, since $\Box true$ is a K′ axiom, MP yields $\Box\phi$. Finally, since *true* is a K-theorem by completeness, the rule RN yields $\Box true$, so K′s distinguishing axiom is valid in K.

The logic C, which is merely K′ without the axiom $\Box true$, qualifies as the minimal non-normal logic. Given the semantics to be introduced below, it can be seen that K and C are inequivalent. Models exist which validate all C theorems but for which certain K theorems (in particular, $\Box true$) fail. See below. C has no theorems of the form $\Box\phi$: if it did, it would be equivalent to K. Suppose that $\vdash_C \Box\phi$, for some ϕ. Since $\vdash_C \phi \rightarrow true$ by PC, $\vdash_C \Box\phi \rightarrow \Box true$ by RM. Then $\vdash_C \Box true$ by MP. But if $\vdash_C \phi$, then $\vdash_C true \rightarrow \phi$ by PC. Applying RM, $\vdash_C \Box true \rightarrow \Box\phi$. Since by assumption, $\vdash_C true$, C is closed under RN. Thus the presence of a single theorem of the form $\Box\phi$ makes C equivalent to K.

Remark:

The Logic C is equivalent to that logic having as its only axiom the so-called principle of *aggregation*

$$\Box(\phi \wedge \psi) \rightarrow (\Box\phi \wedge \Box\psi)$$

and RM (and MP) as its only rules of inference. The axiom K and the aggregation principle are inter-derivable in the presence of RM.

It is clear that $\Box true$ is valid in any model for K. In order to provide a semantics appropriate for C it is necessary to generalise the notion of model. The appropriate generalisation of the Kripke-style semantics we have used so far is a model with distinguished elements, or *queer points*.

A model for C is a tuple $M = (S,v,R,Q)$ where (S,v,R) is a model as defined in §§ 3.1, and Q is a subset of S. Members of Q are called *queer* or (non-normal) points. States in $S - Q$ are *normal*. The interpretation of C-formulae is similar to that of §§ 3.1. The interpretation of propositional formulae and of implication remains unchanged. Otherwise,

$M \models_s \Box\phi$ iff $s \in \{S - Q\}$ and $\forall t \in S$, sRt implies $M \models_t \phi$

$M \models_q \Diamond\phi \quad \forall q \in Q$.

Notice that the relation R plays no role in the evaluation of formulae at queer points. By definition, $M \models_q \neg\Box\phi$, $\forall q \in Q$.

From this point, the development of C follows the course of §§ 3.1. We first show that the schema K is valid in any C-model M. K holds at all of M's normal points by the same reasoning as in §§ 3.1. For queer points, every instance of K is true, because the antecedent is false by definition. To quote Fitting [10, p301],

> The extension to queer worlds is thus by a trivial argument, but that's what comes of saying truth isn't necessary.

To see that the rule RM preserves validity in C-Models is easy. Suppose $(\phi \rightarrow \psi)$ is valid. Suppose that $(\Box\phi \rightarrow \Box\psi)$ fails at state t, *i.e* that $(\Box\phi \wedge \neg\Box\psi)$ holds. Then t is not queer, and there must be some u at which both ϕ and $\neg\psi$ hold, which contradicts the validity of $(\phi \rightarrow \psi)$. It follows that all C-theorems are valid in any model M.

The results used in the completeness proof for K4 carry over directly. The only alteration that is needed is in the proof of (3.1.1). It is necessary to show that the Λ-theory Δ is closed under rule RM, instead of RN. This is left as an exercise for the reader. (3.1.2), (3.1.3), (3.1.4) and (3.1.5) carry over without modification. To show completeness, it is necessary only to prove the Fundamental Theorem for C.

The canonical model $M_C = (S_C, v_C, R_C, Q_C)$ is defined as before, except that Q_C is exactly that subset of S_C whose members do not contain the formula $\Box true$. In consequence of the definition of R_C, queer points are related to every member of S_C.

The Fundamental Theorem for C $\quad \forall\phi$ and $\forall u \in S_C$

$$M_C \models_u \phi \text{ iff } \phi \in u.$$

Proof:

Proceeds as the proof of the Fundamental Theorem for K. ■

Define $C4^*$ to be the logic containing, in addition to C, all instances of the schema

4^*: $\Box\phi \rightarrow \Box(\Box true \rightarrow \Box\phi)$.

Likewise, define CT^* to be the logic containing C and all instances of the schema

T^*: $\Diamond true \rightarrow (\Box\phi \rightarrow \phi)$.

The notation is intended to suggest that these principles are analogous to the better known principles 4, which corresponds to transitivity, and T $(\Box\phi \rightarrow \phi)$, which corresponds to reflexivity. We now show that the modal principles 4^* and T^* correspond to the principles $TRANS^*$ and IREF respectively.

We have hitherto been concerned with models. The usual development of completeness proofs employs more general structures known as *frames*. Frames are the underlying structure upon which models are based.

A frame F is a pair $<S, <, Q>$, where S is a set of states, Q is a subset of S and $<$ is a binary relation on S. A model M on a frame F is the frame, together with a valuation function. A formula ϕ is valid on a frame F if it is valid in every model on F. We say that F is *a frame for* a logic Λ if every Λ-theorem is valid on F.

A class of frames T conditions is said to *determine* a logic Λ if the following conditions hold:

a. $F \in T$ implies F is a model of Λ. *Soundness*

b. $\nvdash_{\Lambda} \phi$ implies $F \not\models \phi$ for some $F \in T$. *Completeness*

The class of frames of interest here are all those satisfying the conditions TRANS* and IREF.

Equivalently, we may say that the logic Λ is sound and complete with respect to the class of frames T. That is, every Λ-theorem is valid in every member of T, and every non-theorem fails in some model on some frame in T.

We first show that the class T' of C-frames which satisfy TRANS* determines the logic C4*.

5.1.5 **Theorem** T' determines the logic C4*

Proof:

Let M be a model on some frame $F \in T'$. Suppose that $\Box\phi \rightarrow \Box(\Box true \rightarrow \Box\phi)$ fails at s in M. Then

$M \models_s \Box\phi$ and for some t, $s < t$ and $M \models_t \Box true \wedge \neg\Box\phi$.

Then for some u, $t < u$ and $M \models_u \neg\phi$.

But s and t are both normal, so by TRANS*, sRu.

The assumption that $M \models_s \Box\phi$ then gives a contradiction.

To show that C4* is complete for T, it is enough to show that the canonical C4* model is TRANS*. The proof is almost identical to the demonstration in §§ 3.1 that the canonical K4 model is transitive. ■

Let T'' be the class of C-frames satisfying the principle IREF.

5.1.6 **Theorem** The class T'' determines the logic CT*

Proof:

Suppose the T* instance $(\Diamond true \rightarrow (\Box\phi \rightarrow \phi))$ is not valid in T''. Then for some state s of some model M on some frame F in T'',

$$M \not\models_s \Diamond true \rightarrow (\Box\phi \rightarrow \phi)$$

If s is queer, we get a contradiction directly, since for any queer point q, $M \models_q \Diamond\phi$ and $M \not\models_q \Box\phi$.

If s is normal, the fact that $M \models_s \Diamond true$ requires that there exist some t with $s < t$. Then by IREF, $s < s$. But $M \not\models_s \Box\phi \rightarrow \phi$ requires that $M \models_s \Box\phi \wedge \neg\phi$. Then $s < s$ and the truth condition for $\Box$ yield the desired contradiction.

It follows that every instance of schema T* is valid for every F in T''.

To complete the proof, suppose that the canonical CT* model M_Λ is not internally reflexive. Then there exist distinct maximal consistent CT*-theories s and t such that $s <_\Lambda t$ and not $s <_\Lambda s$. By the definition of the canonical relation, s is not queer (or else $s <_\Lambda s$.) It follows, since s is a maximal consistent set, that $\Box\phi \in s$ and $\neg\phi \in s$, for some formula ϕ. The assumption that $s <_\Lambda t$ ensures that

$\Diamond true \in s$, whether s is queer or not. But the T^* instance $(\Diamond true \rightarrow (\Box\phi \rightarrow \phi))$ is in s, so by deductive closure of s, the formula $\Box\phi \rightarrow \phi$ is in s also. But the assumption of ir-reflexivity at s required that $\Box\phi \in s$ and by deductive closure again, $\phi \in s$, which makes s inconsistent. It follows that the canonical model is internally reflexive. ■

The main result of this section concerns the logic $C4^*T^*$.

5.1.7 **Theorem**

The class of U-frames C satisfying TRANS* and IREF determines the logic $U4^*T^*$.

Proof:

It is necessary to show that the canonical $C4^*T^*$ model satisfies the principles $TRANS^*$ and IREF.

The details are left to the reader. ■

Finally, consider the formula $\Box\Box true$. It is easy to see that if this formula holds at a state s in any model M on any frame in T then all successors of s in M must be normal. This observation justifies the use of $\Box\Box true$ to characterise computations which never abort.

5.2. Language and Semantics

In this section we develop the language of and a semantics for a logic of the dynamic behaviour of programmes. We first introduce a language L and define a set of well formed formulae. We then say what a model for L is like. With the help of some properties of models that we define and derive, we show how to evaluate a formula in a model. The remainder of the section is devoted to establishing the validity of certain formulae required for the axiomatisation.

5.2.1. Syntax

We now present the formal language L used to describe the ongoing behaviour of processes.

The categories Boolean expression (Bxp) and Programmes (Pgm) are exactly as in §§ 4.1. We add two new operators on formulae. The first of these is the now familiar $\Box$. The second operator is $\approx_\alpha$, for every $\alpha \in Pgm$. Informally, the operator $\approx_\alpha$ causes its operand to be interpreted relatively to a computation tree of the programme α. For this reason, $\approx_\alpha$ is referred to as an *indexical* operator or modality. This operator is the distinguishing feature of the system under development.

The set of formulae Fma is defined inductively.

1. $Bxp \subset Fma$,
2. If ϕ and ψ are formulae, so is $(\phi \rightarrow \psi)$,
3. If α is in Pgm and ϕ is a formula, $[\alpha]\phi$ is a formula,
4. If ϕ is a formula, so is $\Box\phi$,
5. If α is in Pgm and ϕ is a formula, $\approx_\alpha \phi$ is a formula.

The use of brackets is informal, as always. The other logical connectives may be defined as in Chapter 3.

5.2.2. Semantics

Models

A model for the language L is a confabulation of an $L_{I/O}$ model and a frame with queer points as defined in §§ 5.1.4. Formally

5.2.1 **Definition** A model M is a structure $(S,v,[\![\cdot]\!],<,Q)$ such that

$(S,v,[\![\cdot]\!])$ is a standard $L_{I/O}$model and

$(S,<,Q)$ is a relational frame with queer points, satisfying TRANS* and IREF.

■

Defining the satisfaction relation $\models$ requires some elementary model theory.

Generated Submodels

By $M_{I/O}$ we mean an $L_{I/O}$-model. Unless otherwise specified, ϕ is understood to be an $L_{I/O}$ formula when interpreted in any $L_{I/O}$-model. We reiterate the definition of generated submodel.

5.2.2 **Definition** Let $M_{I/O} = (S,v,[\![\cdot]\!])$ be an $L_{I/O}$model,s a state in $M_{I/O}$.

$M_{I/O}(s) = (S',v',[\![\ ]\!])$ is an *s-generated sub-model* of $M_{I/O}$ if S' is the least subset of S such that

1. $s \in S'$ and

2. $\forall u \in S,\ \forall t \in S',\ \forall \alpha \in Pgm\ (t[\![\alpha]\!]u$ implies $u \in S')$.

while v' is the restriction of v to members of S' and $[\![\cdot]\!]'$ is the restriction of $[\![\cdot]\!]$ to members of S'. That is,

$\forall s \in S',\ v'_s = v_s$ and

$\forall s,t \in S'\ \forall \alpha \in Pgm\ (s[\![\alpha]\!]'t$ iff $s[\![\alpha]\!]t)$.

■

We denote the s-generated submodel of $M_{I/O}$ by $M_{I/O}(s)$. Intuitively, $M_{I/O}$(s) contains just those states that are reachable from s by means of some terminating programme

or other. $M_{I/O}(s)$ is "just like" $M_{I/O}$ except that points irrelevant to the execution of any programme from s are stripped away. The next theorem gives formal life to this intuition.

5.2.3 **Theorem** *For any model $M_{I/O}$ and any state t in $M_{I/O}(s)$,*

$$M_{I/O} \models_t \phi \text{ iff } M_{I/O}(s) \models_t \phi$$

for any $L_{I/O}$ formula ϕ.

Proof:

The easy proof is by induction on ϕ. ■

The import of this theorem is that no $L_{I/O}$ formula can tell a state in a generated submodel from the same state in the model. The next result shows that validity is preserved by submodel generation.

5.2.4 **Corollary** *For any model $M_{I/O}$ and any state s in $M_{I/O}$,*

$$M_{I/O} \models \phi \text{ implies } M_{I/O}(s) \models \phi$$

Proof:

$M_{I/O} \models \phi$ means that $M_{I/O} \models_s \phi$ for every state s in $M_{I/O}$. The previous theorem and the obvious fact that $S' \subseteq S$ establishes the theorem.. ■

It is also easy to show that the property of being a standard $L_{I/O}$-model is preserved under submodel generation.

5.2.5 **Theorem** *For every state s in every standard model $M_{I/O}$, $M_{I/O}(s)$ is a standard $L_{I/O}$ model*

Proof:

We again omit the tedious proof, which is by induction on the formation rules for the set *Pgm* of programmes. Each part of the proof requires checking that the standard model conditions for each type of programme are preserved in the submodel. ■

The following Corollary follows immediately from (5.2.2).

5.2.6 **Corollary** *For any model $M_{I/O}$ and any state s in $M_{I/O}$,*

$$M_{I/O}(t) = M_{I/O}(s)(t) \quad , \text{ for any } t \text{ in } M_{I/O}(s)$$

■

As a consequence, $M_{I/O}(s) = M_{I/O}(s)(s)$, which is to say that an s-generated submodel of $M_{I/O}$ is the unique s-generated submodel of itself.

When M is an L-model, the notation $M(s)$ refers to the s-generated submodel of the $L_{I/O}$-model $M_{I/O}$ which is contained in M. Of course the relation $<$ and the set Q of queer points is restricted in $M(s)$ to just those states in $M_{I/O}$.

p-morphisms

Let $M = (S,v,<)$ and $M' = (S',v',<')$ be two L_K models. f is a *p-morphism* from M to M' if

1. f is a surjection from S to S'
2. $s < t$ implies $fs < ft$
3. $fs < ft$ implies $\exists u (s < u$ and $fu = ft)$

Clauses (1) and (2) say that f is a homomorphism. Clause (3) is sometimes known as the *backwards* condition. Following [41], we say that a p-morphism f is *reliable* if $v_s = v'_{fs}$ for every state $s \in S$. Versions of the following well known preservation theorem may be found in [41, 45]

5.2.7 **Theorem**

If f is a reliable p-morphism between L_K models M and M' then, for any L_K formula ϕ,

$$M \models_s \phi \text{ iff } M' \models_{fs} \phi .$$

■

We now introduce the obvious generalisation of p-morphisms to $L_{I/O}$ models.

5.2.8 **Definition** Let $M = (S,v,[\![\ ]\!])$ and $M' = (S',v',[\![\ ]\!]')$ be $L_{I/O}$ models.

$f: M \to M'$ is a p-morphism if

1. f is a surjection from S to S'
2. $\forall \alpha \in Pgm,\ s [\![\alpha]\!] t$ implies $fs [\![\alpha]\!]' ft$
3. $\forall \alpha \in Pgm,\ fs [\![\alpha]\!]' ft$ implies $\exists u \in S (s [\![\alpha]\!] u$ and $fu = ft)$

■

Again, f is reliable if $v_s = v'_{fs}$.

It is easy to generalise the preservation theory for L_K to one for $L_{I/O}$.

5.2.9 The p-morphism **Theorem**

If f is a reliable p-morphism between the $L_{I/O}$ models M and M' then

$$M \models_s \phi \text{ iff } M' \models_{fs} \phi$$

for any $L_{I/O}$ formula ϕ.

Proof:

The only non-trivial step in the inductive proof is for ϕ of the form $[\alpha]\psi$.

Suppose that $M \models_s [\alpha]\psi$. If $M' \not\models_{fs} [\alpha]\psi$, then for some ft, $fs [\![\alpha]\!]' ft$ and $M' \not\models_{ft} \psi$. Then by clause (3) of the definition, there is a state u such that $s [\![\alpha]\!] u$ and $fu = ft$. By the induction hypothesis, $M \not\models_u \psi$, contradicting the assumption that $M \models_s [\alpha]\psi$.

On the other hand, if $M \not\models_s [\alpha]\psi$ then for some state t, $s [\![\alpha]\!] t$ and $M \not\models_t \psi$. Since f is a homomorphism, $fs [\![\alpha]\!]' ft$. By the induction hypothesis, $M' \not\models_{ft} \psi$. It follows by the truth condition on $[\alpha]\psi$ that $M' \not\models_{fs} [\alpha]\psi$. ■

In standard models, the relational operator $[\![\ ]\!]$ is defined with reference to the valuation function v. Because of this dependence we may prove the following interesting result.

5.2.10 **Theorem** *Any p-morphism between standard models is reliable.*

Proof:

Suppose f is a p-morphism between M and M'. If f is not reliable, then for some boolean expression ϵ and some state s in M, $v_s(\epsilon) \neq v'_{fs}(\epsilon)$. Assume that $v_s(\epsilon) = 1$. Consider the programme $(\epsilon \Rightarrow \mathbf{skip}, \mathbf{abort})$. Since M is standard, $s [\![\epsilon \Rightarrow \mathbf{skip}, \mathbf{abort}]\!] s$. Then $fs [\![\epsilon \Rightarrow \mathbf{skip}, \mathbf{abort}]\!]' fs$ in M'. By assumption on v'_{fs}, either $v'_{fs}(\epsilon) = \omega$ or $v'_{fs}(\epsilon) = 0$. But then $fs [\![\mathbf{abort}]\!]' ft$ in M', which is impossible.

For the case that $v_s(\epsilon) = 0$, use the programme $[\epsilon \Rightarrow \mathbf{abort}, \mathbf{skip}]$.

If $v_s(\epsilon) = \omega$, then either $v'_{fs}(\epsilon) = 1$ or $v'_{fs}(\epsilon) = 0$. Then the programme $[\epsilon \Rightarrow \mathbf{skip}, \mathbf{skip}]$ and the backwards clause in Def'n. (5.2.8) yields the required contradiction. ■

This result does not hold in general for p-morphisms between L_K models, or for p-morphisms between non-standard $L_{I/O}$ models.

Images of Models

We now define an important relation between models.

5.2.11 **Definition** *Let* $M = (S,v,[\![\cdot]\!],<,Q)$ *be a model.*

A model $M' = (S',v',[\![\cdot]\!]',<',Q')$ is an *(α,s)-image of M* if and only if there exists a function f such that

1. f is a reliable p-morphism between the s-generated submodel of $(S,v,[\![\cdot]\!])$ and $(S',v',[\![\cdot]\!]')$.
2. $\forall fs,ft \in S'$, $fs <' ft$ iff $\langle s,t\rangle \in ct^M(\alpha,s)$
3. $ft \in Q'$ iff $fs <' ft$ implies $t \in Q^M(\alpha,s)$.

■

Condition (3) says that any queer point related to fs in the image must be an aborting (queer) point in the computation tree $ct^M(\alpha,s)$. If N is an (α,s)-image of M, N is just like the generated submodel $M(s)$ except that the relation $<$ in N is defined by the computation tree for programme α starting at state s in M. Note that the relation $<$ in M plays no role in the definition of any image of M. It may help to think of an (α,s)-image of M as the world M as "experienced by" α starting at s.

$M(\alpha,s)$ denotes an arbitrary (α,s)-image of M. We sometimes write "α-image of $M(s)$" for "(α,s) image of M".

5.2.12 **Lemma**

Let N be an (α,s)-image of model M, under the reliable p-morphism f. Then

$$<t,u> \in ct^M(\alpha,s) \text{ iff } <ft,fu> \in ct^N(\alpha,fs)$$

Proof:

The proof is by induction on the complexity of α. We illustrate the basis step, for elementary programmes π.

Suppose that $<s,t> \in ct^M(\pi,s)$. By definition of computation trees, either $s = t$ or $s [\![\pi]\!] t$. If $s = t$ then $fs = ft$ and again by definition, $<fs,ft> \in ct^N(\pi,fs)$. If $s [\![\pi]\!] t$ in M, then $fs [\![\pi]\!] ft$ in N. Again, $<fs,ft> \in ct^N(\pi,fs)$ by definition.

Suppose on the other hand that $<fs,ft> \in ct^N(\pi,fs)$. If $fs \neq ft$ then for some t' in M, $s [\![\pi]\!] t'$ and $ft' = ft$. But $<s,t'> \in ct^M(\pi,s)$, because $s [\![\pi]\!] t'$ in M. It follows, since N is an (π,s) image of M, that $fs < ft'$ in N, which is to say, since $ft' = ft$, $fs < ft$. Then, of course, $<s,t> \in ct^M(\pi,s)$. ■

5.2.13 **Theorem**

Let N be an (α,s)-image of model M under a reliable p-morphism f. Let N' be a (β,ft)-image of N under g. Then N' is a (β,t)-image of M.

Proof:

It is easy to verify that the function gf is a reliable p-morphism from $M(t)$ to N'.

That $gft < gfu$ in N' iff $<t,u> \in ct^M(\beta,t)$ follows directly from the previous lemma.

N' is a (β,ft)-image of N. Suppose that gfu is a queer point in N' This can be true if and only if $gft < gfu$ in N' implies that fu is queer in $ct^N(\beta,ft)$. By the previous lemma, this is equivalent to the condition: $t < u$ in M implies u is queer in $ct^M(\beta,t)$ ■

Satisfaction

We now define the satisfaction relation $M \models_s \phi$ by induction on the members of *Fma*:

1. If $\phi \in Bxp$, $M \models_s \phi$ iff $v_s(\phi) = 1$

2. $M \models_s (\phi \rightarrow \psi)$ iff $M \models_s \phi$ implies $M \models_s \psi$.

3. $M \models_s [\alpha]\phi$ iff $s [\![\alpha]\!] t$ implies $M \models_t \phi$.

4. $M \models_s \Box\phi$ iff $s < t$ implies $M \models_t \phi$.

5. $M \models_s \approx_\alpha \phi$ iff for every N an (α,s)-image of M (under p-morphism f), $N \models_{fs} \phi$.

Clause (5) reveals the hitherto dark purpose in the definitions of submodels and images, and of all the related theorems. The formula $\approx_\alpha \Box\phi$ is intended to mean that $\Box\phi$ is somehow true of the execution of programme α. This required first that the durational notion *execution* be given a formal meaning. This was done by defining computation trees. Temporal formulae such as $\Box\phi$ are given meaning by clause (4). Clause (5) then says that $\approx_\alpha \Box\phi$ is true at state s in M if and only if $\Box\phi$ is true in a model N which is in all respects like M except that its relation $<$ is determined by $ct^M(\alpha,s)$.

From previous results and truth conditions just defined, we may now prove an important result.

5.2.14 **Theorem** Let $N = fM$ and $N' = gM$ be two (α,s)-images of M

$$N \models_{ft} \phi \text{ iff } N' \models_{gt} \phi.$$

for every formula ϕ and every t in $M(s)$.

Proof:

The proof is by induction on ϕ.

For boolean expressions, the result follows from the definition of images, which requires that f and g be reliable.

The result is obvious for formula of the form $(\phi \rightarrow \psi)$

Let ϕ be of the form $[\beta]\psi$, and suppose $N \models_{ft} [\beta]\psi$ and $N' \not\models_{gt} [\beta]\psi$ Then, for some gu in N', $gt [\![\beta]\!] gu$ and $N' \not\models_{gu} \psi$. By the backwards clause for g, there is an u' in M such that $t [\![\beta]\!] u'$ and $gu = gu'$. Then $N' \not\models_{gu'} \psi$. By the fact that f is a homorphism, $ft [\![\beta]\!] fu'$ in N. Applying the induction hypothesis to ψ and the states fu' and gu', $N \not\models_{fu'} \psi$. This contradicts the assumption that $N \models_{ft} [\beta]\phi$.

It follows immediately from the definition of (α,s)-images that $ft < fu$ in N if and only if $gt < gu$ in N'. This establishes the result for ϕ of the form $\Box\psi$.

Let ϕ be of the form $\approx_\beta \psi$. Suppose $N \models_{ft} \approx_\beta \psi$ and $N' \not\models_{gt} \approx_\beta \psi$. By the truth conditions, $N(\beta,ft) \models_{h_1 ft} \psi$, for any (β,ft)-image of N under h_1. Equally, $N'(\beta,gt) \not\models_{h_2 gt} \psi$ for some (β,gt)-image of N' under p-morphismn h_2. But by (5.2.13), $N(\beta,ft)$ and $N'(\beta,gt)$ are both (β,t)-images of M. The result follows by the induction hypothesis. ■

In view of the Theorem, clause (5) of the truth conditions may be written

5′. $M \models_s \approx_\alpha \phi$ iff for *some* f, fM is an (α,s)-image of M and $fM \models_{fs} \phi$.

We now prove some important properties of the indexical modality $\approx$.

5.2.15 **Theorem**

The following schemata are valid in all models.

1. $\approx_\alpha(\phi \rightarrow \psi) \leftrightarrow (\approx_\alpha \phi \rightarrow \approx_\alpha \psi)$
2. $\neg\approx_\alpha \phi \leftrightarrow \approx_\alpha \neg\phi$
3. $\approx_\alpha \approx_\beta \phi \leftrightarrow \approx_\beta \phi$
4. $\approx_\alpha \phi \leftrightarrow \phi$ for any $L_{I/O}$ formula ϕ.

Proof:

1. Let $M(\alpha,s)$ be some (α,s) image of M under f. Then

$M \models_s \approx_\alpha(\phi \rightarrow \psi)$ iff

$M(\alpha,s) \models_{fs} (\phi \rightarrow \psi)$ iff

$M(\alpha,s) \models_{fs} \phi$ implies $M(\alpha,s) \models_{fs} \psi$ iff

$M \models_s \approx_\alpha \phi$ implies $M \models_s \approx_\alpha \psi$, by the truth condition on $\approx_\alpha$.

The result follows by the truth condition on $\rightarrow$.

2. $M \models_s \neg\approx_\alpha \phi$ iff $M \not\models_s \approx_\alpha \phi$ iff $M(\alpha,s) \not\models_{fs} \phi$ iff $M(\alpha,s) \models_{fs} \neg\phi$ iff $M \models_s \approx_\alpha \neg\phi$

3. Let f be the image function from M to some image $M(\alpha,s)$ and g the function from $M(\alpha,s)$ to one of its (β,fs) images $M(\alpha,s)(\beta,fs)$. Then

$$M \models_s \approx_\alpha \approx_\beta \phi \text{ iff } M(\alpha,s) \models_{fs} \approx_\beta \phi \text{ iff } M(\alpha,s)(\beta,fs) \models_{gfs} \phi .$$

By Theorem (5.2.13), $M(\alpha,s)(\beta,fs)$ is a (β,s)-image of M. The result follows by (5.2.14) and the truth condition on $\approx_\beta$.

4. By Theorem (5.2.9) and clause (2) of Definition (5.2.11).

■

Remark:

The theorem demonstrates why the $\approx$ operator is given the suggestive title of indexical modality. In the presence of a rule of normality, $\approx$ bears a striking resemblance to an *alethic* modality. The distinction between $\approx$ and the modality of S5 is that the schema $(\approx_\alpha \phi \leftrightarrow \phi)$ is restricted to $L_{I/O}$ formulae. For, of course, the truth of $\Box\phi$ at state s is independent of the truth of $\Box\phi$ in $ct^M(\alpha,s)$ for some arbitrary programme α.

5.2.16 **Theorem** The following schemata are valid in all models.

1. $\Box(\phi \rightarrow \psi) \rightarrow (\Box\phi \rightarrow \Box\psi)$

2. $\Box\phi \rightarrow \Box(\Box true \rightarrow \Box\phi)$

3. $\Diamond true \rightarrow (\Box\phi \rightarrow \phi)$

Proof:

1. This scheme is valid for any binary relation.

2. Follows from Theorem (5.1.5), since $<$ satisfies the principle TRANS*.

3. Follows from Theorem (5.1.6), since $<$ satisfies IREF.

■

5.2.17 **Theorem**

The following schemata are valid in all standard models.

1. $[\alpha]\phi \leftrightarrow \approx_{\alpha}\Box(\Box \mathit{false} \rightarrow \phi)$, for any $L_{I/O}$ formula ϕ.

2. $\approx_{\mathit{skip}}\Box \mathit{false}$

3. $\approx_{\mathit{abort}} \Diamond \mathit{false}$

4. $\approx_{\pi}\Box\phi \leftrightarrow \approx_{\pi}(\phi \wedge [\pi]\phi)$, for any elementary programme π.

Proof:

1. $M \models_s \approx_{\alpha}\Box(\Box \mathit{false} \rightarrow \phi)$ iff, in any (α,s) image of M,

$$fs < ft \text{ and } M(\alpha,s) \models_{ft} \Box \mathit{false} \text{ implies } M(\alpha,s) \models_{ft} \phi.$$

Choose some such ft. By clause (2) in (5.2.11), $<s,t> \in ct^M(\alpha,s)$. By assumption, ft is a normal state (since $\Box \mathit{false}$ holds), so by clause (3) of the definition $t \notin Q^M(\alpha,s)$. It follows from TRANS* that t is a final state in $ct^M(\alpha,s)$ which is to say $s [\![\alpha]\!] t$ in M. The result follows by (5.2.9).

2. Suppose not. Then for some state s in some model M, $M \models_s \approx_{\mathit{skip}}\Box \mathit{false}$. But this is true iff

$M \models_s \neg\approx_{\mathit{skip}}\Box \mathit{false}$, iff (by Theorem (5.2.15)(2))
$M \models_s \approx_{\mathit{skip}}\neg\Box \mathit{false}$, iff
$M \models_s \approx_{\mathit{skip}} \Diamond \mathit{true}$ iff
$M(\mathbf{skip},s) \models_{fs} \Diamond \mathit{true}$ iff
$\exists\, ft : fs < ft$ iff,

since $M(\mathbf{skip},s)$ is a $(\mathbf{skip},s)$-image of M, $<s,t> \in ct^M(\mathbf{skip},s)$. This is a contradiction since, by definition, $ct^M(\mathbf{skip},s) = \varnothing$.

3. $M \models_s \approx_{abort} \langle\rangle false$ iff

$M(\mathbf{abort},s) \models_{fs} \langle\rangle false$ iff

fs is a queer point in $M(\mathbf{abort},s)$ iff

s is queer in $ct^M(\mathbf{abort},s)$,

which it is by Definition (5.1.3) and clause (4) of Definition (5.2.11).

4. $M \models_s \approx_\pi \Box\phi$ iff

$M(\pi,s) \models_{fs} \Box\phi$ iff, $\forall ft$ in $M(\pi,s)$, $fs < ft$ implies $M(\pi,s) \models_{ft} \phi$.

But $fs < ft$ iff $\langle s,t\rangle \in ct^M(\pi,s)$ which means that either $t = s$ or $s [\![\pi]\!] t$.

Thus, in $M(\pi,s)$, either $fs = ft$ or $fs [\![\pi]\!]' ft$.

It follows that $M(\pi,s) \models_{fs} \Box\phi$ iff $M(\pi,s) \models_{fs} \phi \wedge [\pi]\phi$

iff $M \models_s \approx_\pi (\phi \wedge [\pi]\phi)$.

■

5.3. Representation of execution properties

Now that the language and semantics have been defined, we show how this system is capable of representing those properties of programme execution introduced in earlier chapters.

Representation of the *after* modality is of course inherited from the system in § 4. And, as Theorem (5.2.17)(1) shows, the formula $[\alpha]\phi$, which asserts "after α, ϕ is true", is equivalent to $\approx_\alpha \Box(\Box false \rightarrow \phi)$.

Reviewing the semantic clauses, it is easy to see that formula $\approx_\alpha \Box\phi$ is valid if and only if ϕ is true at every state in every computation tree of α. Thus $\approx_\alpha \Box\phi$ corresponds to the property "*throughout* α, ϕ." The property "*during* α, ϕ" is merely the dual of "*throughout* α, ϕ". But $\neg\approx_\alpha \Box\neg\phi$ is equivalent, by Thm. (5.2.15)(2) and the duality of $\Box$ and $\langle\rangle$, to $\approx_\alpha \langle\rangle\phi$.

Properties such as Pratt's *preserves* or Lamport's rendition of liveness may be represented by indexing the appropriate UT^*4^* formula by the operator $\approx_\alpha$. In particular, if programme α *preserves* formula ϕ, then in our system, $\approx_\alpha \Box(\phi \rightarrow \Box\phi)$ will be a valid formula.

Finally, we show how a fair execution property may be represented. As remarked earlier, the weakest commonly accepted version of fair process execution is

> If a Process component is infinitely often enabled, then it will be executed infinitely often.

It is necessary at last to clarify this notion. Recall that Π_k stands for the Process $(\epsilon_1 \ldots \epsilon_k \# \alpha_1 \ldots \alpha_k)$. There is no way to say directly in our language that component $[\alpha_j]$ is ever selected during the execution of Π_k. However, suppose that

$$\epsilon_j \rightarrow \approx_{\alpha_j} \Diamond\phi$$

is a valid formula. It says that ϕ is true at some state in any computation of programme α_j which starts at a state in which its guard ϵ_j is true. If, under some conditions, we can then deduce that

$$\approx_{\Pi_k} \Diamond\phi ,$$

a connection between the execution of a Process as a whole and the execution of its separate components has been established. We would then have an axiom schema which says

> *if* ϕ occurs during the execution of α_j *and*
> *if* α_j is ever selected during the execution of Π_k,
> *then* ϕ occurs during the execution of Π_k

A component is enabled if its guard is true when "the scheduler" is selecting the

next component to execute. There is no formula in our language which characterises a state in which this choice is being made. That is, there is no formula which says "guard evaluation happening now". Reviewing the definition of a Process computation tree, it is easy to see that the selection of the next component occurs when the previous component has finished executing. The selection is made by choosing one of the components whose guard is true.

The formula

$$\approx_{\Pi_k} \Box\Diamond \epsilon_j$$

asserts that ϵ_j occurs infinitely often during the execution of Π_k. It does not follow from this that α_j need ever be enabled, let alone executed. For instance, some other component could repeatedly be switching the guard on and off. Moreover, guards will be employed to detect the occurrence of asynchronous events which occur independently of Process execution, such as the arrival of a message from some other Process in a network of parallel event servers. For these reasons, a fairness specification must include a provision that a guard, once true, remains true at least until the next selection event. The formula

$$\epsilon_i \rightarrow \approx_{\alpha_i} \Box(\epsilon_j \rightarrow \Box(\Box \mathit{false} \rightarrow \epsilon_j))$$

is a slightly weaker preservation specification. It says that the guard ϵ_j will be true in any final state of programme α_i if it becomes true at any point during an enabled execution of α_i. The much weaker specification

$$\epsilon_i \rightarrow (\epsilon_j \rightarrow [\alpha_i]\epsilon_j)$$

is an easy consequence.

If the preceding schema is true for every component α_i, then if ϵ_j ever becomes

true during the execution of Π_k, we may be confident that α_j will eventually be enabled (as long as no component aborts or diverges.)

We may now present a fairness specification.

$$\begin{aligned}((\wedge_{i \neq j} \epsilon_i \rightarrow \approx_{\alpha_i} \Box(\epsilon_j \rightarrow \Box(\Box \mathit{false} \rightarrow \epsilon_j)) \\ \wedge (\epsilon_j \rightarrow \approx_{\alpha_j} \Diamond \phi)) \\ \rightarrow (\approx_{\Pi_k} \Box(\epsilon_j \rightarrow \Diamond \phi)))\end{aligned}$$

This says that if ϵ_j is preserved by every other component other than α_j, and if ϕ is a consequence of the enabled execution of α_j, then if ϵ_j ever becomes true during Π_k, then ϕ will become true at some later point in the execution. It is a simple consequence of this formulation that the consequent $\approx_{\Pi_k} \Box(\epsilon_j \rightarrow \Diamond \phi)$ may be replaced by

$$\approx_{\Pi_k} \Box \Diamond \epsilon_j \rightarrow \approx_{\Pi_k} \Box \Diamond \phi .$$

However the first formulation is actually stronger. It says that if a guard becomes true, it remains true at least until it's sequential component is selected (but not necessarily any longer), and that this selection must occur. This specification corresponds to the weak equi-fairness of [22]. Every time a guard becomes true, the component which it guards is executed.

Other notions of fairness could probably be expressed in this language system, but we pursue the matter no further.

The standard model conditions for this and other notions of fairness will not be formally developed here. Suffice to remark that they would amount to the restriction that certain infinite suborderings could not occur in the computation tree of any process.

The system under development is powerful enough to express at least some notion of execution fairness. It is able to do so without employing any predicates which refer to programme location.

5.4. Proof Theory

We take as our axioms, in addition to those of § 4, all instances of the following schemata.

Axioms

$A\approx 1$ $\quad \approx_\alpha(\phi \to \psi) \leftrightarrow (\approx_\alpha \phi \to \approx_\alpha \psi)$

$A\approx 2$ $\quad \neg\approx_\alpha \phi \leftrightarrow \approx_\alpha \neg\phi$

$A\approx 3$ $\quad \approx_\alpha \approx_\beta \phi \leftrightarrow \approx_\beta \phi$

$A\approx 4$ $\quad \approx_\alpha \phi \leftrightarrow \phi$ for any $L_{I/O}$ formula ϕ.

UT^*4^* axioms

K $\quad \Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$

T^* $\quad \Diamond true \to (\Box\phi \to \phi)$

4^* $\quad \Box\phi \to \Box(\Box true \to \Box\phi)$

Bridge Principles

$A\approx 6$ $\quad \approx_{skip} \Box true$

$A\approx 7$ $\quad \approx_{abort} \Diamond false$

$A\approx 8$ $\quad \approx_\pi \Box\phi \leftrightarrow \approx_\pi (\phi \wedge [\pi]\phi)$

All instances of these axioms are shown to be valid in Theorems (5.2.15), (5.2.16) and (5.2.17).

The so called *bridge* axioms provide the linkage between the three distinct modalities of our system.

As rules of inference, we take, in addition to MP, OI, GOI and TR, the rule RM (see §§ 5.1.4) and the rule of normality for the indexical modality,

R≈N: from $\vdash_{\Lambda}\phi$ infer $\vdash_{\Lambda}\approx_{\alpha}\phi$

We now define, as usual, a logic Λ to be any subset of *Fma* which contains all instances of the axioms, and which is closed under the rules of inference.

The deducibility relation $\vdash_{\Lambda}$ is defined just as in § 4. An Λ-theory is any subset of *Fma* which contains Λ and which is closed under the rules MP, OI, and GOI.

5.5. Completeness

We have been unable to finish the completeness proof for Λ. This section presents a snapshot of a Henkin proof in progress.

5.5.1. The Canonical Model

The canonical Λ-model is the structure $M_{\Lambda} = (S_{\Lambda}, v_{\Lambda}, [\![\cdot]\!], <_{\Lambda}, Q_{\Lambda})$, where S_{Λ} and v_{Λ} are defined as in §§ 4.4.1, and for all $\Gamma,\Delta \in S_{\Lambda}$,

(i) for each elementary programme π, $\Gamma [\![\pi]\!] \Delta$ iff $\{\phi : [\pi]\phi \in \Gamma\} \subseteq \Delta$. The relation $[\![\alpha]\!]$ is defined inductively for arbitrary members of *Pgm* according to the standard $L_{I/O}$ model conditions.

(ii) $\Gamma <_{\Lambda} \Delta$ iff $\{\phi : \Box\phi \in \Gamma\} \subseteq \Delta$.

(iii) $\Gamma \in Q_{\Lambda}$ iff $\Diamond\mathit{false} \in \Gamma$.

M_{Λ} is a standard $L_{I/O}$ model by definition. That $<_{\Lambda}$ satisfies the principles TRANS* and IREF follows just as in §§ 5.1.4.

Of course the goal of this section is to prove the fundamental theorem for Λ, *viz:*

The Fundamental Theorem $\quad \forall \phi \in Fma \ , \ \forall \Gamma \in S_\Lambda$

$$\phi \in \Gamma \ \text{ iff } \ M_\Lambda \models_\Gamma \phi \qquad ,$$

■

The completeness of Λ then follows just as in § 4. However, in this instance, the actual proof of the fundamental theorem is somewhat more complex than in § 4. As an aid to the reader, we pause to outline the strategy of the proof.

The proof proceeds by induction on the complexity of formulae, in which the only difficult case is for formulae of the form $\approx_\alpha \phi$. Recalling the truth condition for the $\approx$ operator,

$$M_\Lambda \models_\Gamma \approx_\alpha \phi \ \text{ iff } \ N \models_{f\Gamma} \phi \, ,$$

where N is an α-image of $M_\Lambda(\Gamma)$ under a p-morphism f. The burden of our proof is to construct just such an image.

Consider the set $\approx_\alpha(\Gamma) = \{\phi : \approx_\alpha \phi \in \Gamma\}$ We prove first that $\approx_\alpha(\Gamma)$ is a maximal consistent Λ-theory, and so a member of S_Λ. We then construct a p-morphism f between the two submodels $M_\Lambda(\Gamma)$ and $M_\Lambda(\approx_\alpha(\Gamma))$. By the construction, $f\Gamma = \approx_\alpha(\Gamma)$. Next, we show that

$$f\Gamma <_\Lambda f\Delta \quad \text{iff} \quad <\Gamma,\Delta> \in ct^{M_\Lambda}(\alpha,\Gamma) \, ,$$

establishing that $M_\Lambda(f\Gamma)$ is an α-image of $M_\Lambda(\Gamma)$. (It is this part of the proof which remains elusive.) Given the result, it follows easily that $M_\Lambda \models_{f\Gamma} \phi$ iff $M_\Lambda(f\Gamma) \models_{f\Gamma} \phi$. The inductive step for $\approx_\alpha \phi$ in the proof of the fundamental theorem is finally polished off by observing that $\approx_\alpha \phi \in \Gamma$ iff $\phi \in f\Gamma$, and then applying the induction hypothesis on ϕ to the state $f\Gamma$. We now establish the necessary theorems.

We first re-state a theorem which carries over directly from § 4.

5.5.1 **Theorem** $\forall \epsilon \in Bxp$ and $\forall \Gamma \in S_\Lambda$, *[2.5.5]*

$$\epsilon \in \Lambda \quad \text{iff} \quad M_\Lambda \models_\Gamma \epsilon$$

■

This is the Fundamental Theorem for boolean expressions.

5.5.2 **Theorem** $\forall \alpha \in Pgm$ and $\forall \Gamma \in S_\Lambda$,

$$\approx_\alpha(\Gamma) = \{\phi : \approx_\alpha \phi \in \Gamma\} \text{ is a maximal consistent } \Lambda\text{-theory.}$$

Proof:

By the rule $RN\approx$, $\Lambda \subseteq \approx_\alpha(\Gamma)$. By maximality and consistency of Γ and axiom $A\approx 2$, $\approx_\alpha(\Gamma)$ is maximal and consistent. Axiom $A\approx 1$ guarantees closure under the rule MP.

We claim, but can not as yet prove, that $\approx_\alpha(\Gamma)$ is closed under the rules OI and GOI. A proof of this probably requires a generalisation of the results on Admissible Forms (§ 4). Should no generalisation be necessary, then all admissible forms will be $L_{I/O}$ formulae, and the result will follow via $A\approx 4$. ■

Since $\approx_\alpha(\Gamma)$ is a maximal consistent Λ-theory, it is a member of S_Λ.

We now state, without proof, the analogue of theorem (4.4.2).

5.5.3 **Theorem** $\models_\Lambda \phi$ iff $\forall \Gamma \in S_\Lambda, \phi \in \Gamma$.

■

The details of this (unfinished) proof depend on whether or not the axiomatisation requires additional rules of inference.

5.5.2. Proof of the Fundamental Theorem

Our first objective is to establish a p-morphism between the generated submodels $M_\Lambda(\Gamma)$ and $M_\Lambda(\approx_\alpha(\Gamma))$. The construction depends upon the existence of an enumeration of the states in a generated submodel, which we first define.

It is obvious that, in standard models, if $s[\![\alpha]\!]t$ then there exists a programme

$$\gamma = (\pi_1 ; \pi_2 ; \ldots ; \pi_n) \text{ such that } s[\![\gamma]\!]t,$$

where each π_i is an elementary programme.

This observation suggests an (infinite) iterative procedure for constructing generated submodels. Before demonstrating this procedure, it is necessary to place some restrictions upon the behaviour of elementary programmes. In particular, we require that each elementary programme be *deterministic* and also that it *terminate*. These properties are determined by adding the following axiom schemata to those of § 4:

ADπ: $\quad [\pi]\phi \vee [\pi]\neg\phi$

ATπ: $\quad \neg[\pi]\mathit{false}$

The corresponding standard model condition is that, for each elementary programme π, $[\![\pi]\!]$ be a total function on the state set S. Formally,

$$\forall \pi \in Pgm\ ,\ \forall s \in S\,(\exists!\, t : s[\![\pi]\!]t).$$

For a proof of this fact, see [14,§2.6]. Henceforth, assume that all standard models satisfy this condition, and that the logic Λ has been suitably re-defined to include all instances of the schemata ADπ and ATπ.

These requirements smack of *ad-hoc*ery. There is, however, a justification beyond immediate need. Elementary programmes are designed to be replaced with assignment statements in the first order extension of our system. These statements take the form

$$(x := \sigma)$$

where x is a variable in the first order language of some algebra, and σ is an algebraic expression, naming some element of the algebra. An example is the programme $(p := \mathit{false})$, where $p \in \mathit{Bvb}$. Of course *false* is a boolean expression. The analogues of the schemata ADπ and ATπ are required in the completeness proof for the first order system of [14,§3]. By introducing them at this stage we merely anticipate. It is a consequence of these axioms that, in the canonical model,

$$\Gamma [\![\pi]\!] \Delta \text{ iff } \{\phi : [\pi]\phi \in \Gamma\} = \Delta .$$

Let $M = (S,v,[\![\cdot]\!],<,Q)$. We now describe an enumeration for the generated submodel $M(s)$.

5.5.4 **Definition** For any state s in a model M, let

$$s_0 = \{s\}$$

$$s_{i+1} = \bigcup_{t \in s_1} \bigcup_{\pi \in \mathit{Pgm}} \{u : t [\![\pi]\!] u\}$$

$$s^* = \bigcup_{i < \omega} s_i$$ ■

By axioms ADπ and ATπ, s_{i+1} is well defined at each stage.

5.5.5 **Theorem** For any model $M = (S,v,[\![\cdot]\!],<,Q)$ and $s \in S$

s^* is exactly the set of states in the generated submodel $M(s)$.

Proof:

Immediate from the definition of generated submodels (5.2.2), and the observation on page 117. ■

We may now define a function f from $M_\Lambda(\Gamma)$ to $M_\Lambda(\approx_\alpha(\Gamma))$ in terms of the sets Γ^* and $\approx_\alpha(\Gamma)^*$.

5.5.6 **Definition**

$$f(\Gamma_0) = f\Gamma = \{\phi : \approx_\alpha \phi \in \Gamma\} = \approx_\alpha(\Gamma) = \approx_\alpha(\Gamma_0)$$

If $\Delta' \in \Gamma_{i+1}$ then for some elementary programme π and some state $\Delta \in \Gamma_i$, $\Delta [\![\pi]\!] \Delta'$. By AD$\pi$ and ATπ, $f\Delta [\![\pi]\!] \Delta''$ for some unique Δ''. Put $f\Delta' = \Delta''$. ■

We sometimes refer to f as the *Canonical function*, or where specificity is required, as the canonical (α,Γ) function.

It is necessary to verify that the function f preserves the value of boolean expressions. In fact, we prove the slightly more general:

5.5.7 **Lemma** For every $\Delta \in M_\Lambda(\Gamma)$ and every $\phi \in L_{I/O}$

$$\phi \in \Delta \text{ implies } \phi \in f\Delta$$

Proof:

By induction on the construction of f.

For the basis step, if $\phi \in \Gamma$ then, by A$\approx$4, $\approx_\alpha \phi \in \Gamma$. Then $\phi \in f\Gamma$.

Assume the result for step i. Suppose that $\Delta [\![\pi]\!] \Delta'$ and that $\phi \in \Delta'$. Then by functionality of $[\![\pi]\!]$, $[\pi]\phi \in \Delta$. Since $[\pi]\phi$ is an $L_{I/O}$-formula if ϕ is, the hypothesis of induction applied to Δ assures us that $[\pi]\phi \in f\Delta$. Again by functionality of $[\![\pi]\!]$, there exists a unique Δ'' such that $f\Delta [\![\pi]\!] \Delta''$. The definition of M_Λ requires that $\phi \in \Delta''$. But by construction of f, $\Delta'' = f\Delta'$. ■

We are now able the prove the first of two vital preliminary theorems.

5.5.8 **Theorem** For every $\Gamma \in S_\Lambda$ and $\alpha \in Pgm$,

$$f : M_\Lambda(\Gamma) \to M_\Lambda(\approx_\alpha(\Gamma))$$

is a p-morphism, for f defined as above.

Proof:

An equivalent statement of the theorem is that f is a p-morphism from Γ^* to $\approx_\alpha(\Gamma)^*$.

That f is really a function follows from the functionality of the relations $[\![\pi]\!]$ *and Theorem* (5.5.5).

Since there are three clauses in the definition of a p-morphism, the proof takes the form of three sublemmata

Lemma 1. We prove that f is surjective by induction on the construction of Γ^*. Specifically, we show that f is a surjection from Γ_i to $\approx_\alpha(\Gamma)_i$, for $0 \leqslant i < \omega$

For the basis step, f is clearly a surjection; there is only one element in $\approx_\alpha(\Gamma)_0$, namely $f\Gamma$.

Assume the result for stage i, in order to prove it for stage $i+1$. Suppose that Δ'' is in $\approx_\alpha(\Gamma)_{i+1}$. Then by construction, and the induction hypothesis, $f\Delta\,[\![\pi]\!]\,\Delta''$, for some π. By ADπ and ATπ, these is a unique $\Delta' \in \Gamma_{i+1}$ such that $\Delta\,[\![\pi]\!]\,\Delta'$. By definition of f, $\Delta'' = f\Delta'$.

This establishes that f is a surjection from Γ^* onto $\approx_\alpha(\Gamma)^*$. ■

Lemma 2. We now show that f is a homomorphism. That is, $\Delta\,[\![\alpha]\!]\,\Delta'$ implies $f\Delta\,[\![\alpha]\!]\,f\Delta'$, for every programme α. The proof is by induction on α.

The steps for elementary programmes, and for **skip** and **abort** are trivial.

Assume the result for α and β, in order to prove it for $(\alpha;\beta)$. If $\Delta\,[\![\alpha;\beta]\!]\,\Delta'$ then, since by Theorem (5.2.5) $M_\Lambda(\Gamma)$ is standard, there is some Δ_0 such that $\Delta\,[\![\alpha]\!]\,\Delta_0\,[\![\beta]\!]\,\Delta'$. The induction hypothesis, and the standardness of $M_\Lambda(\approx_\alpha(\Gamma))$ yields the result.

Assume the result for α and β, in order to prove it for $[\epsilon \Rightarrow \alpha,\beta]$. By the standard model conditions, if $\Delta\,[\![\epsilon \Rightarrow \alpha,\beta]\!]\,\Delta'$ then either $v_\Delta(\epsilon) = 1$ and $\Delta\,[\![\alpha]\!]\,\Delta'$ or $v_\Delta(\epsilon) = 0$ and $\Delta\,[\![\beta]\!]\,\Delta'$. Then by Theorem (5.5.1), either $\epsilon \in \Delta$ or *not*—$\epsilon \in \Delta$. The induction hypothesis applied to α (or β), Lemma (5.5.7) and the standardness of $M_\Lambda(\approx_\alpha(\Lambda))$ establishes the result.

We leave to the reader the pleasant diversion of proving the result for iteration and generalised iteration. ■

Lemma 3. Finally, we must estabish the backwards clause for f. That is, we must show that if $f\Delta [\![\alpha]\!] f\Delta''$ then for some Δ', $\Delta [\![\alpha]\!] \Delta'$ and $f\Delta' = f\Delta''$. Again, we proceed by induction on α.

The bases steps for the programmes **abort** and **skip** are trivial. For, by standardness, $[\![\textbf{abort}]\!]$ is empty, while $[\![\textbf{skip}]\!]$ is the identity relation.

Suppose that $f\Delta [\![\pi]\!] f\Delta'$. By AD$\pi$ and ATπ, $\Delta [\![\pi]\!] \Delta''$, for some Δ''. But by the definition of f, $f\Delta''$ is the (unique) state π-related to $f\Delta$, namely $f\Delta'$.

Assume the result for α and β. Suppose that $f\Delta [\![\alpha ; \beta]\!] f\Delta'$. By standardness, $f\Delta [\![\alpha]\!] f\Theta' [\![\beta]\!] f\Delta'$, for some $f\Theta'$. The induction hypothesis applied to $[\![\alpha]\!]$ yields that, for some Θ, $\Delta [\![\alpha]\!] \Theta$ and $f\Theta = f\Theta'$. Then $f\Theta [\![\beta]\!] f\Delta'$. Again by the induction hypothesis, $\Theta [\![\beta]\!] \Delta''$ for some Δ'' such that $f\Delta'' = f\Delta'$. By standardness, $\Delta [\![\alpha ; \beta]\!] \Delta''$ in M_Λ.

Assume the result for α and β. Suppose that $f\Delta [\![\epsilon \Rightarrow \alpha,\beta]\!] f\Delta'$. If $v_\Gamma(\epsilon) = 1$, then Theorem (5.5.7) and the standard model conditions yield that $f\Delta [\![\alpha]\!] f\Delta'$. From the hypothesis on α we obtain the desired result. For the case that $v_\Gamma(\epsilon) = 0$, the result follows similarly.

The remaining cases are foundlings, left to the mercy of the reader, ■

■

Recall that, by Theorem (5.2.5), the property of being a standard model is preserved under sub-model generation. Thus, since M_Λ is standard, so are $M_\Lambda(\Gamma)$ and $M_\Lambda(f\Gamma)$. f has just been shown to be a p-morphism between two standard models, and so, by Theorem (5.2.10), f is reliable.

This development suggests some interesting questions.

1. Lemma (5.2.12) suggests that f may be stronger than a p-morphism. In exactly what sense is this true? Might f really be an *isomorphism*? If not, what extra conditions would be required to make it one?

2. Is it possible to relax the conditions of determinacy and/or termination for elementary programmes and still prove the theorem with f as presently defined? If not, what kind of a morphism would f need to be? Would it be strong enough to prove an appropriate preservation theorem?

3. The proof depends upon a construction that has countably many stages, each of which involves evaluating countably many relations. Can a more finitary construction be found?

4. Is a direct existence proof available?

Some exploration of the model theory of our system would likely shed light on these questions.

We now turn our attention to the proof of the second key result needed for the Fundamental Theorem. The following 'theorem' should be regarded as a conjecture.

5.5.9 **Theorem** $\forall \alpha \in Pgm$ and $\forall \Gamma,\Delta \in S_\Lambda$,

$$f_\alpha\Gamma <_\Lambda f_\alpha\Delta \text{ iff } <\Gamma,\Delta> \in ct^{M_\Lambda}(\alpha,\Gamma),$$

where f_α is the canonical (α,Γ) function.

Proof:

By induction on α. Only the cases for elementary programmes can yet be established.

Case 1: **skip**
By Definition (5.1.2), $ct^{M_\Lambda}(\mathbf{skip},\Gamma) = \varnothing$. By A$\approx$6, $\approx_{skip}\Box false \in \Gamma$, whence $\Box false \in f\Gamma$. It follows that $f\Gamma \not<_\Lambda f\Delta$ for any state $f\Delta$.

Case 2: **abort**
By definition, $<\Gamma,\Delta> \in ct^{M_\Lambda}(\mathbf{abort},\Gamma)$, for every $\Delta \in \Gamma^*$, *i.e.* for every $\Delta \in M_\Lambda(\Gamma)$. By A$\approx$7, $\approx_{abort}\Diamond false \in \Gamma$. So, $\Diamond false \in f\Delta$. By the definition of the canonical relation $<_\Lambda$, $f\Gamma <_\Lambda \Delta$, for every state $\Delta \in S_\Lambda$. So, in particular, $f\Gamma <_\Lambda f\Delta$, $\forall f\Delta \in f(\Gamma^*)$. (Notational Reminder: $f(\Gamma^*)$ denotes the set of images under f of the states in the Γ-generated submodel.)

Case 3: an elementary programme π
By Definition (5.1.2),

$$ct^{M}_{\Lambda}(\pi,\Gamma) = \{<\Gamma,\Gamma> , <\Gamma,\Delta>\}$$

where Δ is the unique state such that $\Gamma [\![\pi]\!] \Delta$.

We prove first that $f\Gamma <_{\Lambda} f\Gamma$ and that $f\Gamma <_{\Lambda} f\Delta$. By A$\approx$8 and PC, every instance of $(\Box\phi \rightarrow \phi)$ and $(\Box\phi \rightarrow [\pi]\phi)$ is in $f\Gamma$. Clearly $f\Gamma <_{\Lambda} f\Gamma$, by the definition of $<_{\Lambda}$. And, since $f\Gamma [\![\pi]\!] f\Delta$, $[\pi]\phi \in f\Gamma$ implies $\phi \in f\Delta$. Then, $\Box\phi \in f\Gamma$ implies $\phi \in f\Delta$, whence $f\Gamma <_{\Lambda} f\Delta$.

We observe that $f\Gamma$ is not queer. *true* is an axiom (A12) so, by the rule TR, $[\pi]$*true* is a theorem. Applying Rule RN$\approx_{\pi}$ we obtain $(\approx_{\pi} true \wedge [\pi]true)$. Then by an instance of A$\approx$8, $\approx_{\pi}\Box true$ is a theorem of Λ. So, $\Box true \in f\Gamma$.

To complete the case for π, we show that

$$f\Gamma <_{\Lambda} \Theta \text{ implies } \Theta = f\Gamma \text{ or } \Theta = f\Delta .$$

The proof requires a sublemma.

Sublemma: If $f\Gamma <_{\Lambda} \Theta$ then $\Theta \subseteq \{f\Gamma \cup f\Delta$

Proof:

Suppose not. Then $\psi \in \Theta$, but $\psi \notin f\Gamma$ and $\psi \notin f\Delta$. Since $f\Gamma$ is a maximal consistent Λ-theory, $\neg\psi \in f\Gamma$ and $\neg\psi \in f\Delta$. Furthermore, since, by definition of ϕ, $f\Gamma [\![\pi]\!] f\Delta$, it follows by axiom AD$\pi$ that $[\pi]\neg\psi \in f\Gamma$. By the A$\approx$8 instance

$$\approx_{\pi}\Box\neg\psi \leftrightarrow \approx_{\pi}(\neg\psi \wedge [\pi]\neg\psi) ,$$

and the fact the $f\Gamma = \{\phi : \approx_{\pi}\phi \in \Gamma\}$, $\Box\psi \in f\Gamma$. This contradicts the assumption that $f\Gamma <_{\Lambda} \Theta$, since $\psi \in \Theta$. Therefore, $\Theta \subseteq \{f\Gamma \cup f\Delta\}$ ■

Now suppose that $\Theta \neq f\Gamma$. Since both $f\Gamma$ and Θ are maximal consistent Λ-theories, there must be some formula ϕ such that $\phi \in \Theta$ and $\phi \notin f\Gamma$. Then by the Sublemma, $\phi \in f\Delta$. By axiom ADπ, $[\pi]\phi \in f\Gamma$.

Choose an arbitrary ψ in $f\Delta$. We show that ψ must be a member of Θ. By construction of f and ADπ, $[\pi]\psi \in f\Gamma$. If $\psi \in f\Gamma$, we get the result directly, by the A$\approx$8 instance $\approx_\pi \Box\psi \leftrightarrow \approx_\pi (\psi \wedge [\pi]\psi)$. So let's assume that $\psi \notin f\Gamma$. Reviewing the situation, we find that

ϕ and ψ are in $f\Delta$.

$\neg\phi$ and $\neg\psi$ are members of $f\Gamma$.

It follows that $(\phi \rightarrow \psi)$ is a formula in $f\Gamma$ and $f\Delta$.

Thus $[\pi](\phi \rightarrow \psi) \in f\Gamma$.

Applying A$\approx$8 once again, we find that $\Box(\phi \rightarrow \psi) \in f\Gamma$. But by assumption, $f\Gamma <_\Lambda \Theta$, so $(\phi \rightarrow \psi) \in \Theta$. But $\phi \in \Theta$ under the assumption that $f\Gamma \neq \Theta$. So $\psi \in \Theta$ by the fact that Θ is closed under the rule MP.

This reasoning has established that if $f\Gamma \neq \Theta$, then $f\Delta \subseteq \Theta$. Since no maximal consistent set may be the proper subset of another, it follows that $f\Delta = \Theta$.

We have shown that if $f\Gamma <_\Lambda \Theta$ then $f\Gamma \neq \Theta$ implies $f\Delta = \Theta$. Equivalently,

$$f\Gamma <_\Lambda \Theta \text{ implies } \Theta = f\Gamma \text{ or } \Theta = f\Delta$$

■

Proving the result for iteration and generalised iteration depends upon proving it for composite programmes of the form $(\alpha;\beta)$. The 'obvious' bridge principle

$$\approx_{(\alpha;\beta)}\Box\phi \leftrightarrow \approx_\alpha \Box\phi \wedge [\alpha] \approx_\beta \Box\phi ,$$

which is the one employed in [42, 35], fails. While it would be valid if ϕ were restricted to $L_{I/O}$ formula, we can not live with this restriction. Even the formulae introduced in §§ 5.1.2 and §§ 5.1.3 require U4*T* subformula of depth two or three (*e.g.* $\approx_\alpha \Box\Diamond true$.) If the reader considers the formula

$$\approx_{(\alpha;\beta)}\Box\Diamond\phi \leftrightarrow \approx_\alpha \Box\Diamond\phi \wedge [\alpha] \approx_\beta \Box\Diamond\phi ,$$

the problem will become clear. It says that if ϕ happens infinitely often in $(\alpha;\beta)$ then ϕ also occurs infinitely often during α. Clearly this principle will not do. The proof of the theorem has cetaceously beached upon this inhospitable shoal.

Of course, the fact that the theorem is not completed explains the absence of bridge axioms for iteration and the rest. These principles will emerge as the inductive steps in the theorem are completed.

The proof of the fundamental theorem requires the statement of one further Lemma. We have established (conjectured) that $M_\Lambda(f\Gamma)$ is an (α,Γ)-image of the canonical model. We still need to show that

5.5.10 **Lemma** For every $\alpha \in Pgm$, $\Gamma \in S_\Lambda$ and $\phi \in Fma$,

$$M_\Lambda \models_{f_\alpha\Gamma} \phi \text{ iff } M_\Lambda(f_\alpha\Gamma) \models_{f_\alpha\Gamma} \phi .$$

Proof:

By an induction on the complexity of ϕ, in which the only non trivial step is for formulae of the form $\Box\psi$. It is obvious that

$$M_\Lambda \models_{f\Gamma} \Box\phi \text{ implies } M_\Lambda(f\Gamma) \models_{f\Gamma} \Box\phi ,$$

since a submodel is a subset of the entire model.

To see the other direction, suppose that $M_\Lambda \not\models_{f_\alpha\Gamma} \Box\phi$. Then for some Δ, $f_\alpha\Gamma <_\Lambda \Delta$ and $M_\Lambda \not\models_\Delta \phi$. By the definition of the canonical relation $<_\Lambda$, it follows that $\Box\phi \notin f_\alpha\Gamma$. Since $f_\alpha\Gamma$ is a maximal consistent set, $\neg\Box\phi \in f_\alpha\Gamma$, whence $\approx_\alpha \neg\Box\phi \in \Gamma$. By Theorem (5.2.15)(2), $\neg\approx_\alpha \Box\phi \in \Gamma$.

A separate induction on α, similar to that of the previous theorem, establishes that there must be some state Δ' within $ct^{M_\Lambda}(\alpha,\Gamma)$ such that ϕ is false at $f_\alpha\Delta'$. By the theorem, $f_\alpha\Gamma <_\Lambda f_\alpha\Delta'$. It follows directly from the truth condition that

$$M_\Lambda(f_\alpha\Gamma) \not\models_{f_\alpha\Gamma} \Box\phi ,$$

as required.

This completes the Lemma. ■

Assuming the result (5.5.9), we may now prove the

Fundamental Theorem for Λ: $\forall \phi \in Fma$ and $\forall \Gamma \in S_\Lambda$,

$$\phi \in \Gamma \text{ iff } M_\Lambda \models_\Gamma \phi.$$

Proof:

By induction on ϕ.

For ϕ a boolean expression, the result is Thm (5.5.1).

For $(\phi \rightarrow \psi)$ and $[\alpha]\phi$ the result is just as in § 4.

For $\Box\phi$ the result follows as in §§ 5.1.4.

Finally, we consider the formula $\approx_\alpha \phi$. Let f be the α canonical function from the generated submodel $M_\Lambda(\Gamma)$ to $M_\Lambda(\approx_\alpha(\Gamma))$. Then

$\approx_\alpha \phi \in \Gamma$
iff $\phi \in \approx_\alpha(\Gamma) = f\Gamma$ by definition of f,
iff $M_\Lambda \models_{f\Gamma} \phi$ by hypothesis on ϕ and $f\Gamma$
iff $M_\Lambda(f\Gamma) \models_{f\Gamma} \phi$ by Lemma (5.5.10).

But, by theorems (5.5.8) and (5.5.9) $M_\Lambda(f\Gamma)$ is an (α,Γ)-image of M_Λ. By theorem (5.2.14) and the truth condition on $\approx_\alpha \phi$, it follows that $M_\Lambda \models_\Gamma \approx_\alpha \phi$.

This completes the inductive step for $\approx_\alpha \phi$, and proves the Fundamental Theorem. ■

Chapter 6

Conclusion

The principle contributions of this thesis are

1. An extension of an existing PDL-like programming logic to include an account of generalised iteration. A generalised iterator is the syntactic representation of a process. A process is an individual component within a distributed network.

2. The development of a semantics for non-termination and failure.

3. The partial axiomatisation of a logic for the dynamic behaviour of processes.

We claim that the system presented qualifies as a logic for individual processes.

6.1. Directions for Further Work

We conclude this thesis by indicating how to extend the system to account for distributed computation. Of course the first thing to do is to finish the completeness proof begun in § 5, or show that this can not be done. We believe that it *is* possible to complete the axiomatisation.

One obvious extension is to increase the power of the modal fragment L_K. This could be done by adding the operators required for reasoning about partial, as opposed to linear, orders. These operators are those employed in the temporal logic of branching time, for which see [38]. Alternatively, an Until or **atnext** operator (see §§ 3.2) could be added. In either case the logical system would require more axioms and bridge principles. If an operator as powerful as Until *were* to be added, the language would approach Process Logic in expressive power.

According to the model of distributed computation presented in §§ 2.1, a distributed network is a set of processes connected by communication channels. Distributed computation consists in the interaction of the various processes in the network. This interaction occurs exclusively via the exchange of messages between pairs of processes. In order to develop a full blown logic of distributed computation, it is necessary to account for the receiving and sending of messages by processes.

6.1.1. An Algebra for Data Buffers

The programming construct $C?x$ was introduced in §§ 2.1. Its meaning is "If channel C has a message waiting, read the message into variable x." As was indicated in §§ 2.2.2 this operation should be decomposed into two separate operations. There should be a boolean expression of some kind which can test for the availability of a message on a given channel. This expression may then serve as a guard within a generalised iterator. There should also be a read command, which reads a value from the channel into a variable.

A data type may be interpreted as an algebra. That is, a data type is an underlying structure and a set of operations on the structure. For instance, the class of Boolean expressions with the function symbol $\supset$ is a data type. A possible underlying structure for the Positive Integers is any infinite well ordered set. We propose that channels be represented as a data type, which, suggestively, we name *buffers*. The elements of the buffer algebra are (all of the) finite well ordered lists of integers. A channel is a variable of sort *buffer*, or *Bff*. If C is a channel variable, the value of C in a given state s is a list of messages which have arrived on Channel C, but which have not yet been read. Let λ represent the empty list, the identity element in the *buffer* algebra. We add to the algebra the undefined element ω. Thus, channel variables not "bound" to some existing physical channel may be said to be undefined. Call this augmented algebra, in the manner of §§ 4.1 Bff^+.

One possible operator on this new data type is the equality

$$f_{=} : Bff^{+} \times Bff^{+} \rightarrow \mathbb{B}^{+} .$$

Then, if C is a buffer variable, the expression $C f_{=} \lambda$ evaluates to the boolean constants *true*, *false* or ω according as C is empty, contains one or more messages, or is undefined. An appropriate guard for a process component required to read from a data buffer C is the boolean expression

$$not\text{–}(C f_{=} \lambda) .$$

Alternatively, it may turn out to be desirable not to have equality in the algebra. Given that our language supports the data type *integer* (as it must), a *length* function could be defined, which would return the current (integral) length of a buffer variable. The appropriate guard would then be

$$length(C) > 0 .$$

Next, a function *read* could be defined, mapping buffers to integers. The precise definition of *read* would vary, depending as a particular buffer was intended to be a queue, a stack, or some other list structure. An attempted read from an empty or undefined buffer should return the undefined element of the *integer* data type, which would result in programme failure.

Let C be a buffer variable, and x an integer variable. Read would in general be defined as a function

$$read : C \rightarrow \langle C' , x \rangle$$

where C' is the element of *Bff* resulting from removing an element from C. Suppose C is a queue. If the value of C in state s is (x_1,x_2,x_3), then the binary relation of the programme $(read(C,x))$ should include a pair $\langle s,t \rangle$ such that the value of C in t

is $(x_2,x_3$ and the value of x is x_1. On the other hand, if C is a stack, then the value of C at t should be x_1,x_2, while the value of x should be x_3.

In [14, §3], Goldblatt extends the logic which we presented in § 4 to the first order case. The result is quite general; it provides for first order expressions in arbitrary data types. The first step in accounting for data buffers is then to carry out the (direct) extension of these results to our system, and suitably axiomatise the behaviour of at least one variety of *read* command. Some indications of the algebraic structure of finite and infinite queues and stacks is to be found in [46]. The work cited shows that stacks can not be characterised in linear temporal logic using a language more powerful than that of §§ 5.1.4 - the language employed contains the *Until* operator. Since Goldblatt has shown that the first order extension of our system can define the integers up to isomorphism, it seems likely that the behaviour of infinite stacks should prove no obstacle. This, of course, needs to be proved.

6.1.2. Message Arrival

Accounting for the arrival of new messages presents an interesting logical problem. We propose that message arrivals be regarded as events occurring *in time*. A path in time, as witnessed by a particular process, would record the history of message arrivals.

The proposed solution is to *tense* the modal logic already developed. After the fashion of [20], we introduce a partial ordering on the class of all models of the first order extension proposed above.

Let M_i, M_j be two models. By $v_{i,s}$ we denote the valuation function at state s in model M_i.

We say that $M_i \leqslant M_j$ if there exists some function f which maps the states of M_i onto the states of M_j such that

1. the valuation $v_{i,s}$ agrees with $v_{j,fs}$ for all variables except buffer variables, and
2. for all buffer variables C, $v_{i,s}(C)$ is a *suffix of* $v_{j,fs}(C)$.

Buffer C is a suffix of buffer D if C is an initial segment of D. So, if fs is a state in M_j, it means that fs is just like state s in M_i except that more messages have arrived on one or more channels.

The precise nature of the function f can not be guessed. It is probable that it must be at least a p-morphism with respect to the relations $[\![\cdot]\!]$. The interaction of the time ordering relation, and the relation between models implicit in the modality $\approx_\alpha$ will determine the nature of f.

The language of the logic would be extended by adding the usual tense logical modality G. The interpretation of a formula $G\phi$ is

$$M_i \models_s G\phi \quad \text{iff} \quad \forall M_j, M_i \leqslant M_j \ \textit{implies} \ M_j \models_{fs} \phi$$

This extended language could account for certain network behaviours not presently representable. Deadlock freedom, for instance, could be expressed by an assertion that certain guards will become true at some later instant in time. That is, the arrival of a new message is guaranteed to occur at some later time.

6.1.3. Message Transmission

The final step needed to account for process interaction must be to provide a semantics for the message sending command $C!x$. Recall that this meant: transmit the value of variable x on Channel C. The intended result of the execution of $C!x$ by one process is that, at least sometimes, a message should arrive at the process at the other end of C. Moreover, the value received at one end should sometimes be "the same as" the value that was transmitted. A solution to this problem seems to require a modal representation of causation. Beyond this, the author has not even a conjecture to offer.

Appendix A

Statements of Theorems Referenced

2.4.1(1) Rule of Terminal Implication

TI: *If* $\vdash_{\Lambda} \phi \rightarrow \psi$ *then* $\vdash_{\Lambda} [\alpha]\phi \rightarrow [\alpha]\psi$ ■

2.4.1(3)

$\vdash_{\Lambda} [\alpha](\phi \wedge \psi) \leftrightarrow [\phi] \wedge [\psi]$ ■

2.4.2(1)

$\vdash_{\Lambda}$ *not*$-\epsilon \rightarrow \neg\epsilon$ ■

2.4.4(1)

If $\phi \in \Sigma$ *then* $\Sigma \vdash_{\Lambda} \phi$. ■

2.4.4(2)

If $\Sigma \vdash_{\Lambda} \phi$ *and* $\Sigma \subseteq \Delta \subseteq Fma$, *then* $\Sigma \vdash_{\Lambda} \phi$. ■

2.4.4(3)

If $\vdash_{\Lambda} \phi$ *then* $\Sigma \vdash_{\Lambda} \phi$. ■

2.4.6(2) *If* Γ *is a* Λ*-theory, then*

Γ *is* deductively closed, *i.e.* $\Gamma \vdash_{\Lambda} \phi$ *only if* $\phi \in \Gamma$. ■

2.4.6(4) *If* Γ *is a* Λ*-theory, then*

Γ *is* Λ*—consistent iff* **false** $\notin \Gamma$ *iff* $\Gamma \neq Fma$. ■

2.4.6(5) *If* Γ *is a* Λ*-theory, then*

$\phi \wedge \psi \in \Gamma$ *iff* $\phi \in \Gamma$ *and* $\psi \in \Gamma$. ■

2.4.8(1) (Corollary to the Deduction Theorem)

$\Sigma \cup \{\phi\}$ *is* Λ*—consistent iff* $\Sigma \nvdash_{\Lambda} \neg\phi$. ■

2.4.8(2) (Corollary to the Deduction Theorem)

$\Sigma \cup \{\neg\phi\}$ *is* Λ*—consistent iff*$\Sigma \nvdash_{\Lambda} \phi$. ■

2.4.10 (Corollary to the α-Deduction Lemma)

If Γ *is a* Λ*-theory, then*
$[\alpha]\phi \in \Gamma$ *iff* $\Gamma(\alpha) \vdash_{\Lambda} \phi$, *where* $\Gamma(\alpha) = \{\psi : [\alpha]\psi \in \Gamma\}$. ■

2.5.1(6)

If Γ *is a maximal* Λ*-theory, then* Exactly *one of* $\neg D\epsilon, \epsilon$, *and not—*ϵ belongs toΓ
for all $\epsilon \in Bxp$. ■

2.5.5 Completeness for Boolean Expressions

For any $\epsilon \in Bxp$, $M_\Lambda \models_T \epsilon$ *iff* $\epsilon \in \Gamma$.

■

References

1. Howard Barringer, Ruurd Kuiper and Amir Pnueli. Now You May Compose Temporal Logic Specifications. Proc. 16th ACM Symp. Theory of Computing, ACM, 1984, pp. 51-63.

2. Brian F. Chellas. *Modal Logic - an Introduction.* Cambridge University Press, 1980.

3. Edmund M. Clarke and E. Allen Emerson. *Lecture Notes in Computer Science.* Volume 131: Design and Sythesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Dexter Kozen, Ed., *Logics of Programmes,* Springer-Verlag, 1982, pp. 52-71. Proceedings of Logics of Programs Workshop, May 1981.

4. J. N. Crossley, *ed.*. Reminiscences of Logicians. Algebra and Logic, Australian Mathematical Society, 1974, pp. 1-62. Springer Verlag Lecture Notes in Mathematics Vol. 450.

5. Edsger W. Dijkstra. "Self-Stabilizing Systems in Spite of Distributed Control". *CACM 17,* 11 (November 1974), 643-644.

6. Edsger W. Dijkstra. *A Discipline for Computer Programming.* Prentice-Hall, 1976.

7. E. Allen Emerson. "Alternative Semantice for Temporal Logics". *Theoretical Computing Science 26,* 1 (September 1983), 121-130.

8. E. Allen Emerson and A. Prasad Sistla. Deciding Branching Time Logic. Proc. 16th ACM Symp. Theory of Computing, ACM, 1984, pp. 14-23.

9. Michael J. Fischer and Richard E. Ladner. "Propositional Dynamic Logic of Regular Programs". *Journal of Computer and System Sciences 18* (1979), 194-211.

10. Melvin Fitting. *Synthese Library.* Volume 169: *Proof Methods for Modal and Intuitionistic Logics.* D. Reidel, 1983.

11. Lawrence Flon and Morihisa Suzuki. Consistent and Complete Proof Rules for the Total Correctness of Parallel Programms. 19th FOCS, IEEE, 1978, pp. 184-192.

12. L. Flon, and N. Suzuki. "The Total Correctness of Parallel Programs". *SIAM Journal of Computation 10,* 2 (May 1981), 227-246.

13. Dov M. Gabbay. *Synthese Library.* Volume 92: *Investigations in Modal and Tense Logics with Applications.* D.Reidel, 1976.

14. Robert Goldblatt. *Lecture Notes in Computer Science.* Volume 130: *Axiomatizing the Logic of Computer Programming.* Springer-Verlag, 1982.

15. David Harel. *Lecture Notes in Computer Science*. Volume 68: *First-Order Dynamic Logic*. Springer-Verlag, 1979.

16. David Harel, Dexter Kozen and Rohit Parikh. "Process Logic: Expressiveness, Decidability, Completeness". *Journal of Computer and System Sciences 25* (1982), 144-170.

17. Matthew Hennessy. "Axiomatizing the Logic of Delay". *Acta Informatica 26*, Fasc. 1 (1984), 61-88.

18. C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". *Communications of the ACM 12* (1969), 576-583.

19. . A Formal Description Technique Based on Extended State Transition Model. ISO /TC 96/SC 21/WG 1 Subgroup B, ISO, December, 1984.

20. Ray Jennings. Lecture Notes. Mimeographed. Course notes for Philosophy 310 (Modal Logic and its Applications) at Simon Fraser University.

21. F. Kroger. "A Generalized Nexttime Operator in Temporal Logic". *Journal of Computer and System Sciences 29* (1984), 80-98.

22. R. Kuiper and W. P. de Roever. Fairness Assumptions for CSP in a Temporal Logic Framework. Formal Description of Programming Concepts - II, IFIP, June, 1982, pp. 159-167. Published by North Holland.

23. I. M. Lakatos. *Proofs and Refutations*. Cambridge University Press, 1977.

24. Leslie Lamport. "The "Hoare Logic" of Concurrent Programs". *Acta Informatica 14*, 1 (1980), 21-37.

25. Zohar Manna and Pierre Wolper. "Synthesis of Communicating Processes from Temporal Logic Specifications". *TOPLAS 6*, 1 (January 1984), 68-93.

26. Albert Meyer. *Lecture Notes in Mathematics*. Volume 453: The Weak Second-Order Theory of One-Succesor is not Elementary Recursive. In *????*, Springer Verlag, 1974. Logic Colloqium Proceedings.

27. H. Nishimura. "Descriptively Complete Process Logic". *Acta Informatica 14*, 4 (1980), 359-369.

28. Maurice Nivat. *NATO Advanced Studies Institutes Series C*. Volume 91: Behaviours of Processes and Synchronized Systems of Processes. In Manfred Broy, Ed., *Theoretical Foundations of Programming Methodology*, D. Reidel, 1982, pp. 473-550.

29. Susan Owicki and Leslie Lamport. "Proving Liveness Properties of Concurrent Programs". *TOPLAS 4*, 3 (July 1982), 455-498.

30. Rohit Parikh. A Decidability Result for a Second-Order Process Logic. 19th FOCS, IEEE, 1978, pp. 177-183.

31. Shlomit S Pinter and Pierre Wolper. A Temporal Logic for Reasoning About Partially Ordered Computations. Third Annual Symposium on Principles of Distributed Computing, ACM, August, 1984, pp. 28-37.

32. Amir Pnueli. The Temporal Logic of Programs. Proceedings of the 18th Symposium on the Foundations of Computer Science, IEEE, Nov., 1977, pp. 46-57.

33. Amir Pnueli. "The Temporal Semantics of Concurrent Programmes". *Theoretical Computer Science 13* (1981), 45-60.

34. Vaughn Pratt. Semantical Considerations on Floyd-Hoare Logic. Proc. 17th IEEE FOCS, IEEE, October, 1976, pp. 109-121.

35. Vaughan R. Pratt. A Practical Decision Method for Propositional Dynamic Logics. Proc. 10th ACM Symp. on the Theory of Computing, ACM, 1977, pp. 326-337.

36. M. Presburger. Uber die Vollstandigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. Comptes-rendus du I Congres des Mathematiciens des Pays Slaves, Kongrezu Matematykow Krajow Slowianskich, 1930, pp. 92-101.

37. Michael O. Rabin. "Decidability of Second-Order Theories and Automata on Infinite Trees". *Trans. Amer. Math. Society 141* (July 1969), 1-35.

38. Nicholas Rescher and Alasdair Urquhart. *Library of Exact Philosophy.* Volume 3: *Temporal Logic.* Springer-Verlag, 1971.

39. Hartley Rogers Jr.. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

40. Richard Schwartz and P.M. Melliar-Smith. From State Machinesa to Temporal Logic: Specification Methods for Protocol Standards. Protocol Specification, Testing and Verification, IFIP WG 6.1, May, 1982, pp. 3-20.

41. Krister Segerberg. An Essay in Classical Modal Logic (in three Volumes). Filosofiska Studier nr 13, Uppsala Universitet, 1971.

42. Krister Segerberg. "Applying Modal Logic". *Studia Logica XXXIX*, 2/3 (1980), 275-295.

43. Joseph E. Stoy. *NATO Advanced Studies Institutes Series C.* Volume 91: Semantic Models. In Manfred Broy, Ed., *Theoretical Foundations of Programming Methodology*, D. Reidel, 1982, pp. 293-324.

44. Dolph Ulrich. "The Finite Model Property and Recursive Bounds on the Size of Counter-Models". *Journal of Philosophical Logic 12* (1983), 477-480.

45. J. F. A. K. van Benthem. *Synthese Library*. Volume 156: *The Logic of Time*. D. Reidel, 1983.

46. Moshe Y. Vardi and Pierre Wolper. Automata Theoretic Techniques for Modal Logics of Programs. Proc. 16th ACM Symp. Theory of Computing, ACM, 1984, pp. 446-456.

47. Moshe Y. Vardi and Pierre Wolper. Yet Another Process Logic. Logics of Programs, 1984, pp. 501-513. Proceedings of a workshop at CMU, June 1983.

48. Job Zwiers, Arie de Bruin and Willem Paul de Roever. *Lecture Notes in Computer Science*. Volume 164: A Proof System for Partial Correctness of Dynamic Networks of Processors. In *Logics of Programs*, Springer-Verlag, 1984, pp. 513-527. Proceedings of a workshop at CMU, June 1983.